Java Object-Oriented Design

Principles, Patterns, and Practices

readbytes.github.io

2025-07-06

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Java and Object-Oriented Programming

1. Why Java?
2. What is Object-Oriented Programming?
3. The Four Pillars of OOP
4. Java Tools and Environment Setup
5. Writing Your First Java Class

1 Introduction to Java and Object-Oriented Programming

1.1 Why Java?

Java is one of the most enduring and influential programming languages in the software industry. Since its introduction in the mid-1990s, Java has grown into a powerful, versatile, and widely-used language, trusted by millions of developers and organizations across the world. But what makes Java such a compelling language—especially for those beginning their journey in object-oriented programming (OOP)? To understand this, it helps to explore Java's origins, its technical strengths, and its place in today's technology ecosystem.

1.1.1 Historical Background and Evolution

Java was originally developed by James Gosling and his team at Sun Microsystems in 1995. The language was created as part of a project called "Green" aimed at building software for embedded devices, but it quickly evolved into a general-purpose programming language. What set Java apart from the outset was its "write once, run anywhere" philosophy. By compiling code into bytecode that could be executed on the Java Virtual Machine (JVM), Java allowed developers to write code that would run consistently on any platform that supported a JVM—Windows, macOS, Linux, and more.

Over time, Java became a staple in enterprise software development, aided by the growth of frameworks like Java EE (Enterprise Edition). It found new life with the rise of Android in the late 2000s, where Java became the primary language for mobile development. Today, Java continues to evolve, with regular feature updates and performance improvements through OpenJDK and the stewardship of Oracle and the broader Java community. The language has modernized with features like lambdas, streams, records, and modules, making it suitable for both legacy systems and modern cloud-native applications.

1.1.2 Key Features of Java

Java's continued popularity is largely due to its thoughtful design and practical features, many of which make it especially well-suited for object-oriented design.

Platform Independence via the JVM

Perhaps Java's most celebrated feature is its platform independence. Instead of compiling code directly to machine instructions (like C or C++), Java compiles source code into an intermediate form called **bytecode**. This bytecode is executed by the **Java Virtual Machine (JVM)**, which is available on almost every modern operating system. This architecture

allows Java programs to be **portable**, meaning the same compiled code can run on any platform without modification.

Strong Typing and Compile-Time Safety

Java is a **statically-typed** language, meaning variable types are known and checked at compile time. This helps catch many common programming errors—such as type mismatches—before the program even runs. Compared to dynamically-typed languages like Python, Java provides a stricter and more structured development experience, which is especially beneficial for large-scale or long-term projects where type safety reduces bugs and enhances maintainability.

Automatic Memory Management with Garbage Collection

Java includes **automatic garbage collection**, which manages memory allocation and deallocation for you. This reduces the burden on developers to manually free memory (as is necessary in languages like C++), helping to avoid memory leaks and dangling pointers. Java's garbage collector continuously runs in the background, reclaiming unused memory and contributing to more robust and stable applications.

Rich Standard Libraries and APIs

Java comes with an extensive **standard library** (also known as the Java API) that supports a wide range of application needs: file I/O, networking, data structures, cryptography, user interfaces, and more. This reduces the need to write boilerplate code from scratch and accelerates development. Additionally, Java has a vibrant ecosystem of third-party libraries and frameworks—like Spring, Hibernate, and Apache libraries—that extend its capabilities for enterprise, web, and microservice development.

1.1.3 Java in the Real World

Java's footprint in industry is massive and diverse:

- **Enterprise Software**: Java remains the go-to language for enterprise systems, particularly in banking, insurance, government, and large-scale backend systems.
- **Android Development**: Until recently, Java was the primary language for Android app development (with Kotlin now sharing that space).
- **Web Applications**: Java powers many web applications through frameworks like Spring Boot and JavaServer Faces (JSF).
- **Big Data and Cloud**: Java is widely used in distributed systems and big data technologies like Hadoop and Apache Spark.
- **Academia and Education**: Many universities use Java as the introductory programming language due to its balance of complexity and readability.

1.1.4 Why Java Over Other Languages?

While there are many modern languages to choose from, Java offers a unique blend of practicality and structure that makes it ideal for learning object-oriented programming.

- **C++** offers fine-grained control over system resources but at the cost of complexity and manual memory management, which can overwhelm beginners.
- **Python** is known for its simplicity and readability, but its dynamic typing can obscure object-oriented principles and lead to runtime errors that are harder to trace.
- **Java**, by contrast, offers a strong balance: it's readable, strongly-typed, object-oriented, and supported by a massive ecosystem. Its widespread industry adoption ensures that Java skills are not only educational but also highly marketable.

1.1.5 A Simple Example: "Hello, World!"

Let's conclude with a simple program that showcases Java's syntax and structure:

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

This example highlights a few things:

- Every Java application starts with a **class** (`HelloWorld`).
- The `main` method is the **entry point** of the application.
- `System.out.println()` is a standard way to print output to the console.

To run this program, you would save the file as `HelloWorld.java`, compile it with `javac HelloWorld.java`, and run it with `java HelloWorld`. Thanks to Java's robust toolchain and platform independence, this process works the same on almost any system.

1.2 What is Object-Oriented Programming?

Object-Oriented Programming, or **OOP**, is a programming paradigm built around the concept of **"objects"**—self-contained entities that combine data and behavior. Unlike procedural programming, where programs are written as a series of step-by-step instructions, OOP organizes code into reusable and modular units, each representing something meaningful from the real world. This shift in thinking—from writing instructions to modeling systems—is one of the most important ideas in modern software development.

1.2.1 From Procedures to Objects

To understand the impact of OOP, consider procedural programming first. In a procedural approach (used in languages like C), programs are composed of functions and procedures that operate on data. The focus is on **what the program should do**, often leading to code that is linear and harder to scale as the project grows. Data is typically kept separate from the procedures that manipulate it, which can make it difficult to manage complex systems or prevent unintended side effects.

Object-Oriented Programming flips this model by bundling **data and behavior together** into units called **objects**. These objects represent entities—like a bank account, a book, or a car—and they know **how to operate on themselves**. This leads to code that is closer to how we think about the real world: self-contained components that interact with each other through well-defined interfaces.

1.2.2 Objects as Real-World Models

One of the key strengths of OOP is its ability to model **real-world concepts** in code. For example, think about a **Car**. In OOP, a car might be represented as an **object** with:

- **Attributes** (also called fields or properties): `color`, `speed`, `fuelLevel`
- **Methods** (also called behaviors or functions): `accelerate()`, `brake()`, `refuel()`

Each **object** is a specific instance of a **class**, which acts as a blueprint. For example, you can have multiple car objects—each with different colors and speeds—but all based on the same `Car` class definition.

This way of structuring code makes it much easier to reason about behavior, organize logic, and scale software systems without overwhelming complexity.

1.2.3 Key Terms in OOP

Before diving further, it's important to understand some core terminology:

- **Class**: A **template** or **blueprint** for creating objects. It defines the structure (attributes) and behavior (methods) that its instances will have.
- **Object**: An individual **instance** of a class. Objects have actual values assigned to their attributes and can perform actions defined by the class.
- **Attributes (Fields)**: The **state** or **data** stored in an object (e.g., `name`, `balance`, `temperature`).
- **Methods**: The **actions** or **functions** that an object can perform (e.g., `deposit()`,

`withdraw()`, `displayInfo()`).

When you combine these building blocks, you get modular, reusable code components that are easy to work with.

1.2.4 Benefits of Object-Oriented Programming

OOP isn't just a change in syntax—it's a philosophy of how to build better software. Its benefits include:

- **Modularity**: Code is divided into independent, self-contained objects. Each class or module can be developed and tested in isolation.
- **Reusability**: Once a class is written, it can be reused across multiple programs. For example, a `User` class can be used in both a website and a mobile app.
- **Maintainability**: Changes to one part of the code are less likely to affect others, thanks to clear interfaces and encapsulation. This makes large projects easier to manage and evolve.
- **Scalability**: As systems grow, new functionality can be added by creating new objects or extending existing ones, rather than rewriting entire functions.

1.2.5 A Simple Example: Modeling a Dog

Let's look at a basic example of an object in Java. Suppose we want to model a **Dog**.

```java
// This is a class definition
public class Dog {
    // Attributes
    String name;
    int age;

    // Method
    void bark() {
        System.out.println(name + " says: Woof!");
    }
}
```

This `Dog` class defines what a dog is: it has a `name`, an `age`, and the ability to `bark()`.

Now let's create an **object** (an instance of the class) and use it:

```java
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Dog class
        Dog myDog = new Dog();
```

```
        myDog.name = "Buddy";
        myDog.age = 3;

        myDog.bark();  // Output: Buddy says: Woof!
    }
}
```

In this example:

- `Dog` is the class.
- `myDog` is an object created from that class.
- We set values for the object's attributes and call its method.

This simple example already demonstrates several core OOP ideas: encapsulating state and behavior, using real-world metaphors, and promoting reuse.

Full runnable code:

```
public class Dog {
    // Attributes
    String name;
    int age;

    // Method
    void bark() {
        System.out.println(name + " says: Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of the Dog class
        Dog myDog = new Dog();
        myDog.name = "Buddy";
        myDog.age = 3;

        myDog.bark();  // Output: Buddy says: Woof!
    }
}
```

1.2.6 Conclusion

Object-Oriented Programming is a powerful paradigm that helps developers think in terms of real-world models. By focusing on objects and their interactions, it enables code that is more intuitive, modular, and adaptable. For a language like Java, which is built around OOP principles, mastering this approach is essential. In the chapters ahead, you'll learn how to define your own classes, use inheritance, encapsulate behavior, and apply patterns that make your code clean, flexible, and maintainable. Understanding the core ideas of OOP is the first step in that journey.

1.3 The Four Pillars of OOP

Object-Oriented Programming (OOP) is built on four foundational concepts often referred to as the **Four Pillars of OOP**: **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction**. These principles work together to create software that is modular, flexible, and easier to maintain. By understanding and applying these concepts, developers can model complex systems in a way that is both intuitive and scalable.

Let's explore each pillar in depth with clear explanations and Java code examples.

1.3.1 Encapsulation Hiding the Internal Details

Encapsulation is the process of **bundling data (attributes) and methods (behavior)** that operate on the data into a single unit—usually a class. It also means **restricting direct access** to some of the object's components, typically by making fields `private` and exposing controlled access through **getters and setters**.

Encapsulation ensures that the internal representation of an object is hidden from the outside world, which protects the integrity of the data and promotes modularity.

Example: Encapsulation in Java

```
public class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        if (initialBalance >= 0) {
            this.balance = initialBalance;
        }
    }

    public double getBalance() {
        return balance;
    }

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

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

In this example:

- The `balance` field is `private`, meaning it cannot be accessed directly from outside the class.
- Public methods like `deposit()` and `withdraw()` provide controlled access to the balance, enforcing rules such as not allowing negative deposits or overdrafts.

Encapsulation supports **data integrity** and encourages separating **what** an object does from **how** it does it.

1.3.2 Inheritance Reusing Code Through Hierarchies

Inheritance allows a class (called a **subclass** or **child class**) to inherit the attributes and methods of another class (called a **superclass** or **parent class**). It promotes **code reuse** and enables hierarchical classification.

Example: Inheritance in Java

```java
// Superclass
public class Animal {
    public void eat() {
        System.out.println("This animal eats food.");
    }
}

// Subclass
public class Dog extends Animal {
    public void bark() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();   // Inherited from Animal
        dog.bark();  // Defined in Dog
    }
}
```

Here, `Dog` **inherits** the `eat()` method from `Animal` and also defines its own method `bark()`. Inheritance reduces duplication and makes it easier to maintain shared behavior in a central place.

However, inheritance must be used thoughtfully. Overusing it can lead to fragile designs. Prefer **"is-a"** relationships—for example, a dog **is an** animal—for inheritance to make logical sense.

1.3.3 Polymorphism One Interface, Many Forms

Polymorphism allows objects to be treated as instances of their **parent class rather than their actual class**. It comes in two forms:

- **Compile-time polymorphism** (method overloading)
- **Runtime polymorphism** (method overriding)

The key benefit of polymorphism is **flexibility**—code can work on objects of different types as long as they share a common parent type.

Example: Polymorphism in Action

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal1 = new Dog();
        Animal myAnimal2 = new Cat();

        myAnimal1.makeSound();  // Dog barks
        myAnimal2.makeSound();  // Cat meows
    }
}
```

In this example, both `myAnimal1` and `myAnimal2` are declared as type `Animal`, but the actual method that gets called is determined at **runtime** based on the object's real type (`Dog` or `Cat`). This is **dynamic dispatch**, a key part of runtime polymorphism.

Polymorphism supports **open-ended design**, allowing new behaviors to be added without modifying existing code.

1.3.4 Abstraction Focusing on the Essentials

Abstraction means hiding complex implementation details and exposing only the necessary parts of an object or system. In Java, this can be achieved using **abstract classes** or **interfaces**.

Abstraction helps manage complexity by letting developers focus on **what** an object does instead of **how** it does it.

Example: Abstraction with Interface

```
public interface Shape {
    double getArea();
}
public class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
public class Square implements Shape {
    private double side;

    public Square(double side) {
        this.side = side;
    }

    public double getArea() {
        return side * side;
    }
}
public class Main {
    public static void printArea(Shape shape) {
        System.out.println("Area: " + shape.getArea());
    }

    public static void main(String[] args) {
        Shape circle = new Circle(3);
        Shape square = new Square(4);

        printArea(circle); // Area: 28.27...
        printArea(square); // Area: 16.0
    }
}
```

In this example:

- `Shape` is an **interface** defining a common behavior (`getArea()`).
- `Circle` and `Square` **implement** the interface and provide specific details.
- The `printArea()` method operates on the **abstract type** `Shape`, not on the concrete

implementations.

This abstraction allows new shapes to be added later without modifying existing code—demonstrating the **Open/Closed Principle** in practice.

1.3.5 How These Pillars Work Together

These four pillars are not isolated—they **reinforce and complement** one another in a complete object-oriented design:

- **Encapsulation** protects internal state and ensures consistency.
- **Inheritance** allows reuse and the building of logical hierarchies.
- **Polymorphism** makes it possible to handle different types uniformly.
- **Abstraction** hides implementation detail and promotes cleaner APIs.

For example, a `Vehicle` class hierarchy might use **abstraction** to define general behavior (e.g., `startEngine()`), **inheritance** to specialize behavior for `Car` and `Truck`, **polymorphism** to treat all vehicles generically, and **encapsulation** to manage internal properties like `fuelLevel`.

These principles are central to writing code that is **scalable, maintainable, and extensible**—qualities that are essential in real-world software development.

1.3.6 Reflective Thought

Think of a **remote control** and a **television**. The remote control is like an **abstract interface**—you don't need to know how the TV works internally, only how to use the remote. The TV hides its complex circuits and logic—just like **encapsulation**. A remote might work with different TV brands—like **polymorphism**. And newer remotes might build upon older ones with added features—just like **inheritance**.

Question: *How might you design a software system for a ride-sharing app using these four principles? What would be the "objects"? What features would benefit from inheritance, abstraction, or encapsulation?*

Understanding and mastering the four pillars of OOP will give you the tools to build robust and elegant systems. In the coming chapters, we will explore how Java brings these concepts to life and how you can apply them to real-world problems.

1.4 Java Tools and Environment Setup

Before you can begin writing Java programs, you'll need to set up a development environment. Fortunately, Java has excellent tooling support and is easy to configure. In this section, we'll walk you through the essential tools and steps required to start coding.

1.4.1 Essential Tools for Java Development

1. **Java Development Kit (JDK)** The JDK includes everything needed to compile and run Java programs: the Java compiler (`javac`), the Java Virtual Machine (`java`), and standard libraries.
2. **Integrated Development Environment (IDE)** IDEs simplify development by providing features like code completion, debugging, and project management.
   - **IntelliJ IDEA** (Community Edition – free): Powerful and widely used.
   - **Eclipse**: Popular in enterprise settings, fully featured and open-source.
   - **Visual Studio Code (VS Code)**: Lightweight editor with Java extensions.

1.4.2 Step-by-Step: Setting Up Java

Install the JDK

- Go to the official OpenJDK site: https://jdk.java.net
- Download the latest LTS (e.g., JDK 17 or JDK 21) for your OS.
- Install it using the installer.
- **Verify installation**: Open a terminal (or command prompt) and run:

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

  You should see the installed version printed.

Install an IDE

- **IntelliJ IDEA**: Download from https://www.jetbrains.com/idea
- **Eclipse**: Download from https://www.eclipse.org/downloads
- **VS Code**: Download from https://code.visualstudio.com, then install the **Java Extension Pack**.

After installation, create a new Java project using the IDE's wizard or "New Project" feature.

1.4.3 Compiling and Running a Java Program

From the Command Line

1. Create a file named `HelloWorld.java`:

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

2. Compile the file:

```
javac HelloWorld.java
```

   This creates a `HelloWorld.class` bytecode file.

3. Run the program:

```
java HelloWorld
```

From an IDE

1. Open or create a project.
2. Create a new Java class named `HelloWorld`.
3. Paste the code above into the editor.
4. Click the "Run" or "Play" button in the toolbar.
5. The output will appear in the console panel.

1.4.4 Final Notes

Once your environment is ready, you can focus on writing and experimenting with code instead of wrestling with setup issues. Whether you prefer the full power of an IDE or the simplicity of a text editor and command line, Java offers flexibility and robust tooling to match your workflow.

1.5 Writing Your First Java Class

Now that your development environment is ready, it's time to write your first Java class. This section walks you through creating a simple, runnable Java program, explains the structure of a class, and highlights best practices that will serve as a foundation for more complex designs later in the book.

1.5.1 A Basic Java Class Example

Let's start with a minimal Java program that prints a message to the console.

```java
// This is a simple Java class
public class HelloWorld {

    // The main method: program entry point
    public static void main(String[] args) {
        // Print a message to the console
        System.out.println("Hello, Java OOP World!");
    }
}
```

1.5.2 Anatomy of a Java Class

Let's break down the key parts of this program:

Class Declaration

```java
public class HelloWorld {
```

- `public`: This access modifier means the class is visible to all other classes.
- `class`: This keyword is used to declare a class.
- `HelloWorld`: The name of the class. It **must match the file name** (`HelloWorld.java`).

Main Method

```java
public static void main(String[] args) {
```

- This is the **entry point** of every standalone Java application.
- `public`: So the JVM can access it from outside the class.
- `static`: So it can run without creating an instance of the class.
- `void`: It does not return a value.
- `String[] args`: Accepts command-line arguments.

Statements and Comments

```java
System.out.println("Hello, Java OOP World!");
```

- This line prints a message to the console.
- `System.out` is a standard output stream.
- `println()` prints the text and moves to a new line.

Single-line comment:

```
// This is a comment
```

Multi-line comment:

```
/*
   This is a multi-line comment
   that spans more than one line.
*/
```

Use comments to explain logic, not to restate obvious code. Clean code should be mostly self-explanatory.

1.5.3 Compiling and Running the Class

From the Command Line

1. Save the file as `HelloWorld.java`.
2. Open your terminal and navigate to the directory containing the file.
3. Compile the code:
   ```
   javac HelloWorld.java
   ```
   This creates a file named `HelloWorld.class`.
4. Run the program:
   ```
   java HelloWorld
   ```

From an IDE

1. Create a new Java project and class named `HelloWorld`.
2. Paste the example code into the editor.
3. Click the **Run** button.
4. View the output in the console window.

1.5.4 Best Practices for Class Structure

- **Use meaningful class names** that describe their purpose.
- **One public class per file**, and the file name must match the class name.
- **Follow Java naming conventions**: class names start with an uppercase letter (e.g., `BankAccount`), method and variable names use camelCase.
- **Keep the main method clean**: offload logic to other methods or classes as your

programs grow.

- **Write comments only where they clarify non-obvious behavior**.

1.5.5 Summary

You've just written and run your first Java class! This example demonstrated how to define a class, use the `main` method, add comments, and follow basic coding conventions. In the chapters ahead, you'll build on this structure to create classes with fields, methods, constructors, and full object-oriented designs. Keep this simple structure in mind—it's the starting point for every Java application.

Chapter 2.

Classes and Objects

1. Defining Classes and Creating Objects
2. Fields and Methods
3. The `this` Keyword
4. Constructors and Constructor Overloading
5. Object Lifecycle and Scope

2 Classes and Objects

2.1 Defining Classes and Creating Objects

In Java, everything begins with **classes and objects**. These two concepts are at the heart of object-oriented programming (OOP). In this section, you'll learn how to define your own classes and create objects from them—laying the foundation for building modular, reusable, and maintainable software.

2.1.1 What Is a Class?

A **class** is a **blueprint** for creating objects. It defines the **structure** (fields or attributes) and **behavior** (methods or functions) that its instances will have. Think of a class as a template—like architectural drawings for a house.

Syntax of a Class Declaration

```
public class Car {
    // Fields (attributes)
    String model;
    int year;

    // Method (behavior)
    void startEngine() {
        System.out.println("Engine started.");
    }
}
```

In this example:

- `Car` is a class with two fields: `model` and `year`.
- It also has one method: `startEngine()`.

This class doesn't actually do anything until we create **objects** from it.

2.1.2 What Is an Object?

An **object** is a **specific instance** of a class. When you create an object, you're telling Java: "Make me one of these, with memory allocated for its fields."

Each object has its own copies of the class's fields, but it shares the behavior (methods) defined in the class.

2.1.3 Creating Objects in Java

To create an object in Java, you use the `new` keyword followed by a call to the class's constructor (a special method we'll cover in the next section).

```
Car myCar = new Car();      // Object 1
Car yourCar = new Car();    // Object 2
```

- `new Car()` creates a new object in memory.
- `myCar` and `yourCar` are **references** to the respective objects.

You can then assign values to the fields and call methods:

```
myCar.model = "Toyota Corolla";
myCar.year = 2020;

yourCar.model = "Honda Civic";
yourCar.year = 2022;

myCar.startEngine();  // Output: Engine started.
yourCar.startEngine();  // Output: Engine started.
```

Each object maintains its own state (`model` and `year`), even though they share the same structure and behavior.

2.1.4 Behind the Scenes: Memory and Referencing

When you write:

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

here's what happens under the hood:

1. `new Car()` allocates memory in the heap for the new object.
2. The object is initialized (default values are assigned).
3. A **reference** to that memory location is returned and stored in `myCar`.

It's important to understand that `myCar` is **not** the object itself—it's a reference (like a pointer) to the object in memory. You can have multiple references to the same object:

```
Car anotherCar = myCar;
anotherCar.model = "Tesla Model 3";

System.out.println(myCar.model);  // Output: Tesla Model 3
```

Since both `myCar` and `anotherCar` point to the same object in memory, changes made through one reference affect the other.

2.1.5 A Complete Example

Let's tie it all together with a complete, simple program:

```
public class Car {
    String model;
    int year;

    void displayInfo() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        car1.model = "Ford Mustang";
        car1.year = 2023;

        Car car2 = new Car();
        car2.model = "Chevrolet Camaro";
        car2.year = 2021;

        car1.displayInfo();  // Output: Model: Ford Mustang, Year: 2023
        car2.displayInfo();  // Output: Model: Chevrolet Camaro, Year: 2021
    }
}
```

2.1.6 Summary

- A **class** defines the structure and behavior of objects.
- An **object** is an actual instance of a class with its own state.
- The `new` keyword allocates memory and returns a reference to the object.
- Fields hold data; methods define behavior.
- Each object has its **own copy of fields**, but they share **method behavior**.

Understanding the difference between **class as blueprint** and **object as instance** is critical for writing effective Java programs. In the next section, you'll dive deeper into **fields and methods**—the components that give objects their unique identity and purpose.

2.2 Fields and Methods

In object-oriented programming, **fields** and **methods** are the core building blocks of a class. Fields define the data or state that an object holds, while methods define the behaviors or actions that the object can perform. Together, they form the structure and functionality of Java classes.

Understanding how to declare and use fields and methods is essential for writing well-structured and reusable code. This section covers the fundamentals of fields and methods in Java, including access control, method parameters, and variable scope.

2.2.1 Fields: Representing Object State

Fields, also known as **instance variables**, are declared inside a class but outside any method. Each object created from a class gets its own copy of the class's fields. Fields hold the data that represents the **state** of the object.

Declaring Fields

```
public class Person {
    // Fields (instance variables)
    public String name;
    private int age;
}
```

In this example:

- `name` is a `public` field, accessible from outside the class.
- `age` is `private`, meaning it can only be accessed within the `Person` class.

It's a best practice to **make fields private** and access them via public methods (getters and setters) to preserve **encapsulation**.

2.2.2 Methods: Representing Object Behavior

Methods define what an object can **do**. A method is a block of code that performs a specific task, and it can operate on the fields of the class.

Declaring Methods

```
public class Person {
    private String name;

    // Method with no return value
    public void greet() {
        System.out.println("Hello, my name is " + name);
    }

    // Method with parameters and a return type
    public void setName(String newName) {
        name = newName;
    }
```

```
    public String getName() {
        return name;
    }
}
```

Here:

- `greet()` is a method that prints a message using the `name` field.
- `setName()` takes a parameter and sets the internal `name` field.
- `getName()` returns the current value of the `name` field.

2.2.3 Access Modifiers

Java uses access modifiers to control the visibility of class members (fields and methods):

- `public`: Accessible from any other class.
- `private`: Accessible only within the same class.
- `protected`: Accessible within the same package or subclasses.
- Default (no modifier): Accessible only within the same package.

Encapsulation encourages using `private` fields and providing controlled access via `public` methods.

2.2.4 Instance Variables vs Local Variables

It's important to distinguish between:

- **Instance variables (fields)**: Declared in the class body and belong to the object.
- **Local variables**: Declared inside a method and exist only during that method's execution.

Example:

```
public class Calculator {
    private int lastResult; // Instance variable

    public int add(int a, int b) {
        int sum = a + b;     // Local variable
        lastResult = sum;    // Store in instance variable
        return sum;
    }
}
```

Here, `sum` is a **local variable** inside the `add()` method. It exists temporarily during method execution. `lastResult` is an **instance variable**, maintaining a persistent state across

method calls.

2.2.5 Method Invocation and Parameter Passing

Once an object is created, its methods can be invoked using **dot notation**:

```
public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.setName("Alice");
        person.greet(); // Output: Hello, my name is Alice
    }
}
```

Here, `setName("Alice")` passes a **parameter** (`"Alice"`) to the method, which is then used to update the internal state (`name`) of the object.

Java uses **pass-by-value** for parameter passing:

- For **primitive types** (e.g., `int`, `double`), the value is copied.
- For **objects**, the reference is passed by value—meaning the reference itself is copied, not the object.

Example:

```
public class Demo {
    public void updateValue(int x) {
        x = 10; // Does not affect original variable
    }

    public void updatePersonName(Person p) {
        p.setName("Updated"); // Affects original object
    }
}
```

Calling `updateValue()` won't change the original `int`, but `updatePersonName()` will update the actual `Person` object's name, since the object reference is passed.

Full runnable code:

```
public class Person {
    // Fields (instance variables)
    private String name;
    private int age;

    // Method with no return value
    public void greet() {
        System.out.println("Hello, my name is " + name);
    }
```

```
    // Setter for name
    public void setName(String newName) {
        name = newName;
    }

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for age
    public void setAge(int newAge) {
        age = newAge;
    }

    // Getter for age
    public int getAge() {
        return age;
    }
}

public class Calculator {
    private int lastResult; // Instance variable

    public int add(int a, int b) {
        int sum = a + b;     // Local variable
        lastResult = sum;    // Store in instance variable
        return sum;
    }

    public int getLastResult() {
        return lastResult;
    }
}

public class Demo {
    public void updateValue(int x) {
        x = 10; // Does not affect original variable
    }

    public void updatePersonName(Person p) {
        p.setName("Updated"); // Affects original object
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.setName("Alice");
        person.setAge(30);
        person.greet(); // Output: Hello, my name is Alice

        Calculator calc = new Calculator();
        int result = calc.add(5, 7);
        System.out.println("Sum: " + result);           // Sum: 12
        System.out.println("Last Result: " + calc.getLastResult()); // Last Result: 12

        Demo demo = new Demo();
```

```
        int original = 5;
        demo.updateValue(original);
        System.out.println("Original after updateValue: " + original); // 5

        demo.updatePersonName(person);
        System.out.println("Person's name after updatePersonName: " + person.getName()); // Updated
    }
}
```

2.2.6 Summary

- **Fields** (instance variables) store object state; **methods** define object behavior.
- Access modifiers (`private`, `public`, etc.) control visibility and enforce encapsulation.
- **Instance variables** live with the object; **local variables** exist only inside methods.
- Methods can take parameters, return values, and operate on fields.
- Java passes parameters **by value**, with objects passed by reference to the same memory.

Understanding and using fields and methods correctly is crucial for designing meaningful and maintainable Java classes. In the next section, you'll learn about the `this` keyword and how it helps distinguish between instance fields and parameters.

2.3 The `this` Keyword

In Java, the keyword `this` is a special reference variable that refers to the **current object**—the instance of the class in which it is used. It is especially useful when differentiating between **instance variables** (fields) and **parameters** or **local variables** with the same name. It also allows a method to return the current object, which can be useful for method chaining and fluent APIs.

2.3.1 Disambiguating Fields from Parameters

A common use case for `this` arises in constructors or setter methods where the **parameter name is the same as the field name**.

Example: Without `this`

```
public class Student {
    String name;

    public void setName(String name) {
```

```
        name = name; // This assigns the parameter to itself, not the field!
    }
}
```

In the above example, `name = name` does **not** do what we expect—it just reassigns the method parameter to itself. The field `name` remains unchanged.

Corrected: Using `this`

```
public class Student {
    String name;

    public void setName(String name) {
        this.name = name; // this.name refers to the field, name is the parameter
    }
}
```

Here, `this.name` clearly refers to the instance variable, while `name` refers to the method parameter. This eliminates ambiguity and ensures the field is correctly updated.

2.3.2 Returning the Current Object

The `this` keyword can also be used to **return the current object** from a method. This technique is often used to **enable method chaining**—calling multiple methods on the same object in a single statement.

Example:

```
public class Book {
    private String title;

    public Book setTitle(String title) {
        this.title = title;
        return this;
    }

    public void printTitle() {
        System.out.println("Title: " + title);
    }
}

public class Main {
    public static void main(String[] args) {
        new Book()
            .setTitle("Clean Code")
            .printTitle();
    }
}
```

In this example, `setTitle()` returns `this`, allowing `printTitle()` to be chained directly.

2.3.3 Common Mistakes Avoided with `this`

- **Accidentally assigning a parameter to itself** (`name = name`) instead of updating the instance field.
- **Improper method chaining** where you forget to return `this`.
- **Shadowing of fields** by local variables or constructor parameters with the same name.

Using `this` is not always required, but it is essential whenever you need to clarify that you're referring to the **current object's field** or when enabling **fluent programming styles**.

2.3.4 Summary

The `this` keyword is a simple yet powerful tool that helps you:

- **Disambiguate** between fields and parameters with the same name.
- **Return the current object** for method chaining.
- **Avoid logic errors** caused by variable shadowing.

As your classes grow in complexity, understanding and using `this` properly will help make your code clearer and more maintainable.

2.4 Constructors and Constructor Overloading

In Java, a **constructor** is a special kind of method used to **initialize objects** when they are created. Unlike regular methods, constructors have the same name as the class and **do not have a return type** (not even `void`). Every time you use the `new` keyword to create an object, a constructor is called to set up the object's initial state.

Understanding how constructors work—and how to write flexible, reusable ones through **overloading**—is essential to building robust and readable Java classes.

2.4.1 What Is a Constructor?

A constructor is executed **once** when an object is created. It typically sets default or user-defined values for the object's fields.

Syntax:

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

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

You can now create a `Person` object like this:

```
Person p1 = new Person(); // name = "Unknown", age = 0
```

2.4.2 Default Constructor

If **no constructor** is explicitly defined, Java automatically provides a **default constructor** with no parameters, which does nothing but allow the object to be instantiated.

However, if **any constructor** is defined, Java does **not** generate a default one. You must explicitly create it if needed.

```
public class Animal {
    String species;

    // No explicit constructor → Java provides a default one.
}
```

But if you write:

```
public class Animal {
    String species;

    public Animal(String s) {
        species = s;
    }
}
```

You **must** now explicitly add a no-argument constructor if you want to create an object like `new Animal()`.

2.4.3 Parameterized Constructors

A **parameterized constructor** allows you to pass initial values when creating an object.

```
public class Car {
    String model;
    int year;

    public Car(String m, int y) {
        model = m;
        year = y;
    }
}
```

Now you can create a car object like this:

```
Car car1 = new Car("Toyota", 2020);
```

Parameterized constructors make your code more expressive and reduce the need to call separate setter methods after object creation.

2.4.4 Constructor Overloading

Java allows you to define **multiple constructors** in the same class, as long as they differ in the number or types of parameters. This is called **constructor overloading**, and it lets you create objects in different ways depending on available data.

Example:

```
public class Book {
    String title;
    String author;

    // Default constructor
    public Book() {
        this("Unknown", "Unknown");
    }

    // Constructor with one parameter
    public Book(String title) {
        this(title, "Unknown");
    }

    // Constructor with two parameters
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public void printInfo() {
```

```
        System.out.println("Title: " + title + ", Author: " + author);
    }
}
```

Usage:

```
Book b1 = new Book();
Book b2 = new Book("1984");
Book b3 = new Book("Brave New World", "Aldous Huxley");

b1.printInfo(); // Output: Title: Unknown, Author: Unknown
b2.printInfo(); // Output: Title: 1984, Author: Unknown
b3.printInfo(); // Output: Title: Brave New World, Author: Aldous Huxley
```

Full runnable code:

```
public class Book {
    String title;
    String author;

    // Default constructor
    public Book() {
        this("Unknown", "Unknown");
    }

    // Constructor with one parameter
    public Book(String title) {
        this(title, "Unknown");
    }

    // Constructor with two parameters
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public void printInfo() {
        System.out.println("Title: " + title + ", Author: " + author);
    }
}

public class Main {
    public static void main(String[] args) {
        Book b1 = new Book();
        Book b2 = new Book("1984");
        Book b3 = new Book("Brave New World", "Aldous Huxley");

        b1.printInfo(); // Output: Title: Unknown, Author: Unknown
        b2.printInfo(); // Output: Title: 1984, Author: Unknown
        b3.printInfo(); // Output: Title: Brave New World, Author: Aldous Huxley
    }
}
```

2.4.5 Constructor Chaining with `this()`

In the `Book` example above, one constructor calls another using the `this()` keyword. This is called **constructor chaining** and helps reduce code duplication.

Rules of constructor chaining:

- `this()` must be the **first statement** in the constructor.
- You can only call **one other constructor** using `this()` in a constructor.

2.4.6 Best Practices

- Use **constructor overloading** to give flexibility to object creation.
- Use **constructor chaining** to avoid repeating initialization logic.
- Use **`this` to differentiate parameters** from fields when names overlap.
- Avoid doing complex logic in constructors—keep them focused on initialization.
- If an object must always have certain data, prefer a constructor over setter methods to enforce mandatory values at creation time.

