OBJECT-ORIENTED PROGRAMMING IN JAVA

A BEGINNER'S GUIDE

Luís Simões da Cunha

Table of Contents

📘 Introduction

This document was designed as a practical and accessible guide to support students taking their first steps in **Object-Oriented Programming (OOP)**.
It is a complementary resource created by the instructor of the course with the same name, aiming to:

- 📚 Follow the topics covered in class
- 👨‍💻 Provide clear and functional examples in Java
- 🔍 Help students understand abstract OOP concepts
- 🛠️ Offer a starting point for project development

👩‍🏫 Who is this manual for?

This manual is intended for first-year Computer Science (or related) students who are beginning to explore the fundamentals of object-oriented programming:

- Classes and Objects
- Inheritance and Polymorphism
- Encapsulation and Abstraction
- Interfaces and Abstract Classes
- Data structures and collections handling
- GUI development with Swing
- Error handling and good practices

🧭 How to use this document?

The manual is divided into short, pedagogical, and progressive chapters, with: ✅ Step-by-step explanations
✅ Commented examples
✅ Mini-labs and challenges
✅ Code that can be tested directly in Visual Studio Code
✅ A complete project that can be used as a reference for assessment

---

🎓 One of many learning tools

This manual is just one of several tools developed by the OOP course instructor as part of a broader pedagogical strategy, including:

- Practical classes with individual support
- Interactive resources on Moodle
- A question bank for self-assessment
- Guided projects and oral defense guides

🙌 Wishing you success

I hope this resource helps you:

- Learn with joy
- Explore with curiosity
- Program with confidence

Because programming well is not about memorizing commands — it's about understanding ideas. And this manual is here to help you do just that.

📘 Chapter 1 – Introduction to Programming and the Java Language

---

🧠 1.1 What is programming?

Programming is giving instructions to a computer so it can solve problems for us.
But careful: the computer only does **exactly** what we tell it to do — not what we "think" it should understand.

💡 *Simple analogy*: imagine giving instructions to a robot that has no common sense. If you say "make dinner", it'll just stand there. But if you give step-by-step instructions like "grab a pot", "fill it with water", "turn on the stove", it can do it!

☕ 1.2 Why Java?

Java is one of the most widely used programming languages in the world, for several reasons:

✅ Runs on almost any system (Windows, macOS, Linux...)
✅ Used in mobile apps (like Android), web apps, and even in banks!
✅ Built around OOP — which helps organize projects well
✅ Huge community and lots of learning resources

🔁 1.3 How does a Java program work?

Unlike interpreted languages like Python, Java runs in two steps:

1. **Compile** the code (convert it into bytecode)
2. **Run** it using the Java Virtual Machine (JVM)

🧱 Compilation + Execution = Portability

Write once, run anywhere!

⚙️ 1.4 Basic Tools

To start programming in Java, you'll need:

- 🧰 JDK – Java Development Kit (the engine)
- ✍️ A text editor (like VS Code or IntelliJ)
- 💻 Terminal/command line

💡 *Tip*: Use VS Code with the “Extension Pack for Java” — lightweight, free, and great for beginners.

💬 1.5 Your First Java Program

Let’s see the world’s most famous program: "Hello, World!" 😄

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

🔍 **Step-by-step explanation**:

- `public class HelloWorld`: defines a class named HelloWorld
- `public static void main(...)`: the program’s entry point
- `System.out.println(...)`: prints a message to the screen

📥 To compile and run:

```
javac HelloWorld.java   # compile
java HelloWorld         # run
```

👩‍🏫 1.6 Learning Java with Objects from the Start

Java was designed with OOP at its core. That means everything revolves around classes and objects — as you’ll soon see.

🔁 **Fast-forward summary**:

- You create **classes** → like recipes
- Then create **objects** from them → like real cakes
- Each object has **attributes** (state) and **methods** (actions)

✅ 1.7 Key Concepts from this Chapter

Concept	Simple Explanation
Program	Set of instructions for the computer
Java	Cross-platform, object-oriented language
Compile	Convert code into a machine-readable format
Class	Blueprint or recipe to create objects
Object	Instance created from a class
main Method	Where a Java program starts executing

🧪 1.8 Mini-Lab

Create a program called `Introduction.java` that prints your name, age, and degree:

```
public class Introduction {
    public static void main(String[] args) {
        System.out.println("My name is John.");
        System.out.println("I am 18 years old.");
        System.out.println("I study Computer Science.");
    }
}
```

➡️ Try changing the messages.

➡️ Add more lines with fun facts about yourself.

🎯 1.9 Challenge

Create a program named `SimpleCalculator` that displays the result of:

- an addition
- a subtraction
- a multiplication
- a division

Use only `System.out.println()` with simple operations inside.

📘 Chapter 2 – First Steps with Java: Variables, Types, and Operators

---

🧠 2.1 What are variables?

A variable is like a "little box" where we store data. That box has a name and a type of data it can hold, like numbers or text.

💡 *Simple analogy*: Imagine you have several drawers in your office. Each drawer has a name, and inside each one, you keep something specific: pens, documents, etc.
In Java, variables work similarly. For example, a variable called `age` can store a person's age, and `name` can store someone's name.

🔠 2.2 Data Types

In Java, there are two main categories of data types: **primitive types** and **reference types**.

Primitive types:

- `int`: whole numbers (e.g. 10, -7, 0)
- `double`: decimal numbers (e.g. 3.14, -0.5)
- `char`: a single character (e.g. 'a', '1', '#')
- `boolean`: true or false values

💡 *Example*:

```
int age = 20;
double weight = 70.5;
char letter = 'A';
boolean isRaining = false;
```

Reference types:

- `String`: a sequence of characters (e.g. "Hello World")
- `Arrays`: a list of values of the same type

💡 *Example*:

```
String name = "John";
int[] ages = {20, 21, 22};
```

➗ 2.3 Operators: How to do calculations in Java

Operators are used to perform operations on variables or values. Java includes several types:

2.3.1 Arithmetic Operators (for calculations):

- `+`: addition (e.g. `5 + 3`)
- `-`: subtraction (e.g. `5 - 3`)
- `*`: multiplication (e.g. `5 * 3`)
- `/`: division (e.g. `6 / 2`)
- `%`: modulo – remainder of division (e.g. `7 % 3` gives `1`)

💡 *Example*:

```
int a = 5;
int b = 3;
int sum = a + b;       // sum = 8
int remainder = a % b; // remainder = 2
```

2.3.2 Comparison Operators (to compare values):

- `==`: equal to
- `!=`: not equal
- `>`: greater than
- `<`: less than
- `>=`: greater or equal
- `<=`: less or equal

💡 *Example*:

```
int a = 5;
int b = 3;
boolean isEqual = (a == b);  // false
boolean isGreater = (a > b); // true
```

2.3.3 Logical Operators (to make decisions):

- `&&`: and (e.g. `a > b && b > 0`)
- `||`: or (e.g. `a > b || b > 0`)
- `!`: not (e.g. `!(a > b)`)

💡 *Example*:

```
boolean result = (a > b) && (b > 0);  // true
```

🖊️ 2.4 How to declare variables in Java?