2.4.7 Summary

Constructors are essential for initializing Java objects. Java supports both **default** and **parameterized** constructors, and lets you overload them for flexibility. Constructor chaining using `this()` makes your code cleaner and reduces repetition.

As you design more complex classes, mastering constructor overloading and chaining will help ensure objects are created in a consistent, predictable way. In the next section, you'll explore how Java manages the **lifecycle and scope** of objects in memory.

2.5 Object Lifecycle and Scope

Every object in Java goes through a lifecycle: it is **created**, **used**, and eventually **removed from memory** when no longer needed. Understanding this lifecycle—and the different scopes that variables can have—is essential for writing efficient, bug-free code.

2.5.1 Object Creation and Garbage Collection

Objects in Java are created using the `new` keyword. This allocates memory on the **heap**, where dynamic memory resides:

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

The reference `myCar` points to the memory location of the `Car` object. Java manages memory using an automatic **garbage collector** (GC), which periodically reclaims memory used by objects that are **no longer reachable**.

For example:

```
Car tempCar = new Car();
tempCar = null; // The object is now eligible for garbage collection
```

You do not need to (and cannot) manually delete objects in Java. However, understanding when objects become unreachable is important to avoid memory leaks in long-running programs.

2.5.2 Variable Scope: Local, Instance, and Class-Level

Local Variables

Declared inside methods, constructors, or blocks. They exist only during the method's execution.

```
public void drive() {
    int speed = 60; // local variable
    System.out.println("Speed: " + speed);
}
```

- Scope: Inside the method/block.
- Lifetime: Until the method completes.

Instance Variables

Defined in a class but outside any method. They belong to **each object**.

```
public class Car {
    String model; // instance variable
}
```

- Scope: Accessible by all methods within the object.
- Lifetime: As long as the object exists.

Class-Level Variables (Static)

Declared with the `static` keyword. They belong to the **class itself**, not any instance.

```
public class Car {
    static int numberOfCars; // class-level variable
}
```

- Shared across all instances of the class.
- Lifetime: From class loading until the program ends or the class is unloaded.

2.5.3 Scope in Practice

```
public class Example {
    int instanceVar = 10;
    static int staticVar = 20;

    public void methodScope() {
        int localVar = 30;
        System.out.println(instanceVar); // Allowed
        System.out.println(staticVar);   // Allowed
        System.out.println(localVar);    // Allowed
    }

    public void anotherMethod() {
        // System.out.println(localVar); // Error: not visible here
    }
}
```

This illustrates that local variables are not accessible outside their defining method, while instance and static variables are accessible throughout the class.

2.5.4 Finalization and Memory Management

Java used to provide a `finalize()` method, which was called by the garbage collector before reclaiming an object. However, it's now deprecated and **not recommended** due to unpredictability and performance concerns.

Instead, use **try-with-resources** or implement **AutoCloseable** for releasing non-memory resources (like file handles or network connections).

2.5.5 Summary

Java simplifies memory management through **automatic garbage collection**, ensuring objects are cleaned up when no longer in use. Understanding variable scope—**local**, **instance**, and **class-level**—helps you write clear, maintainable code and avoid common pitfalls such as unintended variable reuse or memory retention.

In later chapters, you'll learn how this knowledge supports encapsulation, modularity, and efficient resource use in object-oriented design.

Chapter 3.

Encapsulation

1. Access Modifiers (`private`, `public`, `protected`, default)
2. Getters and Setters
3. Immutable Objects
4. JavaBeans and Encapsulation in Practice

3 Encapsulation

3.1 Access Modifiers (`private`, `public`, `protected`, default)

Access modifiers in Java are fundamental to the concept of **encapsulation**, one of the core principles of object-oriented programming. They control the **visibility** of classes, fields, methods, and constructors, defining which parts of a program can access or modify them. Proper use of access modifiers helps create **secure**, **maintainable**, and **well-structured** code by restricting direct access to internal implementation details.

3.1.1 The Four Access Levels in Java

Java provides four access levels: `private`, `public`, `protected`, and the default (also called **package-private** or **no modifier**). Each modifier controls visibility differently:

`private`

- **Scope:** Accessible **only within the defining class**.
- **Purpose:** To hide sensitive data and implementation details from other classes, enforcing strict encapsulation.
- **Usage:** Typically used for **fields** and helper methods that should not be exposed outside the class.

Example:

```
public class BankAccount {
    private double balance; // hidden from other classes

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

    public double getBalance() {
        return balance;
    }
}
```

Here, `balance` cannot be accessed directly outside `BankAccount`; it's only modifiable through controlled methods, protecting the integrity of the data.

`public`

- **Scope:** Accessible from **anywhere** in the program.
- **Purpose:** To expose the class, methods, or fields as part of the public API.
- **Usage:** For methods or constants that should be available to all other classes, such as

interfaces, constructors, or utility methods.

Example:

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

Anyone can call `add()` because it is `public`.

`protected`

- **Scope:** Accessible within the **same package** and **subclasses**, even if they are in different packages.
- **Purpose:** To allow limited visibility that supports inheritance while restricting broad access.
- **Usage:** For fields or methods that subclasses need to access or override but should not be fully public.

Example:

```
public class Animal {
    protected String species;

    protected void makeSound() {
        System.out.println("Some generic sound");
    }
}

public class Dog extends Animal {
    public void bark() {
        makeSound();  // Allowed because of protected access
        System.out.println("Bark!");
    }
}
```

Here, `species` and `makeSound()` are accessible to `Dog`, even if it's in a different package.

Default (Package-Private)

- **Scope:** Accessible only within **the same package**.
- **Purpose:** To allow classes within the same package to collaborate without exposing members publicly.
- **Usage:** When you want to restrict access to a logical grouping of classes (a package), but don't want to make them fully private.

Example:

```
class Helper {
    void assist() {
        System.out.println("Assisting...");
```

```
    }
}

public class Service {
    public void execute() {
        Helper helper = new Helper();
        helper.assist();  // Allowed, same package access
    }
}
```

Note: The `Helper` class and its `assist()` method have no explicit modifier, so they are package-private.

3.1.2 Why Access Control Is Vital for Encapsulation

Encapsulation is about **hiding internal details** and **exposing only what is necessary**. Access modifiers enforce this by:

- **Protecting data integrity:** By restricting direct access to fields (`private`), you prevent unwanted changes that could corrupt the object state.
- **Reducing complexity:** Clients interact only with public methods, simplifying the interface.
- **Supporting maintainability:** Internal implementation can change without affecting external code.
- **Enhancing security:** Sensitive information is shielded from external classes.

3.1.3 Package-Level Access and Inheritance

Java's package system groups related classes, and access modifiers work closely with it:

- Classes without an explicit access modifier are **package-private** and invisible outside their package.
- Subclasses inherit `protected` members and can access them even across packages, while default members are inaccessible outside the package.
- Public classes and members form the API boundary, intended for unrestricted use.

3.1.4 Summary Example Demonstrating All Modifiers

```
package com.example;
```

```java
public class Vehicle {
    private String engineNumber;     // Accessible only inside Vehicle
    protected int maxSpeed;          // Accessible in subclasses and package
    String model;                    // Package-private: accessible in same package
    public String brand;             // Accessible everywhere

    private void startEngine() {
        System.out.println("Engine started.");
    }

    protected void accelerate() {
        System.out.println("Accelerating...");
    }

    void displayModel() {
        System.out.println("Model: " + model);
    }

    public void showBrand() {
        System.out.println("Brand: " + brand);
    }
}
```

- `engineNumber` is tightly guarded to protect critical data.
- `maxSpeed` can be used or overridden by subclasses.
- `model` is package-private, so only classes in `com.example` can see it.
- `brand` is public for anyone to access.

3.1.5 Conclusion

Access modifiers are a vital part of Java's approach to **encapsulation**. By carefully choosing `private`, `protected`, `public`, or default access, you can design classes that hide their internal workings, expose clear interfaces, and support flexible, secure, and maintainable codebases. Mastery of these modifiers sets the foundation for building professional-grade Java applications.

3.2 Getters and Setters

In Java, **getters** and **setters** are methods that provide controlled access to the private fields of a class. They are essential tools for **encapsulation**, allowing you to hide the internal data of an object while still providing a way to read or modify it safely.

3.2.1 The Role of Getters and Setters

When you declare fields as `private` to protect the object's internal state, other classes cannot access or modify those fields directly. To enable controlled interaction, you create **getter** methods to retrieve the field values and **setter** methods to update them.

This approach ensures:

- **Validation:** You can check input values before setting a field.
- **Immutability:** You can restrict which fields are modifiable.
- **Abstraction:** You hide how data is stored internally.
- **Maintainability:** Future changes in field implementation won't affect external code.

3.2.2 Standard Syntax and Naming Conventions

By convention, getter and setter methods follow a simple naming pattern:

Field type	Getter method name	Setter method name
Any type (except boolean)	`getFieldName()`	`setFieldName(Type value)`
Boolean	`isFieldName()` or `getFieldName()`	`setFieldName(boolean value)`

- The first letter of the field name is capitalized in method names.
- Getters return the field value.
- Setters return `void` and accept one parameter to assign to the field.

3.2.3 Example: Basic Getter and Setter

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

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }
```

```
    // Getter for age
    public int getAge() {
        return age;
    }

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

3.2.4 Adding Validation Logic in Setters

One of the biggest benefits of using setters is the ability to **validate data before updating a field**. This ensures the object remains in a consistent, valid state.

```
public class Person {
    private int age;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age < 0) {
            System.out.println("Age cannot be negative.");
        } else {
            this.age = age;
        }
    }
}
```

Now, setting a negative age will not corrupt the object:

```
Person p = new Person();
p.setAge(-5);  // Outputs: Age cannot be negative.
```

3.2.5 Read-Only and Write-Only Properties

Depending on design requirements, some fields should only be **readable** or **writable**:

- **Read-only property:** Provide a **getter** but **no setter**. External code can view but cannot modify the field.

```
public class Employee {
    private final String employeeId;
```

```
    public Employee(String id) {
        this.employeeId = id;
    }

    public String getEmployeeId() {
        return employeeId;
    }
}
```

- **Write-only property:** Provide a **setter** but no getter (less common). Useful when you want to accept input but keep it hidden.

```
public class SecureData {
    private String password;

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Full runnable code:

```
public class Main {
    public static void main(String[] args) {
        // Example: Basic Person with validation
        Person person = new Person();
        person.setName("Alice");
        person.setAge(30);
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());

        person.setAge(-5); // Triggers validation warning

        // Example: Read-only employee ID
        Employee emp = new Employee("EMP123");
        System.out.println("Employee ID: " + emp.getEmployeeId());

        // Example: Write-only secure data
        SecureData data = new SecureData();
        data.setPassword("s3cr3t");  // No way to read it from outside
    }
}

class Person {
    private String name;
    private int age;

    // Getter and Setter with validation
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

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

    public void setAge(int age) {
        if (age < 0) {
            System.out.println("Age cannot be negative.");
        } else {
            this.age = age;
        }
    }
}

class Employee {
    private final String employeeId;

    public Employee(String id) {
        this.employeeId = id;
    }

    public String getEmployeeId() {
        return employeeId;
    }
}

class SecureData {
    private String password;

    public void setPassword(String password) {
        this.password = password;
    }
}
```

3.2.6 Summary

Getters and setters are vital for controlling access to class fields, allowing you to:

- Protect internal data by using `private` fields.
- Provide clear, standardized ways to read or update data.
- Include validation logic in setters to maintain object integrity.
- Create read-only or write-only fields as needed by omitting setters or getters.

Mastering getters and setters lays the groundwork for writing safe, encapsulated Java classes that promote robust software design. In the next section, we will explore how to create **immutable objects**, which take encapsulation even further by preventing changes to the object's state after creation.

3.3 Immutable Objects

3.3.1 What Is Immutability?

In Java and object-oriented design, an **immutable object** is an object whose state cannot be changed after it is created. Once you set the values of its fields, those values remain constant throughout the object's lifetime.

Immutability is a powerful design concept that promotes simplicity, safety, and predictability in software development.

3.3.2 Advantages of Immutability

- **Thread safety:** Immutable objects are inherently thread-safe because their state cannot change, eliminating synchronization issues.
- **Simpler reasoning:** Since objects don't change, you don't have to track modifications across the program.
- **Safe sharing:** You can freely share immutable objects between different parts of a program or threads without worrying about side effects.
- **Cache and reuse:** Immutable objects can be cached or reused without concerns about unexpected changes.
- **Prevents bugs:** Reduces accidental data corruption by disallowing state changes.

3.3.3 How to Create Immutable Classes in Java

To make a class immutable, follow these guidelines:

1. **Declare the class as `final` (optional but recommended):** Prevents subclassing which might alter immutability.
2. **Make all fields `private` and `final`:** Fields cannot be reassigned after initialization.
3. **Do not provide setter methods:** No way to modify fields after construction.
4. **Initialize all fields via constructor:** Set all field values once at object creation.
5. **If fields reference mutable objects, ensure defensive copies:** Return copies or immutable views instead of the original references.
6. **Provide only getters:** Methods that expose field values without allowing modifications.

3.3.4 Example: The Immutable `String` Class

Java's built-in `String` class is a classic example of an immutable class. Once a `String` object is created, its value cannot be changed.

```
String greeting = "Hello";
greeting.toUpperCase(); // Returns a new String; original is unchanged
System.out.println(greeting); // Prints "Hello"
```

Calling methods like `toUpperCase()` returns new `String` instances rather than modifying the original, ensuring immutability.

3.3.5 Creating a Custom Immutable Class

Here's an example of a simple immutable `Person` class:

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

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

    public String getName() {
        return name;
    }

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

Because the class and fields are `final`, and no setters are provided, once a `Person` object is created, its `name` and `age` cannot be changed.

```
Person p = new Person("Alice", 30);
// p.name = "Bob"; // Compile error: cannot assign a value to final field
```

3.3.6 Defensive Copies for Mutable Fields

If your class contains fields that refer to mutable objects (like arrays or collections), you should return copies to maintain immutability:

```java
public final class Team {
    private final List<String> members;

    public Team(List<String> members) {
        // Create a new list to prevent external modification
        this.members = new ArrayList<>(members);
    }

    public List<String> getMembers() {
        // Return a copy to prevent caller from modifying internal list
        return new ArrayList<>(members);
    }
}
```

Full runnable code:

```java
import java.util.ArrayList;
import java.util.List;

public final class Person {
    private final String name;
    private final int age;

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

    public String getName() { return name; }

    public int getAge() { return age; }
}

final class Team {
    private final List<String> members;

    public Team(List<String> members) {
        // Defensive copy to protect internal state
        this.members = new ArrayList<>(members);
    }

    public List<String> getMembers() {
        // Return a copy to maintain immutability
        return new ArrayList<>(members);
    }
}

public class Main {
    public static void main(String[] args) {
        // Immutable Person
        Person p = new Person("Alice", 30);
        System.out.println(p.getName() + ", age " + p.getAge());

        // Mutable list outside
        List<String> list = new ArrayList<>();
        list.add("Bob");
        list.add("Carol");
```

```
        Team team = new Team(list);
        System.out.println("Team members: " + team.getMembers());

        // Modify external list after creation
        list.add("Dave");
        System.out.println("After modifying original list:");
        System.out.println("Team members: " + team.getMembers());

        // Try modifying the returned list from getter
        List<String> membersFromGetter = team.getMembers();
        membersFromGetter.add("Eve");
        System.out.println("After modifying getter's list:");
        System.out.println("Team members: " + team.getMembers());
    }
}
```

3.3.7 Thread-Safety Benefits

Immutable objects are naturally thread-safe because:

- Their state cannot change, so no synchronization is needed.
- Multiple threads can access the same immutable object without risk of data races.
- This simplifies concurrent programming and reduces bugs related to shared mutable state.

3.3.8 Summary

Immutability is a key design strategy that enhances reliability and safety in Java applications. By creating immutable classes with `final` fields, no setters, and proper encapsulation, you gain benefits such as:

- Simplified reasoning about code behavior.
- Thread-safe objects without synchronization overhead.
- Prevention of accidental or unauthorized data changes.

3.4 JavaBeans and Encapsulation in Practice

3.4.1 What Are JavaBeans?

JavaBeans are a widely used convention in Java programming that defines a standard way to create **reusable software components**. The JavaBeans specification sets guidelines on

how classes should be designed to work smoothly with development tools, frameworks, and libraries. Understanding JavaBeans is important because many Java technologies—like GUI builders, enterprise frameworks, and persistence libraries—expect components to follow these conventions.

3.4.2 Key JavaBeans Conventions

A **JavaBean** class typically follows these rules:

1. **Private fields:** All properties are declared as private to enforce encapsulation.
2. **Public getter and setter methods:** Each property has public methods named following a `getXxx()` / `setXxx()` pattern for accessing and modifying values.
3. **No-argument constructor:** The class provides a public default constructor without parameters, allowing tools to instantiate the bean easily.
4. **Serializable:** Implements the `Serializable` interface to allow objects to be converted into a byte stream for storage, communication, or caching.

3.4.3 Properties in JavaBeans

A property represents a logical attribute or characteristic of a JavaBean. The property name corresponds to the field name, and access is provided by getters and setters. For example, a field named `age` would have:

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

This standardized naming convention enables automated tools and frameworks to manipulate bean properties without needing to know their internal implementation.

3.4.4 Example JavaBean Class

Here's a simple example of a JavaBean called `Student`:

```
import java.io.Serializable;

public class Student implements Serializable {
    private String name;
    private int grade;

    // No-argument constructor
    public Student() {
```

```
    }

    // Getter for name property
    public String getName() {
        return name;
    }

    // Setter for name property
    public void setName(String name) {
        this.name = name;
    }

    // Getter for grade property
    public int getGrade() {
        return grade;
    }

    // Setter for grade property
    public void setGrade(int grade) {
        if (grade >= 0 && grade <= 100) {
            this.grade = grade;
        }
    }
}
```

This class meets the JavaBeans standards: it has private fields, public getters/setters, a no-arg constructor, and implements `Serializable`.

3.4.5 How JavaBeans Support Encapsulation and Practical Development

JavaBeans put encapsulation into practice by enforcing **private fields and controlled access through getters and setters**. This hides internal data and allows validation or logic to be inserted in setters, helping maintain object integrity.

Moreover, the conventions facilitate:

- **Tool compatibility:** IDEs, GUI builders, and frameworks can introspect JavaBeans using reflection to build UIs, map data, or generate code.
- **Component reuse:** Beans can be easily reused across different applications.
- **Standardization:** Having a common pattern makes it easier for developers to understand and work with various Java libraries.

3.4.6 Summary

JavaBeans offer a practical embodiment of encapsulation principles with standardized patterns for fields, constructors, and methods. They are an integral part of professional Java development, ensuring your classes work well with many tools and frameworks while promoting safe,

maintainable, and reusable code. As you progress, you'll frequently encounter JavaBeans, especially in enterprise and GUI applications, making mastery of this pattern invaluable.

Chapter 4.

Inheritance

1. `extends` Keyword and Inheriting Methods
2. Method Overriding
3. `super` Keyword
4. Constructors and Inheritance
5. Inheritance Best Practices

4 Inheritance

4.1 `extends` Keyword and Inheriting Methods

Inheritance is a foundational concept in object-oriented programming that enables one class to acquire properties and behaviors of another class. In Java, this is accomplished using the **`extends`** keyword.

4.1.1 What Is the `extends` Keyword?

The `extends` keyword is used to declare that a new class (called a **subclass** or **child class**) is derived from an existing class (called a **superclass** or **parent class**). This establishes an **is-a relationship**, meaning the subclass is a specialized type of the superclass.

Syntax:

```
class SubClass extends SuperClass {
    // additional fields and methods
}
```

For example:

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}
```

Here, `Dog` extends `Animal`, meaning `Dog` inherits all accessible members (fields and methods) from `Animal`.

4.1.2 What Does a Subclass Inherit?

When a class extends another:

- **All non-private fields and methods** of the superclass are inherited by the subclass.
- This includes public, protected, and package-private (default) members if the subclass is in the same package.
- **Private members are not inherited** directly, though their public or protected

accessors can be used.
- Constructors are **not inherited**, but the subclass constructor can invoke superclass constructors using `super()`.

In the `Animal` and `Dog` example, `Dog` inherits the `eat()` method and can call it directly:

```
Dog d = new Dog();
d.eat();  // Output: Eating...
d.bark(); // Output: Barking...
```

4.1.3 Simple Class Hierarchy Example

Consider a class hierarchy representing vehicles:

```
class Vehicle {
    String brand;

    void start() {
        System.out.println("Vehicle is starting");
    }
}

class Car extends Vehicle {
    int numberOfDoors;

    void honk() {
        System.out.println("Car horn sounds");
    }
}
```

Here, `Car` inherits the `brand` field and the `start()` method from `Vehicle`. It also adds its own field `numberOfDoors` and method `honk()`.

Usage:

```
Car myCar = new Car();
myCar.brand = "Toyota";  // inherited field
myCar.start();           // inherited method
myCar.honk();            // subclass method
```

4.1.4 What Is Not Inherited?

- **Private members** of the superclass are hidden and not accessible in the subclass.
- Constructors are unique to each class and must be explicitly called.
- Static members belong to the class itself and are accessed differently.

For example:

```
class Person {
    private String ssn;  // private, not inherited

    public String getSsn() {
        return ssn;
    }
}

class Employee extends Person {
    void printSsn() {
        // System.out.println(ssn); // Error: ssn is private in Person
        System.out.println(getSsn()); // Allowed via public getter
    }
}
```

Full runnable code:

```
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // Inherited method
        d.bark();  // Dog's own method

        Car myCar = new Car();
        myCar.brand = "Toyota";   // inherited field
        myCar.start();            // inherited method
        myCar.honk();             // Car's own method

        Employee emp = new Employee();
        emp.setSsn("123-45-6789");
        emp.printSsn();           // prints SSN via getter
    }
}

class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

class Vehicle {
    String brand;

    void start() {
        System.out.println("Vehicle is starting");
    }
}

class Car extends Vehicle {
```

```
    int numberOfDoors;

    void honk() {
        System.out.println("Car horn sounds");
    }
}

class Person {
    private String ssn;

    public String getSsn() {
        return ssn;
    }

    public void setSsn(String ssn) {
        this.ssn = ssn;
    }
}

class Employee extends Person {
    void printSsn() {
        // System.out.println(ssn); // Not accessible: ssn is private in Person
        System.out.println("SSN: " + getSsn()); // Access via public getter
    }
}
```

4.1.5 Why Use Inheritance in Design?

Inheritance is useful to:

- **Promote code reuse:** Common functionality is placed in a superclass, reducing duplication.
- **Model real-world hierarchies:** Reflects natural relationships like Animal → Dog, Vehicle → Car.
- **Support polymorphism:** Subclasses can be treated as instances of the superclass, enabling flexible code.

However, it should be used thoughtfully; overusing inheritance can lead to rigid and tightly coupled designs.

4.1.6 Summary

The `extends` keyword in Java defines inheritance, allowing a subclass to acquire accessible methods and fields from a superclass. This mechanism enables efficient code reuse and models hierarchical relationships naturally. While private members and constructors are not inherited, subclasses can access superclass behavior and extend it with additional features.

Understanding what is inherited—and how to leverage inheritance properly—is critical for designing clean, maintainable object-oriented systems.

4.2 Method Overriding

4.2.1 What Is Method Overriding?

Method overriding is a fundamental feature of object-oriented programming where a subclass provides its own implementation of a method that is already defined in its superclass. The goal is to **change or extend the behavior** inherited from the parent class.

This allows a subclass to customize or replace the behavior of methods it inherits, enabling flexible and dynamic behavior.

4.2.2 How Is Overriding Different from Overloading?

It's important not to confuse **method overriding** with **method overloading**:

- **Overriding** means a subclass has a method with the **same name, return type, and parameter list** as a method in its superclass. It replaces the superclass's behavior at runtime.
- **Overloading** means defining multiple methods in the same class with the **same name but different parameter lists**. This is resolved at compile-time.

Example:

```
class Calculator {
    int add(int a, int b) { return a + b; }       // Overloading example
    int add(int a, int b, int c) { return a + b + c; }
}

class AdvancedCalculator extends Calculator {
    @Override
    int add(int a, int b) {                       // Overriding example
        System.out.println("Adding two numbers:");
        return super.add(a, b);
    }
}
```

4.2.3 The `@Override` Annotation

Java provides the `@Override` annotation, which you place just before a method to indicate it overrides a method from the superclass.

Benefits of `@Override`:

- **Compile-time checking:** The compiler verifies that the method correctly overrides a superclass method. If it does not, it will produce an error, helping catch typos or signature mismatches early.
- **Improved readability:** It clearly signals to readers and tools that the method overrides another.

Example:

```
class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("Dog barks");
    }
}
```

If you accidentally write `speek()` instead of `speak()`, the compiler will flag it if you use `@Override`.

4.2.4 Example: Overriding Methods to Change Behavior

Consider a base class `Vehicle` and a subclass `Car`:

```
class Vehicle {
    void start() {
        System.out.println("Vehicle starting...");
    }
}

class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car ignition started!");
    }
}
```

Usage:

```
Vehicle myVehicle = new Vehicle();
myVehicle.start();  // Output: Vehicle starting...

Car myCar = new Car();
myCar.start();      // Output: Car ignition started!
```

Here, the `Car` class overrides the `start()` method to provide behavior specific to cars, replacing the generic vehicle start message.

4.2.5 Rules for Overriding Methods

When overriding a method in Java, the following rules apply:

1. **Method Signature Must Match Exactly:** The method name, return type, and parameter types must be identical to the superclass method.
2. **Access Level Cannot Be More Restrictive:** The overriding method must have the **same or broader** access modifier. For example, if the superclass method is `protected`, the overriding method can be `protected` or `public` but not `private`.
3. **Return Type Must Be Compatible:** The overriding method's return type must be the same or a subtype (covariant return type) of the superclass method's return type.
4. **Exception Handling:** The overriding method can throw the same exceptions or fewer (more specific), but not new or broader checked exceptions.
5. **Static Methods Cannot Be Overridden:** Static methods belong to the class, not instances, so they cannot be overridden. They can be re-declared in the subclass, which hides the superclass static method (called method hiding).

4.2.6 Polymorphism and Overriding

Method overriding is the key enabler of **runtime polymorphism** (also called dynamic dispatch) in Java. Polymorphism allows the JVM to decide at runtime which method implementation to invoke based on the actual object type, not the declared reference type.

Example:

```
Vehicle myVehicle = new Car();
myVehicle.start();  // Output: Car ignition started!
```

Even though `myVehicle` is declared as type `Vehicle`, the actual object is a `Car`. Because `start()` is overridden, the `Car` version is called at runtime, demonstrating polymorphism.

Full runnable code:

```java
public class Main {
    public static void main(String[] args) {
        Vehicle myVehicle = new Vehicle();
        myVehicle.start();  // Output: Vehicle starting...

        Car myCar = new Car();
        myCar.start();      // Output: Car ignition started!

        Vehicle polyVehicle = new Car();
        polyVehicle.start(); // Output: Car ignition started!
    }
}

class Vehicle {
    void start() {
        System.out.println("Vehicle starting...");
    }
}

class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car ignition started!");
    }
}
```

4.2.7 Summary

Method overriding lets subclasses provide specialized behavior for methods inherited from their superclasses. By using the `@Override` annotation, developers get compile-time safety and improved code clarity.

The rules for overriding ensure consistent behavior and proper access control. Combined with polymorphism, overriding enables flexible and extensible designs where objects behave according to their actual types, not just their declared types.

Mastering method overriding is essential for building dynamic, maintainable, and scalable Java applications.

4.3 `super` Keyword

In Java, the `super` keyword is a special reference that allows a subclass to access members (methods, constructors, or fields) of its immediate superclass. It acts as a bridge between the child and parent classes, enabling reuse and extension of functionality.

4.3.1 Using `super()` to Call Superclass Constructors

One of the most common uses of `super` is to invoke a superclass constructor from within a subclass constructor. This is important because constructors are **not inherited**; each class must explicitly initialize itself. Using `super()` ensures the superclass is properly initialized before the subclass adds its own initialization.

Example:

```
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called");
    }
}

class Dog extends Animal {
    public Dog(String name) {
        super(name);  // Calls Animal's constructor
        System.out.println("Dog constructor called");
    }
}
```

When you create a new `Dog`, the output will be:

```
Animal constructor called
Dog constructor called
```

This demonstrates that the superclass constructor runs first, setting up the `name` field, followed by the subclass constructor.

If you don't explicitly call `super()`, Java inserts a no-argument `super()` call automatically—but this only works if the superclass has a no-argument constructor. Otherwise, a compile-time error occurs.

4.3.2 Invoking Superclass Methods Using `super`

Besides constructors, `super` allows subclasses to **invoke overridden methods** in the superclass. This is useful when you want to extend or reuse behavior rather than completely replace it.

Example:

```
class Vehicle {
    void start() {
        System.out.println("Vehicle is starting");
    }
```

```
}

class Car extends Vehicle {
    @Override
    void start() {
        super.start();  // Call superclass method
        System.out.println("Car ignition turned on");
    }
}
```

When calling `start()` on a `Car` object, the output is:

```
Vehicle is starting
Car ignition turned on
```

Here, the `Car` method calls the original `Vehicle` method first, then adds its own functionality, demonstrating behavior extension.

4.3.3 Accessing Superclass Fields with `super`

If the subclass has fields with the same name as the superclass, you can use `super` to disambiguate and access the superclass field:

```
class Parent {
    int value = 10;
}

class Child extends Parent {
    int value = 20;

    void printValues() {
        System.out.println("Child value: " + value);
        System.out.println("Parent value: " + super.value);
    }
}
```

Output:

```
Child value: 20
Parent value: 10
```

Full runnable code:

```
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        System.out.println();

        Car car = new Car();
```

```
        car.start();
        System.out.println();

        Child child = new Child();
        child.printValues();
    }
}

class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called");
    }
}

class Dog extends Animal {
    public Dog(String name) {
        super(name);  // Calls Animal's constructor
        System.out.println("Dog constructor called");
    }
}

class Vehicle {
    void start() {
        System.out.println("Vehicle is starting");
    }
}

class Car extends Vehicle {
    @Override
    void start() {
        super.start();  // Call superclass method
        System.out.println("Car ignition turned on");
    }
}

class Parent {
    int value = 10;
}

class Child extends Parent {
    int value = 20;

    void printValues() {
        System.out.println("Child value: " + value);
        System.out.println("Parent value: " + super.value);
    }
}
```

4.3.4 Common Pitfalls and Best Practices

- **Calling `super()` must be the first statement in a constructor.** You cannot perform other operations before calling the superclass constructor.
- Avoid overusing `super` to access fields; prefer encapsulation through getters/setters.
- Remember `super` refers only to the immediate superclass, not higher ancestors.
- Use `super` thoughtfully to enhance readability and maintain clear relationships between classes.

4.3.5 Summary

The `super` keyword is a vital tool in inheritance, providing controlled access to superclass constructors, methods, and fields. It helps build clean, maintainable hierarchies by allowing subclasses to initialize and reuse superclass behavior effectively. Mastering `super` unlocks powerful ways to extend and customize class functionality in Java.

4.4 Constructors and Inheritance

In Java, constructors are special methods responsible for initializing new objects. When inheritance is involved, understanding how constructors work in a class hierarchy is crucial for writing robust, maintainable code.

4.4.1 Constructor Invocation Order in Inheritance

When you create an instance of a subclass, constructors are called starting from the **topmost superclass down to the subclass**. This ensures that the superclass part of the object is fully initialized before the subclass adds its own initialization.

For example, if class `A` is the superclass and class `B` extends `A`, creating a new `B` object calls the constructor of `A` first, then `B`'s constructor.

This top-down invocation order maintains the integrity of the inheritance chain and prevents partially initialized objects.

4.4.2 Implicit and Explicit Calls to Superclass Constructors

- **Implicit call:** If a subclass constructor does **not** explicitly call a superclass constructor, Java automatically inserts a call to the **no-argument constructor** of the superclass, `super()`, at the start of the subclass constructor.
- **Explicit call:** A subclass can explicitly call a specific superclass constructor using `super(arguments)` with parameters.

Example:

```
class Animal {
    Animal() {
        System.out.println("Animal constructor");
    }

    Animal(String name) {
        System.out.println("Animal constructor with name: " + name);
    }
}

class Dog extends Animal {
    Dog() {
        super();  // Implicit if omitted
        System.out.println("Dog constructor");
    }

    Dog(String name) {
        super(name);  // Explicit call to superclass constructor with argument
        System.out.println("Dog constructor with name");
    }
}
```

4.4.3 Constructor Chaining Example

```
Dog dog1 = new Dog();
// Output:
// Animal constructor
// Dog constructor

Dog dog2 = new Dog("Buddy");
// Output:
// Animal constructor with name: Buddy
// Dog constructor with name
```

The `Dog` constructors invoke the `Animal` constructors explicitly with `super()`. This chaining ensures proper initialization of inherited state.

Full runnable code:

```java
public class Main {
    public static void main(String[] args) {
        Dog dog1 = new Dog();
        System.out.println();

        Dog dog2 = new Dog("Buddy");
    }
}

class Animal {
    Animal() {
        System.out.println("Animal constructor");
    }

    Animal(String name) {
        System.out.println("Animal constructor with name: " + name);
    }
}

class Dog extends Animal {
    Dog() {
        super();  // Implicit if omitted
        System.out.println("Dog constructor");
    }

    Dog(String name) {
        super(name);  // Explicit call to superclass constructor with argument
        System.out.println("Dog constructor with name");
    }
}
```

4.4.4 Why Constructors Are Not Inherited

Unlike regular methods, **constructors are not inherited** because:

1. **Constructors initialize the specific class they belong to.** The superclass constructor prepares the superclass part of the object, while the subclass constructor handles subclass-specific initialization.
2. **Each class defines how its objects are constructed.** Allowing constructor inheritance would blur the distinct initialization responsibilities between classes.
3. **Java enforces explicit constructor calls.** This avoids confusion and ensures developers consciously decide how to initialize both superclass and subclass parts.

4.4.5 Summary

When creating objects in an inheritance hierarchy, constructors are invoked in a top-down manner—from superclass to subclass. Subclass constructors either implicitly or explicitly call superclass constructors using `super()`. This ensures the entire object is properly initialized.

Understanding that constructors are not inherited emphasizes the unique role constructors play: each class controls its own initialization. Constructor chaining through `super()` calls is a key tool for coordinating initialization across the inheritance tree.

Mastering constructor behavior in inheritance will help you write clear, predictable, and reliable Java code.

4.5 Inheritance Best Practices

Inheritance is a powerful mechanism in Java, but it comes with responsibilities and potential pitfalls. Knowing **when and how to use inheritance effectively** is key to designing clean, maintainable software.

4.5.1 When to Use Inheritance vs. Composition

Inheritance expresses an **"is-a"** relationship—meaning the subclass should be a specialized version of the superclass. For example, a `Car` **is a** `Vehicle`, so inheritance makes sense.

However, if you want to express a **"has-a"** relationship, such as a `Car` **has an** `Engine`, composition (embedding objects as fields) is a better choice. Composition promotes greater flexibility by allowing objects to be assembled from interchangeable parts rather than fixed class hierarchies.

Rule of thumb:

- Use **inheritance** for **type specialization**.
- Use **composition** for **code reuse and flexibility**.

4.5.2 Common Pitfalls of Inheritance

1. **Tight Coupling:** Subclasses are tightly coupled to the superclass implementation. Changes in the superclass can inadvertently break subclass behavior, leading to fragile designs.
2. **Fragile Base Class Problem:** If the superclass is modified, all subclasses might be

affected. This problem grows as hierarchies become deep and complex.

3. **Inheritance for Code Reuse Alone:** Avoid using inheritance solely to reuse code. This often leads to inappropriate relationships and violates the **Liskov Substitution Principle** (an "is-a" relationship must hold).
4. **Overusing Inheritance:** Deep inheritance hierarchies become hard to understand and maintain. Favor composition and interfaces where appropriate.

4.5.3 Guidelines for Designing Inheritance Hierarchies

- **Keep hierarchies shallow:** Deep inheritance trees are hard to follow and maintain.
- **Model real-world relationships accurately:** Ensure subclasses truly represent specialized versions of the superclass.
- **Prefer interfaces or abstract classes:** Use interfaces to define capabilities without forcing implementation inheritance.
- **Design for extension:** Write superclass methods that subclasses can safely override.
- **Encapsulate superclass data:** Avoid exposing internal state; provide protected or public methods to safely interact with it.
- **Use final classes and methods sparingly:** Final prevents inheritance but can increase safety when extension is not intended.

4.5.4 Examples of Good Inheritance Design Patterns

- **Template Method Pattern:** Define the skeleton of an algorithm in the superclass and allow subclasses to override specific steps.
- **Factory Method Pattern:** Use inheritance to defer object creation to subclasses, enabling flexible instantiation.
- **Strategy Pattern (using interfaces):** Favor composition over inheritance by defining interchangeable behaviors as separate classes.

Chapter 5.

Polymorphism

1. Compile-time vs Runtime Polymorphism
2. Upcasting and Downcasting
3. Dynamic Method Dispatch
4. Practical Use Cases for Polymorphism

5 Polymorphism

5.1 Compile-time vs Runtime Polymorphism

Polymorphism is one of the core concepts in object-oriented programming. It means **"many forms"**, allowing the same code to behave differently depending on the context. In Java, polymorphism manifests in two main forms: **compile-time polymorphism** and **runtime polymorphism**. Understanding their distinctions is crucial to mastering Java's flexible and powerful design.

5.1.1 Compile-Time Polymorphism (Static Polymorphism)

Compile-time polymorphism occurs when the compiler determines which method to invoke **before the program runs**. This is also called **static binding** or **early binding**.

The primary way to achieve compile-time polymorphism in Java is through **method overloading** — defining multiple methods with the same name but different parameter lists within the same class.

Example of Method Overloading:

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

    int add(int a, int b, int c) {
        return a + b + c;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

Here, the method `add` is overloaded three times. When you call `add(2, 3)`, the compiler knows to use the first method. If you call `add(2, 3, 4)`, it uses the second, and for `add(2.0, 3.0)`, the third.

How does compile-time polymorphism work?

- The compiler selects the method to invoke based on the method signature (name + parameters).
- Since this decision happens at compile time, it is fast and efficient.
- However, the method cannot be changed dynamically during program execution.

5.1.2 Runtime Polymorphism (Dynamic Polymorphism)

Runtime polymorphism occurs when the decision about which method to call is deferred **until the program is running**. This is called **dynamic binding** or **late binding**.

The key mechanism behind runtime polymorphism is **method overriding** combined with **inheritance** and **upcasting**.

Example of Method Overriding and Runtime Polymorphism:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

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

public class Test {
    public static void main(String[] args) {
        Animal animal = new Dog();  // Upcasting
        animal.sound();             // Runtime polymorphism in action
    }
}
```

Output:

```
Dog barks
```

Even though the reference type is `Animal`, the actual object is a `Dog`. At runtime, Java invokes the `sound()` method of `Dog`, not `Animal`. This dynamic method dispatch allows flexible and extensible code.

5.1.3 Key Differences

Aspect	Compile-time Polymorphism	Runtime Polymorphism
Binding	Static (at compile time)	Dynamic (at runtime)
Mechanism	Method Overloading	Method Overriding
Performance	Faster due to static binding	Slightly slower due to lookup
Flexibility	Less flexible (fixed method)	More flexible (dynamic behavior)

Aspect	Compile-time Polymorphism	Runtime Polymorphism
Use Cases	Multiple methods with different parameters in same class	Extending and customizing behavior in subclass

5.1.4 Practical Implications and Benefits

- **Compile-time polymorphism** improves code readability by allowing methods to perform similar tasks with different inputs, reducing method name clutter.
- It provides **type safety and better performance**, as method calls are resolved early.
- **Runtime polymorphism** enables **extensibility and maintainability** by allowing new subclasses to change behavior without altering existing code.
- It is essential for implementing frameworks and APIs where objects of different types are handled through a common interface or superclass.

5.1.5 Summary

Both compile-time and runtime polymorphism enable flexible code, but they operate differently:

- **Compile-time polymorphism** relies on method overloading and static binding to decide method calls during compilation.
- **Runtime polymorphism** uses method overriding and dynamic binding to decide method calls during execution.

Mastering both types helps you design clean, reusable, and scalable Java applications that leverage the full power of polymorphism.

5.2 Upcasting and Downcasting

In Java's object-oriented world, **casting** between types is an essential concept that allows flexibility when working with class hierarchies. Two important forms of casting are **upcasting** and **downcasting**. Understanding these helps you write polymorphic and safe code.

5.2.1 What is Upcasting?

Upcasting is when a subclass object is referenced by a superclass type. This is an **implicit and safe** operation because every subclass object "is-a" superclass object.

Example:

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

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }

    void fetch() {
        System.out.println("Dog fetches");
    }
}

public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Animal animal = dog;  // Upcasting (implicit)
        animal.sound();       // Calls Dog's overridden method
    }
}
```

Output:

```
Dog barks
```

Here, the `Dog` object is referenced as an `Animal`. The compiler allows this automatically (implicit cast). Upcasting is useful when you want to treat different subclasses uniformly, such as storing them in an array or passing them to methods expecting the superclass type.

5.2.2 What is Downcasting?

Downcasting is the reverse: casting a superclass reference back to a subclass type. Unlike upcasting, **downcasting is explicit and potentially unsafe** because the superclass reference may not actually point to an instance of the subclass.

Example:

```
Animal animal = new Dog(); // Upcasting
Dog dog = (Dog) animal;    // Downcasting (explicit)
dog.fetch();               // Now accessible
```

Here, the explicit cast `(Dog)` tells the compiler to treat the `Animal` reference as a `Dog`.

5.2.3 Risks of Downcasting and `instanceof` Checks

If you downcast incorrectly — for example, casting a superclass reference that does **not** point to the subclass — the program throws a **`ClassCastException`** at runtime.

```
Animal animal = new Animal();
Dog dog = (Dog) animal;  // Causes ClassCastException!
```

To avoid this, always perform an **`instanceof` check** before downcasting:

```
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    dog.fetch();
} else {
    System.out.println("Not a Dog!");
}
```

The `instanceof` operator returns `true` only if the object referenced by `animal` is actually an instance of `Dog` or its subclass.

5.2.4 When to Use Upcasting and Downcasting

- **Upcasting** is common and recommended when working with polymorphism. It allows you to write general code that works with any subclass of a common superclass or interface.
- **Downcasting** should be used sparingly and carefully, typically when you need access to subclass-specific methods or fields that are not part of the superclass interface.

5.2.5 Summary Example

```
public class CastingDemo {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();  // Upcasting (safe and implicit)
        myAnimal.sound();             // Polymorphic call: Dog's sound()

        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal;  // Downcasting (explicit and safe)
            myDog.fetch();
        }
```

```
    }
}
```

Output:

```
Dog barks
Dog fetches
```

Full runnable code:

```
public class CastingDemo {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();  // Upcasting (safe and implicit)
        myAnimal.sound();             // Polymorphic call: Dog's sound()

        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal;  // Downcasting (explicit and safe)
            myDog.fetch();
        } else {
            System.out.println("Not a Dog!");
        }

        // Example of unsafe downcast causing ClassCastException
        Animal justAnimal = new Animal();
        if (justAnimal instanceof Dog) {
            Dog dog = (Dog) justAnimal;
            dog.fetch();
        } else {
            System.out.println("justAnimal is not a Dog!");
        }
    }
}

class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

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

    void fetch() {
        System.out.println("Dog fetches the ball");
    }
}
```

5.2.6 Conclusion

Upcasting simplifies your code by allowing subclass objects to be handled as superclass types, enabling polymorphism and flexible APIs. Downcasting lets you access subclass-specific features but requires caution with runtime type checks to avoid exceptions.

Mastering these casts makes your Java programs more powerful and adaptable while maintaining type safety.

5.3 Dynamic Method Dispatch

Dynamic method dispatch is a fundamental mechanism in Java that enables **runtime polymorphism** — the ability of a program to decide at runtime which method implementation to execute, based on the actual object's type rather than the reference type.

5.3.1 What is Dynamic Method Dispatch?

In Java, when you call a method on an object, the method executed is determined by the **actual type of the object** in memory, not by the type of the reference variable that points to it.

This behavior is called **dynamic method dispatch** or **late binding** because the decision about which method to call is deferred until runtime.

5.3.2 How Does Dynamic Method Dispatch Work?

Consider the following class hierarchy:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

class Dog extends Animal {
```

```
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Now, look at this example:

```
public class Test {
    public static void main(String[] args) {
        Animal myAnimal;

        myAnimal = new Cat();
        myAnimal.sound();  // Output: Cat meows

        myAnimal = new Dog();
        myAnimal.sound();  // Output: Dog barks
    }
}
```

Even though the reference variable `myAnimal` is of type `Animal`, the method that gets called is the one overridden in the actual object instance (`Cat` or `Dog`).

Full runnable code:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

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

public class Test {
    public static void main(String[] args) {
        Animal myAnimal;

        myAnimal = new Cat();
        myAnimal.sound();  // Output: Cat meows

        myAnimal = new Dog();
        myAnimal.sound();  // Output: Dog barks
    }
}
```

5.3.3 Why Does This Enable Runtime Polymorphism?

The ability for the method call to **dynamically dispatch** to the correct subclass implementation allows Java programs to be flexible and extensible. You can write code that works with superclass types, but the actual behavior will depend on the subclass objects used at runtime.

For example, a method that processes an array of `Animal` objects can call `sound()` on each one, and each animal will make its own unique sound without the method needing to know the specific subclass.

5.3.4 Behind the Scenes: Virtual Method Table (VMT)

Java uses a **virtual method table** (VMT) internally to keep track of overridden methods for classes. When a method is called, the JVM looks up the actual object's method in the VMT and invokes the correct implementation.

This lookup is done at runtime, enabling dynamic dispatch.

5.3.5 Benefits of Dynamic Method Dispatch

- **Extensibility:** New subclasses can be added without changing existing code.
- **Code Reuse:** Superclass references can refer to any subclass object.
- **Simplified Code:** Polymorphic calls eliminate the need for conditional checks or casting to invoke subclass-specific behavior.

5.3.6 Summary

Dynamic method dispatch is a core Java feature that allows methods to be selected based on an object's runtime type. It enables polymorphism by allowing you to write general, reusable code that adapts its behavior dynamically to different subclasses.

This powerful mechanism is the foundation for many design patterns and best practices in object-oriented design.

5.4 Practical Use Cases for Polymorphism

Polymorphism is not just a theoretical concept—it is the backbone of many real-world software designs that demand flexibility, extensibility, and maintainability. By allowing objects of different classes to be treated uniformly through a common interface or superclass, polymorphism empowers developers to write scalable and adaptable applications. Let's explore some common scenarios where polymorphism shines in practice.

5.4.1 Plugin Architectures

Modern applications often support **plugins or modules** that can be added or removed without modifying the core system. Polymorphism allows the application to interact with plugins through a common interface, while each plugin implements its own behavior.

Example:

Suppose you have a media player that supports different audio formats. Define a common interface:

```
interface AudioPlayer {
    void play();
}

class MP3Player implements AudioPlayer {
    public void play() {
        System.out.println("Playing MP3 file");
    }
}

class WAVPlayer implements AudioPlayer {
    public void play() {
        System.out.println("Playing WAV file");
    }
}
```

The media player can work with any `AudioPlayer` implementation:

```
public class MediaPlayer {
    public void playAudio(AudioPlayer player) {
        player.play();  // Polymorphic call
    }

    public static void main(String[] args) {
        MediaPlayer player = new MediaPlayer();
        player.playAudio(new MP3Player());
        player.playAudio(new WAVPlayer());
    }
}
```

Here, the `MediaPlayer` does not need to know the specific audio format. It simply in-

vokes `play()`, and the right method executes dynamically. New formats can be added by implementing `AudioPlayer` without changing existing code.

Full runnable code:

```
interface AudioPlayer {
    void play();
}

class MP3Player implements AudioPlayer {
    public void play() {
        System.out.println("Playing MP3 file");
    }
}

class WAVPlayer implements AudioPlayer {
    public void play() {
        System.out.println("Playing WAV file");
    }
}

public class MediaPlayer {
    public void playAudio(AudioPlayer player) {
        player.play();  // Polymorphic call
    }

    public static void main(String[] args) {
        MediaPlayer player = new MediaPlayer();
        player.playAudio(new MP3Player());
        player.playAudio(new WAVPlayer());
    }
}
```

5.4.2 Event Handling in GUIs

Graphical user interfaces (GUIs) heavily rely on polymorphism to handle events like button clicks or keyboard inputs. Event listeners implement a common interface, allowing the GUI framework to notify different objects uniformly.

Example:

```
interface ClickListener {
    void onClick();
}

class SaveButtonListener implements ClickListener {
    public void onClick() {
        System.out.println("Saving file...");
    }
}

class CancelButtonListener implements ClickListener {
```

```
    public void onClick() {
        System.out.println("Cancelling operation...");
    }
}
```

The GUI code triggers the appropriate response polymorphically:

```
public class Button {
    private ClickListener listener;

    public void setClickListener(ClickListener listener) {
        this.listener = listener;
    }

    public void click() {
        if (listener != null) listener.onClick();
    }
}
```

This design allows adding new event handlers without modifying the button or GUI code, supporting maintainable and extensible applications.

Full runnable code:

```
interface ClickListener {
    void onClick();
}

class SaveButtonListener implements ClickListener {
    public void onClick() {
        System.out.println("Saving file...");
    }
}

class CancelButtonListener implements ClickListener {
    public void onClick() {
        System.out.println("Cancelling operation...");
    }
}

class Button {
    private ClickListener listener;

    public void setClickListener(ClickListener listener) {
        this.listener = listener;
    }

    public void click() {
        if (listener != null) listener.onClick();
    }
}

public class GUITest {
    public static void main(String[] args) {
        Button saveButton = new Button();
        saveButton.setClickListener(new SaveButtonListener());
```

```
        Button cancelButton = new Button();
        cancelButton.setClickListener(new CancelButtonListener());

        saveButton.click();   // Output: Saving file...
        cancelButton.click(); // Output: Cancelling operation...
    }
}
```

5.4.3 Strategy Pattern for Flexible Algorithms

The **Strategy pattern** is a classic design pattern that leverages polymorphism to select algorithms or behaviors at runtime. You define a family of interchangeable algorithms encapsulated in classes implementing a common interface.

Example:

```
interface SortingStrategy {
    void sort(int[] array);
}

class BubbleSort implements SortingStrategy {
    public void sort(int[] array) {
        // Implementation of bubble sort
        System.out.println("Sorting with Bubble Sort");
    }
}

class QuickSort implements SortingStrategy {
    public void sort(int[] array) {
        // Implementation of quicksort
        System.out.println("Sorting with Quick Sort");
    }
}

class Sorter {
    private SortingStrategy strategy;

    public Sorter(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sortArray(int[] array) {
        strategy.sort(array);
    }
}
```

Switching strategies is seamless:

```
public class StrategyDemo {
    public static void main(String[] args) {
        int[] data = {5, 3, 8, 1};
```

```
        Sorter sorter = new Sorter(new BubbleSort());
        sorter.sortArray(data);

        sorter = new Sorter(new QuickSort());
        sorter.sortArray(data);
    }
}
```

Full runnable code:

```
interface SortingStrategy {
    void sort(int[] array);
}

class BubbleSort implements SortingStrategy {
    public void sort(int[] array) {
        // Simplified bubble sort for demonstration
        System.out.println("Sorting with Bubble Sort");
        for (int i = 0; i < array.length - 1; i++) {
            for (int j = 0; j < array.length - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
}

class QuickSort implements SortingStrategy {
    public void sort(int[] array) {
        System.out.println("Sorting with Quick Sort");
        quickSort(array, 0, array.length - 1);
    }

    private void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pi = partition(array, low, high);
            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
        }
    }

    private int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int i = low -1;
        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                i++;
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        int temp = array[i+1];
        array[i+1] = array[high];
```

```
        array[high] = temp;
        return i + 1;
    }
}

class Sorter {
    private SortingStrategy strategy;

    public Sorter(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sortArray(int[] array) {
        strategy.sort(array);
        System.out.print("Sorted array: ");
        for (int num : array) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}

public class StrategyDemo {
    public static void main(String[] args) {
        int[] data = {5, 3, 8, 1};

        Sorter sorter = new Sorter(new BubbleSort());
        sorter.sortArray(data);

        int[] data2 = {5, 3, 8, 1};  // reset data
        sorter = new Sorter(new QuickSort());
        sorter.sortArray(data2);
    }
}
```

5.4.4 Benefits of Polymorphism in Real-world Designs

- **Extensibility:** Adding new behaviors or types rarely requires changing existing code. Just add new classes that implement the expected interface or inherit from the base class.
- **Maintainability:** Polymorphism promotes clean separation of concerns, making code easier to understand and maintain.
- **Flexibility:** Systems can adapt at runtime by switching implementations dynamically.
- **Reduced Complexity:** Polymorphic interfaces hide complex conditional logic from the client code.

5.4.5 Reflective Thought

Consider a restaurant: a waiter takes orders (calls methods) without knowing how the chef will prepare the dish. Different chefs (subclasses) prepare meals differently, but the waiter interacts uniformly. This analogy highlights how polymorphism decouples caller and implementation, enabling flexible collaboration.

5.4.6 Summary

Practical uses of polymorphism permeate many software systems—from plugin architectures and event handling to strategy selection and beyond. Embracing polymorphism leads to software that is easier to extend, maintain, and scale—qualities essential for modern, robust applications.

By designing with polymorphism in mind, you create a foundation for growth and change, adapting gracefully to new requirements without costly rewrites.

Chapter 6.

Abstraction

1. Abstract Classes
2. Abstract Methods
3. Interfaces and `implements`
4. Differences Between Abstract Classes and Interfaces
5. Functional Interfaces (Intro)

6 Abstraction

6.1 Abstract Classes

In Java programming, **abstraction** is a powerful principle that lets you focus on what an object *does* instead of *how* it does it. One key tool to achieve abstraction is the **abstract class**. Abstract classes provide a way to define common behavior and structure for a group of related classes, while leaving some implementation details to be filled in by subclasses.

6.1.1 What Is an Abstract Class?

An **abstract class** in Java is a class that cannot be instantiated on its own but can be subclassed. It serves as a blueprint for other classes, defining **common characteristics** and **behaviors** that related subclasses share. Abstract classes can contain both:

- **Abstract methods** — method declarations without bodies, which must be implemented by subclasses.
- **Concrete methods** — fully implemented methods that subclasses inherit as-is or override.

Because abstract classes cannot be instantiated directly, you can only create objects from concrete subclasses that provide implementations for all abstract methods.

6.1.2 When and Why Use Abstract Classes?

Abstract classes are appropriate when:

- You want to define a **common base class** with shared fields and methods.
- You expect multiple subclasses to have **common behavior** but differ in some method implementations.
- You want to enforce a **contract** where subclasses must implement specific behaviors.
- You need to provide **default method implementations** alongside abstract declarations.
- You want to take advantage of **code reuse** in the base class.

Abstract classes are more flexible than interfaces (which we'll discuss later) because they allow you to define state (fields) and fully implemented methods.

6.1.3 Syntax and Example

Here is an example of an abstract class that represents a general `Vehicle`:

```
abstract class Vehicle {
    private String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }

    // Concrete method
    public void displayBrand() {
        System.out.println("Brand: " + brand);
    }

    // Abstract method - no implementation here
    public abstract void startEngine();

    // Abstract method
    public abstract void stopEngine();
}
```

In this example:

- `Vehicle` is declared `abstract`.
- It has a private field `brand` and a constructor to initialize it.
- The `displayBrand()` method is concrete and shared by all vehicles.
- `startEngine()` and `stopEngine()` are abstract methods—each subclass must provide its own implementation.

6.1.4 Subclassing an Abstract Class

Any class extending `Vehicle` must implement the abstract methods:

```
class Car extends Vehicle {
    public Car(String brand) {
        super(brand);
    }

    @Override
    public void startEngine() {
        System.out.println("Car engine started");
    }

    @Override
    public void stopEngine() {
        System.out.println("Car engine stopped");
    }
}
```

```
class Motorcycle extends Vehicle {
    public Motorcycle(String brand) {
        super(brand);
    }

    @Override
    public void startEngine() {
        System.out.println("Motorcycle engine started");
    }

    @Override
    public void stopEngine() {
        System.out.println("Motorcycle engine stopped");
    }
}
```

6.1.5 Using the Abstract Class and Subclasses

```
public class TestVehicles {
    public static void main(String[] args) {
        Vehicle car = new Car("Toyota");
        car.displayBrand();
        car.startEngine();
        car.stopEngine();

        Vehicle bike = new Motorcycle("Harley-Davidson");
        bike.displayBrand();
        bike.startEngine();
        bike.stopEngine();
    }
}
```

Output:

```
Brand: Toyota
Car engine started
Car engine stopped
Brand: Harley-Davidson
Motorcycle engine started
Motorcycle engine stopped
```

Full runnable code:

```
abstract class Vehicle {
    private String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }
```

```
    // Concrete method
    public void displayBrand() {
        System.out.println("Brand: " + brand);
    }

    // Abstract methods
    public abstract void startEngine();
    public abstract void stopEngine();
}

class Car extends Vehicle {
    public Car(String brand) {
        super(brand);
    }

    @Override
    public void startEngine() {
        System.out.println("Car engine started");
    }

    @Override
    public void stopEngine() {
        System.out.println("Car engine stopped");
    }
}

class Motorcycle extends Vehicle {
    public Motorcycle(String brand) {
        super(brand);
    }

    @Override
    public void startEngine() {
        System.out.println("Motorcycle engine started");
    }

    @Override
    public void stopEngine() {
        System.out.println("Motorcycle engine stopped");
    }
}

public class TestVehicles {
    public static void main(String[] args) {
        Vehicle car = new Car("Toyota");
        car.displayBrand();
        car.startEngine();
        car.stopEngine();

        Vehicle bike = new Motorcycle("Harley-Davidson");
        bike.displayBrand();
        bike.startEngine();
        bike.stopEngine();
    }
}
```

6.1.6 Important Rules About Abstract Classes

- **Cannot Instantiate:** You cannot create an object of an abstract class directly. For example, `new Vehicle()` is illegal and causes a compile-time error.
- **Must Subclass and Implement Abstract Methods:** Any concrete subclass must provide implementations for all abstract methods, or else it too must be declared abstract.
- **Can Have Constructors:** Abstract classes can have constructors, which are called during the instantiation of subclasses.
- **Can Contain Fields and Methods:** Unlike interfaces, abstract classes can have instance variables and fully implemented methods, allowing code reuse.

6.1.7 Summary

Abstract classes strike a balance between a completely abstract interface and a fully concrete class. They let you:

- Define common behavior and state.
- Enforce method implementation contracts.
- Promote code reuse through shared code.
- Design flexible and extensible class hierarchies.

By leveraging abstract classes in your Java designs, you create clear and maintainable architectures that encourage thoughtful subclassing and consistent behavior across related objects. This forms a cornerstone of effective object-oriented design.

6.2 Abstract Methods

6.2.1 What Are Abstract Methods?

An **abstract method** is a method declared without an implementation (i.e., no method body) inside an abstract class. It defines a **method signature**—the method's name, return type, and parameters—without specifying *how* it works.

The primary purpose of abstract methods is to **enforce a contract**: any concrete subclass of the abstract class *must* provide its own implementation of these methods. This ensures that certain behaviors are guaranteed while allowing subclasses the freedom to decide *how* to implement those behaviors.

6.2.2 Declaring Abstract Methods

Abstract methods are declared using the `abstract` keyword and end with a semicolon rather than a body:

```
abstract class Vehicle {
    // Abstract method - no body
    public abstract void startEngine();

    // Another abstract method
    public abstract void stopEngine();
}
```

In this example, `startEngine()` and `stopEngine()` declare *what* must be done but not *how*. The `Vehicle` class itself provides no implementation for these methods.

6.2.3 Enforcing Subclass Contracts

Any concrete subclass of an abstract class **must override** and implement all abstract methods, or else the subclass itself must be declared abstract.

```
class Car extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Car engine started");
    }

    @Override
    public void stopEngine() {
        System.out.println("Car engine stopped");
    }
}
```

If `Car` omitted one of these methods, the Java compiler would report an error, enforcing that `Car` fulfill the contract established by `Vehicle`.

6.2.4 Abstract Methods and Design Flexibility

Abstract methods allow you to define **common interfaces** for related classes without dictating the details. This promotes:

- **Flexibility:** Subclasses can implement behaviors differently based on their specific needs.
- **Polymorphism:** Client code can work with the abstract class or interface type and rely on subclass implementations without knowing the details.
- **Clearer Architecture:** Abstract methods communicate which behaviors subclasses

must provide, improving code clarity.

6.2.5 Example: Abstract Method Declaration and Implementation

```
abstract class Shape {
    // Abstract method to calculate area
    public abstract double area();

    // Concrete method to display area
    public void display() {
        System.out.println("Area is: " + area());
    }
}

class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    private double width, height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}
```

Here, the abstract method `area()` forces every `Shape` subclass to provide its own area calculation. Meanwhile, the `display()` method in the abstract class can call `area()` polymorphically.

Full runnable code:

```
abstract class Shape {
    // Abstract method to calculate area
    public abstract double area();

    // Concrete method to display area
```

```
    public void display() {
        System.out.println("Area is: " + area());
    }
}

class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    private double width, height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}

public class TestShapes {
    public static void main(String[] args) {
        Shape circle = new Circle(5);
        circle.display();  // Area is: 78.53981633974483

        Shape rectangle = new Rectangle(4, 7);
        rectangle.display();  // Area is: 28.0
    }
}
```

6.2.6 Impact on Code Reuse

While abstract methods require subclasses to implement certain behaviors, the abstract class can still provide **concrete methods** that use those abstract methods to offer reusable functionality.

For example, `Shape`'s `display()` method relies on the `area()` abstract method but provides a shared implementation to display results. This balance encourages **code reuse** and **design consistency**.

6.2.7 Summary

Abstract methods are a fundamental part of Java's abstraction mechanism. By declaring abstract methods:

- You define a **clear contract** for subclasses.
- You **enforce consistency** across related classes.
- You **enable flexible and extensible designs** that support polymorphism.
- You create opportunities for **code reuse** by combining abstract and concrete methods in abstract classes.

Understanding abstract methods helps you build robust, maintainable Java applications that leverage the full power of object-oriented design.

6.3 Interfaces and `implements`

6.3.1 What Is an Interface?

In Java, an **interface** is a special kind of reference type that defines a **contract** for what a class can do, without prescribing *how* it does it. Interfaces specify **method signatures** that implementing classes must provide, but they do not hold state (fields) or method implementations—at least traditionally.

Interfaces are a core part of Java's approach to **abstraction**, allowing different classes to share common behavior without forcing a strict class hierarchy. They promote **loose coupling** and support multiple inheritance of behavior, something classes alone cannot do.

6.3.2 How Interfaces Differ from Classes

While both classes and interfaces define types, they differ in several important ways:

- **No instance fields:** Interfaces cannot have instance variables. They can only have constants (`public static final` fields).
- **No constructors:** You cannot instantiate an interface or create a constructor for one.
- **Method declarations:** Before Java 8, interfaces could only declare abstract methods (methods without implementation). From Java 8 onward, interfaces can also have **default** and **static** methods with implementations.
- **Multiple inheritance:** A class can implement multiple interfaces but can extend only one class (abstract or concrete).
- **Purpose:** Interfaces represent *capabilities* or *roles* a class can play, rather than concrete objects.

6.3.3 Syntax: Implementing Interfaces

To use an interface, a class declares that it **implements** the interface and provides concrete implementations for all its abstract methods.

Here is the basic syntax:

```
interface Drivable {
    void drive();
}

class Car implements Drivable {
    @Override
    public void drive() {
        System.out.println("Car is driving");
    }
}
```

`Car` promises to fulfill the contract of `Drivable` by implementing the `drive()` method.

6.3.4 Implementing Multiple Interfaces

One major advantage of interfaces is that a single class can implement **multiple interfaces**, gaining the behaviors of several types simultaneously. This is Java's way to achieve multiple inheritance of type.

Example:

```
interface Drivable {
    void drive();
}

interface Electric {
    void chargeBattery();
}

class Tesla implements Drivable, Electric {
    @Override
    public void drive() {
        System.out.println("Tesla is driving silently");
    }

    @Override
    public void chargeBattery() {
        System.out.println("Tesla battery is charging");
    }
}
```

Here, `Tesla` acts as both a `Drivable` and an `Electric` vehicle, implementing methods from both interfaces.

6.3.5 Default and Static Methods in Interfaces (Since Java 8)

Prior to Java 8, interfaces could only declare abstract methods. This limitation made evolving interfaces challenging, as adding new methods would break existing implementations.

Java 8 introduced two important features to address this:

- **Default methods:** These are methods within an interface that provide a **default implementation**. Classes that implement the interface can inherit these methods without overriding them, but they can also override if needed.

```
interface Printable {
    default void print() {
        System.out.println("Printing from Printable interface");
    }
}

class Document implements Printable {
    // Inherits default print() or can override
}
```

- **Static methods:** Interfaces can also have **static methods** that belong to the interface itself, not to instances.