To declare a variable, specify its type and name. You can assign a value right away using =.

💡 *Example*:

```
int age = 18;
String name = "Maria";
```

If you don't want to assign a value immediately:

```
int height;
```

💡 *Best practice*: It's usually better to initialize your variable when you declare it, to avoid errors later.

⚙️ 2.5 How to use variables in calculations?

Now that you know how to declare and assign values, let's use them in calculations.

💡 *Example*:

```
int a = 10;
int b = 5;
int result = a + b;
System.out.println(result);  // outputs: 15
```

📑 2.6 Practical Example

Let's build a program that calculates the average of three numbers entered by the user:

```
import java.util.Scanner;

public class Average {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the first number: ");
        double num1 = scanner.nextDouble();
```

```
        System.out.print("Enter the second number: ");
        double num2 = scanner.nextDouble();
        System.out.print("Enter the third number: ");
        double num3 = scanner.nextDouble();

        double average = (num1 + num2 + num3) / 3;
        System.out.println("The average is: " + average);
    }
}
```

➡️ This program uses the `Scanner` class to read numbers from the user, then calculates their average.

💡 2.7 Challenge

Create a program that calculates the tax on a product.
The program should ask for the price and the tax rate, and then show the final price with tax.

🎯 2.8 Summary of Concepts

Concept	Simple Explanation
Variables	"Boxes" to store data (numbers, text, etc.)
Data types	Specify what type of data a variable can store
Arithmetic operators	Tools to do calculations (+, -, *, /, %)
Comparison operators	Tools to compare values (==, !=, >, <, etc.)
Logical operators	Tools to make decisions (&&, `

📘 Chapter 3 – Control Flow: Making Decisions

🧠 3.1 What is "control flow"?

In programming, not everything should happen in a fixed order. Sometimes, you want your program to make decisions or repeat actions.

➡️ Control flow lets your code:

- Make decisions using conditions (`if`, `else`)
- Choose between multiple paths (`switch`)
- Repeat code blocks (`while`, `for`, `do-while`)

💡 *Simple analogy*: Imagine you're following a recipe.
If you have eggs, you make a cake. If not, you make pancakes.
That choice — based on a condition — is control flow.

🧩 3.2 The if Statement

`if` is the simplest way to make decisions in Java.

```
int age = 18;

if (age >= 18) {
    System.out.println("You are an adult.");
}
```

➕ **if-else**: choose between two paths

```
if (age >= 18) {
    System.out.println("You are an adult.");
} else {
    System.out.println("You are a minor.");
}
```

🪜 3.3 if-else-if: multiple cases

```
int grade = 15;

if (grade >= 18) {
    System.out.println("Excellent!");
} else if (grade >= 10) {
    System.out.println("Passed");
} else {
    System.out.println("Failed");
}
```

🔍 Java checks the conditions from top to bottom. As soon as one is true, it ignores the rest.

🔁 3.4 Repeating with loops

When you need to repeat a block of code, use loops.

while: repeats while the condition is true

```
int i = 1;
while (i <= 5) {
    System.out.println("Count: " + i);
    i++;
}
```

do-while: runs at least once

```
int i = 1;
do {
    System.out.println("Count: " + i);
    i++;
} while (i <= 5);
```

➡️ Key difference: `do-while` runs the block at least once, even if the condition is false at the start.

for: best when you know how many times to repeat

```
for (int i = 1; i <= 5; i++) {
    System.out.println("Count: " + i);
}
```

💡 *The for loop has three parts*:

- `int i = 1`: initializes the variable
- `i <= 5`: loop condition
- `i++`: increment at each repetition

🎛️ 3.5 The switch Statement

`switch` lets you test several options — like a menu.

```
int day = 3;

switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

⚠️ *Important*: without `break`, Java keeps executing the next cases — this is called *fall-through*.

⛔ 3.6 break and continue

- `break`: exits the loop or switch immediately
- `continue`: skips the rest of the current loop and moves to the next iteration

Example with continue:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    System.out.println(i);
}
// Output: 1, 2, 4, 5 (3 is skipped)
```

📑 3.7 Practical Example: Grade Classifier

```
import java.util.Scanner;

public class Classifier {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter grade (0-20): ");
        int grade = scanner.nextInt();

        if (grade >= 18) {
            System.out.println("Excellent!");
        } else if (grade >= 10) {
            System.out.println("Passed.");
        } else {
            System.out.println("Failed.");
        }
    }
}
```

🧪 3.8 Mini-Lab

Create a program called `Counter` that:

- Shows numbers from 1 to 10
- Skips the multiples of 3 (use `continue`!)

🎯 3.9 Challenge

Create a program that:

- Asks the user what operation they want to perform (+, -, *, /)
- Reads two numbers
- Uses `switch` to apply the correct operation

🧠 3.10 Summary of Concepts

Concept	Explanation
if / else	Choose between two or more paths
while, for, do-while	Repeat code blocks multiple times
switch	Alternative to `if-else-if` for multiple options
break	Exit a loop or block
continue	Skip an iteration and continue the loop

📘 Chapter 4 – Introduction to Classes, Objects, and Methods

🎯 4.1 We've reached the most important part: OOP

🚨 Everything we've learned so far (variables, operators, loops) are essential tools…
But note: we haven't been coding "Java-style" yet! 🤷

👉 From this chapter on, you're stepping into **Object-Oriented Programming (OOP)** — the way Java projects are meant to be built.

🧱 4.2 What is a class, really?