```
interface MathOperations {
    static int add(int a, int b) {
        return a + b;
    }
}
```

These methods can be called via `MathOperations.add(5, 3);`.

Full runnable code:

```
interface Drivable {
    void drive();
}

interface Electric {
    void chargeBattery();
}

interface Printable {
    default void print() {
        System.out.println("Printing from Printable interface");
    }
}

interface MathOperations {
    static int add(int a, int b) {
        return a + b;
    }
}

class Tesla implements Drivable, Electric, Printable {
```

```
    @Override
    public void drive() {
        System.out.println("Tesla is driving silently");
    }

    @Override
    public void chargeBattery() {
        System.out.println("Tesla battery is charging");
    }

    // Inherits default print() method from Printable
}

public class TestInterfaces {
    public static void main(String[] args) {
        Tesla myTesla = new Tesla();
        myTesla.drive();
        myTesla.chargeBattery();
        myTesla.print();  // default method from Printable

        int sum = MathOperations.add(10, 20);  // static method from interface
        System.out.println("Sum: " + sum);
    }
}
```

6.3.6 Why Use Interfaces?

- **Multiple inheritance of type:** Unlike classes, Java allows implementing multiple interfaces, enabling flexible design.
- **Decoupling:** Interfaces separate *what* something does from *how* it does it, supporting loosely coupled architectures.
- **Design by contract:** They define clear expectations for behavior, improving maintainability.
- **Polymorphism:** Interfaces allow objects of different classes to be treated uniformly if they share the same interface.

6.3.7 Example: Interface in Action

```
interface Animal {
    void makeSound();
}

interface Runnable {
    void run();
}
```

```
class Dog implements Animal, Runnable {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }

    @Override
    public void run() {
        System.out.println("Dog is running");
    }
}
```

Client code can interact with `Dog` objects through either `Animal` or `Runnable` references:

```
Animal animal = new Dog();
animal.makeSound();

Runnable runner = new Dog();
runner.run();
```

This flexibility is one of the core strengths of interfaces in Java's OOP design.

6.3.8 Summary

Interfaces are a powerful abstraction tool in Java, enabling classes to promise certain behaviors without restricting class inheritance. By using the `implements` keyword, classes commit to providing concrete implementations for interface methods, fostering polymorphism and decoupled designs.

With the introduction of default and static methods, interfaces have become even more flexible, allowing method evolution without breaking existing code. Mastering interfaces is essential for writing scalable, maintainable Java applications that follow modern object-oriented design principles.

6.4 Differences Between Abstract Classes and Interfaces

In Java, **abstract classes** and **interfaces** are both tools for abstraction that allow you to define contracts and share behavior among related types. However, they serve different purposes, have distinct capabilities, and fit different design scenarios. Understanding their differences is crucial to making informed design decisions.

6.4.1 Key Differences

Feature	Abstract Class	Interface
Inheritance Type	Single inheritance (a class can extend only one abstract class)	Multiple inheritance (a class can implement multiple interfaces)
Method Types	Can have abstract and concrete methods	Before Java 8: only abstract methods; since Java 8: can have default and static methods
Fields	Can have instance variables (state)	Only `public static final` constants
Constructors	Can have constructors	Cannot have constructors
Access Modifiers	Methods and fields can have any access level	Methods are implicitly `public`; fields are `public static final`
When to Use	When classes share common base behavior and state	To define roles or capabilities that unrelated classes can implement
Flexibility	Less flexible due to single inheritance	More flexible, allows multiple behaviors
Backward Compatibility	Adding methods can break subclasses unless they are default methods	Default methods since Java 8 allow evolving interfaces without breaking implementations

6.4.2 Usage Scenarios

- **Abstract Classes:** Use abstract classes when you want to provide a **common base implementation** or state (fields) for a group of closely related classes. For example, if you have a family of shapes (`Shape` abstract class) sharing common fields like `color` or methods like `move()`, an abstract class is appropriate. Abstract classes help you avoid code duplication by allowing concrete shared code.
- **Interfaces:** Interfaces define **capabilities or roles** that can be added to classes from different inheritance trees. For instance, `Serializable` or `Comparable` are interfaces describing behavior unrelated to a class's main hierarchy. Interfaces excel at enabling **multiple inheritance of behavior**, which abstract classes cannot provide.

6.4.3 Single Inheritance vs Multiple Interfaces

Java allows a class to extend only one class (abstract or concrete) due to the complexity and ambiguity that can arise from multiple inheritance. However, a class can implement

multiple interfaces, enabling it to assume multiple roles or behaviors.

This is a crucial difference: **interfaces provide flexibility**, while **abstract classes provide structure and shared code**.

6.4.4 When to Prefer Abstract Classes

- You want to share code among several closely related classes.
- You need to define non-static or non-final fields (state).
- You want to define constructors for initializing shared fields.
- You expect your base type to evolve with more shared code.

6.4.5 When to Prefer Interfaces

- You want to specify a contract that many unrelated classes can implement.
- You need to support multiple inheritance of type.
- Your design emphasizes capability over hierarchy.
- You want to ensure loose coupling and flexible code evolution.

6.4.6 Decision Checklist

Question	Choose Abstract Class	Choose Interface
Do you need to share common code (method bodies)?		(prior to Java 8, limited support)
Do you need to define instance fields?		
Should the type support multiple inheritance?		
Are you defining a capability or role?		
Do you need constructors in the base type?		
Is backward compatibility critical?	Use interfaces with default methods	Use interfaces with default methods
Are the classes closely related in hierarchy?		

6.4.7 Summary

While both abstract classes and interfaces provide abstraction, they are designed for different situations. Abstract classes are best when there is a clear "is-a" relationship with shared implementation and state. Interfaces are ideal when defining roles or capabilities that can crosscut various unrelated classes.

Knowing when and how to use each effectively enables you to design robust, flexible, and maintainable Java applications. In practice, a combination of both often yields the best results—using abstract classes to share code and interfaces to define contracts.

Understanding these differences is a foundational skill for mastering Java's object-oriented design principles.

6.5 Functional Interfaces (Intro)

With the introduction of **Java 8**, the concept of **functional interfaces** became a cornerstone for supporting **functional programming** features in Java. Functional interfaces enable a new, concise way of writing code using **lambda expressions**, which are anonymous functions that can be passed around like objects.

6.5.1 What Are Functional Interfaces?

A **functional interface** is an interface that contains exactly **one abstract method**. This single-method requirement allows instances of the interface to be created with lambda expressions or method references, dramatically simplifying code for cases where you would otherwise create an anonymous class.

6.5.2 The `@FunctionalInterface` Annotation

While any interface with one abstract method can be considered functional, Java provides a special annotation, `@FunctionalInterface`, to **explicitly declare** your intention. This annotation helps:

- **Clarify intent** for readers and tools.
- **Enable compile-time checking** to ensure the interface remains functional (i.e., it cannot accidentally gain a second abstract method).

Example:

```java
@FunctionalInterface
public interface MyFunction {
    void apply();
}
```

If you add another abstract method to `MyFunction`, the compiler will flag an error.

6.5.3 Common Functional Interfaces in Java

Java's standard library includes many widely used functional interfaces, making it easy to adopt functional programming idioms:

- **`Runnable`** Has a single method `void run()`. Often used for tasks executed by threads or background jobs.
- **`Callable<V>`** Defines `V call() throws Exception`, allowing tasks that return results and can throw exceptions.
- **`Comparator<T>`** Defines `int compare(T o1, T o2)`, used to compare two objects for sorting and ordering.

These interfaces became even more useful with lambda expressions, eliminating boilerplate anonymous classes.

6.5.4 How Functional Interfaces Enable Lambda Expressions

Lambda expressions provide a concise syntax to implement functional interfaces. Instead of writing an anonymous class, you can write a short, readable expression representing the method implementation.

For example, with `Runnable`:

Without Lambda:

```java
Runnable task = new Runnable() {
    @Override
    public void run() {
        System.out.println("Task running");
    }
};
```

With Lambda:

```java
Runnable task = () -> System.out.println("Task running");
```

Both create a `Runnable` instance, but the lambda is much cleaner.

6.5.5 Simple Example Using Lambda and Functional Interface

Suppose we define a custom functional interface:

```
@FunctionalInterface
interface Greeting {
    void sayHello(String name);
}
```

Using a lambda, you can implement it like this:

```
Greeting greet = (name) -> System.out.println("Hello, " + name + "!");
greet.sayHello("Alice");
```

This will output:

```
Hello, Alice!
```

Full runnable code:

```
@FunctionalInterface
interface Greeting {
    void sayHello(String name);
}

public class LambdaDemo {
    public static void main(String[] args) {
        Greeting greet = (name) -> System.out.println("Hello, " + name + "!");
        greet.sayHello("Alice");
    }
}
```

6.5.6 Summary

Functional interfaces are a key Java 8+ feature that allow developers to write more expressive and compact code using lambda expressions. By defining interfaces with a single abstract method and optionally marking them with `@FunctionalInterface`, Java enables powerful functional programming constructs while maintaining strong type safety and readability.

Mastering functional interfaces unlocks new design possibilities and modern coding styles in Java, which will be explored further in later chapters.

Chapter 7.

Composition and Aggregation

1. “Has-a” Relationships
2. Composition vs Inheritance
3. Designing for Reuse
4. Example: Modeling a Car with Components

7 Composition and Aggregation

7.1 "Has-a" Relationships

One of the foundational principles in Object-Oriented Design (OOD) is understanding the **relationships between objects**. Two primary types of relationships often discussed are **"is-a"** and **"has-a"**. This section focuses on the "has-a" relationship, which plays a crucial role in designing flexible, maintainable systems by favoring **composition** over inheritance.

7.1.1 What Is a Has-a Relationship?

A **"has-a" relationship** describes a situation where one object **contains** or **owns** another object as part of its state or behavior. It models real-world entities by capturing the fact that some objects are made up of or possess other objects. This kind of relationship is also known as **composition** or **aggregation**, depending on the strength of the relationship.

For example:

- A **Car has an Engine**. The engine is a component of the car, and the car cannot function without it.
- A **House has Rooms**. Rooms exist as part of the house but may also have some independence.

These relationships reflect **part-whole hierarchies** where the whole object "has" one or more parts.

7.1.2 Contrasting Has-a and Is-a Relationships

Understanding the difference between **"has-a" (composition)** and **"is-a" (inheritance)** relationships is essential for sound design:

- **"Is-a" (Inheritance):** When an object is a specialized type of another, it uses inheritance. For example, a `SportsCar` **is a** `Car`, so it inherits properties and behaviors from `Car`. The relationship indicates that the subclass **is** a kind of the superclass.
- **"Has-a" (Composition):** When an object contains or is composed of other objects, it uses composition. For example, a `Car` **has an** `Engine`. The car is not a type of engine but contains one.

In summary:

- Use **inheritance** when creating a clear hierarchical relationship based on "is-a".
- Use **composition** to model objects that **own or contain** other objects, representing

"has-a".

7.1.3 Why Favor Has-a Relationships?

While inheritance is powerful, relying too heavily on it can lead to rigid, tightly coupled designs. In contrast, composition via "has-a" relationships offers greater **flexibility**, **modularity**, and **reuse**. Here's why:

- **Encapsulation:** Each component object encapsulates its own behavior and data. Changes in one component typically do not affect others.
- **Flexibility:** You can easily replace or modify parts without altering the whole. For example, changing the type of engine in a car object is straightforward if the engine is a separate object.
- **Avoids Inheritance Pitfalls:** Overusing inheritance can create deep, fragile class hierarchies that are hard to maintain and extend.
- **Realistic Modeling:** Most real-world objects are naturally described as being composed of parts, making "has-a" a natural fit.

7.1.4 Examples of Has-a Relationships

Here are some simple Java examples illustrating "has-a" relationships:

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

class Car {
    private Engine engine; // Car "has-a" Engine

    public Car() {
        this.engine = new Engine();
    }

    public void startCar() {
        engine.start(); // Delegates behavior to the engine
        System.out.println("Car is ready to go!");
    }
}
```

In this example, the `Car` class **has an** `Engine` object. The car delegates engine-related tasks to the engine object rather than inheriting from it.

Another example:

```
class Room {
    private String name;

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

    public String getName() {
        return name;
    }
}

class House {
    private Room[] rooms; // House "has-a" collection of Rooms

    public House(Room[] rooms) {
        this.rooms = rooms;
    }

    public void listRooms() {
        for (Room room : rooms) {
            System.out.println("Room: " + room.getName());
        }
    }
}
```

The `House` object contains an array of `Room` objects, showing a "has-a" relationship with multiple components.

Full runnable code:

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

class Car {
    private Engine engine; // Car "has-a" Engine

    public Car() {
        this.engine = new Engine();
    }

    public void startCar() {
        engine.start(); // Delegates behavior to the engine
        System.out.println("Car is ready to go!");
    }
}

class Room {
    private String name;

    public Room(String name) {
        this.name = name;
    }
```

```
    public String getName() {
        return name;
    }
}

class House {
    private Room[] rooms; // House "has-a" collection of Rooms

    public House(Room[] rooms) {
        this.rooms = rooms;
    }

    public void listRooms() {
        for (Room room : rooms) {
            System.out.println("Room: " + room.getName());
        }
    }
}

public class HasARelationshipDemo {
    public static void main(String[] args) {
        Car car = new Car();
        car.startCar();

        Room[] rooms = { new Room("Living Room"), new Room("Bedroom"), new Room("Kitchen") };
        House house = new House(rooms);
        house.listRooms();
    }
}
```

7.1.5 Reflecting on Has-a Relationships

When designing software, try to think in terms of how objects are composed of other objects rather than forcing everything into an inheritance hierarchy. Ask yourself:

- Does this object logically contain or own another?
- Is this a whole-part relationship rather than a kind-of relationship?

By focusing on "has-a" relationships, your designs will tend to be more adaptable to change, easier to test, and clearer to understand.

7.1.6 Summary

- **"Has-a" relationships** represent objects containing other objects as parts or components.
- They contrast with "is-a" relationships, which represent inheritance hierarchies.
- Favoring composition over inheritance leads to more modular and flexible designs.

- Real-world modeling often naturally follows “has-a” relationships.

Mastering these concepts will help you design better, more maintainable object-oriented systems as you progress through this book.

7.2 Composition vs Inheritance

In object-oriented design, two fundamental ways to model relationships between classes are **inheritance** and **composition**. Both techniques allow one class to reuse or extend the behavior of another, but they do so in very different ways, each with distinct advantages and trade-offs. Understanding when to use each is crucial to designing flexible, maintainable, and reusable Java software.

7.2.1 What Is Inheritance?

Inheritance is a mechanism where one class (called the **subclass** or **child class**) **inherits** fields and methods from another class (the **superclass** or **parent class**). It represents an **“is-a”** relationship: the subclass **is a specialized type of** the superclass.

For example, consider a class hierarchy where `Car` extends `Vehicle`:

```
class Vehicle {
    void start() {
        System.out.println("Vehicle starting");
    }
}

class Car extends Vehicle {
    void openTrunk() {
        System.out.println("Opening trunk");
    }
}
```

Here, `Car` inherits the `start()` method from `Vehicle` and adds its own behavior.

7.2.2 What Is Composition?

Composition models a **“has-a”** relationship where one class contains an instance of another as a **component**. Instead of inheriting behavior, the containing class **delegates** tasks to the component objects it holds.

Using the same example:

```
class Engine {
    void start() {
        System.out.println("Engine starting");
    }
}

class Car {
    private Engine engine;

    public Car() {
        engine = new Engine();
    }

    void start() {
        engine.start();
        System.out.println("Car is ready to go");
    }
}
```

Here, `Car` **has an** `Engine`, and it uses that engine to perform the start operation.

Full runnable code:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Inheritance example:");
        CarInheritance car1 = new CarInheritance();
        car1.start();         // Inherited from Vehicle
        car1.openTrunk();     // Defined in CarInheritance

        System.out.println("\nComposition example:");
        CarComposition car2 = new CarComposition();
        car2.start();         // Delegates to Engine
    }
}

// Inheritance example
class Vehicle {
    void start() {
        System.out.println("Vehicle starting");
    }
}

class CarInheritance extends Vehicle {
    void openTrunk() {
        System.out.println("Opening trunk");
    }
}

// Composition example
class Engine {
    void start() {
        System.out.println("Engine starting");
    }
}

class CarComposition {
```

```
    private Engine engine;

    public CarComposition() {
        engine = new Engine();
    }

    void start() {
        engine.start();
        System.out.println("Car is ready to go");
    }
}
```

7.2.3 Comparing Composition and Inheritance

Aspect	Inheritance	Composition
Relationship	"Is-a" — subclass is a specialized type of superclass	"Has-a" — class contains component objects
Coupling	Tight coupling to superclass implementation	Loose coupling; components are replaceable
Flexibility	Less flexible; changes in superclass affect subclasses	More flexible; components can be changed or replaced independently
Reuse	Reuse by inheriting behavior and overriding methods	Reuse by delegating behavior to composed objects
Maintenance	Can lead to fragile base classes if superclass changes	Easier to maintain as components are independent
Multiple inheritance	Java does not support multiple class inheritance	Allows mixing different components freely

7.2.4 Pros and Cons of Inheritance

Pros:

- Easy to express natural hierarchies.
- Code reuse is straightforward through inheritance.
- Polymorphism is naturally supported.

Cons:

- Tight coupling to the superclass can cause cascading changes.
- Inheritance hierarchies can become deep and complex, leading to brittle designs.
- Subclasses inherit all superclass behavior, even if not all is relevant.
- Java supports only single inheritance for classes, limiting flexibility.

7.2.5 Pros and Cons of Composition

Pros:

- Promotes loose coupling and encapsulation.
- Components can be swapped or extended independently.
- Encourages building small, focused classes.
- Supports multiple behaviors by composing multiple objects.

Cons:

- Can require more boilerplate delegation code.
- Designing component interfaces requires upfront planning.

7.2.6 Modeling the Same Scenario: Inheritance vs Composition

Suppose you want to model different types of payment processing.

Inheritance example:

```
class PaymentProcessor {
    void processPayment(double amount) {
        System.out.println("Processing payment: " + amount);
    }
}

class CreditCardProcessor extends PaymentProcessor {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing credit card payment: " + amount);
    }
}
```

Composition example:

```
interface PaymentMethod {
    void pay(double amount);
}

class CreditCard implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Paying with credit card: " + amount);
    }
}

class PaymentProcessor {
    private PaymentMethod paymentMethod;

    public PaymentProcessor(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }
```

```
    void processPayment(double amount) {
        paymentMethod.pay(amount);
    }
}
```

In the composition example, `PaymentProcessor` can work with any `PaymentMethod` implementation, making it flexible and extensible without changing the processor class.

7.2.7 Favor Composition Over Inheritance: The Guiding Principle

Modern object-oriented design strongly advocates **"favor composition over inheritance"**. The rationale is:

- **Inheritance creates strong coupling** between subclass and superclass, making code fragile and difficult to evolve.
- **Composition promotes flexibility** by allowing objects to be composed dynamically at runtime or easily swapped.
- **Composition supports better encapsulation**, hiding component implementations behind well-defined interfaces.
- It helps avoid the pitfalls of deep and complex inheritance trees.

This does not mean inheritance is bad—it's still valuable for clear "is-a" relationships and polymorphism. However, composition should be the default choice when designing relationships between classes.

7.2.8 Summary

- **Inheritance** expresses an "is-a" relationship with code reuse via subclassing but can lead to tight coupling and fragile designs.
- **Composition** expresses "has-a" relationships by assembling objects with distinct responsibilities, offering greater flexibility and maintainability.
- Favor composition for most cases, using inheritance carefully when natural hierarchies exist.
- Designing with composition leads to more adaptable and reusable Java programs.

Understanding these concepts deeply equips you to make better design choices as you build complex, scalable object-oriented systems.

7.3 Designing for Reuse

In software development, **code reuse** is a key goal—it saves time, reduces errors, and promotes consistency across projects. Designing classes and components for reuse requires thoughtful planning, especially when using **composition**, which is the preferred way to build flexible and maintainable systems.

7.3.1 Why Composition Enhances Reuse

Unlike inheritance, which tightly couples subclasses to their parent classes, **composition enables reuse by assembling independent, well-defined components**. This modular approach makes it easier to reuse and extend parts of a system without affecting unrelated areas.

For example, a logging component designed as a standalone class can be reused across many different parts of an application or even across different projects simply by including and configuring it, rather than forcing all clients to inherit from a base class that provides logging.

7.3.2 Strategies for Designing Reusable Components

Single Responsibility Principle (SRP)

Design classes to have **one clear responsibility**. When a class does one thing well, it becomes easier to understand, test, and reuse in different contexts.

```
class Logger {
    void log(String message) {
        System.out.println("LOG: " + message);
    }
}
```

This `Logger` class is simple and focused—perfect for reuse anywhere logging is needed.

Programming to Interfaces

Define components using **interfaces** rather than concrete classes. Interfaces specify *what* a component does without prescribing *how* it does it. This abstraction enables multiple implementations that can be swapped easily.

```
interface PaymentMethod {
    void pay(double amount);
}

class CreditCardPayment implements PaymentMethod {
    public void pay(double amount) {
```

```
        System.out.println("Paying $" + amount + " by credit card");
    }
}
```

By depending on `PaymentMethod` rather than `CreditCardPayment` directly, your system becomes more flexible and reusable.

Favor Composition and Delegation

Build complex behavior by composing small, reusable components. Use **delegation** to forward requests to component objects, avoiding the pitfalls of inheritance.

Example: A `Car` class composed of an `Engine` and `Transmission`:

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

class Transmission {
    void shiftGear(int gear) { System.out.println("Shifted to gear " + gear); }
}

class Car {
    private Engine engine = new Engine();
    private Transmission transmission = new Transmission();

    void drive() {
        engine.start();
        transmission.shiftGear(1);
        System.out.println("Car is driving");
    }
}
```

Each component is reusable on its own, and `Car` simply composes these parts.

Use Immutability Where Possible

Immutable objects (objects whose state cannot change after construction) are naturally reusable because they are thread-safe and side-effect free.

7.3.3 Examples of Reusable Components

- **Utility Classes:** Common helper classes for string manipulation, file operations, or math calculations.
- **Service Classes:** Authentication, caching, or notification services that encapsulate functionality usable in many contexts.
- **UI Components:** Reusable buttons, dialogs, or form elements in GUI applications.

By designing these components with clear interfaces and minimal dependencies, you enable reuse across many parts of your application or even across projects.

7.3.4 Design Patterns That Emphasize Composition for Reuse

Several well-known design patterns explicitly promote composition to maximize reuse:

- **Decorator Pattern:** Adds responsibilities to objects dynamically by wrapping them with decorator classes rather than subclassing. This promotes flexible, runtime composition.

```
interface Coffee {
    double cost();
}

class SimpleCoffee implements Coffee {
    public double cost() { return 2.0; }
}

class MilkDecorator implements Coffee {
    private Coffee coffee;
    public MilkDecorator(Coffee coffee) { this.coffee = coffee; }
    public double cost() { return coffee.cost() + 0.5; }
}
```

- **Strategy Pattern:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The client composes with different strategy objects to vary behavior.

```
interface SortingStrategy {
    void sort(int[] array);
}

class BubbleSort implements SortingStrategy {
    public void sort(int[] array) { /* bubble sort implementation */ }
}

class QuickSort implements SortingStrategy {
    public void sort(int[] array) { /* quick sort implementation */ }
}

class Sorter {
    private SortingStrategy strategy;
    public Sorter(SortingStrategy strategy) { this.strategy = strategy; }
    public void sortArray(int[] array) { strategy.sort(array); }
}
```

Full runnable code:

```
interface SortingStrategy {
    void sort(int[] array);
}

class BubbleSort implements SortingStrategy {
    public void sort(int[] array) {
        // Simple bubble sort implementation
        int n = array.length;
        for(int i = 0; i < n-1; i++) {
```

```
            for(int j = 0; j < n-i-1; j++) {
                if(array[j] > array[j+1]) {
                    int temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }
            }
        }
    }
}

class QuickSort implements SortingStrategy {
    public void sort(int[] array) {
        quickSort(array, 0, array.length - 1);
    }

    private void quickSort(int[] arr, int low, int high) {
        if(low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi -1);
            quickSort(arr, pi + 1, high);
        }
    }

    private int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);
        for(int j = low; j < high; j++) {
            if(arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }
}

class Sorter {
    private SortingStrategy strategy;

    public Sorter(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sortArray(int[] array) {
        strategy.sort(array);
    }
}

public class MainSort {
    public static void main(String[] args) {
        int[] data = {5, 3, 8, 1, 2};
```

```
        Sorter bubbleSorter = new Sorter(new BubbleSort());
        bubbleSorter.sortArray(data);
        System.out.print("BubbleSorted array: ");
        for(int num : data) System.out.print(num + " ");
        System.out.println();

        data = new int[]{5, 3, 8, 1, 2}; // reset array

        Sorter quickSorter = new Sorter(new QuickSort());
        quickSorter.sortArray(data);
        System.out.print("QuickSorted array: ");
        for(int num : data) System.out.print(num + " ");
        System.out.println();
    }
}
```

7.3.5 Summary

Designing for reuse with composition involves:

- Creating **small, focused components** with clear responsibilities.
- Using **interfaces** to define contracts and enable flexible implementations.
- **Composing** behavior by delegating to these components rather than inheriting from monolithic base classes.
- Applying design patterns like **Decorator** and **Strategy** to structure reusable and extensible systems.

By following these strategies, you build software that is easier to maintain, extend, and adapt—key qualities for modern, scalable Java applications.

7.4 Example: Modeling a Car with Components

In this section, we will explore a practical, real-world example of **composition** by modeling a `Car` composed of several smaller components such as an `Engine`, `Wheel`, and `Transmission`. This example will demonstrate how objects can be assembled from other objects to build flexible and maintainable designs.

7.4.1 Why Use Composition to Model a Car?

A car is not just one big object; it is made up of many parts working together. Instead of trying to cram all functionality into one monolithic class, composition allows us to break down the problem into manageable, reusable pieces.

By modeling a car as a **has-a** relationship—i.e., a car *has an* engine, *has wheels*, *has a transmission*—we reflect the real-world relationships clearly and create components that can be developed and tested independently.

7.4.2 Defining Component Classes

First, let's define each component class with simple functionality.

```
// Engine.java
public class Engine {
    private boolean running;

    public void start() {
        running = true;
        System.out.println("Engine started.");
    }

    public void stop() {
        running = false;
        System.out.println("Engine stopped.");
    }

    public boolean isRunning() {
        return running;
    }
}
```

Wheel.java

```
public class Wheel {
    private String position;

    public Wheel(String position) {
        this.position = position;
    }

    public void rotate() {
        System.out.println(position + " wheel is rotating.");
    }
}
```

Transmission.java

```
public class Transmission {
    private int currentGear = 0;

    public void shiftGear(int gear) {
        currentGear = gear;
        System.out.println("Shifted to gear " + currentGear);
    }
```

```
    public int getCurrentGear() {
        return currentGear;
    }
}
```

7.4.3 Defining the Car Class with Composition

The `Car` class **has** an `Engine`, multiple `Wheel` objects, and a `Transmission`. We compose these inside `Car` as fields and use their behavior to implement the car's functionality.

Car.java

```
public class Car {
    private Engine engine;
    private Wheel[] wheels;
    private Transmission transmission;

    public Car() {
        // Initialize components
        engine = new Engine();
        wheels = new Wheel[] {
            new Wheel("Front Left"),
            new Wheel("Front Right"),
            new Wheel("Rear Left"),
            new Wheel("Rear Right")
        };
        transmission = new Transmission();
    }

    public void startCar() {
        engine.start();
        transmission.shiftGear(1);
        for (Wheel wheel : wheels) {
            wheel.rotate();
        }
        System.out.println("Car is started and ready to drive.");
    }

    public void stopCar() {
        transmission.shiftGear(0);
        engine.stop();
        System.out.println("Car has stopped.");
    }

    // Additional behaviors can be added here...
}
```

7.4.4 Running the Example

Let's create a `Main` class to run this car simulation.

```
// Main.java
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();

        myCar.startCar();
        // Simulate driving...

        myCar.stopCar();
    }
}
```

Expected Console Output:

```
Engine started.
Shifted to gear 1
Front Left wheel is rotating.
Front Right wheel is rotating.
Rear Left wheel is rotating.
Rear Right wheel is rotating.
Car is started and ready to drive.
Shifted to gear 0
Engine stopped.
Car has stopped.
```

Full runnable code:

```
// Engine.java
class Engine {
    private boolean running;

    public void start() {
        running = true;
        System.out.println("Engine started.");
    }

    public void stop() {
        running = false;
        System.out.println("Engine stopped.");
    }

    public boolean isRunning() {
        return running;
    }
}

// Wheel.java
class Wheel {
    private String position;
```

```
    public Wheel(String position) {
        this.position = position;
    }

    public void rotate() {
        System.out.println(position + " wheel is rotating.");
    }
}

// Transmission.java
class Transmission {
    private int currentGear = 0;

    public void shiftGear(int gear) {
        currentGear = gear;
        System.out.println("Shifted to gear " + currentGear);
    }

    public int getCurrentGear() {
        return currentGear;
    }
}

// Car.java
class Car {
    private Engine engine;
    private Wheel[] wheels;
    private Transmission transmission;

    public Car() {
        engine = new Engine();
        wheels = new Wheel[] {
            new Wheel("Front Left"),
            new Wheel("Front Right"),
            new Wheel("Rear Left"),
            new Wheel("Rear Right")
        };
        transmission = new Transmission();
    }

    public void startCar() {
        engine.start();
        transmission.shiftGear(1);
        for (Wheel wheel : wheels) {
            wheel.rotate();
        }
        System.out.println("Car is started and ready to drive.");
    }

    public void stopCar() {
        transmission.shiftGear(0);
        engine.stop();
        System.out.println("Car has stopped.");
    }
}

// Main.java
public class Main {
```

```
    public static void main(String[] args) {
        Car myCar = new Car();

        myCar.startCar();
        // Simulate driving...

        myCar.stopCar();
    }
}
```

7.4.5 Design Reflection and Advantages of Composition

1. **Modularity:** Each component (`Engine`, `Wheel`, `Transmission`) encapsulates its own behavior and state. This modularity makes it easier to maintain and enhance individual parts without affecting the whole car.
2. **Reusability:** The same `Engine` or `Transmission` classes could be reused in other vehicle types like trucks or motorcycles with little or no modification.
3. **Flexibility:** If we want to model different types of wheels or engines (e.g., electric vs combustion), we can subclass or swap components without changing the `Car` class's overall structure.
4. **Clear Responsibilities:** Each class has a clear responsibility—`Engine` starts and stops, `Wheel` rotates, `Transmission` shifts gears—promoting better separation of concerns.
5. **Testability:** You can unit test each component independently before integrating them into the `Car`. This reduces bugs and improves reliability.

7.4.6 Summary

This example showcases how **composition models real-world "has-a" relationships** effectively in Java. Instead of one large class trying to do everything, breaking down a complex object like a `Car` into components leads to a cleaner, more manageable design.

Composition encourages building systems by assembling smaller parts with defined roles—making code easier to understand, extend, and maintain. This principle is central to modern object-oriented design and is preferred over inheritance when modeling "part-of" relationships.

Chapter 8.

Inner Classes and Anonymous Classes

1. Member Classes
2. Static Nested Classes
3. Local Classes
4. Anonymous Classes and Functional Use Cases

8 Inner Classes and Anonymous Classes

8.1 Member Classes

In Java, a **member class**—often called an **inner class**—is a class defined *within* another class. Unlike top-level classes, member classes have a special relationship with their outer class, allowing them to access its members directly, even the private ones. This tight coupling is useful for logically grouping classes that are only relevant within the context of their outer class, enhancing encapsulation and code organization.

8.1.1 What is a Member (Inner) Class?

A member class is declared inside the body of another class without the `static` keyword. This means each instance of the inner class is implicitly associated with an instance of the outer class. The inner class can directly access all members (fields and methods) of the outer class, including those marked `private`.

This relationship makes member classes ideal for modeling "part-of" or helper objects that belong specifically to their outer class.

8.1.2 Syntax of Member Classes

Here is the basic syntax illustrating a member class inside an outer class:

```
public class OuterClass {
    private String outerField = "Outer field value";

    // Member (inner) class
    public class InnerClass {
        public void display() {
            // Accessing outer class's private field directly
            System.out.println("Accessing: " + outerField);
        }
    }
}
```

8.1.3 Instantiating Member Classes

To create an instance of the inner class, you first need an instance of the outer class:

```
OuterClass outer = new OuterClass();
OuterClass.InnerClass inner = outer.new InnerClass();
inner.display();
```

This syntax highlights the strong link: the inner class instance is tied to its specific outer class instance.

8.1.4 Use Cases for Member Classes

Member classes provide logical grouping of classes that are not useful outside the context of their outer class. For example:

- **Helper classes**: Classes that support the main functionality of the outer class but don't need to be exposed elsewhere.
- **Encapsulation**: Keeping related code close together and hiding inner workings from the outside.
- **Event handlers**: In GUI programming, inner classes often handle events related to the outer class.

8.1.5 Example: Member Class in Action

Consider a `Library` class with an inner class `Book`:

```
public class Library {
    private String name;

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

    public class Book {
        private String title;

        public Book(String title) {
            this.title = title;
        }

        public void showDetails() {
            // Inner class accessing outer class member
            System.out.println("Book: " + title + ", Library: " + name);
        }
    }
}
```

Usage:

```
Library lib = new Library("City Library");
Library.Book book = lib.new Book("Java Fundamentals");
book.showDetails();
```

Output:

```
Book: Java Fundamentals, Library: City Library
```

Here, `Book` logically belongs to a `Library` and accesses the outer class's `name` directly, showing the close coupling.

Full runnable code:

```
public class Library {
    private String name;

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

    public class Book {
        private String title;

        public Book(String title) {
            this.title = title;
        }

        public void showDetails() {
            // Inner class accessing outer class member
            System.out.println("Book: " + title + ", Library: " + name);
        }
    }

    public static void main(String[] args) {
        Library lib = new Library("City Library");
        Library.Book book = lib.new Book("Java Fundamentals");
        book.showDetails();
    }
}
```

8.1.6 Access Rules Between Inner and Outer Classes

- The **inner class can access all members** of the outer class, including `private` ones, as if it were part of the same class.
- The **outer class cannot directly access the inner class's members** unless it has an instance of the inner class.
- Inner class instances carry a reference to their enclosing outer class instance, allowing seamless interaction.

8.1.7 Summary

Member classes are a powerful feature for logically grouping classes that are tightly related. They enable:

- Encapsulation of helper or subordinate classes.
- Easy access to outer class members without exposing internals to the outside.
- Clean organization of code when classes are only relevant within a specific context.

In the next sections, we will explore other types of nested classes, including static nested classes and local classes, which offer additional design flexibility in Java.

8.2 Static Nested Classes

In Java, a **static nested class** is a nested class declared with the `static` keyword inside another class. Unlike member (inner) classes, static nested classes do *not* hold an implicit reference to an instance of the enclosing outer class. This key difference affects how static nested classes are used, instantiated, and what members they can access.

8.2.1 What Are Static Nested Classes?

A static nested class is associated with its outer class itself rather than with an instance of the outer class. Because of this, static nested classes:

- Cannot access non-static members (fields or methods) of the outer class directly.
- Can access `static` members of the outer class.
- Are instantiated without needing an instance of the outer class.

Here's the syntax for declaring a static nested class:

```
public class OuterClass {
    static class StaticNestedClass {
        void display() {
            System.out.println("Inside static nested class");
        }
    }
}
```

8.2.2 How Static Nested Classes Differ From Member Classes

Feature	Member (Inner) Class	Static Nested Class
Implicit reference to outer	Yes, to an instance of outer class	No
Access to outer class members	Can access all, including instance	Can only access static members
Instantiation	Requires an instance of outer class	Can be instantiated without outer instance
Use cases	Closely tied to outer class instance	More independent, utility or helper classes

8.2.3 When to Use Static Nested Classes

Static nested classes are useful when the nested class:

- Does **not** need access to instance variables or methods of the outer class.
- Can logically belong to the outer class as a static helper or utility.
- Should avoid the overhead of referencing the outer instance, saving memory.

For example, if you have a complex class and want to organize related helper classes without exposing them at the top level, static nested classes offer a clean, encapsulated solution.

8.2.4 Example: Using a Static Nested Class

Let's look at an example where a static nested class helps encapsulate functionality inside an outer class:

```
public class Computer {
    private String brand;

    public Computer(String brand) {
        this.brand = brand;
    }

    // Static nested class representing the CPU component
    static class CPU {
        private int cores;

        public CPU(int cores) {
            this.cores = cores;
        }

        public void displaySpecs() {
            System.out.println("CPU with " + cores + " cores");
        }
```

```
    }
}
```

Usage:

```
Computer.CPU cpu = new Computer.CPU(8);
cpu.displaySpecs();
```

Notice that we instantiate `CPU` without creating a `Computer` object. This makes sense because the `CPU` class here doesn't rely on a specific `Computer` instance.

Full runnable code:

```
public class Computer {
    private String brand;

    public Computer(String brand) {
        this.brand = brand;
    }

    // Static nested class representing the CPU component
    static class CPU {
        private int cores;

        public CPU(int cores) {
            this.cores = cores;
        }

        public void displaySpecs() {
            System.out.println("CPU with " + cores + " cores");
        }
    }

    public static void main(String[] args) {
        Computer.CPU cpu = new Computer.CPU(8);
        cpu.displaySpecs();
    }
}
```

8.2.5 Design Scenarios Favoring Static Nested Classes

Static nested classes are well-suited for:

- **Helper or utility classes** that perform functions related to the outer class but do not need access to its instance data.
- **Builder patterns**, where a builder class is nested inside the class it builds, but is static to avoid unnecessary coupling.
- **Grouping related classes** logically without cluttering the top-level namespace.
- **Immutable data structures**, where the nested class represents a part or state of the outer class in a static, reusable way.

8.2.6 Summary

Static nested classes provide a way to organize related classes under an outer class without binding them to instances. Their independence from outer instances offers efficiency and clarity when the nested class's behavior is more general or static in nature. This distinction between member and static nested classes enables better design decisions depending on how tightly the nested class needs to be coupled to the outer class.

In the next section, we'll explore **local classes**, which are nested classes declared inside methods and have their own specific use cases.

8.3 Local Classes

In Java, a **local class** is a nested class defined *within a method*, constructor, or initializer block. Unlike member or static nested classes, local classes exist only during the execution of that method, and their scope is limited to the enclosing block.

8.3.1 What Are Local Classes?

Local classes behave like regular classes but have a restricted scope. They are declared inside methods and cannot be accessed outside the method in which they are defined. This makes them ideal for encapsulating helper functionality that's only relevant within a specific method.

Here is the syntax for a local class inside a method:

```
public void process() {
    class Helper {
        void assist() {
            System.out.println("Helping inside process method");
        }
    }

    Helper helper = new Helper();
    helper.assist();
}
```

In this example, the class `Helper` is local to the method `process`. You cannot create an instance of `Helper` outside of `process`.

8.3.2 Scope and Restrictions

Local classes:

- Can access **final** or **effectively final** local variables and parameters from the enclosing method.
- Cannot have access modifiers (e.g., `public`, `private`) because they are local to the method.
- Cannot be declared `static` since they belong to a method's instance.
- Are not visible outside the method they belong to, limiting their lifetime to the method's execution.

Since Java 8, local classes can access local variables if they are effectively final, meaning the variable is not modified after initialization.

8.3.3 Practical Use Cases

Local classes are particularly useful for:

- **Simplifying complex methods:** Breaking down logic into smaller helper classes without exposing them globally.
- **Callbacks and event handling:** For example, inside a GUI method or when dealing with listeners, you might use a local class to implement an event handler specific to that method's context.
- **Encapsulating temporary behavior:** When the behavior is short-lived and doesn't warrant a full member or top-level class.

8.3.4 Example: Local Class in Action

Imagine you have a method that processes a list of items and needs to track progress or perform intermediate steps:

```
public void processItems(List<String> items) {
    final int total = items.size();

    class ProgressTracker {
        int processed = 0;

        void increment() {
            processed++;
            System.out.println("Processed " + processed + " out of " + total);
        }
    }

    ProgressTracker tracker = new ProgressTracker();
```

```
    for (String item : items) {
        // Process item here
        tracker.increment();
    }
}
```

Here, `ProgressTracker` is a local class tailored for use only within the `processItems` method, enhancing modularity and readability.

Full runnable code:

```
import java.util.List;
import java.util.Arrays;

public class LocalClassExample {

    public void processItems(List<String> items) {
        final int total = items.size();

        class ProgressTracker {
            int processed = 0;

            void increment() {
                processed++;
                System.out.println("Processed " + processed + " out of " + total);
            }
        }

        ProgressTracker tracker = new ProgressTracker();

        for (String item : items) {
            // Simulate processing the item
            tracker.increment();
        }
    }

    public static void main(String[] args) {
        LocalClassExample example = new LocalClassExample();
        example.processItems(Arrays.asList("Item1", "Item2", "Item3"));
    }
}
```

8.3.5 Summary

Local classes provide a neat way to define helper classes scoped to a method, keeping the global class structure clean and focused. By limiting visibility and lifetime, local classes promote encapsulation and reduce clutter. They are perfect for short-lived, method-specific logic such as callbacks, progress tracking, or simplifying complex workflows.

In the next section, we'll explore **anonymous classes**—a concise way to implement classes on the fly without naming them explicitly.

8.4 Anonymous Classes and Functional Use Cases

8.4.1 What Are Anonymous Classes?

Anonymous classes in Java are a special type of inner class without a name. They are declared and instantiated all at once, typically used to implement an interface or extend a class in a concise way. Since they have no explicit class name, they are ideal for quick, one-off implementations where defining a full-fledged class would be cumbersome.

The syntax of an anonymous class looks like this:

```
InterfaceOrClass instance = new InterfaceOrClass() {
    // Override methods here
};
```

This creates a new unnamed subclass or implementation on the spot, allowing you to override methods or provide implementation details inline.

8.4.2 Typical Use Cases

Anonymous classes are widely used for:

- **Event handling in GUI applications:** For example, attaching listeners like mouse or button click handlers where you only need a small custom implementation.
- **Implementing simple interfaces or abstract classes:** When you want to quickly provide behavior for interfaces like `Runnable`, `Comparator`, or callback interfaces without cluttering your codebase with multiple class files.
- **Quick customization of existing classes:** For example, overriding a method in a one-off subclass without the need for a named class.

Anonymous classes are a staple in Java programming, especially before Java 8 introduced lambdas.

8.4.3 Example 1: Anonymous Class for Runnable

Here's a classic example using an anonymous class to create a `Runnable` task for a new thread:

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread running using anonymous class.");
    }
});
```

```
thread.start();
```

Instead of creating a separate `Runnable` implementation class, we directly provide the implementation inline. This makes the code concise and focused on the task at hand.