A class is a **blueprint**. It defines:

- What **attributes** (data) it has
- What **methods** (actions) it can perform

💡 *Simple analogy*:
Think of a cookie cutter. It defines the shape of cookies, but it's not a cookie itself.
The actual cookies are the **objects** created from that shape.

🧊 4.3 What is an object?

An object is a **concrete instance** of a class.

Real-world example:

- Class: Car
- Object: My red car from brand X with license plate Y

In code:

```
Car myCar = new Car();
```

🤌 4.4 Creating your first class

```
public class Person {
    String name;
    int age;

    void introduce() {
```

```
        System.out.println("Hi, my name is " + name + " and I am " + age
+ " years old.");
    }
}
```

- `name` and `age` → attributes (data)
- `introduce()` → a method (action)

🧪 4.5 Creating and using an object

```
public class TestPerson {
    public static void main(String[] args) {
        Person p1 = new Person();  // Create object
        p1.name = "Maria";
        p1.age = 20;
        p1.introduce();  // Call method
    }
}
```

🧠 What's happening here?

- `new Person()` → creates a new object in memory
- `p1.name = "Maria"` → assigns data to an attribute
- `p1.introduce()` → performs the method's action

🧰 4.6 One class per file

Golden rule:

🗂️ Each public class should be in its **own file**, with the same name.

Example:

- Class `Person` → file `Person.java`
- Class `TestPerson` → file `TestPerson.java`

📦 4.7 Mental organization: a class ≠ a complete program

A class on its own doesn't run.
You need a class with a `main()` method to start execution.

So we have:

- **Models (classes)** → `Person`, `Car`, `Book`

- **Executors (testers)** → `Main`, `CarTest`, `LibraryApp`

🔁 **4.8 Reusing methods: avoid repeated code** You can create methods to manipulate the class's own data — improving clarity and reuse.

```
public class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}
```

Use in `main()`:

```
Calculator calc = new Calculator();
int result = calc.add(3, 5);
System.out.println("Result: " + result);
```

🔎 **4.9 Summary of terms (with examples!)**

Concept	Definition	Example
Class	Blueprint or model	`class Person { ... }`
Object	Instance of a class	`Person p1 = new Person();`
Attribute	Data belonging to the class	`String name;`
Method	Action the class performs	`void introduce() { ... }`
Instantiate	Create an object	`new Person()`

🧪 **4.10 Mini-lab**

Create a class `Book` with:

- Attributes: `title`, `author`, `year`
- A method `showInfo()` that prints the book details

Then, in `main()`, create two books and display their information.

🎯 **4.11 Challenge**

Create a class `Dog` with:

- Attributes: `name`, `breed`
- Methods: `bark()` (prints "Woof woof!") and `introduce()`

In `main()`, create two dogs and call their methods.

🧠 4.12 Final summary

🧩 This is the true starting point of OOP. Until now, we were only learning Java syntax. Now we're thinking like **object-oriented programmers**.

🔑 Remember:

- A class defines what objects are and do
- An object is a concrete instance
- We can create methods to act upon the object’s data

📘 Chapter 5 – Constructors: Giving Life to Objects with Initial Values

---

🧠 5.1 What is a constructor?

A constructor is a **special method** automatically called when you create an object with `new`.
Its role is simple but crucial: **give initial values** to the object's attributes.

💡 *Simple analogy*:
When assembling furniture, you follow a manual (the class), and the finished item (the object) already has screws in place.
The constructor is the **assembly step**.

🧪 5.2 Creating a custom constructor

```
public class Person {
    String name;
    int age;

    // Constructor
    public Person(String initialName, int initialAge) {
        name = initialName;
        age = initialAge;
    }

    void introduce() {
        System.out.println("Hi, I'm " + name + " and I'm " + age + "
years old.");
    }
}
```

In `main()`:

```
Person p = new Person("John", 25);
p.introduce();
```

🧠 The constructor takes two parameters and uses them to fill in the object's attributes.

🧲 5.3 The `this` keyword

When the parameter name matches the attribute name, use `this` to distinguish them:

```
public Person(String name, int age) {
    this.name = name;
    this.age = age;
}
```

📌 `this.name` refers to the object's attribute

📌 `name` refers to the parameter

🆓 5.4 Default constructor

If you don't write any constructor, Java adds a default one (no parameters):

```
Person p = new Person();  // Works only if no custom constructor exists
```

❗ But if you define a constructor with parameters, the default one disappears.

You can write your own default constructor like this:

```
public Person() {
    name = "Unknown";
    age = 0;
}
```

🌀 5.5 Constructor overloading

You can have multiple constructors in the same class, with different parameter lists:

```
public class Person {
    String name;
    int age;

    public Person(String name) {
        this.name = name;
        this.age = 0;  // default value
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

➡️ Now you can create:

```
Person p1 = new Person("Ana");
Person p2 = new Person("Carlos", 30);
```

⚠️ 5.6 Mind the parameter order!

Constructors are distinguished by the **number and type of parameters**, in **order**.

So:

```
Person(String name, int age)
```

is **not** the same as:

```
Person(int age, String name)
```

🧰 5.7 Advantages of using constructors

✔️ Makes object creation clearer
✔️ Prevents missing important attributes
✔️ Ensures objects start in a valid state

🧪 5.8 Mini-lab

Create a class `Student` with:

- Attributes: `name`, `course`, `year`
- Two constructors:
  - One receiving all values
  - One receiving only `name` and using defaults for the rest

🎯 5.9 Challenge

Create a class `Product` with attributes `name`, `price`, and `stock`.

Create **three constructors**:

- One with all parameters
- One with only `name` and `price`
- One default constructor with base values

In `main()`, create three products and print their details.

🧠 5.10 Final Summary

Concept	Explanation
Constructor	Special method that initializes objects
`this`	Refers to the current object
Default constructor	Created by Java if none is declared
Overloading	Having multiple constructors with different parameters

📘 Chapter 6 – Encapsulation: Hiding to Protect

🧠 6.1 What is encapsulation?

Encapsulation is one of the **pillars of OOP**.
It means hiding a class's internal details, exposing **only what's necessary**.

💡 *Simple analogy*:
A microwave has buttons to heat, stop, and open.
You use the buttons — you don't need to see the wires inside.
That's encapsulation.

🧱 6.2 How is encapsulation done in Java?

We use **access modifiers** to control visibility:

Modifier	Visible to...
`private`	Only inside the class
`public`	Everywhere
`protected`	In the class and its subclasses
(none)	Only within the same package

🔒 6.3 Hiding attributes (best practice)

The good practice in OOP is:

🔐 Make attributes `private` and provide public methods to access them.

```
public class Account {
    private double balance;

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }
}
```

📌 The balance cannot be changed directly — only through allowed methods.

🎙️ 6.4 The role of getters and setters

- `getX()` → returns the value of an attribute
- `setX(value)` → changes the value (if allowed)

```
public class Person {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String newName) {
        if (!newName.isEmpty()) {
            name = newName;
        }
    }
}
```

🎯 The setter ensures that the name isn't empty — keeping the object in a valid state.

🤔 6.5 When not to use setters

Not every attribute should be changeable!

You can have **read-only** (getter only) or **write-only** attributes.

💡 *Example*:

```
public class Sensor {
    private final String id;

    public Sensor(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }
}
```

📌 Since `id` never changes, there is no `setId()`.

⚠️ 6.6 "But can't I just make everything public?"

You can… but you **shouldn't**.

Making attributes public allows this:

```
account.balance = -1000;
```

🧨 Result: the object ends up in an invalid and dangerous state.

Encapsulation **prevents that**.

🎯 6.7 Benefits of encapsulation

- ✔ Protects data integrity
- ✔ Allows validation rules
- ✔ Makes maintenance easier
- ✔ Hides unnecessary details from users

🧪 6.8 Mini-lab

Create a class `Temperature` with:

- A private attribute `celsius`
- Methods:
  - `getCelsius()`
  - `setCelsius(double value)` → only accepts values from -100 to 100
  - `getFahrenheit()` → converts to Fahrenheit

🎯 6.9 Challenge

Create a class `Product` with:

- Private attributes: `name`, `price`, `stockQuantity`
- Getters and setters with validation:
  - `price` cannot be negative
  - `stockQuantity` must be ≥ 0

🧠 6.10 Final Summary

Concept	Explanation
Encapsulation	Hiding the inner workings of a class
`private`	Protects internal data
Getters/Setters	Public methods to access or update attributes
Validation	Keeps objects in a consistent and safe state

📘 Chapter 7 – Inheritance: Reuse to Grow

🧠 7.1 What is inheritance?

Inheritance is a mechanism that allows one class to **inherit attributes and methods** from another.
The idea is simple: if several classes share common behaviors, you can define them in a **superclass** and let others inherit from it.

💡 *Simple analogy*:
Imagine a `Vehicle` class.
`Car`, `Motorbike`, and `Truck` are all vehicles. Instead of repeating code in each one, you place shared code in `Vehicle` and reuse it.

🧬 7.2 How is inheritance defined in Java?

```
public class Vehicle {
    int speed;

    void accelerate() {
        speed += 10;
        System.out.println("Accelerating: " + speed + " km/h");
    }
}

public class Car extends Vehicle {
    void honk() {
        System.out.println("Honk! 🚗");
    }
}
```

🔑 The keyword `extends` indicates that `Car` **inherits** from `Vehicle`.

📋 7.3 What does a subclass inherit?

A subclass inherits everything that is `public` or `protected` from the superclass:

- Attributes

- Methods

But:

- It **does not inherit constructors**
- It **cannot access private members** directly

🔁 7.4 Overriding methods (@Override)

A subclass can **change the behavior** of an inherited method by writing its own version:

```
public class Dog {
    void makeSound() {
        System.out.println("Generic sound");
    }
}

public class GermanShepherd extends Dog {
    @Override
    void makeSound() {
        System.out.println("Woof woof!");
    }
}
```

💬 7.5 The `super` keyword

`super` is used to:

1. Call a method from the superclass that was overridden
2. Invoke the constructor of the superclass

```
public class Animal {
    Animal(String name) {
        System.out.println("Animal: " + name);
    }
}

public class Cat extends Animal {
    Cat() {
        super("Felix");  // calls the superclass constructor
    }
}
```

🧱 7.6 Class hierarchies

You can build inheritance chains:

```
class Animal { ... }
class Mammal extends Animal { ... }
class Human extends Mammal { ... }
```

➡️ Each class inherits from the one above, accumulating and possibly customizing behaviors.

👥 7.7 Polymorphism with inheritance

With inheritance, you can treat subclass objects as if they were instances of the superclass:

```
Animal a = new Cat();
a.makeSound();  // calls the Cat version, if overridden
```

➡️ This is **polymorphism**, which we'll explore more in the next chapter.

🧪 7.8 Mini-lab

Create:

- A superclass `Employee` with `name` and a method `showInfo()`
- A subclass `Teacher` that adds `subject` and **overrides** `showInfo()`
- A `main()` that creates a teacher and displays their data

🎯 7.9 Challenge

Build a hierarchy:

- `Vehicle` with `brand`, `model`, and `accelerate()`
- `Car` inherits from `Vehicle` and has `honk()`
- `Truck` inherits from `Vehicle` and has `loadCargo()`
  Test the methods in `main()` using two different vehicles.

❗ 7.10 Be careful with inheritance

Although powerful, inheritance should be used when there is a true **"is-a"** relationship:

✔️ A Cat **is an** Animal → OK
❌ A Car **is an** Engine → No (use composition instead: a car **has an** engine)

💡 *Golden tip*: If the phrase "X is a Y" makes sense, then inheritance might be appropriate.

🧠 7.11 Final summary

Concept	Explanation
Inheritance	Allows one class to inherit from another
`extends`	Keyword used to inherit
`super`	Access superclass members or constructors
`@Override`	Indicates that a method is being redefined
Hierarchy	A chain of inheritance with multiple levels

📘 Chapter 8 – Polymorphism: One Name, Many Behaviors

🧠 8.1 What is polymorphism?

Polymorphism comes from Greek: **poly** (many) + **morphos** (forms).
In OOP, it means a single method name can behave **differently** depending on the object that executes it.

💡 *Simple analogy*:
Tell several people to "play music":

- A pianist plays the piano
- A guitarist plays the guitar
  Everyone responds to the same command, **in their own way**.

🧬 8.2 Polymorphism with inheritance

When you have a superclass and multiple subclasses that override the same method, you can use a reference of the superclass type to call the method — and Java will **automatically choose** the correct version:

```
class Animal {
    void makeSound() {
        System.out.println("Generic sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow");
    }
}
```

🧪 8.3 Using polymorphism in practice

```
public class Test {
    public static void main(String[] args) {
        Animal a1 = new Dog();
        Animal a2 = new Cat();

        a1.makeSound();  // Woof
        a2.makeSound();  // Meow
    }
}
```

➡️ Even though the reference is of type `Animal`, Java picks the **correct implementation at runtime**.

🔁 8.4 Advantage: generic and extensible code

Polymorphism lets us write code that works with **any object type** that shares a common structure (inherits or implements the same interface):

```
public class Zoo {
    public static void makeSound(Animal a) {
        a.makeSound();
    }

    public static void main(String[] args) {
        makeSound(new Dog());
        makeSound(new Cat());
    }
}
```

🤖 8.5 Polymorphism with interfaces

Polymorphism doesn't only come from inheritance!
It also works with interfaces — we'll explore this more in Chapter 10.