8.4.4 Example 2: Anonymous Class for Event Listener

Consider a simple GUI button listener using Swing:

```
JButton button = new JButton("Click Me");

button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});
```

The `ActionListener` interface is implemented anonymously here, defining the action performed when the button is clicked, all inline without additional classes.

8.4.5 Anonymous Classes and Functional Interfaces

Starting with Java 8, **functional interfaces** (interfaces with a single abstract method) opened the door for even more concise syntax using **lambda expressions**, which can be seen as a streamlined alternative to anonymous classes.

For example, the above `Runnable` example can be rewritten with a lambda like this:

```
Thread thread = new Thread(() -> System.out.println("Thread running using lambda."));
thread.start();
```

While lambdas provide brevity and clarity, anonymous classes remain valuable when you need to:

- Implement interfaces with multiple methods (non-functional interfaces).
- Use `this` to refer to the anonymous class itself (lambdas cannot do this).
- Override multiple methods or provide complex implementations.

We'll explore functional interfaces and lambdas in detail in a later chapter.

8.4.6 Readability and When to Use Anonymous Classes

Anonymous classes keep code compact and focused, but overusing them or placing large blocks of code inline can reduce readability. If the implementation is complex, consider:

- Extracting the anonymous class into a named class for clarity and reusability.
- Using lambda expressions when applicable for simpler, functional-style implementations.

Anonymous classes are a middle ground: they provide encapsulation without the boilerplate of a named class but require care to maintain code readability.

8.4.7 Summary

Anonymous classes enable quick, inline implementations of interfaces or subclasses without naming overhead, making them invaluable for event handling, callbacks, and simple behavior customization. They bridge the gap between traditional OOP and functional programming styles, especially when combined with functional interfaces and lambdas introduced in later Java versions. By understanding anonymous classes, you gain a powerful tool for flexible, modular Java design.

Chapter 9.

Packages and Modular Design

1. Organizing Code with Packages
2. Access Control Across Packages
3. Introduction to Java Modules (`module-info.java`)

9 Packages and Modular Design

9.1 Organizing Code with Packages

9.1.1 The Purpose and Benefits of Packages

In Java, **packages** serve as a fundamental way to organize and structure your code. Think of packages as folders or namespaces that group related classes and interfaces together. They help in managing large codebases by:

- **Organizing code logically:** Grouping related classes (e.g., all data models, utilities, or services) together makes your project easier to navigate and maintain.
- **Avoiding naming conflicts:** Packages create unique namespaces, so two classes with the same name can coexist in different packages without clashes.
- **Encapsulating code:** Packages help define visibility and access control, limiting which classes can see or use certain members and classes.

By structuring your code into packages, you create a modular and maintainable project that scales well as complexity grows.

9.1.2 Declaring Packages and Placing Classes

To declare a package, use the `package` keyword as the very first statement in your Java source file. For example:

```
package com.example.utils;

public class StringUtils {
    public static boolean isEmpty(String s) {
        return s == null || s.isEmpty();
    }
}
```

This declaration states that the class `StringUtils` belongs to the package `com.example.utils`.

Important notes:

- The package statement must be the first line (except comments) in the file.
- The source file should be located in a directory structure that matches the package name. For example, `com.example.utils.StringUtils` would reside in `/com/example/utils/StringUtils.java`.
- When compiling and running, the Java compiler and JVM expect this folder structure to reflect the package hierarchy.

9.1.3 Example Project Structure with Multiple Packages

Consider a small project organized like this:

```
project-root/
 +- src/
     +- com/
         +- example/
             +- model/
                  +- User.java
             +- service/
                  +- UserService.java
             +- utils/
                   +- StringUtils.java
     +- Main.java
```

- `User.java` in `com.example.model` contains your data model classes.
- `UserService.java` in `com.example.service` holds business logic.
- `StringUtils.java` in `com.example.utils` contains utility functions.
- `Main.java` may be in the default (unnamed) package or in a root package like `com.example`.

This hierarchical structure improves clarity, helps prevent class name conflicts, and enforces logical separation of concerns.

9.1.4 Conventions for Package Naming

Java package names follow a widely adopted convention based on the **reverse domain name** of your organization or project. For example:

- If your company website is `example.com`, your base package would be `com.example`.
- You then add sub-packages for different layers or modules (`service`, `model`, `utils`, etc.).

Using reverse domain names ensures uniqueness across packages worldwide, reducing the risk of naming collisions when combining code from different sources.

Additional tips:

- Use all lowercase letters for package names to avoid conflicts with class names.
- Separate words with dots to create meaningful sub-packages.
- Avoid using Java reserved keywords as package names.

9.1.5 Package Visibility and Encapsulation

Packages also influence **access control** in Java. Access modifiers such as `public`, `protected`, and `private` interact with package structure to control visibility:

- **Public members** are accessible everywhere.
- **Private members** are accessible only within the same class.
- **Package-private** (default, no modifier) members are accessible only within the same package—this allows classes inside a package to interact closely while hiding details from other packages.
- **Protected members** are accessible within the same package and also by subclasses, even if they are in different packages.

Using package-private visibility promotes **encapsulation at the package level**, allowing you to hide implementation details and expose only what is necessary for other parts of your program.

9.1.6 Summary

Packages are essential in Java for organizing your code into coherent, manageable modules. They prevent naming collisions, promote encapsulation, and reflect your project's logical structure. By following package naming conventions and placing classes appropriately in directories that match package names, you ensure your code is clear, maintainable, and scalable.

9.2 Access Control Across Packages

Java's access control mechanisms are central to building well-encapsulated and maintainable software. When working with multiple packages, understanding how Java's access modifiers govern visibility **across package boundaries** becomes critical for designing robust APIs and preserving internal implementation details.

9.2.1 Overview of Java Access Modifiers

Java provides four primary access levels for classes, methods, and variables:

- `public`: Accessible from any other class, regardless of package. This is the most permissive access level and is typically used for APIs intended for broad usage.
- `protected`: Accessible within the **same package** and also accessible to subclasses

even if they reside in different packages.

- **Default (package-private)**: If no access modifier is specified, the member or class is accessible only within its own package. This means classes outside the package cannot see or use it.
- `private`: Accessible only within the **declaring class** itself, no matter what package it is in. It is the most restrictive access level.

9.2.2 Package-Private (Default) Access and Its Importance

The default or package-private access level is often overlooked but is essential for **encapsulation at the package level**. When you omit any modifier, Java treats the class or member as package-private. This means:

- It is **visible to all classes inside the same package**.
- It is **not visible to classes in other packages**, even if those classes inherit from the class.

This behavior encourages grouping related classes into the same package so they can interact freely, while hiding implementation details from the outside world.

9.2.3 Cross-Package Access Examples

Let's consider a simple example with two packages: `com.example.model` and `com.example.service`.

```
// File: com/example/model/User.java
package com.example.model;

public class User {
    String username;            // package-private field
    protected String email;     // protected field
    private int id;             // private field

    public User(String username, String email, int id) {
        this.username = username;
        this.email = email;
        this.id = id;
    }

    String getUsername() {       // package-private method
        return username;
    }

    protected String getEmail() {
        return email;
```

```
    }

    private int getId() {
        return id;
    }
}
```

Now, in another package:

```
// File: com/example/service/UserService.java
package com.example.service;

import com.example.model.User;

public class UserService extends User {
    public UserService(String username, String email, int id) {
        super(username, email, id);
    }

    public void printUserDetails() {
        // System.out.println(username); // Compile error: package-private not visible outside com.exam
        System.out.println(email);       // Allowed: protected visible to subclass
        // System.out.println(id);       // Compile error: private
        System.out.println(getEmail());  // Allowed: protected method accessible
        // System.out.println(getUsername()); // Compile error: package-private method not visible
    }
}
```

Key takeaways from the example:

- `username` (package-private) is **not accessible** from `UserService` in a different package. This applies to the field and its getter method `getUsername()`.
- `email` (protected) is accessible in `UserService` because it is a subclass, even though it is in a different package.
- `id` (private) and `getId()` are **not accessible** outside the `User` class, regardless of subclassing or package.

This illustrates how package boundaries and inheritance interact with access control modifiers.

9.2.4 Design Considerations for Exposing APIs Across Packages

When designing your Java applications, especially large ones split across multiple packages or modules, carefully deciding which members and classes to expose is vital.

1. Use `public` sparingly for APIs: Only classes and methods that are truly part of your external API should be `public`. Overusing public access risks exposing internal details, leading to fragile code dependent on implementation specifics.

2. Favor package-private for internal collaboration: Related classes within the same

package can freely share package-private methods and fields, promoting cohesion without exposing internals to the outside.

3. Leverage `protected` for inheritance hierarchies: When designing abstract classes or frameworks meant to be extended, use `protected` members to allow subclasses to access necessary methods or fields, but keep them hidden from unrelated classes.

4. Encapsulate with `private`: Always restrict fields and helper methods to `private` unless broader access is explicitly needed. This minimizes unintended usage and makes future changes safer.

5. Provide controlled access with getters/setters: Even when fields are private, exposing them via `public` or `protected` getters and setters lets you enforce validation, logging, or other logic—offering a flexible interface without compromising encapsulation.

9.2.5 Summary

Understanding Java's access modifiers in the context of package boundaries helps create well-structured, maintainable codebases:

- **`public`** members are accessible everywhere and form your external API.
- **`protected`** members are accessible within the same package and to subclasses across packages.
- **Default (package-private)** access restricts visibility to the same package, promoting tight encapsulation among related classes.
- **`private`** members are visible only inside their own class.

This nuanced access control supports modular design by protecting implementation details, encouraging clear boundaries, and guiding API exposure. By carefully combining these modifiers, you ensure that your Java application is robust, secure, and easier to evolve.

9.3 Introduction to Java Modules (`module-info.java`)

With the release of Java 9, the Java Platform Module System (JPMS) was introduced to address the growing complexity of large-scale applications and to enhance modularity beyond traditional packages. Modules provide a higher level of structure, allowing developers to explicitly declare dependencies and control the visibility of packages across the application, improving maintainability, security, and performance.

9.3.1 What Is the Java Platform Module System (JPMS)?

JPMS, often referred to as the **Java Modules System**, is a framework that organizes Java code into **modules**—self-describing collections of packages with explicit dependencies. It complements packages by grouping related packages and resources together under a module boundary.

Modules solve problems that packages alone cannot:

- **Strong encapsulation**: Unlike packages, which only control access at the package level, modules explicitly state which packages are visible to other modules, hiding internal implementation packages completely.
- **Reliable configuration**: By declaring dependencies at the module level, the runtime can verify at startup that all required modules are present, preventing classpath issues.
- **Improved performance**: The module system can optimize startup and reduce memory footprint by loading only needed modules.

9.3.2 The Module Descriptor: `module-info.java`

At the heart of JPMS is the **module descriptor**, a special file named `module-info.java` placed at the root of the module source folder. This file declares:

- The module's name.
- The packages the module **exports** (makes accessible to other modules).
- The modules it **requires** (depends on).

This descriptor enables the compiler and runtime to enforce module boundaries and dependencies.

9.3.3 Basic Syntax of `module-info.java`

Here is a simple example of a module descriptor:

```
module com.example.library {
    exports com.example.library.api;
    requires java.logging;
}
```

- `module com.example.library` declares a module named `com.example.library`.
- `exports com.example.library.api` makes the package `com.example.library.api` accessible to other modules.
- `requires java.logging` declares a dependency on the `java.logging` module provided by the JDK.

If a package within the module is not exported, it remains inaccessible outside the module, even if public classes exist in that package. This enforces **strong encapsulation** at the module level.

9.3.4 Example: Simple Modular Project Structure

Suppose we have a project with two modules:

- `com.example.app` (main application)
- `com.example.utils` (utility library)

Each module has its own `module-info.java`.

`com.example.utils/module-info.java`:

```
module com.example.utils {
    exports com.example.utils.helpers;
}
```

`com.example.app/module-info.java`:

```
module com.example.app {
    requires com.example.utils;
}
```

The `com.example.utils` module exports the `com.example.utils.helpers` package, allowing `com.example.app` to use those classes. The `com.example.app` module declares that it requires `com.example.utils`.

9.3.5 Benefits of Using Modules in Large Projects

1. Improved Encapsulation Modules explicitly control what is exposed outside their boundaries, preventing accidental access to internal packages. This is a stronger guarantee than package-private access since it applies even if classes are public.

2. Reliable Configuration At both compile-time and runtime, the system verifies module dependencies, reducing the infamous "classpath hell" problems where classes fail to load due to missing dependencies.

3. Better Maintainability Clearly defined module boundaries help teams understand and manage dependencies, making large codebases easier to navigate and evolve.

4. Performance Optimizations The module system can optimize which parts of the application are loaded, improving startup time and reducing resource usage.

9.3.6 Running a Modular Application

To compile and run modular applications, you use the `javac` and `java` commands with module-specific options.

Assuming the source files are structured as:

```
/project-root
  /com.example.utils
    /src
      /module-info.java
      /com/example/utils/helpers/Utility.java
  /com.example.app
    /src
      /module-info.java
      /com/example/app/Main.java
```

Compile modules:

```
javac -d mods/com.example.utils com.example.utils/src/module-info.java com.example.utils/src/com/exampl
```

```
javac --module-path mods -d mods/com.example.app com.example.app/src/module-info.java com.example.app/s
```

Run the application:

```
java --module-path mods -m com.example.app/com.example.app.Main
```

Here, `--module-path` specifies where compiled modules are located, and `-m` specifies the module and main class to run.

9.3.7 Summary

The Java Platform Module System, introduced in Java 9, marks a significant evolution in organizing and structuring Java applications beyond packages. By defining explicit module boundaries with `module-info.java`, developers gain fine-grained control over what is visible and what is hidden, making large codebases more modular, secure, and easier to maintain.

Incorporating modules encourages **strong encapsulation**, **reliable dependency management**, and sets the foundation for scalable Java applications. While the initial learning curve can be steep, understanding and applying JPMS is essential for modern Java development, especially for enterprise-scale projects.

Chapter 10.

Exception Handling and Design

1. The Exception Hierarchy
2. Checked vs Unchecked Exceptions
3. Creating Custom Exceptions
4. Exception Handling as a Design Tool

10 Exception Handling and Design

10.1 The Exception Hierarchy

In Java, exception handling is an essential mechanism that helps manage errors and unexpected behavior gracefully. Rather than crashing abruptly, Java programs can detect, catch, and handle problems, improving robustness and reliability. To achieve this, Java provides a structured and extensible exception hierarchy rooted in the `Throwable` class. Understanding this hierarchy is fundamental to writing clean, maintainable error-handling code.

10.1.1 The Root: `Throwable`

At the top of the exception hierarchy is the abstract class `Throwable`, which is the superclass of all errors and exceptions in Java. It has two main direct subclasses:

- `Error`
- `Exception`

Only instances of `Throwable` or its subclasses can be thrown using the `throw` statement or caught using `try-catch` blocks.

10.1.2 Branch 1: `Error`

`Error` represents serious issues that are beyond the control of the program. These typically signal problems with the Java Virtual Machine (JVM) itself or the system environment. Examples include:

- `OutOfMemoryError`
- `StackOverflowError`
- `NoClassDefFoundError`

Because these errors indicate fundamental failures, they should **not** be caught or handled in most cases. Attempting to recover from an `Error` is generally considered bad practice, as they reflect conditions where the JVM cannot continue reliably.

10.1.3 Branch 2: `Exception`

The more commonly used branch is `Exception`. This class and its subclasses represent conditions that a program **should** catch and handle. For example:

- `IOException` – Signals problems with input/output operations.
- `SQLException` – Related to database access.
- `FileNotFoundException` – A file requested does not exist.

Within `Exception`, Java distinguishes between two categories: **checked exceptions** and **unchecked exceptions**.

10.1.4 Checked vs. Unchecked Exceptions

Checked Exceptions

Checked exceptions are subclasses of `Exception` **excluding** `RuntimeException` and its subclasses. These exceptions are checked at compile time. The compiler forces you to either handle them using a `try-catch` block or declare them using the `throws` clause in method signatures.

Example:

```
import java.io.*;

public class FileReaderExample {
    public void readFile(String filename) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        String line = reader.readLine();
        reader.close();
    }
}
```

Here, `IOException` is a checked exception, and the method must declare it with `throws`.

Unchecked Exceptions

Unchecked exceptions are subclasses of `RuntimeException`. These are **not** checked at compile time, meaning the compiler does not force you to handle or declare them. They often indicate programming bugs such as logic errors or improper API use.

Common unchecked exceptions include:

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `IllegalArgumentException`
- `ArithmeticException`

Example:

```
public class Calculator {
    public int divide(int a, int b) {
        return a / b; // May throw ArithmeticException if b == 0
    }
}
```

The method above may throw an `ArithmeticException`, but the compiler doesn't require you to handle it explicitly.

Full runnable code:

```
import java.io.*;

// Demonstrates checked exception (IOException)
class FileReaderExample {
    public void readFile(String filename) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        String line = reader.readLine();
        System.out.println("First line: " + line);
        reader.close();
    }
}

// Demonstrates unchecked exception (ArithmeticException)
class Calculator {
    public int divide(int a, int b) {
        return a / b; // May throw ArithmeticException if b == 0
    }
}

public class ExceptionHierarchyDemo {
    public static void main(String[] args) {
        FileReaderExample fileReader = new FileReaderExample();
        Calculator calculator = new Calculator();

        // Handling checked exception with try-catch
        try {
            // Change to a file path that exists or not to see behavior
            fileReader.readFile("nonexistentfile.txt");
        } catch (IOException e) {
            System.out.println("Caught IOException: " + e.getMessage());
        }

        // Handling unchecked exception with try-catch (optional)
        try {
            int result = calculator.divide(10, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Caught ArithmeticException: " + e.getMessage());
        }

        // Unchecked exceptions can be left unhandled (program will crash)
        // int crash = calculator.divide(10, 0); // Uncomment to see runtime crash
    }
}
```

10.1.5 The RuntimeException Branch

`RuntimeException` is the superclass of exceptions that are unchecked. These typically arise from logical flaws that should be avoided through proper code rather than being caught.

Developers should be cautious not to misuse unchecked exceptions as a way to skip error-handling responsibility. While convenient, relying too heavily on unchecked exceptions can make code brittle and harder to debug.

10.1.6 Java Exception Hierarchy Diagram (Textual)

Here's a simplified hierarchy view:

```
Object
 +-- Throwable
      +-- Error
           +-- (e.g., OutOfMemoryError, StackOverflowError)
      +-- Exception
           +-- RuntimeException
                +-- NullPointerException
                +-- IllegalArgumentException
                +-- IndexOutOfBoundsException
           +-- (Checked Exceptions)
                +-- IOException
                     +-- FileNotFoundException
                +-- SQLException
```

10.1.7 Practical Perspective

A robust application should:

- Use checked exceptions to signal conditions that callers can reasonably be expected to recover from (e.g., missing files).
- Use unchecked exceptions to signal programming errors or violations of method contracts (e.g., invalid arguments).
- Avoid catching `Error` unless absolutely necessary.

Understanding the structure of the exception hierarchy helps developers create more maintainable and predictable applications. It also enables precise exception handling strategies, which will be explored further in the rest of this chapter.

In the next section, we'll dive deeper into the contrast between **checked and unchecked**

exceptions, and how to choose between them when designing your own application or API.

10.2 Checked vs Unchecked Exceptions

Exception handling in Java is more than just catching and displaying error messages—it plays a vital role in designing robust and maintainable applications. One of the key distinctions in Java's exception model is between **checked** and **unchecked** exceptions. This section explores their differences, practical implications, and how they shape application and API design.

10.2.1 Checked Exceptions

Definition: Checked exceptions are exceptions that are **checked at compile time**. These exceptions derive from the `Exception` class **but not** from the `RuntimeException` class. The compiler forces developers to either handle these exceptions using a `try-catch` block or declare them using the `throws` keyword.

Purpose: Checked exceptions represent conditions that a program **can anticipate and should attempt to recover from**. Examples include file I/O problems, database errors, or network issues.

Compiler Enforcement Example:

```
import java.io.*;

public class FileLoader {
    public void load(String fileName) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        System.out.println(reader.readLine());
        reader.close();
    }
}
```

In the example above, `FileReader` and `readLine()` can throw an `IOException`, so the method must declare it in its `throws` clause. If you omit it, the compiler will produce an error.

Handling Checked Exceptions:

```
import java.io.*;

public class App {
    public static void main(String[] args) {
        FileLoader loader = new FileLoader();
        try {
```

```
            loader.load("data.txt");
        } catch (IOException e) {
            System.err.println("Could not read file: " + e.getMessage());
        }
    }
}
 class FileLoader {
    public void load(String fileName) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        System.out.println(reader.readLine());
        reader.close();
    }
}
```

This enforced handling improves reliability in environments where resource access can fail.

10.2.2 Unchecked Exceptions

Definition: Unchecked exceptions are **not checked at compile time**. They are subclasses of `RuntimeException` and indicate programming logic errors that typically **should not be recovered from**.

Common Types:

- `NullPointerException`
- `IllegalArgumentException`
- `ArrayIndexOutOfBoundsException`
- `ArithmeticException`

These exceptions usually result from violating a method's preconditions or from flaws in program logic.

Example:

```
public class Calculator {
    public int divide(int a, int b) {
        return a / b; // May throw ArithmeticException if b == 0
    }
}
```

The method above does not declare `throws ArithmeticException` because it is unchecked. It's assumed that the caller is responsible for ensuring that `b` is not zero.

Handling Unchecked Exceptions (Optional):

```
try {
    Calculator calc = new Calculator();
    int result = calc.divide(10, 0);
} catch (ArithmeticException e) {
    System.err.println("Cannot divide by zero.");
```

```
}
```

While handling unchecked exceptions is possible, it's not required.

10.2.3 When to Use Checked vs Unchecked Exceptions

Choosing between checked and unchecked exceptions is a critical part of API design and overall architecture.

Scenario	Exception Type	Justification
File not found, network error, I/O	Checked	Recoverable, user can retry or provide alternatives.
Invalid arguments passed to method	Unchecked	Indicates a logic error that the programmer should fix.
Division by zero or null access	Unchecked	Bug in program, not meant to be handled by user logic.
Database connection failure	Checked	Often recoverable; retry or show error to user.

Design Rule of Thumb:

- Use **checked exceptions** for errors that clients **must be made aware of** and can **reasonably recover from**.
- Use **unchecked exceptions** for **programming errors**, such as violating assumptions or failing validations.

10.2.4 Impact on API Design

When designing an API, consider the **burden on the caller**. Declaring a method with multiple checked exceptions makes the client code more complex and verbose:

```
public void readFile(String path) throws IOException, FileNotFoundException
```

This provides more detail but requires explicit `try-catch` handling. Alternatively, using unchecked exceptions can simplify method signatures:

```
public void readFile(String path) {
    if (path == null) {
        throw new IllegalArgumentException("Path cannot be null.");
    }
}
```

However, overusing unchecked exceptions can reduce clarity and make the API harder to use correctly.

10.2.5 Conclusion

Understanding the distinction between checked and unchecked exceptions is crucial for writing robust, predictable Java programs. Checked exceptions encourage handling of external and recoverable issues, while unchecked exceptions highlight bugs and violations of assumptions. By thoughtfully applying both types in your design, you enable clearer contracts, better error handling, and more maintainable code.

10.3 Creating Custom Exceptions

In Java programming, exceptions are used to signal that an unexpected condition or error has occurred during the execution of a program. While Java provides a comprehensive set of built-in exceptions, there are many scenarios where these standard exceptions are insufficient to express the specific problems your application might encounter. This is where **custom exceptions** become invaluable. Custom exceptions allow developers to create meaningful, domain-specific error types that improve code clarity, facilitate better error handling, and enhance maintainability.

10.3.1 What Are Custom Exceptions?

Custom exceptions are user-defined classes that extend the Java exception hierarchy, enabling you to represent application-specific error conditions clearly and precisely. Unlike general exceptions such as `NullPointerException` or `IOException`, custom exceptions convey information tailored to the particular business logic or architectural constraints of your software.

For example, in an e-commerce application, rather than throwing a generic `IllegalArgumentException` when a payment fails, you could define a `PaymentFailedException` that clearly communicates what went wrong. This specificity makes your code easier to understand and helps other developers quickly identify the nature of the problem when handling exceptions.

Custom exceptions fit neatly into Java's exception hierarchy, typically extending either `Exception` or `RuntimeException`. The choice depends on whether the exception is **checked** or **unchecked** — a topic explored in the previous section.

10.3.2 How to Create Custom Exceptions

Creating a custom exception in Java is straightforward and follows these basic steps:

1. **Choose the base class to extend.**
    - Extend `Exception` to create a **checked exception**. Checked exceptions must be declared in a method's `throws` clause and force the caller to handle or propagate them explicitly.
    - Extend `RuntimeException` to create an **unchecked exception**. Unchecked exceptions do not need to be declared or caught, typically reserved for programming errors or conditions that usually cannot be recovered from.
2. **Define constructors.** It's a common practice to include constructors that mirror those of `Exception` or `RuntimeException`:
    - A no-argument constructor.
    - A constructor that accepts a descriptive error message (`String message`).
    - A constructor that accepts both an error message and a cause (`Throwable cause`).
    - Optionally, a constructor that accepts only a cause.
3. **Add any additional methods or fields (optional).** Sometimes, a custom exception might carry extra information relevant to the error, such as error codes, user-friendly messages, or context data.

Here is a simple example of a custom checked exception:

```
public class InsufficientFundsException extends Exception {

    public InsufficientFundsException() {
        super();
    }

    public InsufficientFundsException(String message) {
        super(message);
    }

    public InsufficientFundsException(String message, Throwable cause) {
        super(message, cause);
    }

    public InsufficientFundsException(Throwable cause) {
        super(cause);
    }
}
```

For an unchecked exception, the only difference is extending `RuntimeException`:

```
public class InvalidUserInputException extends RuntimeException {

    public InvalidUserInputException() {
        super();
    }
```

```
    public InvalidUserInputException(String message) {
        super(message);
    }

    public InvalidUserInputException(String message, Throwable cause) {
        super(message, cause);
    }

    public InvalidUserInputException(Throwable cause) {
        super(cause);
    }
}
```

Full runnable code:

```
// Custom checked exception
class InsufficientFundsException extends Exception {
    public InsufficientFundsException() {
        super();
    }

    public InsufficientFundsException(String message) {
        super(message);
    }

    public InsufficientFundsException(String message, Throwable cause) {
        super(message, cause);
    }

    public InsufficientFundsException(Throwable cause) {
        super(cause);
    }
}

// Custom unchecked exception
class InvalidUserInputException extends RuntimeException {
    public InvalidUserInputException() {
        super();
    }

    public InvalidUserInputException(String message) {
        super(message);
    }

    public InvalidUserInputException(String message, Throwable cause) {
        super(message, cause);
    }

    public InvalidUserInputException(Throwable cause) {
        super(cause);
    }
}

// Demo usage
public class CustomExceptionDemo {
    public static void main(String[] args) {
        try {
```

```
            withdraw(50.0, 20.0);  // Succeeds
            withdraw(30.0, 100.0); // Triggers checked exception
        } catch (InsufficientFundsException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }

        // Triggering unchecked exception
        try {
            processInput(null); // Will throw InvalidUserInputException
        } catch (InvalidUserInputException e) {
            System.out.println("Caught runtime exception: " + e.getMessage());
        }
    }

    // Throws a checked exception
    public static void withdraw(double balance, double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Attempted to withdraw $" + amount + ", but only $" + b
        }
        System.out.println("Withdrawal of $" + amount + " successful.");
    }

    // Throws an unchecked exception
    public static void processInput(String input) {
        if (input == null || input.trim().isEmpty()) {
            throw new InvalidUserInputException("Input cannot be null or empty.");
        }
        System.out.println("Processing input: " + input);
    }
}
```

10.3.3 When to Use Custom Exceptions

Knowing **when** to create and throw custom exceptions is key to writing clean, maintainable code. You should consider using custom exceptions in the following scenarios:

1. To Represent Domain-Specific Errors

When your application operates within a particular business domain or context, using domain-specific exceptions helps express error conditions precisely. For example, an airline reservation system might have exceptions like `SeatUnavailableException` or `FlightOverbookedException` to clearly communicate problems unique to flight bookings.

This enhances code readability and communicates intent clearly both within the code and to developers consuming your API.

2. To Distinguish Different Error Conditions

Built-in exceptions are often too generic, causing ambiguity about the exact failure reason. Creating custom exceptions lets you differentiate errors clearly.

For example, you might have multiple failure modes when processing user registration: `UsernameAlreadyExistsException`, `WeakPasswordException`, or `EmailFormatException`. Throwing distinct exceptions allows calling code to respond appropriately to each condition instead of relying on error messages or error codes.

3. To Simplify Exception Handling Logic

Custom exceptions help clients of your code write cleaner error handling blocks by catching specific exception types rather than inspecting exception messages or error codes. This results in fewer error-prone string comparisons and less convoluted code.

For instance, in a payment processing module, catching `PaymentDeclinedException` separately from `NetworkException` allows different retry or compensation strategies.

4. To Encapsulate Low-Level Exceptions

When integrating third-party libraries or lower layers of an application, you might want to encapsulate checked exceptions or exceptions with complex semantics behind a simpler custom exception interface. This technique, sometimes called **exception translation or wrapping**, creates a cleaner and more consistent API.

For example, your data access layer might catch `SQLException` and throw a custom `DataAccessException`, hiding the low-level details from the business logic layer.

5. When You Want to Attach Additional Contextual Information

If an error requires more data than just a message and cause—such as error codes, remediation suggestions, or affected resource identifiers—a custom exception class can include extra fields and methods.

For example, a `ValidationException` might include a list of field errors to give detailed feedback on invalid input.

10.3.4 Best Practices

- Avoid overusing custom exceptions: Create them only when they provide clear benefit. Excessive use can clutter code and make exception handling cumbersome.
- Name exceptions clearly and consistently, ending with `Exception` to follow Java conventions.
- Document your custom exceptions thoroughly to explain when and why they are thrown.
- Use checked exceptions for recoverable errors that callers should handle explicitly.
- Use unchecked exceptions for programming errors or conditions unlikely to be recovered from.

10.4 Exception Handling as a Design Tool

Exception handling is more than just reacting to unexpected errors; it is a powerful design mechanism that models error states explicitly and enforces robustness across your software system. When used thoughtfully, exceptions shape how your code behaves under adverse conditions, enable clear communication between components, and support graceful recovery from failures. This section explores how exceptions serve as a critical design tool, guiding API development, facilitating error recovery strategies, and improving overall software quality.

10.4.1 Modeling Error States and Enforcing Robustness

At its core, exception handling models the reality that no system runs perfectly all the time. Code execution often encounters exceptional situations such as invalid inputs, resource exhaustion, or external failures like network outages. By throwing and catching exceptions, your software explicitly represents these error states rather than ignoring or hiding them. This transparency helps enforce robustness by ensuring that error conditions are surfaced, recognized, and handled appropriately rather than causing silent failures or data corruption.

Using exceptions also forces developers to confront potential failure scenarios during design. For example, a method that declares it throws a checked exception reminds callers that certain risks exist, encouraging them to write code that anticipates and handles those risks. This proactive approach helps reduce bugs and improve system resilience.