```
interface Printable {
    void print();
}

class Document implements Printable {
    public void print() {
```

```
        System.out.println("Printing document...");
    }
}

class Image implements Printable {
    public void print() {
        System.out.println("Printing image...");
    }
}

public class Test {
    public static void main(String[] args) {
        Printable p1 = new Document();
        Printable p2 = new Image();

        p1.print();  // Printing document...
        p2.print();  // Printing image...
    }
}
```

🧱 8.6 Complete example with object list

```
public class Main {
    public static void main(String[] args) {
        Animal[] animals = new Animal[3];
        animals[0] = new Dog();
        animals[1] = new Cat();
        animals[2] = new Dog();

        for (Animal a : animals) {
            a.makeSound();
        }
    }
}
```

➡️ Even with different types of animals, they can all be treated uniformly thanks to polymorphism.

🧪 8.7 Mini-lab

Create:

- A superclass `Employee` with method `calculateSalary()`
- A subclass `Teacher` with fixed salary
- A subclass `Technician` with hourly wage
  Create an array of `Employee` and calculate the salaries of several workers.

🎯 8.8 Challenge

Define an interface `Drawable` with a method `draw()`.

Then create two classes:

- `Circle` that prints "Drawing circle"
- `Square` that prints "Drawing square"

In `main()`, create a list of `Drawable` objects and draw them all in a loop.

❗ 8.9 Polymorphism ≠ Overloading

⚠️ Don't confuse polymorphism with **method overloading**!

- **Polymorphism**: multiple implementations of the same method in **different classes**
- **Overloading**: multiple methods with the same name but **different signatures** (more on this later)

🧠 8.10 Final summary

Concept	Explanation
Polymorphism	Allows using different objects in a generic way
`@Override`	Redefines an inherited method
Interface	Enables polymorphism without inheritance
Extensible code	You can add new classes without modifying existing logic

---

📘 Chapter 9 – Composition: When One Object Has Another

🧠 9.1 What is composition?

Composition is a principle of OOP where a class is built from other **component classes**.
We say there's a **"has-a"** relationship between objects.

💡 *Simple analogy*:
A car **has a** motor.
A school **has** students.
A phone **has a** screen.
→ These parts are **not subclasses** — they're components.

🧱 9.2 Composition in Java

```
public class Engine {
    void start() {
        System.out.println("Engine started!");
    }
}

public class Car {
    private Engine engine = new Engine();  // composition

    void start() {
        engine.start();  // delegates the action to Engine
    }
}
```

➡️ The `Car` contains an `Engine`. It doesn't inherit from it, but uses it as a **part**.

🔁 9.3 Composition vs Inheritance

Inheritance ("is-a")	Composition ("has-a")
Builds hierarchies	Builds internal relationships
Strong coupling	Flexible and modular
Ex: Cat is an Animal	Ex: Car has an Engine

🔑 Composition is generally preferable, especially when:

- The classes don't share the same type
- You want a modular and reusable system

📦 9.4 Encapsulation within composition

You can **encapsulate internal objects**, controlling access via methods.

```
public class Library {
    private Book book;

    public Library(Book b) {
        book = b;
    }

    public void showBook() {
        book.displayInfo();
    }
}
```

➡️ The `Library` doesn't expose the `Book` directly. It uses it **internally with control**.

🧰 9.5 Composition with lists (collections of objects)

```
import java.util.ArrayList;

public class ClassGroup {
    private ArrayList<Student> students = new ArrayList<>();

    public void addStudent(Student s) {
        students.add(s);
    }

    public void listStudents() {
        for (Student s : students) {
            s.introduce();
        }
    }
}
```

➡️ The `ClassGroup` contains multiple `Student` objects — each treated individually but forming a whole.

🧪 9.6 Mini-lab

Create:

- A class `Address` with `street`, `city`, `postalCode`
- A class `Person` with `name` and an `Address` attribute
- In `main()`, create a person and print their name + address

🎯 9.7 Challenge

Create a class `Store` that contains a list of `Product`.
Each product has a name, price, and quantity.
The store must:

- Add products
- Show all products
- Calculate total stock value

⚠️ 9.8 Too much composition? Not always.

Composition is powerful, but should be used wisely:

- Use it when one class **logically belongs** to another
- Avoid it when classes are **unrelated**

Example: ✅ Car has a Wheel
❌ Car has a Dog (makes no sense!)

🧠 9.9 Final summary

Concept	Explanation
Composition	A class contains instances of other classes
"has-a" relationship	Ex: Person has an Address
Modularity	You can change or upgrade parts independently
Delegation	Tasks are passed to internal objects (`engine.start()`)

📘 Chapter 10 – Interfaces: Behavioral Contracts

🧠 10.1 What is an interface?

An interface is a special type of class that defines a **set of methods to be implemented**, but contains no code (just method signatures).
Think of it as a **contract**: whoever signs it agrees to fulfill the defined obligations.

💡 *Simple analogy*:
A power socket has a standard shape. Every device that plugs in must follow that shape — even if their internal workings differ.

📄 10.2 How do you define an interface?

```
public interface Printable {
    void print();  // no body!
}
```

➡️ All interface methods are **public and abstract by default** — no implementation allowed.

🔗 10.3 How do you implement an interface?

Use the keyword `implements`:

```
public class Document implements Printable {
    public void print() {
        System.out.println("Printing document...");
    }
}
```

📌 If the class doesn't implement **all** interface methods, the program won't compile.

🧬 10.4 Interface ≠ Inheritance

Feature	Interface	Inheritance
Keyword	`implements`	`extends`
Purpose	Define required behavior	Reuse existing code

Feature	Interface	Inheritance
Multiple support	Yes (multiple)	No (only one class)
Attributes or logic	❌ (except default in Java 8+)	✅ Yes

➡️ Interfaces are used when **different classes should share behavior**, even if they're unrelated by type.

💼 10.5 Interfaces in the real world

Imagine these situations:

- All documents should be printable → `Printable`
- All game objects should move → `Movable`
- All forms should validate their data → `Validatable`

Each class handles it its own way, but all agree to follow the **interface contract**.

🧪 10.6 Practical example with multiple classes

```
public interface Printable {
    void print();
}

public class Report implements Printable {
    public void print() {
        System.out.println("Report: monthly stats...");
    }
}

public class Invoice implements Printable {
    public void print() {
        System.out.println("Invoice: total due €123.45");
    }
}

public class Printer {
    public static void printAll(Printable p) {
        p.print();
    }
}
```

In `main()`:

```
public class Main {
    public static void main(String[] args) {
        Printable doc1 = new Report();
        Printable doc2 = new Invoice();

        Printer.printAll(doc1);
        Printer.printAll(doc2);
    }
}
```

➡️ Even though `Report` and `Invoice` are totally different classes, we can treat them **polymorphically**.

🎯 10.7 Challenge

Create:

- An interface `Drawable` with a `draw()` method
- Classes `Circle`, `Square`, and `Triangle` that implement `Drawable`
- A method that receives a list of `Drawable` and draws them all

🧠 10.8 Interface with multiple implementations

A class can implement **multiple interfaces**:

```
public class Robot implements Movable, Drawable {
    public void move() {
        System.out.println("Moving forward");
    }

    public void draw() {
        System.out.println("Drawing on the floor");
    }
}
```

➡️ This simulates **multiple inheritance**, which is not allowed with classes in Java — but **is allowed with interfaces**!

📦 10.9 Interfaces = Independence

Programming with interfaces makes your system more:

- ✅ Flexible
- ✅ Testable
- ✅ Maintainable

➡️ Because your code depends not on the **concrete class**, but on the **interface** — the **contract**.

🧠 10.10 Final summary

Concept	Explanation
Interface	Contract defining required methods
`implements`	Keyword to implement an interface
Polymorphism	Lets you treat different objects generically
Flexibility	You can swap implementations freely

📘 Chapter 11 – Abstract Classes: When Completion Comes Later

🧠 11.1 What is an abstract class?

An abstract class:

- Cannot be instantiated directly
- Serves as a **base** for other classes
- May have methods with or without implementation

💡 *Simple analogy*:
Think of a mold for LEGO bricks. The mold defines how pieces should connect, but doesn't specify the color or exact shape. Each specific piece **completes the mold differently**.

📦 11.2 When to use an abstract class?

Use abstract classes when you want to:

- Group common behavior for multiple classes
- Force subclasses to implement certain methods
- Provide **partial** implementations

➡️ Perfect when you have **shared logic**, but each subclass needs custom details.

🧱 11.3 How to declare an abstract class?

```
public abstract class Animal {
    String name;

    public void sleep() {
        System.out.println(name + " is sleeping...");
    }

    public abstract void makeSound();  // no implementation!
}
```

➡️ The keyword `abstract`:

- Can mark a class

- Can mark a method that **has no body**

🚫 11.4 Abstract classes can't be instantiated

```
Animal a = new Animal();  // ❌ ERROR: abstract classes can't be instantiated
```

➡️ You must create a concrete subclass that implements the abstract methods.

✅ 11.5 Implementing a concrete subclass

```
public class Cat extends Animal {
    public Cat(String name) {
        this.name = name;
    }

    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

Now you can do:

```
Animal a = new Cat("Felix");
a.makeSound();  // Meow!
a.sleep();      // Felix is sleeping...
```

🔍 11.6 Abstract class vs Interface

Feature	Abstract Class	Interface
Can have attributes and logic	✅ Yes	❌ (Java 7) / ✅ default (Java 8+)
Can have constructors	✅ Yes	❌ No
Can define partial behavior	✅ Yes	❌ (unless default method)
Multiple inheritance allowed	❌ No	✅ Yes

💡 *Quick tip*:

- Use **interface** → when defining common behavior
- Use **abstract class** → when sharing **data and code**

🧪 11.7 Complete example

```
public abstract class Employee {
    String name;

    public Employee(String name) {
        this.name = name;
    }

    public abstract double calculateSalary();
}

public class Teacher extends Employee {
    private double fixedSalary;

    public Teacher(String name, double salary) {
        super(name);
        this.fixedSalary = salary;
    }

    @Override
    public double calculateSalary() {
        return fixedSalary;
    }
}
```

In `main()`:

```
Employee e = new Teacher("Luis", 1200);
System.out.println("Salary: €" + e.calculateSalary());
```

🎯 11.8 Challenge

Create:

- An abstract class `Animal` with `makeSound()`
- Subclasses `Dog` and `Cat`
- Test in `main()` using **polymorphism**
  - 💡 Bonus: add a shared method `eat()` with default implementation

---

🧠 11.9 Final summary

Concept	Explanation
Abstract class	Incomplete class used as a base
`abstract`	Keyword for abstract methods or classes
Concrete subclass	Must implement abstract methods
Polymorphism	Applies to abstract classes too

📘 Chapter 12 – Collections and Data Structures: Storing and Organizing Objects

---

🧠 12.1 What are collections?

Collections are structures that allow you to **store and organize multiple objects** efficiently.
Unlike basic variables that hold one value, collections can store **many elements** — like lists, sets, or maps.

💡 *Simple analogy*:
Think of a backpack full of books or a shelf filled with DVDs. Each compartment stores an item — and you can add, remove, or search them.

📦 12.2 Most used collection types in Java

Type	Interface	What it does
List	`List`	Maintains insertion order, allows duplicates
Set	`Set`	Disallows duplicate elements
Map	`Map`	Stores key → value pairs

📋 12.3 The `ArrayList` class (dynamic list)

```
import java.util.ArrayList;

ArrayList<String> names = new ArrayList<>();
names.add("Ana");
names.add("Carlos");
names.add("Beatriz");

System.out.println(names.get(1));  // Carlos
```

➡️ The list maintains **insertion order**
➡️ You can access any element by **index**

🔁 12.4 Looping through lists with for-each

```
for (String name : names) {
    System.out.println("Hello, " + name + "!");
}
```

➡️ This loop is cleaner, safer, and easier to read.