10.4.2 Designing APIs with Clear Exception Policies

A well-designed API clearly communicates what exceptions clients can expect and under what conditions. Defining an explicit exception policy is essential to prevent confusion and misuse. Here are some guiding principles for designing APIs with thoughtful exception handling:

- **Document all exceptions:** Every method that can throw exceptions should document which exceptions may be raised and what conditions trigger them. This allows clients to plan proper error handling.
- **Use specific exceptions:** Prefer domain-specific or custom exceptions over generic types like `Exception` or `RuntimeException`. Specific exceptions provide clarity and allow clients to distinguish different error types easily.
- **Limit checked exceptions to recoverable errors:** Checked exceptions should be reserved for errors that the caller can realistically handle, such as validation failures or transient resource issues. Unchecked exceptions are appropriate for programming errors or fatal conditions.
- **Avoid throwing exceptions for normal control flow:** Exceptions should signal

truly exceptional or erroneous situations, not expected or frequent outcomes.

For example, a file reading API might declare it throws `FileNotFoundException` and `IOException`, indicating issues with locating or reading files, while also documenting the precise conditions for these errors.

10.4.3 Graceful Error Recovery and Fallback Strategies

Exception handling enables your software to respond gracefully to failure rather than crashing abruptly. Designing for graceful degradation and recovery often involves layered exception handling and fallback strategies.

For instance, consider a web service client that fetches data from a remote API. Network errors such as timeouts or connection refusals can occur unpredictably. Instead of failing immediately, the client can catch exceptions like `SocketTimeoutException` and implement retry logic:

```
int attempts = 0;
while (attempts < MAX_RETRIES) {
    try {
        return remoteApi.fetchData();
    } catch (SocketTimeoutException e) {
        attempts++;
        log.warn("Timeout, retrying attempt " + attempts);
    }
}
throw new ServiceUnavailableException("Failed to fetch data after retries");
```

Fallback strategies may also involve alternative data sources or degraded functionality when the primary system is unavailable. For example, a caching layer can serve stale but available data if the database is down, catching exceptions and returning cached results as a fallback.

Such strategies demonstrate that exceptions are not just error signals but tools that enable robust systems capable of adapting to real-world failures.

10.4.4 Using Exceptions for Flow Control — And Why to Avoid It

Although exceptions technically can be used for controlling flow—such as breaking out of deeply nested loops or handling expected conditions—this practice is strongly discouraged in Java design for several reasons:

- **Performance overhead:** Throwing and catching exceptions is significantly more expensive than normal conditional checks, impacting performance if used frequently.
- **Obscured intent:** Using exceptions for routine flow control makes code harder to

read and understand, as exceptions signal abnormal conditions, not typical logic paths.

- **Complicated maintenance:** It becomes difficult to distinguish between genuine errors and flow control events, complicating debugging and testing.

Instead, use traditional control structures (if-else, loops, returns) to manage expected scenarios. Reserve exceptions strictly for unexpected or error conditions. For example, instead of throwing an exception to indicate a "not found" condition, consider returning an `Optional` or a sentinel value to communicate the absence of data clearly.

10.4.5 Best Practices for Exception Documentation and Logging

Effective use of exceptions as a design tool requires rigorous documentation and consistent logging:

- **Comprehensive documentation:** Each exception your code throws should be well-documented, ideally in method-level Javadoc comments. Include what the exception signifies, when it is thrown, and how clients should handle it.
- **Consistent logging:** Logging exceptions at appropriate levels (info, warning, error) provides critical insight during debugging and production monitoring. Avoid logging and rethrowing exceptions repeatedly, which leads to log clutter.
- **Include root causes:** When wrapping exceptions, always preserve the original cause by passing it to the new exception's constructor. This chaining maintains the full stack trace and error context.
- **Use meaningful messages:** Exception messages should be clear, actionable, and avoid revealing sensitive information.

Proper documentation and logging create a feedback loop that helps developers identify recurring issues and improve error handling strategies, ultimately enhancing software reliability.

Chapter 11.

SOLID Principles in Java

1. Single Responsibility Principle
2. Open/Closed Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

11 SOLID Principles in Java

11.1 Single Responsibility Principle

The **Single Responsibility Principle (SRP)** is the first and arguably most foundational of the five SOLID principles in object-oriented design. It states:

> **"A class should have only one reason to change."** — Robert C. Martin (Uncle Bob)

This means that a class should have only one job or responsibility. If a class handles multiple responsibilities, changes in one area can inadvertently affect other parts of the class, making it harder to understand, test, and maintain.

11.1.1 Why SRP Matters

As applications grow, so does the complexity of their codebases. Without clear separation of concerns, classes can become "God classes" — bloated, entangled blocks of logic that try to do too much. These are difficult to test, reuse, and extend. Adhering to SRP helps:

- **Isolate changes**: When a class has one reason to change, changes in requirements won't cause ripple effects.
- **Improve readability**: Focused classes are easier to understand and document.
- **Enhance testability**: Unit testing becomes more straightforward when responsibilities are well defined.
- **Promote reuse**: Classes with narrow responsibilities can be reused in different contexts.

11.1.2 Identifying Multiple Responsibilities

You can identify SRP violations by looking at the **reasons a class might change**. For example, consider a class that:

- Validates user input
- Logs to a file
- Saves data to a database

Each of these responsibilities could change independently — logging format might change, database schema could evolve, or validation rules could be updated. Keeping all that logic in one class violates SRP.

11.1.3 Example: A Class Violating SRP

Let's consider a simple `ReportManager` class:

```
public class ReportManager {
    public void generateReport() {
        // Logic to generate report
    }

    public void saveToFile(String reportData) {
        // Logic to save report to file
    }

    public void sendEmail(String reportData) {
        // Logic to send the report via email
    }
}
```

This class handles:

1. Generating reports
2. Saving reports
3. Emailing reports

These are three separate responsibilities. A change in email logic shouldn't impact report generation.

11.1.4 Refactoring to Follow SRP

We can refactor the above into focused classes:

```
public class ReportGenerator {
    public String generateReport() {
        // Logic to generate report
        return "report content";
    }
}
```

```
public class ReportSaver {
    public void saveToFile(String reportData) {
        // Logic to save to file
    }
}
```

```
public class EmailSender {
    public void sendEmail(String reportData) {
        // Logic to send email
    }
}
```

Now, each class has a single responsibility and can change independently. We can combine them in a coordinator class:

```
public class ReportService {
    private ReportGenerator generator = new ReportGenerator();
    private ReportSaver saver = new ReportSaver();
    private EmailSender sender = new EmailSender();

    public void processReport() {
        String report = generator.generateReport();
        saver.saveToFile(report);
        sender.sendEmail(report);
    }
}
```

This design is modular, extensible, and easy to maintain.

Full runnable code:

```
// ReportGenerator.java
class ReportGenerator {
    public String generateReport() {
        System.out.println("Generating report...");
        return "Report Content: Sales Data for Q2";
    }
}

// ReportSaver.java
class ReportSaver {
    public void saveToFile(String reportData) {
        System.out.println("Saving report to file...");
        // Simulate file saving
        System.out.println("Report saved: " + reportData);
    }
}

// EmailSender.java
class EmailSender {
    public void sendEmail(String reportData) {
        System.out.println("Sending report via email...");
        // Simulate sending email
        System.out.println("Email sent with report: " + reportData);
    }
}

// ReportService.java (Coordinator)
class ReportService {
    private ReportGenerator generator = new ReportGenerator();
    private ReportSaver saver = new ReportSaver();
    private EmailSender sender = new EmailSender();

    public void processReport() {
        String report = generator.generateReport();
        saver.saveToFile(report);
        sender.sendEmail(report);
    }
}
```

```java
// Main.java
public class Main {
    public static void main(String[] args) {
        ReportService service = new ReportService();
        service.processReport();
    }
}
```

11.1.5 SRP and Testing

With SRP in place, unit tests become much easier to write and maintain:

- You can test `ReportGenerator` without worrying about file I/O or email configuration.
- You can mock dependencies in `ReportService` for integration-style tests.
- Changes in one component don't break unrelated tests.

For instance, if the email format changes, only `EmailSenderTest` needs to be updated.

11.1.6 SRP and Debugging

When a bug is reported in the way reports are saved, you can directly inspect `ReportSaver`, knowing it's the only class handling that concern. Without SRP, you might have to wade through a large class with unrelated logic, making bug tracking more difficult and time-consuming.

11.1.7 Conclusion

The Single Responsibility Principle encourages **modular, focused class design**. When applied effectively, it leads to software that is **easier to maintain, test, and evolve**. It may initially seem like SRP leads to more classes, but this modularity is a strength, not a weakness. By assigning one responsibility per class, your code becomes a flexible and resilient foundation for long-term software development.

Thought Exercise:

> Look at a class from one of your recent projects. How many different things is it responsible for? Could you split it into smaller, more focused classes to better follow SRP?

11.2 Open/Closed Principle

The **Open/Closed Principle (OCP)** is the second principle in the SOLID acronym and a cornerstone of maintainable software design. It states:

> **"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."** — Bertrand Meyer

At its core, the Open/Closed Principle advocates that once a class is written and tested, it should not be changed. Instead, new functionality should be added by **extending** the existing code rather than **modifying** it.

11.2.1 Why the Open/Closed Principle Matters

When software requirements evolve, developers face a key question: how can we adapt existing code without breaking it? The Open/Closed Principle provides an answer by encouraging designs that allow for **behavioral extensions** through polymorphism rather than rewriting working code.

By doing so, it:

- Preserves existing, tested logic
- Prevents introducing regressions
- Encourages modularity and plug-and-play architecture
- Improves code readability and predictability

11.2.2 Classic Example: Without OCP

Consider a basic `PaymentProcessor` class:

```
public class PaymentProcessor {
    public void processPayment(String type) {
        if (type.equals("credit")) {
            // logic for credit card
        } else if (type.equals("paypal")) {
            // logic for PayPal
        } else {
            throw new IllegalArgumentException("Unsupported payment type");
        }
    }
}
```

If a new payment type (e.g., Bitcoin) is introduced, this class must be modified. Every modification increases the risk of bugs and tightens coupling.

11.2.3 Applying OCP with Abstraction

We can refactor the `PaymentProcessor` to comply with OCP using an interface:

```
public interface PaymentMethod {
    void pay();
}
```

Now, define implementations:

```
public class CreditCardPayment implements PaymentMethod {
    public void pay() {
        System.out.println("Processing credit card payment.");
    }
}

public class PayPalPayment implements PaymentMethod {
    public void pay() {
        System.out.println("Processing PayPal payment.");
    }
}
```

And a flexible processor:

```
public class PaymentProcessor {
    public void processPayment(PaymentMethod method) {
        method.pay();
    }
}
```

To add support for a new type like Bitcoin, you simply implement the `PaymentMethod` interface:

```
public class BitcoinPayment implements PaymentMethod {
    public void pay() {
        System.out.println("Processing Bitcoin payment.");
    }
}
```

No changes to the `PaymentProcessor` class are necessary. The system is now **open for extension** (new payment methods) but **closed for modification** (no touching of core logic).

Full runnable code:

```
// PaymentMethod.java
interface PaymentMethod {
    void pay();
}

// CreditCardPayment.java
class CreditCardPayment implements PaymentMethod {
    public void pay() {
        System.out.println("Processing credit card payment.");
    }
```

```
}

// PayPalPayment.java
class PayPalPayment implements PaymentMethod {
    public void pay() {
        System.out.println("Processing PayPal payment.");
    }
}

// BitcoinPayment.java
class BitcoinPayment implements PaymentMethod {
    public void pay() {
        System.out.println("Processing Bitcoin payment.");
    }
}

// PaymentProcessor.java
class PaymentProcessor {
    public void processPayment(PaymentMethod method) {
        method.pay();
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();

        PaymentMethod creditCard = new CreditCardPayment();
        PaymentMethod paypal = new PayPalPayment();
        PaymentMethod bitcoin = new BitcoinPayment();

        processor.processPayment(creditCard); // Output: Processing credit card payment.
        processor.processPayment(paypal);     // Output: Processing PayPal payment.
        processor.processPayment(bitcoin);    // Output: Processing Bitcoin payment.
    }
}
```

11.2.4 Tools for Applying OCP

Java provides several tools to help developers apply OCP:

- **Abstract classes and interfaces**: Define a common contract and defer behavior to subclasses.
- **Polymorphism**: Enables dynamic method resolution to invoke behavior appropriate for the object at runtime.
- **Dependency injection**: Decouples class dependencies, making it easier to swap implementations.

11.2.5 Benefits of the Open/Closed Principle

- **Improved Maintainability**: You don't need to dig into existing logic to add new features.
- **Increased Scalability**: New behaviors are added as discrete, testable components.
- **Reduced Risk of Breakage**: Since existing code remains untouched, the chance of introducing new bugs is minimized.
- **Flexible Architecture**: OCP naturally leads to plugin-like designs and dynamic behavior.

11.2.6 Trade-Offs and Misuse

While OCP provides many advantages, there are some trade-offs:

- **Over-engineering**: Prematurely abstracting everything "just in case" can lead to unnecessary complexity.
- **Too many small classes**: Excessive application of OCP can fragment the codebase and hinder readability.
- **Rigid hierarchy**: Overreliance on inheritance can create inflexible and tightly coupled hierarchies.

Best Practice: Apply OCP when you anticipate **frequent changes in behavior**, especially in parts of the system that vary independently. Use abstraction only when it solves a real problem, not preemptively.

11.2.7 Conclusion

The Open/Closed Principle is a guiding philosophy for designing systems that are resilient to change. By programming to interfaces and abstracting varying behavior, developers can **extend systems without modifying existing code**, resulting in designs that are **robust, flexible, and easier to maintain**.

Thought Prompt:

> Think of a class you've modified multiple times to add new logic. Could it be refactored using OCP to delegate responsibilities to extendable components?

11.3 Liskov Substitution Principle

The **Liskov Substitution Principle (LSP)** is the third principle in the SOLID acronym and plays a pivotal role in designing robust, extensible object-oriented systems. It was introduced by Barbara Liskov in 1987 and can be formally stated as:

> *"If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering the correctness of the program."*

In simpler terms, **a subclass should be usable anywhere its superclass is expected without causing incorrect behavior**. This principle ensures the integrity of inheritance hierarchies and enables polymorphism to work as intended.

11.3.1 Why Substitutability Matters

Polymorphism is one of the key advantages of object-oriented programming. It allows developers to write flexible, reusable code that operates on general types (like interfaces or abstract classes), while relying on the behavior of concrete implementations at runtime.

If a subclass breaks the expected behavior of a superclass, it compromises the reliability of that substitution. This leads to subtle bugs, tight coupling, and violations of expected contracts.

11.3.2 Understanding Behavioral Contracts

To honor LSP, a subclass must adhere to the **behavioral contract** defined by its superclass. That means:

- **Preconditions** (requirements before a method runs) must not be strengthened.
- **Postconditions** (guarantees after a method runs) must not be weakened.
- **Invariants** (class-level consistency rules) must be preserved.

Let's see how this works through examples.

11.3.3 Example: Obeying LSP

Suppose we have a base class `Bird`:

```
public class Bird {
    public void fly() {
        System.out.println("The bird flies.");
```

```
    }
}
```

Now, we subclass it with `Sparrow`:

```
public class Sparrow extends Bird {
    @Override
    public void fly() {
        System.out.println("The sparrow flutters and flies.");
    }
}
```

This subclass doesn't change the contract. A `Sparrow` can be treated as a `Bird` and it still behaves appropriately. It **honors** the expectations set by the superclass.

```
public void makeBirdFly(Bird b) {
    b.fly();
}
```

This will work correctly whether `b` is a `Bird` or a `Sparrow`.

11.3.4 Example: Violating LSP

Now let's add another subclass:

```
public class Ostrich extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Ostriches can't fly!");
    }
}
```

This subclass violates the contract of `Bird`. While `Bird` promises that `fly()` is a valid operation, `Ostrich` breaks that promise. Substituting an `Ostrich` for a `Bird` could crash the program:

```
Bird b = new Ostrich();
b.fly(); // Throws exception - violates LSP
```

Here, the behavior of `Ostrich` contradicts the expectation that all `Bird`s can fly, making it an improper subclass.

11.3.5 How to Avoid Violating LSP

- **Refactor incorrectly modeled hierarchies**: If `Ostrich` cannot fly, maybe `Bird` shouldn't have a `fly()` method in the base class.
  - Instead, extract a `FlyingBird` subclass and separate flight logic there.
- **Favor composition over inheritance**: If behavior varies drastically between types, consider using strategy patterns or interfaces instead.
- **Use interfaces to enforce appropriate contracts**: Let each type declare only the capabilities it truly supports.

Example refactoring:

```
public interface Flyable {
    void fly();
}

public class Sparrow implements Flyable {
    public void fly() {
        System.out.println("Sparrow flies.");
    }
}

public class Ostrich {
    // Doesn't implement Flyable - correct modeling
}
```

Now, a method that requires a flying creature can safely depend on `Flyable`:

```
public void makeFly(Flyable f) {
    f.fly();
}
```

11.3.6 Consequences of Violating LSP

Ignoring the Liskov Substitution Principle leads to:

- Fragile code with unexpected runtime exceptions
- Breakage in client code that trusts the base class behavior
- Difficulty in testing, reusing, or extending systems

Maintaining the behavioral contract is essential for scalable and maintainable design.

11.3.7 Summary

The **Liskov Substitution Principle** protects your code from the silent failures of poorly designed inheritance. By ensuring that subclasses remain faithful to the behavior promised by their superclasses, you build **trustworthy**, **predictable**, and **modular** systems. When inheritance doesn't naturally fit, remember: **composition and interfaces** offer better alternatives.

11.4 Interface Segregation Principle

The **Interface Segregation Principle (ISP)** is the fourth principle of SOLID design and addresses the size and design of interfaces. It states:

> **"Clients should not be forced to depend on methods they do not use."**

In other words, **interfaces should be specific and focused on what a client needs**, rather than being large and general-purpose. When interfaces become too large—sometimes referred to as "fat" interfaces—they tend to force implementing classes to define methods that may be irrelevant or unused. This leads to bloated, brittle designs and breaks the goal of modularity.

11.4.1 The Problem with Fat Interfaces

Consider the following interface:

```
public interface Machine {
    void print();
    void scan();
    void fax();
}
```

This `Machine` interface bundles multiple responsibilities. Now suppose you have a simple printer that only prints documents:

```
public class BasicPrinter implements Machine {
    @Override
    public void print() {
        System.out.println("Printing document...");
    }

    @Override
    public void scan() {
        // Not supported
        throw new UnsupportedOperationException("Scan not supported");
    }
```

```
    @Override
    public void fax() {
        // Not supported
        throw new UnsupportedOperationException("Fax not supported");
    }
}
```

The `BasicPrinter` is being **forced to implement methods it doesn't support**, violating ISP. This can lead to runtime errors, cluttered code, and maintenance challenges.

11.4.2 Applying ISP: Split Interfaces by Responsibility

To fix this, we can break the `Machine` interface into **more specific interfaces**, each aligned to a particular responsibility:

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

public interface Scanner {
    void scan();
}

public interface Fax {
    void fax();
}
```

Now, `BasicPrinter` can implement only what it supports:

```
public class BasicPrinter implements Printer {
    @Override
    public void print() {
        System.out.println("Printing document...");
    }
}
```

A more advanced device can implement multiple interfaces:

```
public class MultiFunctionPrinter implements Printer, Scanner, Fax {
    @Override
    public void print() {
        System.out.println("Printing...");
    }

    @Override
    public void scan() {
        System.out.println("Scanning...");
    }

    @Override
```

```
    public void fax() {
        System.out.println("Faxing...");
    }
}
```

This approach offers cleaner code and stronger contracts—**each class commits only to what it can actually do**.

11.4.3 Benefits of Interface Segregation

- **Improved Flexibility**: Clients can pick and choose which interfaces to implement without dealing with unrelated methods.
- **Reduced Coupling**: Smaller interfaces isolate dependencies, making changes less risky.
- **Simpler Testing**: It's easier to test focused behaviors individually when classes implement smaller interfaces.
- **Better Reusability**: Specific interfaces can be reused across unrelated contexts without pulling in unnecessary behavior.

11.4.4 Practical Example: Animal Behaviors

Imagine an interface like this:

```
public interface Animal {
    void walk();
    void swim();
    void fly();
}
```

Clearly, not all animals do all these things. Let's apply ISP:

```
public interface Walkable {
    void walk();
}

public interface Swimmable {
    void swim();
}

public interface Flyable {
    void fly();
}
```

Now we can create animals with accurate capabilities:

```
public class Dog implements Walkable, Swimmable {
    public void walk() {
        System.out.println("Dog walking");
    }

    public void swim() {
        System.out.println("Dog swimming");
    }
}

public class Bird implements Walkable, Flyable {
    public void walk() {
        System.out.println("Bird walking");
    }

    public void fly() {
        System.out.println("Bird flying");
    }
}
```

Each class implements only the behaviors relevant to it—no unnecessary baggage.

Full runnable code:

```
// Walkable.java
interface Walkable {
    void walk();
}

// Swimmable.java
interface Swimmable {
    void swim();
}

// Flyable.java
interface Flyable {
    void fly();
}

// Dog.java
class Dog implements Walkable, Swimmable {
    public void walk() {
        System.out.println("Dog walking");
    }

    public void swim() {
        System.out.println("Dog swimming");
    }
}

// Bird.java
class Bird implements Walkable, Flyable {
    public void walk() {
        System.out.println("Bird walking");
    }

    public void fly() {
```

```
        System.out.println("Bird flying");
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.walk();   // Output: Dog walking
        dog.swim();   // Output: Dog swimming

        Bird bird = new Bird();
        bird.walk();  // Output: Bird walking
        bird.fly();   // Output: Bird flying
    }
}
```

11.4.5 Conclusion

The Interface Segregation Principle reinforces the value of **focused, client-specific interfaces**. By avoiding fat interfaces and breaking responsibilities into smaller, composable interfaces, your code becomes more **modular, reusable, and easier to maintain**. ISP promotes **flexible architecture** that's easier to scale and refactor as systems grow in complexity.

Before creating an interface, always ask: *Who will use this, and which methods do they actually need?* Let that answer guide your design.

11.5 Dependency Inversion Principle

The **Dependency Inversion Principle (DIP)** is the fifth and final principle in the SOLID design framework. It focuses on **restructuring dependencies** in a way that leads to **flexible and maintainable software systems**.

> **"High-level modules should not depend on low-level modules. Both should depend on abstractions." "Abstractions should not depend on details. Details should depend on abstractions."**

11.5.1 The Problem with Direct Dependencies

Traditionally, in procedural or tightly coupled designs, high-level modules (such as business logic) directly depend on low-level modules (like file readers, database connectors, or APIs).

This setup makes the codebase brittle—**any change to a low-level detail ripples upward**, breaking business logic and creating rigid structures.

For example:

```
public class ReportService {
    private FileWriter fileWriter = new FileWriter();

    public void generateReport() {
        // Logic to generate report
        fileWriter.write("report.txt", "Report content");
    }
}
```

Here, `ReportService` (a high-level module) depends directly on `FileWriter` (a low-level module). If we wanted to switch to a `DatabaseWriter` or a `CloudStorageWriter`, we'd have to modify `ReportService`—clearly a violation of the **Open/Closed Principle** too.

11.5.2 DIP: Invert the Dependency with Abstractions

To apply DIP, we **introduce an interface or abstract class** that defines the contract for writing:

```
public interface Writer {
    void write(String destination, String content);
}
```

Then the low-level classes implement the abstraction:

```
public class FileWriter implements Writer {
    @Override
    public void write(String destination, String content) {
        System.out.println("Writing to file: " + destination);
        // File writing logic
    }
}

public class DatabaseWriter implements Writer {
    @Override
    public void write(String destination, String content) {
        System.out.println("Writing to database: " + content);
        // DB logic
    }
}
```

Now, `ReportService` depends only on the `Writer` interface:

```
public class ReportService {
    private Writer writer;
```

```
    public ReportService(Writer writer) {
        this.writer = writer;
    }

    public void generateReport() {
        String report = "Report content";
        writer.write("report.txt", report);
    }
}
```

This **decouples the high-level logic** from low-level details. Swapping implementations becomes trivial and doesn't require modifying core logic:

```
Writer writer = new FileWriter();
ReportService service = new ReportService(writer);
service.generateReport();
```

11.5.3 Benefits of DIP

Looser Coupling

By depending on abstractions, your classes are **not hard-wired** to specific implementations. This promotes flexibility in choosing and changing behaviors.

Improved Testability

With abstractions, it's easy to inject **mock or stub implementations** during unit testing:

```
public class MockWriter implements Writer {
    public void write(String destination, String content) {
        System.out.println("Mock writer used for test.");
    }
}
```

This enables you to test `ReportService` in isolation without needing actual files or databases.

Full runnable code:

```
// Writer.java
interface Writer {
    void write(String destination, String content);
}

// FileWriter.java
class FileWriter implements Writer {
    @Override
    public void write(String destination, String content) {
        System.out.println("Writing to file: " + destination);
        System.out.println("Content: " + content);
    }
}
```

```java
// DatabaseWriter.java
class DatabaseWriter implements Writer {
    @Override
    public void write(String destination, String content) {
        System.out.println("Writing to database: " + content);
    }
}

// ReportService.java
class ReportService {
    private Writer writer;

    public ReportService(Writer writer) {
        this.writer = writer;
    }

    public void generateReport() {
        String report = "Report content";
        writer.write("report.txt", report);
    }
}

// MockWriter.java (for testing)
class MockWriter implements Writer {
    public void write(String destination, String content) {
        System.out.println("Mock writer used for test. Skipping actual write.");
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        // Real usage
        Writer fileWriter = new FileWriter();
        ReportService fileReportService = new ReportService(fileWriter);
        fileReportService.generateReport();

        System.out.println();

        // Test usage
        Writer mockWriter = new MockWriter();
        ReportService mockService = new ReportService(mockWriter);
        mockService.generateReport();
    }
}
```

Scalability and Maintenance

Adding a new type of writer (e.g., `CloudStorageWriter`) doesn't affect existing logic. The system **scales horizontally**, making it future-proof and easier to extend.

Aligns with Other SOLID Principles

- Works well with **Open/Closed Principle**—code is open to extension via new implementations.
- Encourages **Single Responsibility** by separating concerns.

- Promotes **Interface Segregation** through fine-grained abstractions.

11.5.4 Dependency Injection as a Realization of DIP

In practice, DIP often manifests through **Dependency Injection (DI)**—a technique where dependencies (e.g., implementations of interfaces) are passed to a class rather than instantiated within it.

There are three common types of DI:

- **Constructor Injection** (as seen in our example)
- **Setter Injection**
- **Interface Injection**

Modern frameworks like **Spring** automate this process using annotations like `@Autowired` to inject dependencies.

11.5.5 Real-World Analogy

Think of a high-level kitchen appliance (e.g., a **blender**) that can accept different **attachments** (blades, jars) via a standardized socket. The blender doesn't care what brand or type of blade is used—as long as it fits the contract (interface). This is **Dependency Inversion in action**: the blender (high-level) and blade (low-level) depend on a common contract (socket shape/interface).

11.5.6 Conclusion

The **Dependency Inversion Principle** encourages **designing around abstractions**, rather than concrete implementations. This practice leads to software that is easier to test, extend, and maintain. By combining DIP with dependency injection techniques and solid interface design, Java developers can build robust, modular, and adaptable systems—essential traits for modern software engineering.

Before you instantiate a class inside your core logic, ask yourself: *Can I depend on an interface instead?* That question lies at the heart of mastering DIP.

Chapter 12.

DRY, KISS, YAGNI, and Other Principles

1. Avoiding Code Duplication (DRY)
2. Keeping It Simple and Modular (KISS)
3. Avoiding Premature Optimization (YAGNI)

12 DRY, KISS, YAGNI, and Other Principles

12.1 Avoiding Code Duplication (DRY)

One of the most fundamental principles in software engineering is **DRY**, which stands for **Don't Repeat Yourself**. Coined by Andy Hunt and Dave Thomas in *The Pragmatic Programmer*, the DRY principle emphasizes that *"every piece of knowledge must have a single, unambiguous, authoritative representation within a system."* In simpler terms, **avoid code duplication**—not just in lines of code, but in logic, knowledge, and behavior.

12.1.1 Why Duplication Is Harmful

Duplication increases the risk of bugs, bloated codebases, and inconsistent behavior. If the same logic is copied across multiple classes or methods, any change to that logic must be applied consistently in all places. Miss just one, and you've introduced a potential bug. Duplication also makes code harder to read, maintain, and extend.

For example, consider the following duplicate code across two methods:

```
public double calculateDiscountedPrice(double price) {
    double discount = price * 0.1;
    return price - discount;
}

public double calculateLoyaltyPrice(double price) {
    double discount = price * 0.1;
    return price - discount;
}
```

Both methods apply the same discount logic. If business rules change tomorrow (e.g., the discount becomes 15%), both methods would need to be updated. This is inefficient and error-prone.

12.1.2 Common Causes of Duplication

Duplication can sneak into codebases through:

- **Copy-pasting code** between classes or methods
- **Similar algorithms** written independently
- **Poor abstraction** or overuse of procedural patterns
- **Lack of refactoring** after implementing features quickly

In object-oriented systems, it's especially common to find the same method logic implemented in multiple classes due to improper use of inheritance or composition.

12.1.3 Refactoring to Eliminate Duplication

There are several strategies to remove duplication effectively:

Extract Common Logic into Methods

Refactor repeated logic into a reusable method:

```
private double applyDiscount(double price) {
    return price - (price * 0.1);
}

public double calculateDiscountedPrice(double price) {
    return applyDiscount(price);
}

public double calculateLoyaltyPrice(double price) {
    return applyDiscount(price);
}
```

Use Inheritance for Shared Behavior

If multiple classes share behavior, move it to a superclass:

```
public class Product {
    protected double applyDiscount(double price) {
        return price - (price * 0.1);
    }
}

public class Electronics extends Product {
    public double getPriceAfterDiscount(double price) {
        return applyDiscount(price);
    }
}

public class Clothing extends Product {
    public double getPriceAfterDiscount(double price) {
        return applyDiscount(price);
    }
}
```

Inheritance helps promote reuse, but use it judiciously to avoid the fragile base class problem.

Prefer Composition for Reusable Logic

When behavior doesn't belong to a hierarchy, create a helper or utility class:

```
public class DiscountCalculator {
    public double applyDiscount(double price) {
        return price - (price * 0.1);
    }
}

public class Order {
```

```
    private DiscountCalculator calculator = new DiscountCalculator();

    public double getFinalPrice(double price) {
        return calculator.applyDiscount(price);
    }
}
```

Composition allows you to **share behavior across unrelated classes** without tying them into a rigid inheritance chain.

12.1.4 When DRY Can Go Too Far

While DRY is important, **overzealous abstraction can lead to confusion**. Sometimes what *looks* like duplication is actually **similar but contextually distinct logic**. If you try to unify two seemingly duplicated methods that serve different purposes, the resulting abstraction may be awkward, harder to maintain, and violate other principles like the **Single Responsibility Principle (SRP)**.

> **Rule of Thumb**: Only refactor duplication when the logic and intent are truly the same and likely to evolve together.

For example, trying to generalize two discount formulas that behave differently based on customer type may obscure business logic.

12.1.5 Summary

The DRY principle helps build cleaner, more maintainable Java code by reducing redundancy and centralizing logic. By identifying duplicated code early—whether through manual review or static analysis tools—you can refactor into smaller, reusable components using methods, inheritance, or composition. However, always balance DRY with clarity and intent. A smart developer knows not only *how* to eliminate duplication but also *when* to leave it alone.

Reflective Tip: As you review your codebase, ask: *If this logic changes, how many places do I have to update?* If the answer is more than one, you likely have a DRY violation.

12.2 Keeping It Simple and Modular (KISS)

In software design, simplicity is not a luxury—it's a necessity. The **KISS principle**, which stands for **Keep It Simple, Stupid**, is a timeless design philosophy that encourages developers to **avoid unnecessary complexity**. The core idea is that most systems work best when they are kept simple rather than made complicated. Therefore, simplicity should

be a primary goal in design, and unnecessary complexity should be avoided.

12.2.1 Why Simplicity Leads to Maintainability

Simple code is **easier to read, understand, test, and maintain**. It reduces the cognitive load on developers who must work with the code weeks, months, or years after it was written. Complex code, on the other hand, can lead to bugs, misunderstandings, and costly rework.

Take the following two Java methods for checking if a number is even:

Overengineered version:

```
public boolean isEven(int number) {
    return ((number / 2) * 2 == number) ? true : false;
}
```

Simple version:

```
public boolean isEven(int number) {
    return number % 2 == 0;
}
```

Both methods perform the same task, but the latter is cleaner and more intuitive. The complex version adds no value—only confusion.

12.2.2 Modular Design: Simplicity through Separation

Keeping it simple often goes hand-in-hand with **modular design**. Modular code divides responsibilities among **small, focused classes** that communicate through **clear interfaces**. This separation of concerns makes code easier to understand and evolve.

Consider a payment processing system. A non-modular design might cram all logic into a single class:

```
public class PaymentProcessor {
    public void processPayment(String method, double amount) {
        if (method.equals("credit")) {
            // credit payment logic
        } else if (method.equals("paypal")) {
            // PayPal logic
        } else if (method.equals("bank")) {
            // bank transfer logic
        }
    }
}
```

This class is **hard to test, extend, or maintain**. Now compare it to a modular version:

```
public interface PaymentMethod {
    void pay(double amount);
}

public class CreditCardPayment implements PaymentMethod {
    public void pay(double amount) {
        // credit card logic
    }
}

public class PayPalPayment implements PaymentMethod {
    public void pay(double amount) {
        // PayPal logic
    }
}

public class PaymentProcessor {
    public void process(PaymentMethod method, double amount) {
        method.pay(amount);
    }
}
```

This design is simpler **because it's modular**. Each class has one responsibility, and new payment methods can be added without changing existing code.

12.2.3 Simplicity vs Functionality: Finding the Balance

Keeping things simple doesn't mean stripping out essential functionality or avoiding abstraction. Simplicity means writing the **most direct, clear, and logical solution** to a problem. Over-abstraction—adding unnecessary design layers or generalization—can violate KISS.

For instance, wrapping a single method call in five classes of delegation adds complexity without benefit. Likewise, introducing a full-blown strategy pattern for toggling a boolean flag is often overkill. Use patterns and principles wisely, not dogmatically.

12.2.4 Best Practices for KISS and Modularity

- **Write small methods and classes**: Each should do one thing well.
- **Use meaningful names**: Clear naming reduces the need for comments or extra logic.
- **Avoid clever code**: Code that's tricky to understand today will become unmaintainable tomorrow.
- **Refactor regularly**: As systems evolve, refactor to keep code readable and manageable.
- **Design APIs with clarity**: Simple, well-defined interfaces make modular code easier to plug together.

12.2.5 Conclusion

The KISS principle reminds us that **clarity is better than cleverness**. By choosing straightforward solutions and designing modular systems, developers produce code that is easier to understand, test, and evolve. In Java and object-oriented design, this often means composing systems with **clear interfaces, small focused classes, and predictable behaviors**. Simplicity doesn't mean less power—it means more control and reliability in the long run.

> **Thought Prompt:** When you write code, ask yourself: "Would another developer understand this in 30 seconds?" If the answer is no, it's time to simplify.

12.3 Avoiding Premature Optimization (YAGNI)

In software development, it's tempting to think ahead—to imagine every possible feature, every potential performance bottleneck, and to begin writing code for situations that may never happen. But a key principle in agile, pragmatic programming is **YAGNI**, which stands for **"You Aren't Gonna Need It."**

12.3.1 What Is YAGNI?

YAGNI is a design principle that reminds developers **not to implement something until it is actually necessary.** It was popularized in the context of agile development and extreme programming (XP), where quick iteration, customer feedback, and adaptability are core values. YAGNI encourages developers to **resist the urge to build features or abstractions "just in case."**

> **Definition**: *"You Aren't Gonna Need It"* means *don't add functionality until it is absolutely required.*

The principle advocates for focusing on what the system needs **right now**, not what it might need later.

12.3.2 The Cost of Premature Optimization

One of the most common violations of YAGNI occurs during **premature optimization**—when developers try to improve performance or scalability before there's any evidence that it's needed.

Consider the classic advice from Donald Knuth:

"Premature optimization is the root of all evil."

Why is this a problem? Because early optimization often introduces:

- **Complex code** that's harder to understand and maintain.
- **Wasted development time** on features or improvements that may never be used.
- **Inflexibility**, making it harder to change or extend the system later.
- **Incorrect assumptions** about performance bottlenecks.

Let's look at a simple example:

12.3.3 Example of Premature Optimization

Suppose you're writing a feature to store and retrieve user comments. A basic, clear implementation might store comments in a `List<String>`.