🔄 12.5 Removing and modifying elements

```
names.remove("Ana");       // by value
names.remove(0);           // by index
names.set(1, "Tiago");     // update index 1
```

🧱 12.6 Working with objects in lists

```
ArrayList<Person> people = new ArrayList<>();

Person p1 = new Person("John", 20);
Person p2 = new Person("Maria", 22);

people.add(p1);
people.add(p2);

for (Person p : people) {
    p.introduce();
}
```

➡️ This allows collections of **custom objects**, with their own behavior.

🌐 12.7 The `Set`: collections with no duplicates

```
import java.util.HashSet;

HashSet<String> colors = new HashSet<>();
colors.add("Blue");
colors.add("Red");
colors.add("Blue");  // Ignored!

System.out.println(colors);  // No duplicates!
```

➡️ Ideal when you want **automatic uniqueness**.

🔑 12.8 The `Map`: key → value pairs

```
import java.util.HashMap;

HashMap<String, String> capitals = new HashMap<>();
capitals.put("Portugal", "Lisbon");
capitals.put("Spain", "Madrid");

System.out.println(capitals.get("Portugal"));  // Lisbon
```

➡️ Great for **quick lookups** (e.g. student names → grades, countries → capitals)

🧪 12.9 Mini-lab

Create:

- A class `Product` with `name`, `price`
- A list of products (`ArrayList<Product>`)
- Loop through the list and show each name and price
- Calculate the total price of all products

🎯 12.10 Challenge

Create a class `Agenda` that uses:

- A `HashMap<String, String>` to store **contacts** (name → number)
- Functions to add, list, and search contacts by name

🧠 12.11 Final summary

Concept	Explanation
`ArrayList`	Dynamic list with indexed access
`HashSet`	Collection without duplicates
`HashMap`	Stores key → value pairs
`for-each`	Practical way to loop through collections
Reusability	You can store custom objects in any collection

📘 Chapter 13 – Exception Handling: Dealing with Errors Safely

---

🧠 13.1 What are exceptions?

Exceptions are **errors that occur while the program is running**.
They can be caused by various situations, such as:

- Dividing by zero
- Accessing an invalid index in a list
- Reading a file that doesn't exist
- Entering invalid input

💡 *Simple analogy*:
Imagine you're driving and suddenly there's a pothole in the road. That's an exception.
If you ignore it, your car (or program) could crash.
But if you anticipate it, you can stop or steer away — that's **exception handling**.

💥 13.2 Runtime errors

```
int a = 10;
int b = 0;
int result = a / b;  // ERROR! Division by zero
```

➡️ This triggers an `ArithmeticException`
Without handling, the program **crashes abruptly**.

🛡️ 13.3 Using try-catch to protect the program

```
try {
    int result = 10 / 0;
    System.out.println("Result: " + result);
} catch (ArithmeticException e) {
    System.out.println("Error: cannot divide by zero!");
}
```

- `try`: where the error might occur
- `catch`: what to do if it happens
- `e`: is the object representing the error (you can check its type, message, etc.)

🪂 13.4 The `finally` block: always runs

```
try {
    System.out.println("Trying division...");
    int r = 10 / 2;
} catch (Exception e) {
    System.out.println("Error!");
} finally {
    System.out.println("End of block.");
}
```

➡️ The `finally` block is **always executed**, whether an error occurred or not.

🪆 13.5 Exceptions with user input

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
System.out.print("Enter an integer: ");

try {
    int number = sc.nextInt();
    System.out.println("Number: " + number);
} catch (Exception e) {
    System.out.println("Error: you must enter a valid integer!");
}
```

➡️ Very useful when the user might type something wrong — and we don't want the program to crash.

⚠️ 13.6 Common exceptions

Exception	Typical Cause
`ArithmeticException`	Division by zero
`NullPointerException`	Using an object that wasn't created
`ArrayIndexOutOfBoundsException`	Accessing an invalid array index
`InputMismatchException`	Wrong input type
`FileNotFoundException`	File doesn't exist

Exception	Typical Cause
`IOException`	Generic input/output error

👨‍🏫 13.7 Creating custom exceptions

```
public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

Use it like this:

```
public void setAge(int age) throws InvalidAgeException {
    if (age < 0) {
        throw new InvalidAgeException("Age cannot be negative!");
    }
    this.age = age;
}
```

➡️ This allows you to define **custom business rules** with clearer, more meaningful messages.

🧪 13.8 Mini-lab

Create a program that:

- Reads two numbers
- Divides the first by the second
- Uses `try-catch` to avoid division by zero
- Displays a **friendly** error message

🎯 13.9 Challenge

Create:

- A class `BankAccount` with a method `withdraw(double amount)`
- If the balance is insufficient, it throws a custom `InsufficientBalanceException`
- In `main()`, test both a successful and a failed withdrawal

🧠 13.10 Final summary

Concept	Explanation
Exception	Error at runtime
`try-catch`	Try an operation and catch the error if it occurs
`finally`	Always runs, with or without an error
Custom exceptions	Define your own error rules
Robustness	A good program doesn't crash with bad input

📘 Chapter 14 – GUI with Swing: Bringing Your Program to Life

From console to interactive windows!

🧠 14.1 What is Swing?

Swing is a built-in Java library for creating graphical interfaces:
windows, buttons, menus, text boxes, lists, and even simple charts.

💡 Until now, we've written **text-based programs**. With Swing, you can create real applications with **windows and visual interaction**, just like modern apps!

🚀 14.2 Essentials to get started

Basic imports:

```
import javax.swing.*;     // Main components (JFrame, JButton, etc.)
import java.awt.*;        // Layouts, colors, fonts
import java.awt.event.*;  // Events (clicks, keys, etc.)
```

Your first window:

```
import javax.swing.*;

public class SimpleWindow {
    public static void main(String[] args) {
        JFrame window = new JFrame("Hello Swing!");
        window.setSize(300, 200);                       // width, height
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setVisible(true);                        // show window
    }
}
```

🧠 `JFrame` is the main window.
You must configure it: set size, title... and make it visible!