```
List<String> comments = new ArrayList<>();
comments.add("Nice article!");
```

But you preemptively decide that users might leave millions of comments, and build a caching system, a database connection pool, and asynchronous message queues. This not only bloats your codebase but also **delays delivery** and introduces new points of failure—**before any real performance problem exists.**

12.3.4 Prioritize Simplicity and Correctness First

A better approach is to:

1. **Implement the simplest solution** that works correctly.
2. **Measure performance** under realistic loads.
3. **Optimize only when benchmarks show actual problems.**

In the previous example, the simple `ArrayList` may be sufficient for months. If performance issues arise later, profiling tools can help identify exactly where the bottlenecks are—maybe it's not the comment storage at all.

12.3.5 Strategies to Delay Optimization Wisely

Here are ways to follow YAGNI without sacrificing long-term quality:

- **Write clean, modular code**: Good structure makes it easier to optimize later.
- **Avoid speculative features**: Only add capabilities when there's a real need.

- **Measure, don't guess**: Use profiling tools to validate performance assumptions.
- **Keep code adaptable**: Favor flexibility and readability over clever tricks.
- **Focus on delivering value**: Working software today is more valuable than hypothetical speed later.

For instance, if you're worried about slow sorting in large datasets, use Java's built-in `Collections.sort()` for now. If the list grows and slows down the application, you can then refactor to a more performant data structure or algorithm—**when it's justified**.

12.3.6 Conclusion

YAGNI is not about ignoring future concerns—it's about **not solving problems you don't have yet**. In modern Java development, it means **prioritizing simplicity and correctness**, especially during early iterations. When performance or complexity becomes a real concern, you'll have the tools, code clarity, and context to optimize effectively.

> **Thought Prompt**: The next time you consider adding a feature "just in case," ask yourself: "Is this solving a real, immediate problem?" If not, you probably aren't gonna need it—yet.

Chapter 13.

Design by Contract and Defensive Programming

1. Preconditions, Postconditions, Invariants
2. Assertions
3. Defensive Copies

13 Design by Contract and Defensive Programming

13.1 Preconditions, Postconditions, Invariants

In the pursuit of building reliable and maintainable software, **Design by Contract (DbC)** stands out as a powerful methodology to ensure program correctness. Introduced by Bertrand Meyer with the Eiffel programming language, Design by Contract establishes formal agreements—or *contracts*—between software components, specifying their mutual obligations and benefits. This methodology focuses on clearly defining what a method expects, guarantees, and maintains, reducing ambiguity and minimizing bugs.

At the core of Design by Contract are three fundamental concepts: **preconditions**, **postconditions**, and **invariants**. These act as rules that must be respected by both the caller of a method and the method implementation itself.

13.1.1 Preconditions: What Must Be True Before Execution

A **precondition** is a condition or requirement that must hold *before* a method executes. It defines the responsibilities of the caller—essentially what the caller guarantees when invoking the method. If a precondition is violated, the method is free from any obligation to behave correctly, meaning the caller is misusing the API.

Example: Consider a method that calculates the square root of a number.

```
public double sqrt(double value) {
    if (value < 0) {
        throw new IllegalArgumentException("Value must be non-negative");
    }
    return Math.sqrt(value);
}
```

Here, the precondition is that `value` must be non-negative. The caller must ensure this before calling `sqrt()`. If this precondition is violated, the method throws an exception indicating improper use.

13.1.2 Postconditions: What Must Be True After Execution

A **postcondition** defines what the method guarantees to be true *after* it finishes execution, provided that the preconditions were met. It specifies the expected state or output that the caller can rely on.

Example: Continuing with the `sqrt` method, a postcondition might be that the returned result is always non-negative.

```java
public double sqrt(double value) {
    if (value < 0) {
        throw new IllegalArgumentException("Value must be non-negative");
    }
    double result = Math.sqrt(value);
    assert result >= 0 : "Postcondition violated: result is negative";
    return result;
}
```

This postcondition asserts that the returned value will be greater than or equal to zero. While Java doesn't enforce contracts natively, developers can use assertions or explicit checks to verify postconditions during testing.

13.1.3 Invariants: Conditions That Always Hold True

An **invariant** is a condition that must always be true for a class's instance—before and after any public method executes. Invariants represent the consistent state of an object, ensuring its internal data remains valid throughout its lifecycle.

Example: Imagine a simple `BankAccount` class:

```java
public class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        if (initialBalance < 0) {
            throw new IllegalArgumentException("Initial balance cannot be negative");
        }
        this.balance = initialBalance;
    }

    public void deposit(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Deposit must be positive");
        }
        balance += amount;
        assert balance >= 0 : "Invariant violated: balance negative";
    }

    public void withdraw(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Withdrawal must be positive");
        }
        if (amount > balance) {
            throw new IllegalArgumentException("Insufficient funds");
        }
        balance -= amount;
        assert balance >= 0 : "Invariant violated: balance negative";
    }

    public double getBalance() {
        return balance;
```

```
    }
}
```

Here, the **invariant** is that `balance` must never be negative. This condition is checked after each operation that modifies the state, ensuring the object remains valid.

13.1.4 Benefits of Using Contracts in APIs

By clearly defining preconditions, postconditions, and invariants, APIs become more **transparent** and **robust**:

- **Clear expectations**: Callers know exactly what input values are valid and what to expect in return.
- **Error localization**: Violations of contracts help detect bugs early, isolating whether the caller or the callee is at fault.
- **Documentation alignment**: Contracts serve as formal documentation, reducing misunderstandings and misuse.
- **Easier debugging and maintenance**: Contract violations are explicit, aiding debugging and improving code maintainability.

When contracts are violated, exceptions or assertion failures provide immediate feedback, preventing obscure bugs that may only appear much later in the system.

13.1.5 Contracts and Formal Specifications

Design by Contract moves beyond informal comments and verbal agreements by establishing **formal specifications** that can be enforced programmatically or at least systematically tested. While Java does not natively enforce contracts, tools and libraries (such as the Java Modeling Language (JML) or runtime assertions) can help integrate DbC practices.

Formal contracts encourage developers to think rigorously about every method's obligations, promoting disciplined and predictable designs. They are especially valuable in large codebases or team environments, where clear interfaces reduce integration errors.

13.1.6 Summary

- **Preconditions** define what a method expects before it runs.
- **Postconditions** specify what a method guarantees after execution.
- **Invariants** maintain consistent object state throughout its life.

Together, these contracts form a foundation for **correctness**, **clarity**, and **reliability** in Java software design. Embracing these principles leads to APIs that are easier to use, debug, and maintain—helping you write code that you and others can trust.

If you've ever wondered how to ensure your methods and classes behave exactly as intended, Design by Contract provides a systematic approach to formalize those expectations. It's a powerful mindset that transforms vague assumptions into explicit agreements, helping prevent bugs before they happen.

13.2 Assertions

In Java programming, **assertions** serve as a built-in mechanism for testing assumptions during development. They play a key role in **Design by Contract (DbC)** by allowing developers to formally verify **preconditions**, **postconditions**, and **invariants**—the essential agreements between a method and its caller. Although assertions are not intended for handling user or runtime errors, they are a valuable tool for improving software correctness and robustness during development and testing phases.

13.2.1 What Are Assertions?

An **assertion** is a statement in code that asserts a condition must be true at a specific point of execution. If the assertion fails—meaning the condition evaluates to `false`—the Java Virtual Machine (JVM) throws an `AssertionError`. Assertions help detect logical flaws early by surfacing broken assumptions as soon as they occur.

Introduced in Java 1.4, the `assert` keyword is not enabled by default in production environments. It is mainly a development-time feature that helps verify the internal consistency of your program.

13.2.2 Syntax and Usage of the `assert` Keyword

Java's `assert` statement comes in two forms:

```
assert condition;
```

```
assert condition : errorMessage;
```

The second form allows you to provide a custom error message that helps identify the problem when the assertion fails.

Example:

```
int result = divide(10, 2);
assert result == 5 : "Expected result to be 5";
```

To enable assertions during runtime, use the `-ea` flag with the `java` command:

```
java -ea MyApp
```

By default, assertions are disabled unless explicitly enabled this way.

13.2.3 Using Assertions for Contracts

Assertions align closely with the **Design by Contract** model and can be used to enforce:

Preconditions What must be true before a method executes.

```
public int divide(int a, int b) {
    assert b != 0 : "Precondition failed: denominator must not be zero";
    return a / b;
}
```

Postconditions What must be true after a method executes.

```
public int increment(int x) {
    int result = x + 1;
    assert result > x : "Postcondition failed: result should be greater than input";
    return result;
}
```

Invariants Conditions that must always hold true for an objects state.

```
public class BankAccount {
    private double balance;

    public void deposit(double amount) {
        balance += amount;
        assert balance >= 0 : "Invariant violated: balance should not be negative";
    }
}
```

Assertions help make these expectations **explicit**, improving code readability and maintainability.

13.2.4 Assertions vs. Exceptions

While both assertions and exceptions signal problems, they serve different purposes:

Feature	Assertions	Exceptions
Purpose	Detect programming errors	Handle expected or unexpected runtime issues
Use in contracts	Precondition/postcondition checks	Input validation, recovery strategies
Enabled in prod	Typically disabled	Always active
Recoverable?	No – indicates bugs	Yes – can often recover gracefully

Use assertions to check for conditions that should never occur if the code is correct. **Use exceptions** to handle recoverable conditions, especially those caused by user input or external factors (e.g., file not found, invalid data).

13.2.5 Benefits and Limitations

Benefits:

- Detect bugs early during development.
- Clearly document assumptions and contract conditions.
- Low-overhead checks that improve code quality.

Limitations:

- Disabled by default in production, so they cannot enforce correctness in deployed code.
- Should not be used to validate public method parameters or handle user-facing errors.
- Cannot substitute robust exception handling or logging.

13.2.6 Summary

Assertions are a lightweight yet powerful tool to reinforce the **contractual logic** in your programs. By expressing assumptions and verifying them during development, you can catch subtle bugs before they manifest in production. Although assertions are not a runtime safety net, they greatly aid in **writing correct, self-validating code**—especially when used to reinforce design principles like preconditions, postconditions, and invariants.

When used wisely, assertions make your codebase not just more robust, but more expressive, readable, and aligned with your design intentions.

13.3 Defensive Copies

In Java, **defensive copying** is a programming technique used to protect the internal state of an object from unintended or malicious modification. It is especially important when exposing **mutable objects** (like arrays, `List`, `Map`, or custom objects) through public methods. By returning or accepting copies instead of references, you ensure that external code cannot alter the internal data of your class, maintaining **encapsulation** and **object integrity**.

13.3.1 Why Defensive Copies Matter

When an object exposes a direct reference to a mutable field, external code can change it, potentially violating class invariants or expected behavior. This breaks the contract between a class and its clients and can lead to hard-to-diagnose bugs.

Example without defensive copy:

```
public class Person {
    private Date birthDate;

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }

    public Date getBirthDate() {
        return birthDate; // Dangerous: exposes internal state
    }
}
```

```
Person p = new Person(new Date());
Date d = p.getBirthDate();
d.setTime(0); // Modifies the Person's internal birthDate!
```

Here, the internal `birthDate` of `Person` is unintentionally modified from outside the class.

13.3.2 Implementing Defensive Copies

To avoid this, return a **copy** of the mutable object:

```
public class Person {
    private Date birthDate;

    public Person(Date birthDate) {
        this.birthDate = new Date(birthDate.getTime()); // Defensive copy on input
    }

    public Date getBirthDate() {
```

```
        return new Date(birthDate.getTime()); // Defensive copy on output
    }
}
```

This ensures that changes to the original `Date` or to the returned `Date` do not affect the internal state of the `Person` object.

With arrays:

```
public class DataHolder {
    private int[] data;

    public DataHolder(int[] data) {
        this.data = Arrays.copyOf(data, data.length); // Defensive copy
    }

    public int[] getData() {
        return Arrays.copyOf(data, data.length); // Defensive copy
    }
}
```

With collections:

```
public class Student {
    private List<String> courses = new ArrayList<>();

    public void setCourses(List<String> courses) {
        this.courses = new ArrayList<>(courses); // Defensive copy
    }

    public List<String> getCourses() {
        return new ArrayList<>(courses); // Defensive copy
    }
}
```

13.3.3 Trade-Offs of Defensive Copying

Benefits:

- Preserves encapsulation and class invariants.
- Prevents external classes from introducing bugs by altering internal state.
- Increases robustness and reduces maintenance issues.

Costs:

- **Performance overhead**: Copying large objects or collections can be expensive.
- **Memory usage**: Each defensive copy consumes additional memory.
- **Complexity**: For deeply nested objects or object graphs, copying can be nontrivial.

Therefore, defensive copying is most valuable in situations where **data integrity outweighs**

performance concerns, such as APIs, libraries, or critical systems.

13.3.4 Conclusion

Defensive copying is a foundational defensive programming technique in Java. When designing classes that deal with **mutable objects**, you should assume that callers might inadvertently—or intentionally—modify what they receive or pass in. By defensively copying, you uphold object boundaries and enforce **contracts** reliably, ensuring your objects remain safe, predictable, and easy to reason about.

Chapter 14.

Object-Oriented Design Patterns (Essentials)

1. What Are Design Patterns?
2. Creational Patterns: Singleton, Factory, Builder
3. Structural Patterns: Adapter, Decorator, Composite
4. Behavioral Patterns: Strategy, Observer, Command

14 Object-Oriented Design Patterns (Essentials)

14.1 What Are Design Patterns?

In software engineering, **design patterns** are proven, general-purpose solutions to common problems encountered in object-oriented design. Rather than providing finished code, a design pattern offers a **template or blueprint** for solving a particular issue in a flexible, reusable way. Just as architects use blueprints to design buildings, developers use design patterns to structure software systems that are scalable, maintainable, and robust.

14.1.1 The Origin: The Gang of Four

The concept of design patterns in software was popularized by the 1994 book *Design Patterns: Elements of Reusable Object-Oriented Software*, authored by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—collectively known as the **Gang of Four (GoF)**. Their book introduced 23 classic design patterns and grouped them into three major categories:

1. **Creational** – Deal with object creation mechanisms.
2. **Structural** – Focus on class and object composition.
3. **Behavioral** – Concerned with communication between objects.

These patterns have since become foundational knowledge for object-oriented developers and are widely taught and applied in modern software engineering.

14.1.2 Why Use Design Patterns?

Design patterns bring multiple advantages to the development process:

- **Reusability**: Patterns are general solutions that can be reused across projects and domains.
- **Best Practices**: They incorporate time-tested approaches, reducing the risk of design flaws.
- **Improved Communication**: Developers can convey complex solutions simply by referring to pattern names (e.g., "Use a Factory here").
- **Maintainability and Scalability**: Patterns promote separation of concerns and reduce tight coupling, making code easier to evolve and extend.

When used appropriately, design patterns lead to **more readable, modular, and adaptable codebases**, which are easier to debug, refactor, and test.

14.1.3 The Three Main Categories of Patterns

Creational Patterns

These deal with object instantiation in ways that promote flexibility and reuse. Instead of instantiating classes directly, creational patterns abstract the instantiation process. Examples include:

- **Singleton**: Ensures a class has only one instance.
- **Factory Method**: Lets subclasses decide which class to instantiate.
- **Builder**: Separates object construction from its representation.

Structural Patterns

These focus on how classes and objects are composed to form larger structures. Structural patterns help ensure that components are organized efficiently and can work together. Examples include:

- **Adapter**: Converts one interface into another expected by a client.
- **Decorator**: Adds behavior to objects dynamically.
- **Composite**: Composes objects into tree structures to represent part-whole hierarchies.

Behavioral Patterns

These deal with object interaction and responsibility. Behavioral patterns help manage communication and control flow among objects. Examples include:

- **Strategy**: Defines a family of algorithms and makes them interchangeable.
- **Observer**: Notifies dependent objects when one object changes state.
- **Command**: Encapsulates a request as an object.

14.1.4 A Simple Example: The Singleton Pattern

One of the simplest design patterns is **Singleton**, which ensures that a class has only one instance and provides a global point of access to it. This is useful for managing shared resources such as a configuration manager or database connection.

```
public class ConfigurationManager {
    private static ConfigurationManager instance;

    private ConfigurationManager() { }

    public static ConfigurationManager getInstance() {
        if (instance == null) {
            instance = new ConfigurationManager();
        }
        return instance;
    }
}
```

With Singleton, the `getInstance()` method always returns the same instance, ensuring consistency across the application.

14.1.5 Conclusion

Design patterns are essential tools in the object-oriented programmer's toolbox. Far from being rigid formulas, they are flexible, proven strategies that help manage complexity and promote sound architectural decisions. By learning and applying design patterns thoughtfully, Java developers can write code that is easier to understand, reuse, and maintain—qualities that are critical in modern software development.

In the following sections, we'll explore specific patterns from each category, how they're implemented in Java, and when to apply them in real-world design scenarios.

14.2 Creational Patterns: Singleton, Factory, Builder

Creational design patterns focus on how objects are created. They abstract the instantiation process, allowing systems to be more flexible and independent of the way objects are created, composed, and represented. In this section, we explore three essential creational patterns in Java: **Singleton**, **Factory**, and **Builder**.

14.2.1 Singleton Pattern

Purpose: Ensure a class has only one instance and provide a global point of access to it.

Use Cases

- Configuration managers
- Logging
- Resource managers (e.g., connection pools)

Java Example

```
public class Logger {
    private static Logger instance;

    private Logger() {
        // private constructor prevents instantiation
    }

    public static Logger getInstance() {
```

```
        if (instance == null) {
            instance = new Logger(); // lazy initialization
        }
        return instance;
    }

    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Logger logger = Logger.getInstance();
        logger.log("Application started.");
    }
}
```

Pros

- Controlled access to a sole instance
- Reduces memory footprint

Cons

- Hard to test (tight coupling)
- Can lead to global state (anti-pattern if overused)
- Not thread-safe without synchronization

Best Practice: Use lazy-loaded, thread-safe variants (e.g., using `synchronized`, or static inner class).

14.2.2 Factory Method Pattern

Purpose: Define an interface for creating an object but allow subclasses to alter the type of objects that will be created. The Factory Method lets a class defer instantiation to subclasses or encapsulate logic in a factory class.

Use Cases

- When the exact type of object is unknown until runtime
- When object creation is complex or needs logic branching

Java Example

```java
// Product interface
interface Notification {
    void notifyUser();
}

// Concrete implementations
class EmailNotification implements Notification {
    public void notifyUser() {
        System.out.println("Sending Email Notification");
    }
}

class SMSNotification implements Notification {
    public void notifyUser() {
        System.out.println("Sending SMS Notification");
    }
}

// Factory
class NotificationFactory {
    public static Notification createNotification(String type) {
        if (type.equalsIgnoreCase("EMAIL")) {
            return new EmailNotification();
        } else if (type.equalsIgnoreCase("SMS")) {
            return new SMSNotification();
        }
        throw new IllegalArgumentException("Unknown notification type");
    }
}
```

Usage:

```java
public class Main {
    public static void main(String[] args) {
        Notification notification = NotificationFactory.createNotification("SMS");
        notification.notifyUser();
    }
}
```

Pros

- Promotes loose coupling
- Centralizes object creation
- Easier to manage future extensions

Cons

- Complexity increases with number of product types
- Can lead to bloated factory classes if not organized well

Best Practice: Keep factory logic clean, consider abstract factories for families of related objects.

14.2.3 Builder Pattern

Purpose: Separate the construction of a complex object from its representation, so the same construction process can create different representations.

Use Cases

- When an object has many optional parameters or configurations
- When object construction is complex or multi-step
- When readability and maintainability matter

Java Example

```
public class Computer {
    // Required parameters
    private final String processor;
    private final int ram;

    // Optional parameters
    private final boolean graphicsCard;
    private final boolean bluetooth;

    private Computer(Builder builder) {
        this.processor = builder.processor;
        this.ram = builder.ram;
        this.graphicsCard = builder.graphicsCard;
        this.bluetooth = builder.bluetooth;
    }

    public static class Builder {
        private final String processor;
        private final int ram;

        private boolean graphicsCard = false;
        private boolean bluetooth = false;

        public Builder(String processor, int ram) {
            this.processor = processor;
            this.ram = ram;
        }

        public Builder graphicsCard(boolean value) {
            this.graphicsCard = value;
            return this;
        }

        public Builder bluetooth(boolean value) {
            this.bluetooth = value;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }
```

```
    public void displayConfig() {
        System.out.println("Processor: " + processor + ", RAM: " + ram +
            "GB, Graphics Card: " + graphicsCard + ", Bluetooth: " + bluetooth);
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Computer customPC = new Computer.Builder("Intel i7", 16)
                .graphicsCard(true)
                .bluetooth(true)
                .build();
        customPC.displayConfig();
    }
}
```

Full runnable code:

```
public class Main {
    public static void main(String[] args) {
        Computer customPC = new Computer.Builder("Intel i7", 16)
                .graphicsCard(true)
                .bluetooth(true)
                .build();
        customPC.displayConfig();
    }
}

class Computer {
    // Required parameters
    private final String processor;
    private final int ram;

    // Optional parameters
    private final boolean graphicsCard;
    private final boolean bluetooth;

    private Computer(Builder builder) {
        this.processor = builder.processor;
        this.ram = builder.ram;
        this.graphicsCard = builder.graphicsCard;
        this.bluetooth = builder.bluetooth;
    }

    public static class Builder {
        private final String processor;
        private final int ram;

        private boolean graphicsCard = false;
        private boolean bluetooth = false;

        public Builder(String processor, int ram) {
            this.processor = processor;
            this.ram = ram;
```

```
        }

        public Builder graphicsCard(boolean value) {
            this.graphicsCard = value;
            return this;
        }

        public Builder bluetooth(boolean value) {
            this.bluetooth = value;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }

    public void displayConfig() {
        System.out.println("Processor: " + processor + ", RAM: " + ram +
                "GB, Graphics Card: " + graphicsCard + ", Bluetooth: " + bluetooth);
    }
}
```

Pros

- Clean and readable construction code
- Avoids constructor telescoping
- Immutable objects can be built more easily

Cons

- Slightly verbose, especially for simple objects
- Requires additional builder class

Best Practice: Use for complex objects or those with many optional parameters (e.g., configuration, GUI components, data transfer objects).

14.2.4 Choosing the Right Pattern

Pattern	Best For	Avoid When…
Singleton	Single shared instance	Multiple instances are required
Factory	Flexible, decoupled object creation	Object creation is trivial
Builder	Complex object construction	Only simple constructors are needed

14.2.5 Conclusion

Creational patterns offer flexible and reusable solutions to object creation problems in Java. Whether you need **controlled instantiation** (Singleton), **decoupled creation logic** (Factory), or **step-by-step configuration** (Builder), these patterns help build more **modular, testable, and maintainable** systems. By applying the appropriate creational pattern in the right context, developers can write cleaner and more expressive code that scales gracefully as complexity grows.

14.3 Structural Patterns: Adapter, Decorator, Composite

Structural design patterns are focused on how classes and objects are composed to form larger structures. These patterns simplify the relationships between entities, allowing developers to create flexible and extensible systems by enabling different parts of a program to work together more effectively.

This section covers three widely used structural patterns in Java: **Adapter**, **Decorator**, and **Composite**.

14.3.1 Adapter Pattern

Intent: Convert the interface of a class into another interface that clients expect. The Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Use Case

When integrating legacy code or third-party libraries that don't match your current interface expectations.

Example Scenario

Suppose you have a legacy `AudioPlayer` that plays `.mp3` files, and a new media library supports `.mp4`. You want your system to adapt the new format without modifying the existing player.

Java Example

```
// Existing interface
interface MediaPlayer {
    void play(String audioType, String fileName);
}

// New library with a different interface
```

```
class AdvancedMediaPlayer {
    public void playMp4(String fileName) {
        System.out.println("Playing MP4 file: " + fileName);
    }
}

// Adapter to bridge both
class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedPlayer = new AdvancedMediaPlayer();

    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("mp4")) {
            advancedPlayer.playMp4(fileName);
        } else {
            System.out.println("Unsupported format: " + audioType);
        }
    }
}

// Client
class AudioPlayer implements MediaPlayer {
    private MediaAdapter adapter = new MediaAdapter();

    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing MP3 file: " + fileName);
        } else {
            adapter.play(audioType, fileName);
        }
    }
}
```

Usage:

```
AudioPlayer player = new AudioPlayer();
player.play("mp3", "song.mp3");
player.play("mp4", "video.mp4");
```

Benefits

- Decouples client from implementation details
- Enables reuse of existing functionality

14.3.2 Decorator Pattern

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Use Case

When you want to add functionality to objects without changing their class or creating a large inheritance hierarchy.

Java Example

```java
// Core interface
interface Coffee {
    String getDescription();
    double cost();
}

// Concrete component
class SimpleCoffee implements Coffee {
    public String getDescription() { return "Simple Coffee"; }
    public double cost() { return 5.0; }
}

// Decorator base class
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;
    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }
}

// Concrete decorators
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    public double cost() {
        return decoratedCoffee.cost() + 1.5;
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    public double cost() {
        return decoratedCoffee.cost() + 0.5;
    }
}
```

Usage:

```
Coffee coffee = new SimpleCoffee();
coffee = new MilkDecorator(coffee);
coffee = new SugarDecorator(coffee);

System.out.println(coffee.getDescription()); // Simple Coffee, Milk, Sugar
System.out.println("Cost: $" + coffee.cost()); // Cost: $7.0
```

Benefits

- Open/Closed Principle: open for extension, closed for modification
- Composable behavior

Full runnable code:

```
// Core interface
interface Coffee {
    String getDescription();
    double cost();
}

// Concrete component
class SimpleCoffee implements Coffee {
    public String getDescription() { return "Simple Coffee"; }
    public double cost() { return 5.0; }
}

// Decorator base class
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;
    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }
}

// Concrete decorators
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    public double cost() {
        return decoratedCoffee.cost() + 1.5;
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
```

```
    }

    public double cost() {
        return decoratedCoffee.cost() + 0.5;
    }
}

public class DecoratorDemo {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        coffee = new MilkDecorator(coffee);
        coffee = new SugarDecorator(coffee);

        System.out.println(coffee.getDescription()); // Simple Coffee, Milk, Sugar
        System.out.println("Cost: $" + coffee.cost()); // Cost: $7.0
    }
}
```

14.3.3 Composite Pattern

Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions uniformly.

Use Case

When building hierarchies such as graphical components (buttons inside panels), file systems (folders containing files), or organizational charts.

Java Example

```
// Component
interface FileSystem {
    void ls();
}

// Leaf
class File implements FileSystem {
    private String name;
    public File(String name) { this.name = name; }

    public void ls() {
        System.out.println("File: " + name);
    }
}

// Composite
class Directory implements FileSystem {
    private String name;
    private List<FileSystem> contents = new ArrayList<>();

    public Directory(String name) {
```

```
        this.name = name;
    }

    public void add(FileSystem fs) {
        contents.add(fs);
    }

    public void ls() {
        System.out.println("Directory: " + name);
        for (FileSystem fs : contents) {
            fs.ls();
        }
    }
}
```

Usage:

```
FileSystem file1 = new File("readme.txt");
FileSystem file2 = new File("data.csv");
Directory dir = new Directory("MyDocs");
dir.add(file1);
dir.add(file2);

Directory root = new Directory("Root");
root.add(dir);
root.ls();
```

Output:

```
Directory: Root
Directory: MyDocs
File: readme.txt
File: data.csv
```

Full runnable code:

```
import java.util.ArrayList;
import java.util.List;

// Component
interface FileSystem {
    void ls();
}

// Leaf
class File implements FileSystem {
    private String name;
    public File(String name) { this.name = name; }

    public void ls() {
        System.out.println("File: " + name);
    }
}
```

```java
// Composite
class Directory implements FileSystem {
    private String name;
    private List<FileSystem> contents = new ArrayList<>();

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

    public void add(FileSystem fs) {
        contents.add(fs);
    }

    public void ls() {
        System.out.println("Directory: " + name);
        for (FileSystem fs : contents) {
            fs.ls();
        }
    }
}

public class CompositeDemo {
    public static void main(String[] args) {
        FileSystem file1 = new File("readme.txt");
        FileSystem file2 = new File("data.csv");
        Directory dir = new Directory("MyDocs");
        dir.add(file1);
        dir.add(file2);

        Directory root = new Directory("Root");
        root.add(dir);

        root.ls();
    }
}
```

Benefits

- Treat individual objects and collections uniformly
- Recursive structures are easier to manage

14.3.4 Summary

Pattern	Intent	When to Use
Adapter	Convert incompatible interfaces	Bridging legacy code or APIs
Decorator	Add behavior dynamically	Customize functionality without changing class
Composite	Treat groups and objects uniformly	Model tree-like structures (e.g., UI, filesystems)

14.3.5 Conclusion

Structural design patterns like Adapter, Decorator, and Composite offer powerful tools for organizing code, reducing coupling, and enhancing system flexibility. These patterns help developers build systems that are **easier to extend, maintain, and scale**—core goals of object-oriented design. By mastering these patterns, Java developers can craft more modular, adaptable, and reusable code architectures.

14.4 Behavioral Patterns: Strategy, Observer, Command

Behavioral design patterns are concerned with how objects interact and communicate to fulfill responsibilities. These patterns increase flexibility in carrying out behaviors and decouple sender and receiver objects. In this section, we'll explore three widely used behavioral patterns in Java: **Strategy**, **Observer**, and **Command**.

14.4.1 Strategy Pattern

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy allows an algorithm's behavior to be selected at runtime.

Use Case

When different algorithms can be applied interchangeably without altering the context class using them—e.g., sorting strategies or payment methods.

Java Example

```
// Strategy interface
interface PaymentStrategy {
    void pay(int amount);
}

// Concrete strategies
class CreditCardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid $" + amount + " using Credit Card.");
    }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid $" + amount + " using PayPal.");
    }
}
```

```
// Context
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

Usage:

```
ShoppingCart cart = new ShoppingCart();
cart.setPaymentStrategy(new PayPalPayment());
cart.checkout(100); // Paid $100 using PayPal
```

Full runnable code:

```
// Strategy interface
interface PaymentStrategy {
    void pay(int amount);
}

// Concrete strategies
class CreditCardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid $" + amount + " using Credit Card.");
    }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid $" + amount + " using PayPal.");
    }
}

// Context
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(int amount) {
        if (paymentStrategy == null) {
            System.out.println("Payment strategy not set!");
        } else {
            paymentStrategy.pay(amount);
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        cart.setPaymentStrategy(new PayPalPayment());
        cart.checkout(100); // Output: Paid $100 using PayPal

        cart.setPaymentStrategy(new CreditCardPayment());
        cart.checkout(250); // Output: Paid $250 using Credit Card
    }
}
```

Benefits

- Easily interchangeable behaviors
- Adheres to Open/Closed Principle
- Reduces conditional logic

14.4.2 Observer Pattern

Intent: Define a one-to-many dependency so that when one object changes state, all its dependents are notified and updated automatically.

Use Case

Widely used in event-driven systems, GUIs, and model-view-controller (MVC) frameworks—e.g., updating multiple UI components when data changes.

Java Example

```
// Observer interface
interface Observer {
    void update(String message);
}

// Subject interface
interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyObservers();
}

// Concrete subject
class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String news;

    public void setNews(String news) {
        this.news = news;
        notifyObservers();
```

```
    }

    public void attach(Observer o) { observers.add(o); }
    public void detach(Observer o) { observers.remove(o); }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(news);
        }
    }
}

// Concrete observer
class NewsChannel implements Observer {
    private String name;

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

    public void update(String message) {
        System.out.println(name + " received: " + message);
    }
}
```

Usage:

```
NewsAgency agency = new NewsAgency();
agency.attach(new NewsChannel("Channel A"));
agency.attach(new NewsChannel("Channel B"));
agency.setNews("Breaking: Strategy Pattern Deployed!");
```

Output:

```
Channel A received: Breaking: Strategy Pattern Deployed!
Channel B received: Breaking: Strategy Pattern Deployed!
```

Full runnable code:

```
import java.util.ArrayList;
import java.util.List;

// Observer interface
interface Observer {
    void update(String message);
}

// Subject interface
interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyObservers();
}

// Concrete subject
```

```java
class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String news;

    public void setNews(String news) {
        this.news = news;
        notifyObservers();
    }

    public void attach(Observer o) { observers.add(o); }
    public void detach(Observer o) { observers.remove(o); }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(news);
        }
    }
}

// Concrete observer
class NewsChannel implements Observer {
    private String name;

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

    public void update(String message) {
        System.out.println(name + " received: " + message);
    }
}

public class ObserverDemo {
    public static void main(String[] args) {
        NewsAgency agency = new NewsAgency();
        agency.attach(new NewsChannel("Channel A"));
        agency.attach(new NewsChannel("Channel B"));
        agency.setNews("Breaking: Strategy Pattern Deployed!");
    }
}
```

Benefits

- Promotes loose coupling between subject and observers
- Easily extensible for additional observers
- Suits asynchronous event handling

14.4.3 Command Pattern

Intent: Encapsulate a request as an object, thereby allowing for parameterization of clients with queues, logs, and undo/redo operations.