🧩 14.3 Adding buttons and interaction

```
import javax.swing.*;
import java.awt.event.*;
```

```
public class ButtonWindow {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button");
        JButton button = new JButton("Click me!");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame, "Hello, world!");
            }
        });

        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

✅ Common components:

Component	Swing Class
Main window	`JFrame`
Button	`JButton`
Label (text)	`JLabel`
Text field	`JTextField`
Large text box	`JTextArea`
Dropdown list	`JComboBox`
Checkbox	`JCheckBox`
Radio button	`JRadioButton`
Layout panel	`JPanel`

🎛️ 14.4 Layouts: organizing components

- `FlowLayout` (default)

  ```
  frame.setLayout(new FlowLayout());
  ```

- `BorderLayout` (5 areas: North, South, Center, East, West)

  ```
  frame.add(button, BorderLayout.SOUTH);
  ```

- `GridLayout` (grid/table)

  ```
  frame.setLayout(new GridLayout(2, 2));  // 2 rows, 2 columns
  ```

🧪 14.5 Simple form with fields and a button

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Form {
    public static void main(String[] args) {
        JFrame window = new JFrame("Form");
        window.setLayout(new FlowLayout());

        JLabel lblName = new JLabel("Name:");
        JTextField txtName = new JTextField(20);
        JButton btnSend = new JButton("Send");

        btnSend.addActionListener(e -> {
            String name = txtName.getText();
            JOptionPane.showMessageDialog(window, "Hello, " + name + "!");
        });

        window.add(lblName);
        window.add(txtName);
        window.add(btnSend);

        window.setSize(300, 150);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setVisible(true);
    }
}
```

📊 14.6 Drawing a simple chart

To draw, subclass `JPanel` and use `Graphics`:

```
import javax.swing.*;
import java.awt.*;

class ChartPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLUE);
        g.fillRect(50, 50, 100, 100);  // x, y, width, height
        g.setColor(Color.RED);
        g.drawString("Simple chart!", 60, 45);
    }
}
```

Then use the panel in a window:

```
public class ChartWindow {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Chart");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        ChartPanel panel = new ChartPanel();
        frame.add(panel);

        frame.setVisible(true);
    }
}
```

➡️ This panel can draw bars, points, shapes — even line charts.

💡 For advanced charts, consider libraries like **JFreeChart** (not covered here, but easy to integrate).

🎯 14.7 Challenge

Create a window with:

- A text field to enter a name
- A button that shows a **welcome message**
- An extra button that **closes the application** (`System.exit(0)`)

🧠 14.8 Final summary

Concept	Explanation
`JFrame`	Main window
`JButton`, `JLabel`, `JTextField`	Visual components
`addActionListener`	Responds to clicks or actions
`JOptionPane`	Displays dialogs
`JPanel` + `Graphics`	Used to draw shapes and graphics manually
Layouts	Organizes components: `Flow`, `Border`, `Grid`

📘 Chapter 15 – Good Practices and Project Structuring

🧠 15.1 Why are structure and good practices so important?

Creating a program that works is only the **first step**.
But well-structured, readable, and reusable code is what makes you a true **object-oriented programmer**.

💡 *Think of your project like a house*:

- If it's poorly built, it collapses at the first issue.
- If it's well designed, it's easy to maintain, expand, and explain to others.

🧱 15.2 Keep classes well separated

➡️ Each class should represent a **clear idea** (an entity, a role, or a function).

Examples:

- `Student`, `Teacher`, `Course` → entities
- `Calculator`, `Report`, `Printer` → utility classes
- `Main`, `App`, `TestXYZ` → entry points or test classes

🧠 **Golden rule**: one public class per file, with the **same name as the class**.

🔒 15.3 Encapsulate everything you can

Never leave attributes public.
Control access with `private` and expose only through `get/set` methods **when it makes sense**.

💬 A good object:

- Protects its data
- Exposes only what is essential
- Validates what comes in

🧩 15.4 Use constructors with logic

Avoid this:

```
Person p = new Person();
p.name = "John";
p.age = 0;
```

Prefer this:

```
Person p = new Person("John", 0);
```

➡️ It makes the code cleaner, more readable, and less error-prone.

🔗 15.5 Relate classes meaningfully

Before using inheritance, ask yourself:

❓ *"Does it make sense to say that X is a Y?"*

- If yes → use `extends`
- If not → you're probably looking at **composition** (X *has a* Y)

Example:

- `Car extends Vehicle` ✅
- `Car has an Engine` ✅
- `Car extends Engine` ❌

💉 15.6 Use meaningful lists and maps

Prefer collections when:

- You have **multiple elements** of the same type (`List<Product>`)
- You need to **associate values to names/keys** (`Map<String, Student>`)

➡️ Give variables good names and pick the right collection type!

💬 15.7 Comment with purpose

❌ Don't comment the obvious:

```
int age = 20; // defines age ← NO
```

✅ Comment where the **logic isn't obvious**:

```
// Validates if user is old enough to vote
if (age >= 18) {
    allowVote();
}
```

And use **JavaDoc** in main classes and methods:

```
/**
 * Represents a student with name and final grade.
 */
public class Student {
    private String name;
    private double grade;

    /**
     * Creates a student with name and grade.
     */
    public Student(String name, double grade) { ... }
}
```

🧼 15.8 Write clean code

✔ Use **meaningful names** (`totalAmount`, `calculateAverage`)

✔ **Avoid code repetition**

✔ **Split large functions** into smaller parts

✔ **Indent your code properly** — makes reading and debugging easier

---

🧪 15.9 Mini-lab

Take one of your previous projects (e.g., product management, library, bank...) and:

- Reorganize the code into **multiple classes**
- Apply **encapsulation** correctly
- Add **useful JavaDoc** comments
- **Avoid repetition** and improve variable names

🎯 15.10 Challenge

Create a simple project with the following features:

- 3 or more **clearly related classes** (`Person`, `Book`, `Loan`)
- Correct use of **constructors**, **get/set**, and **encapsulation**
- A `main()` method that **demonstrates the behaviors**
- JavaDoc on the **main classes and methods**

💡 This project can serve as the **skeleton for your final assignment**! 🎓

🧠 15.11 Final summary

Practice	Explanation
Separate classes	One file and one responsibility per class
Proper encapsulation	Keep attributes private + controlled access via methods
Useful comments	JavaDoc for public methods, inline comments for complex logic
Clean constructors	Create objects with valid data from the start
Logical relationships	Use inheritance and composition **when appropriate**