Use Case

When you want to parameterize operations (e.g., buttons triggering commands), support undo functionality, or queue operations.

Java Example

```
// Command interface
interface Command {
    void execute();
}

// Receiver
class Light {
    public void turnOn() {
        System.out.println("Light turned ON");
    }

    public void turnOff() {
        System.out.println("Light turned OFF");
    }
}

// Concrete commands
class TurnOnCommand implements Command {
    private Light light;

    public TurnOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}

class TurnOffCommand implements Command {
    private Light light;

    public TurnOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOff();
    }
}

// Invoker
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
```

```
        command.execute();
    }
}
```

Usage:

```
Light light = new Light();
Command on = new TurnOnCommand(light);
Command off = new TurnOffCommand(light);

RemoteControl remote = new RemoteControl();
remote.setCommand(on);
remote.pressButton(); // Light turned ON
remote.setCommand(off);
remote.pressButton(); // Light turned OFF
```

Full runnable code:

```
// Command interface
interface Command {
    void execute();
}

// Receiver
class Light {
    public void turnOn() {
        System.out.println("Light turned ON");
    }

    public void turnOff() {
        System.out.println("Light turned OFF");
    }
}

// Concrete commands
class TurnOnCommand implements Command {
    private Light light;

    public TurnOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}

class TurnOffCommand implements Command {
    private Light light;

    public TurnOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOff();
```

```
    }
}

// Invoker
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

public class CommandDemo {
    public static void main(String[] args) {
        Light light = new Light();
        Command on = new TurnOnCommand(light);
        Command off = new TurnOffCommand(light);

        RemoteControl remote = new RemoteControl();
        remote.setCommand(on);
        remote.pressButton(); // Light turned ON
        remote.setCommand(off);
        remote.pressButton(); // Light turned OFF
    }
}
```

Benefits

- Command encapsulation allows flexible invocation
- Easily supports undo, redo, and macros
- Decouples the invoker from the receiver

14.4.4 Summary

Pattern	Intent	Ideal For
Strategy	Choose algorithm dynamically	Behavior switching (e.g., different logics)
Observer	Notify subscribers when subject changes	Event-driven systems, UI updates
Command	Encapsulate requests as objects	Queues, undo/redo, remote execution

14.4.5 Conclusion

Behavioral design patterns such as **Strategy**, **Observer**, and **Command** are essential tools for organizing object communication in a clean and scalable way. They help developers decouple objects, promote flexibility, and design systems that are easier to extend and maintain. When used appropriately, these patterns can significantly enhance the responsiveness, modularity, and clarity of Java applications.

Chapter 15.

Refactoring to Design

1. Code Smells and Refactoring
2. Extract Method, Class, and Interface
3. Replace Conditional with Polymorphism
4. Applying Patterns Through Refactoring

15 Refactoring to Design

15.1 Code Smells and Refactoring

In software development, maintaining clean, modular, and extensible code is essential to long-term success. Over time, however, codebases can deteriorate due to rushed development, unclear requirements, or lack of design awareness. These issues often manifest as *code smells*—symptoms in code that may not be outright bugs but indicate deeper structural problems. Recognizing and resolving these smells through **refactoring** is a vital part of sustaining high-quality object-oriented design.

15.1.1 What Are Code Smells?

A *code smell* is a surface-level indication that something may be wrong in the system's design. While a smell doesn't always mean there's a bug, it suggests the presence of technical debt or future maintainability issues. The term was popularized by Martin Fowler in his seminal book *Refactoring: Improving the Design of Existing Code*, where he emphasizes that smells should trigger deeper inspection.

Smells are not absolute—they depend on context, scale, and the domain—but being able to identify them is a critical skill for any Java developer.

15.1.2 Common Object-Oriented Code Smells

Below are some frequent smells particularly relevant to Java and object-oriented programming:

1. **Long Method** Methods that try to do too much tend to become unreadable and hard to maintain. *Smell indicator*: A method spans dozens of lines and has multiple responsibilities.
2. **Large Class** Classes bloated with fields and methods may indicate poor separation of concerns. *Smell indicator*: The class has too many responsibilities or manages unrelated data.
3. **Duplicated Code** Copy-pasted code across methods or classes leads to inconsistencies and makes maintenance harder. *Smell indicator*: Similar blocks of code appear in multiple places.
4. **Feature Envy** A method that frequently accesses the internals of another class likely belongs there. *Smell indicator*: Calls to another object's getters dominate the method.
5. **Data Clumps** Groups of parameters or fields that tend to appear together suggest the need for a new object. *Smell indicator*: Methods regularly pass the same three or

four arguments.

6. **Switch Statements or Long Conditionals** Repeated conditional logic suggests polymorphism might better serve the design. *Smell indicator*: `switch` or `if-else` blocks that dispatch behavior based on type.

15.1.3 The Role of Refactoring

Refactoring is the process of improving the internal structure of code without changing its observable behavior. It's a disciplined way to clean code while retaining all of its functionality. Refactoring is not about adding features; it's about improving design and maintainability.

Refactoring helps achieve:

- **Improved readability**
- **Reduced duplication**
- **Better encapsulation and modularity**
- **Simpler testing and debugging**

It's crucial to refactor with the support of a good suite of unit tests to ensure that changes don't break existing functionality.

15.1.4 Example: Refactoring a Long Method

Before Refactoring:

```
public void printInvoice(Order order) {
    System.out.println("Invoice for Order #" + order.getId());
    for (Item item : order.getItems()) {
        double price = item.getPrice() * item.getQuantity();
        System.out.println(item.getName() + ": " + price);
    }
    System.out.println("Total: " + order.getTotal());
    System.out.println("Thank you for shopping!");
}
```

Smell: This method mixes multiple responsibilities—printing headers, calculating prices, and formatting output.

After Refactoring (using *Extract Method*):

```
public void printInvoice(Order order) {
    printHeader(order);
    printItems(order);
    printFooter(order);
}
```

```java
private void printHeader(Order order) {
    System.out.println("Invoice for Order #" + order.getId());
}

private void printItems(Order order) {
    for (Item item : order.getItems()) {
        double price = item.getPrice() * item.getQuantity();
        System.out.println(item.getName() + ": " + price);
    }
}

private void printFooter(Order order) {
    System.out.println("Total: " + order.getTotal());
    System.out.println("Thank you for shopping!");
}
```

Result: The main method now reads clearly, and each step can be reused, tested, or modified independently.

15.1.5 When and How to Refactor

You should refactor when:

- You encounter a code smell during development.
- You add new features to legacy code.
- You fix bugs in poorly structured code.

Use IDE features (like in IntelliJ IDEA or Eclipse) to assist with safe refactoring. Always validate your changes with tests.

15.1.6 Conclusion

Code smells are a helpful way to detect areas where your code might be violating design principles such as the Single Responsibility Principle or DRY (Don't Repeat Yourself). Through disciplined refactoring, developers can transform smelly code into clean, modular, and robust systems. Rather than treating refactoring as a chore, embrace it as a key tool for evolving your design, especially in growing or long-lived Java projects.

15.2 Extract Method, Class, and Interface

15.2.1 Refactoring Techniques: Extract Method, Class, and Interface

In object-oriented software development, **refactoring** is essential to keep code maintainable, readable, and extensible. Among the most powerful and frequently used refactoring techniques are **Extract Method**, **Extract Class**, and **Extract Interface**. These techniques help developers simplify complex code, improve cohesion, and reduce coupling — all of which contribute to cleaner design and better software architecture.

Let's explore each of these techniques with practical examples and best practices.

15.2.2 Extract Method: Breaking Down Complexity

The **Extract Method** refactoring technique involves moving a block of code from a larger method into a new, well-named method. This improves readability, reusability, and testability.

Before Refactoring

```
public class InvoicePrinter {
    public void printInvoice(Order order) {
        System.out.println("Invoice for Order #" + order.getId());
        for (Item item : order.getItems()) {
            double price = item.getPrice() * item.getQuantity();
            System.out.println(item.getName() + ": " + price);
        }
        System.out.println("Total: " + order.getTotal());
    }
}
```

This method is doing too much—printing headers, iterating items, computing prices, and printing totals.

After Refactoring

```
public class InvoicePrinter {
    public void printInvoice(Order order) {
        printHeader(order);
        printItems(order);
        printFooter(order);
    }

    private void printHeader(Order order) {
        System.out.println("Invoice for Order #" + order.getId());
    }

    private void printItems(Order order) {
        for (Item item : order.getItems()) {
```

```
            double price = item.getPrice() * item.getQuantity();
            System.out.println(item.getName() + ": " + price);
        }
    }

    private void printFooter(Order order) {
        System.out.println("Total: " + order.getTotal());
    }
}
```

Benefits

- Easier to read and maintain
- Each method has a clear responsibility
- Smaller methods are easier to test independently

15.2.3 Extract Class: Improving Cohesion

When a class has multiple responsibilities or grows too large, it violates the **Single Responsibility Principle**. In such cases, you can use the **Extract Class** technique to move related behavior and data into a new class.

Before Refactoring

```
public class Person {
    private String name;
    private String email;
    private String street;
    private String city;
    private String zip;

    public String getFullAddress() {
        return street + ", " + city + " " + zip;
    }

    // Getters and setters...
}
```

Here, the `Person` class is managing both personal identity and address information.

After Refactoring

```
public class Address {
    private String street;
    private String city;
    private String zip;

    public String getFullAddress() {
        return street + ", " + city + " " + zip;
```

```
    }

    // Getters and setters...
}

public class Person {
    private String name;
    private String email;
    private Address address;

    public String getAddressDetails() {
        return address.getFullAddress();
    }

    // Getters and setters...
}
```

Benefits

- Each class has a focused purpose
- Encourages reuse (the `Address` class might be used elsewhere)
- Makes the codebase more navigable and easier to extend

15.2.4 Extract Interface: Enhancing Flexibility and Decoupling

Interfaces allow code to depend on **abstractions** rather than concrete implementations. The **Extract Interface** technique involves identifying methods that define a contract for interaction and isolating them into a separate interface.

Before Refactoring

```
public class FileLogger {
    public void log(String message) {
        System.out.println("LOG: " + message);
    }
}
```

Suppose you want to allow different logging mechanisms without changing the client code.

After Refactoring

```
public interface Logger {
    void log(String message);
}

public class FileLogger implements Logger {
    public void log(String message) {
        System.out.println("LOG: " + message);
    }
```

```
}

public class DatabaseLogger implements Logger {
    public void log(String message) {
        // Simulated DB log
        System.out.println("DB LOG: " + message);
    }
}
```

Now, client code can work with the `Logger` interface:

```
public class OrderService {
    private Logger logger;

    public OrderService(Logger logger) {
        this.logger = logger;
    }

    public void processOrder() {
        logger.log("Order processed.");
    }
}
```

Benefits

- Promotes **loose coupling**
- Enables easier **testing** with mocks or stubs
- Supports **polymorphism** and **extensibility**

15.2.5 Best Practices and Considerations

- **Name methods and classes clearly** when extracting. Good names communicate intent and make the code self-documenting.
- **Don't over-fragment**. If a method is already short and readable, extracting it may hurt more than help.
- **Use interfaces judiciously**. Not every class needs an interface—extract them when multiple implementations or abstractions are expected.
- **Refactor incrementally**. Apply these techniques in small steps, validating with tests to ensure nothing breaks.

15.2.6 Conclusion

Refactoring with *Extract Method*, *Extract Class*, and *Extract Interface* empowers developers to evolve codebases into modular, clean, and extensible designs. These techniques enhance maintainability by aligning with core object-oriented principles like separation of concerns,

encapsulation, and abstraction. By mastering and routinely applying these techniques, developers can transform even the most tangled code into well-structured and robust software.

15.3 Replace Conditional with Polymorphism

15.3.1 Replacing Conditional Logic with Polymorphism

In object-oriented programming, large conditional statements like `if-else` and `switch` often signal missed opportunities for design improvement. These structures tend to grow unwieldy over time, making code harder to understand, modify, and extend. One powerful refactoring technique is to **replace conditional logic with polymorphism**, allowing behavior to be delegated to objects rather than being controlled by conditional flow.

By leveraging polymorphism through interfaces and class hierarchies, developers can isolate behaviors into dedicated classes. This results in code that adheres more closely to the **Open/Closed Principle** — open for extension but closed for modification.

15.3.2 The Problem with Conditionals

Consider the following code that calculates the shipping cost based on shipping type:

```
public class ShippingCalculator {
    public double calculateShipping(String method, double weight) {
        if (method.equals("standard")) {
            return weight * 1.0;
        } else if (method.equals("express")) {
            return weight * 1.5 + 10;
        } else if (method.equals("overnight")) {
            return weight * 2.0 + 25;
        } else {
            throw new IllegalArgumentException("Unknown shipping method: " + method);
        }
    }
}
```

This design is functional but rigid. Adding new shipping types requires modifying the `calculateShipping` method, which violates the Open/Closed Principle. Furthermore, it's harder to test and maintain each logic path individually.

15.3.3 Step-by-Step Refactoring to Polymorphism

Step 1: Define an Interface

```
public interface ShippingStrategy {
    double calculate(double weight);
}
```

This interface defines a common contract that all shipping strategies will implement.

Step 2: Create Concrete Strategy Classes

```
public class StandardShipping implements ShippingStrategy {
    public double calculate(double weight) {
        return weight * 1.0;
    }
}

public class ExpressShipping implements ShippingStrategy {
    public double calculate(double weight) {
        return weight * 1.5 + 10;
    }
}

public class OvernightShipping implements ShippingStrategy {
    public double calculate(double weight) {
        return weight * 2.0 + 25;
    }
}
```

Each class encapsulates its behavior, removing the need for conditionals.

Step 3: Update the Client Code

```
public class ShippingCalculator {
    private ShippingStrategy strategy;

    public ShippingCalculator(ShippingStrategy strategy) {
        this.strategy = strategy;
    }

    public double calculate(double weight) {
        return strategy.calculate(weight);
    }
}
```

Usage:

```
ShippingStrategy strategy = new ExpressShipping();
ShippingCalculator calculator = new ShippingCalculator(strategy);
double cost = calculator.calculate(5.0);
```

This approach makes it trivial to add new strategies by creating a new class that implements

`ShippingStrategy`.

15.3.4 Benefits of This Refactoring

Improved Maintainability

Each class has a single responsibility. If shipping logic changes, it's isolated to a specific class without affecting others.

Open for Extension

Adding new shipping types doesn't require altering existing logic—just a new class.

Enhanced Testability

Each strategy can be tested independently of the others, improving coverage and reducing side effects.

Cleaner Code

Replacing nested or sprawling conditionals with dedicated classes results in clearer, self-documenting logic.

15.3.5 When to Apply This Pattern

This refactoring is particularly effective when:

- The conditional logic selects behavior based on a known set of values.
- The conditions map directly to different behaviors or algorithms.
- The logic is likely to change or grow over time.

It may be unnecessary for very simple conditionals or when behavior doesn't vary much.

15.3.6 Real-World Examples

- In a **tax calculation** system, different tax strategies (flat, progressive, regional) can be modeled as polymorphic classes.
- For **game design**, different character behaviors (aggressive, defensive, stealthy) can be modeled with behavior interfaces.
- In **UI frameworks**, button actions are often encapsulated in command objects rather than hardcoded with `if-else`.

15.3.7 Conclusion

Replacing conditionals with polymorphism is a fundamental object-oriented design technique. It helps eliminate rigid control flow and replaces it with a flexible, extensible architecture built on interfaces and inheritance. The result is more modular, testable, and scalable code that aligns with core design principles such as Single Responsibility and Open/Closed. As systems grow, this approach pays dividends in reduced complexity and easier evolution of software behavior.

15.4 Applying Patterns Through Refactoring

15.4.1 Applying Patterns Through Refactoring

Refactoring is more than cleaning up code—it's a path to discovering better structure. As systems grow in complexity, refactoring becomes an essential tool to evolve procedural or tightly coupled code into well-structured, maintainable designs. One powerful outcome of thoughtful refactoring is the emergence—or intentional introduction—of **object-oriented design patterns**.

Design patterns are proven solutions to recurring design problems. However, code often begins without patterns, especially in early prototypes or legacy systems. Through refactoring, we can reshape existing code to align with patterns such as Strategy, Observer, Command, or Decorator—enhancing clarity, testability, and extensibility.

15.4.2 Recognizing Opportunities for Patterns

When working with legacy code, you may encounter symptoms like long conditional chains, repeated logic, or tightly coupled classes. These are clues that a design pattern might provide a cleaner solution.

Example: Refactoring to the Strategy Pattern

Consider a legacy tax calculator:

```
public class TaxCalculator {
    public double calculate(String region, double amount) {
        if (region.equals("US")) {
            return amount * 0.07;
        } else if (region.equals("EU")) {
            return amount * 0.2;
        } else if (region.equals("IN")) {
            return amount * 0.18;
        }
        return 0;
```

```
    }
}
```

This procedural design is difficult to extend. To support more regions or changing rules, we would need to continually modify this method.

Refactored Using the Strategy Pattern:

```
public interface TaxStrategy {
    double calculate(double amount);
}

public class USTax implements TaxStrategy {
    public double calculate(double amount) {
        return amount * 0.07;
    }
}

public class EUTax implements TaxStrategy {
    public double calculate(double amount) {
        return amount * 0.2;
    }
}

public class IndiaTax implements TaxStrategy {
    public double calculate(double amount) {
        return amount * 0.18;
    }
}

public class TaxCalculator {
    private TaxStrategy strategy;

    public TaxCalculator(TaxStrategy strategy) {
        this.strategy = strategy;
    }

    public double calculate(double amount) {
        return strategy.calculate(amount);
    }
}
```

Now, new tax regions can be added without changing existing logic, following the **Open/Closed Principle**. The code is cleaner, more modular, and easier to test.

15.4.3 Observer Pattern: Making Code Reactive

Another frequent use case is decoupling event producers and consumers, common in UI, game engines, or monitoring systems. A legacy implementation may have direct method calls like:

```java
public class Button {
    public void click() {
        System.out.println("Button clicked");
        new Logger().log("Clicked");
        new Analytics().track("Click");
    }
}
```

This tightly couples the `Button` to multiple classes, making it rigid.

Refactored Using the Observer Pattern:

```java
public interface ClickListener {
    void onClick();
}

public class Button {
    private List<ClickListener> listeners = new ArrayList<>();

    public void addListener(ClickListener listener) {
        listeners.add(listener);
    }

    public void click() {
        for (ClickListener listener : listeners) {
            listener.onClick();
        }
    }
}

public class Logger implements ClickListener {
    public void onClick() {
        System.out.println("Logged click");
    }
}

public class Analytics implements ClickListener {
    public void onClick() {
        System.out.println("Tracked click");
    }
}
```

Listeners can now be registered or removed dynamically, and the button has no knowledge of who listens. This improves **decoupling** and allows reusability of both the button and its observers.

15.4.4 Benefits of Pattern-Driven Refactoring

- **Improved Clarity**: Patterns make code intent explicit. Other developers can quickly recognize Strategy or Observer patterns and understand the design rationale.
- **Modularity and Testability**: Refactoring to patterns often yields smaller, more

focused classes that are easier to test in isolation.
- **Flexibility**: Patterns promote extensible code by removing rigid control flows and hardcoded dependencies.

15.4.5 When to Apply Patterns vs. Simpler Refactoring

Refactoring to design patterns is not always the first step. In early stages, simpler transformations like extracting methods or reducing duplication might be more appropriate. Use pattern-driven refactoring when:

- A specific problem aligns with a known pattern.
- Future changes are likely (e.g., changing algorithms, growing feature sets).
- You need to isolate variation and make the system more flexible.

Avoid forcing patterns into code where simplicity suffices. The goal is to improve structure, not complicate it with unnecessary abstractions.

15.4.6 Conclusion

Refactoring toward design patterns is a practical and powerful technique for evolving code. Rather than imposing patterns from the outset, let them **emerge through iterative improvements**. When used wisely, patterns transform rigid, duplicated logic into elegant, extensible architecture. Through small, well-directed refactorings, we enable our software to grow gracefully and sustainably.

Chapter 16.

Generics and Type Abstraction

1. Generic Classes and Methods
2. Bounded Type Parameters
3. Wildcards (`? extends`, `? super`)
4. Generics in Collections

16 Generics and Type Abstraction

16.1 Generic Classes and Methods

16.1.1 Introduction to Generics

Generics were introduced in Java 5 to enhance **type safety** and **code reuse**. They allow classes, interfaces, and methods to operate on **typed parameters**, enabling you to write code that is more flexible and less error-prone. Rather than using `Object` references and casting at runtime, generics let you define the expected types at compile time, which can prevent many common programming errors.

Without generics, developers often had to rely on raw types and explicit casting:

```
List names = new ArrayList();
names.add("Alice");
String name = (String) names.get(0); // cast required
```

With generics:

```
List<String> names = new ArrayList<>();
names.add("Alice");
String name = names.get(0); // no cast needed
```

Generics bring clarity, eliminate casting, and reduce the risk of `ClassCastException`.

16.1.2 Declaring Generic Classes

A **generic class** is defined with a type parameter (commonly `T`, `E`, `K`, `V`) that is specified when the class is instantiated. Here's an example of a generic container:

```
public class Box<T> {
    private T item;

    public void set(T item) {
        this.item = item;
    }

    public T get() {
        return item;
    }
}
```

You can use the `Box` class with any type:

```
Box<String> stringBox = new Box<>();
stringBox.set("Hello");
```

```
System.out.println(stringBox.get()); // Output: Hello

Box<Integer> intBox = new Box<>();
intBox.set(42);
System.out.println(intBox.get()); // Output: 42
```

This design promotes code reuse and ensures type safety—`stringBox` can only hold `String` values, while `intBox` can only hold `Integer` values.

Full runnable code:

```
public class Box<T> {
    private T item;

    public void set(T item) {
        this.item = item;
    }

    public T get() {
        return item;
    }

    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.set("Hello");
        System.out.println(stringBox.get()); // Output: Hello

        Box<Integer> intBox = new Box<>();
        intBox.set(42);
        System.out.println(intBox.get()); // Output: 42
    }
}
```

16.1.3 Declaring Generic Methods

You can also create **generic methods**, which declare their own type parameters. These are useful when the method needs to work with various types independent of any class-level type parameters.

```
public class Utility {
    public static <T> void printArray(T[] array) {
        for (T item : array) {
            System.out.println(item);
        }
    }
}
```

To use this method:

```
String[] names = {"Alice", "Bob", "Charlie"};
Integer[] numbers = {1, 2, 3};

Utility.printArray(names);
Utility.printArray(numbers);
```

Here, the method `printArray` can accept arrays of any object type (`T[]`). The compiler infers the type parameter from the arguments passed, allowing one method to work generically with multiple types.

Full runnable code:

```
public class Utility {
    public static <T> void printArray(T[] array) {
        for (T item : array) {
            System.out.println(item);
        }
    }

    public static void main(String[] args) {
        String[] names = {"Alice", "Bob", "Charlie"};
        Integer[] numbers = {1, 2, 3};

        Utility.printArray(names);
        Utility.printArray(numbers);
    }
}
```

16.1.4 Type Parameters and Naming Conventions

Java uses naming conventions to indicate the purpose of type parameters:

- `T`: Type
- `E`: Element (commonly used in collections)
- `K`, `V`: Key and Value (used in maps)
- `S`, `U`, etc.: Additional type parameters

These conventions help developers understand the roles of generic types more clearly.

16.1.5 Eliminating Casts and Reducing Runtime Errors

A primary benefit of generics is the elimination of **unsafe casts**. In pre-generic Java, you had to downcast every object extracted from a collection, which was both tedious and error-prone. Generics enable **compile-time checks**, catching type mismatches early.

For example, without generics:

```
List list = new ArrayList();
list.add("Hello");
Integer number = (Integer) list.get(0); // ClassCastException at runtime
```

With generics:

```
List<String> list = new ArrayList<>();
list.add("Hello");
// List<Integer> integers = list; // Compile-time error
```

Now, the compiler ensures that `list` only holds `String` objects, preventing incorrect assignments.

16.1.6 Use Cases in Real-World Design

Generics are particularly useful in designing reusable libraries. The Java Collections Framework is a great example—classes like `List<E>`, `Map<K,V>`, and `Set<E>` rely on generics to handle any type of object safely and consistently.

Another use case is building data structures:

```
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

Now you can create pairs of any types:

```
Pair<String, Integer> entry = new Pair<>("Apples", 5);
System.out.println(entry.getKey() + ": " + entry.getValue());
```

Full runnable code:

```
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
```

```
    public K getKey() { return key; }
    public V getValue() { return value; }

    public static void main(String[] args) {
        Pair<String, Integer> entry = new Pair<>("Apples", 5);
        System.out.println(entry.getKey() + ": " + entry.getValue());
    }
}
```

16.1.7 Conclusion

Generic classes and methods are powerful tools that make Java code more **robust, reusable, and type-safe**. By using generics, you avoid casting, reduce runtime errors, and create APIs that are flexible and easy to maintain. In the chapters ahead, we'll explore advanced generic features like bounded types and wildcards, which further extend the expressiveness and safety of Java's type system.

16.2 Bounded Type Parameters

16.2.1 What Are Bounded Type Parameters?

In Java generics, **bounded type parameters** restrict the types that can be used as arguments for a type parameter. By placing bounds on type parameters, developers can ensure that the type passed in meets specific criteria—namely, that it is a subtype (or occasionally supertype) of a particular class or interface. This allows methods and classes to take advantage of behaviors guaranteed by those bounds, leading to **safer and more expressive code**.

There are two primary types of bounds:

- **Upper bounds**: Restrict the type to a class or interface and its subclasses.
- **Lower bounds**: Restrict the type to a class or interface and its superclasses.

Let's explore each in more detail.

16.2.2 Upper Bounds (`extends`)

To specify that a type parameter must be a subclass of a certain class or implement a specific interface, you use the `extends` keyword—even for interfaces.

```java
public class Box<T extends Number> {
    private T value;

    public Box(T value) {
        this.value = value;
    }

    public double doubleValue() {
        return value.doubleValue(); // Safe: Number guarantees this method
    }
}
```

Here, `T` is constrained to `Number` or any of its subclasses (e.g., `Integer`, `Double`). This ensures that `value.doubleValue()` is always valid, as it's part of the `Number` class.

Usage:

```java
Box<Integer> intBox = new Box<>(10);
Box<Double> doubleBox = new Box<>(3.14);
// Box<String> stringBox = new Box<>("text"); // Compile-time error
```

Using upper bounds allows generic methods to access methods of the bounded type, improving both **flexibility** and **type safety**.

16.2.3 Lower Bounds (`super`)

While less common, **lower bounds** can be useful when writing methods that consume objects rather than produce them. Lower bounds are used with **wildcards**, typically in method parameters:

```java
public static void addNumbers(List<? super Integer> list) {
    list.add(1);
    list.add(2);
}
```

This method accepts a list of `Integer` or any of its supertypes (e.g., `Number`, `Object`) and safely adds integers. You can't retrieve specific `Integer` elements from the list without casting, but you *can* add known `Integer` values into it.

16.2.4 Why Use Bounded Type Parameters?

Bounded types provide **compile-time guarantees** and allow **generic code to rely on specific behavior** from the type. Consider this example of a method that compares two elements:

```java
public static <T extends Comparable<T>> T max(T a, T b) {
    return (a.compareTo(b) > 0) ? a : b;
}
```

This ensures that any type passed to `max()` implements `Comparable<T>`, so the `compareTo` method is guaranteed to exist.

16.2.5 Trade-Offs and Limitations

While bounds increase safety and flexibility, they come with a few trade-offs:

- **Complexity**: Bounded declarations can make method signatures harder to read, especially with multiple type parameters.
- **Restrictions**: You cannot use both `extends` and `super` in a single type declaration.
- **Verbosity**: Overuse of bounds may clutter otherwise simple code.

Nonetheless, when used appropriately, bounded type parameters help strike the right balance between flexibility and safety.

16.2.6 Common Use Cases

- **Mathematical operations**: Using `T extends Number` ensures access to numeric methods.
- **Sorting and comparison**: Constraining to `Comparable<T>` allows safe comparisons.
- **Custom data structures**: When writing a collection class, bounds ensure only compatible types are allowed.

16.2.7 Summary

Bounded type parameters enhance generics by constraining what types can be used, ensuring that the code operates only on types with the required behavior. Upper bounds (`extends`) are common for accessing specific methods, while lower bounds (`super`) are useful when writing methods that work with collections of varying supertypes. Used wisely, bounded generics lead to cleaner, more reliable, and type-safe APIs.

16.3 Wildcards (`? extends`, `? super`)

16.3.1 Understanding Wildcards in Generics

Java generics allow classes, interfaces, and methods to operate on types specified as parameters. However, sometimes you need more flexibility than strict type parameters provide—especially when working with **collections of unknown but related types**. This is where **wildcards** come into play.

Wildcards enable **covariance** and **contravariance**, allowing generic code to accept broader or more restrictive sets of types. The wildcard `?` represents an unknown type, and with bounds (`extends` or `super`), it can express relationships between types more fluidly.

16.3.2 The Wildcard Syntax

There are three main forms of wildcards:

- `<?>`: An **unbounded wildcard**, representing any type.
- `<? extends Type>`: An **upper bounded wildcard**, representing a type that is `Type` or a subclass.
- `<? super Type>`: A **lower bounded wildcard**, representing a type that is `Type` or a superclass.

These allow flexibility when **reading from** or **writing to** generic structures like lists, particularly when the exact type isn't known or shouldn't be fixed.

16.3.3 Covariance with `? extends`

The `? extends` wildcard allows a method to accept a list of `Type` or any of its subclasses. It is used **when you only need to read from a structure** (i.e., "producer").

Example:

```
public static void printNumbers(List<? extends Number> list) {
    for (Number num : list) {
        System.out.println(num);
    }
}
```

This method can accept a `List<Integer>`, `List<Double>`, or any other `List` whose elements are subtypes of `Number`. However, **you cannot add elements to this list** because the compiler cannot determine the specific subtype.

Usage:

```
List<Integer> intList = Arrays.asList(1, 2, 3);
printNumbers(intList);  // Valid

List<Double> doubleList = Arrays.asList(1.1, 2.2);
printNumbers(doubleList);  // Also valid
```

This is an example of **covariance**—allowing a method to accept more specific subtypes.

Full runnable code:

```
import java.util.Arrays;
import java.util.List;

public class WildcardExample {

    public static void printNumbers(List<? extends Number> list) {
        for (Number num : list) {
            System.out.println(num);
        }
    }

    public static void main(String[] args) {
        List<Integer> intList = Arrays.asList(1, 2, 3);
        printNumbers(intList);  // Valid

        List<Double> doubleList = Arrays.asList(1.1, 2.2);
        printNumbers(doubleList);  // Also valid
    }
}
```

16.3.4 Contravariance with `? super`

The `? super` wildcard is used **when writing to a generic structure** (i.e., "consumer"). It allows adding instances of a specified type or its subtypes to the collection but restricts what can be read.

Example:

```
public static void addIntegers(List<? super Integer> list) {
    list.add(10);
    list.add(20);
}
```

This method can accept `List<Integer>`, `List<Number>`, or `List<Object>`, but **you can only safely read them as `Object`**.

Usage:

```
List<Number> numberList = new ArrayList<>();
addIntegers(numberList);  // Valid
```

```
List<Object> objectList = new ArrayList<>();
addIntegers(objectList);  // Valid
```

This pattern allows writing to a broader type while restricting read operations, demonstrating **contravariance**.

16.3.5 PECS: "Producer Extends, Consumer Super"

A helpful mnemonic is **PECS**:

- **Producer — Extends**
- **Consumer — Super**

Use `? extends` when you only need to **read/consume** data, and `? super` when you only need to **write/produce** data.

16.3.6 Limitations and Trade-offs

While wildcards add flexibility, they come with limitations:

- **Loss of type information**: When using `?`, the exact type is unknown, making some operations (e.g., adding elements) impossible without casting.
- **Verbosity and confusion**: Wildcard syntax can become complex in real-world APIs and may confuse readers.
- **No wildcard in generic method declarations**: Wildcards are mainly for method parameters, not for declaring new types.

Consider the following incorrect use:

```
List<? extends Number> numbers = new ArrayList<Integer>();
numbers.add(5); // Compile-time error
```

Even though `5` is an `Integer`, the compiler disallows adding because the exact subtype of `? extends Number` is unknown—it might not be `Integer`.

16.3.7 Best Practices

- Use **wildcards in method parameters**, not in return types or class-level declarations.
- Apply `? extends` when you **only need to read** from a collection.
- Apply `? super` when you **only need to write** to a collection.
- Use wildcards when designing **flexible APIs**, particularly for libraries or frameworks.

16.3.8 Summary

Wildcards are a critical feature of Java generics, enabling flexible and type-safe operations on collections and generic structures. The distinction between `? extends` and `? super` allows code to safely interact with a hierarchy of types without sacrificing generality. By mastering wildcard usage—and applying the PECS principle—developers can write robust, reusable, and expressive generic code while avoiding pitfalls like unsafe casts or overly rigid type constraints.

16.4 Generics in Collections

16.4.1 Generics in Java Collections Framework

One of the most impactful applications of Java generics is in the **Java Collections Framework (JCF)**. Collections such as `List`, `Set`, `Map`, and `Queue` are all **generic classes** that allow developers to specify the type of elements they store. This enables **compile-time type checking**, improves readability, and **eliminates the need for explicit casting**, leading to safer and more maintainable code.

Before Java 5 introduced generics, collections were untyped and required manual casting, which could lead to runtime `ClassCastException`. With generics, these errors are caught at compile time.

16.4.2 Using Generic Collections

Let's look at some basic generic collection types and how they are used.

`List<T>` Example

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");

for (String name : names) {
    System.out.println(name.toUpperCase());
}
```

Here, the `List<String>` ensures that only `String` values can be added. If you try to add an `Integer`, the compiler will produce an error:

```
names.add(123); // Compile-time error
```

Without generics, this error would only occur at runtime when casting, increasing the chance

of bugs.

`Map<K, V>` Example

Generic maps allow you to define both key and value types:

```
Map<Integer, String> userMap = new HashMap<>();
userMap.put(101, "Alice");
userMap.put(102, "Bob");

for (Map.Entry<Integer, String> entry : userMap.entrySet()) {
    System.out.println("ID: " + entry.getKey() + ", Name: " + entry.getValue());
}
```

This ensures that only an `Integer` can be used as a key and a `String` as a value. Type mismatches are caught during compilation.

16.4.3 Common Pitfalls Avoided with Generics

Generics help eliminate several frequent errors found in non-generic code:

- **Casting Errors:** Without generics, retrieving an object from a collection requires a cast.

  ```
  List names = new ArrayList(); // Non-generic
  names.add("Charlie");
  String name = (String) names.get(0); // Must cast manually
  ```

  If the list contains a non-`String` object, this cast will throw a `ClassCastException`. Generics avoid this.

- **Unchecked Warnings:** Using raw types (non-generic collections) results in compiler warnings.

  ```
  List rawList = new ArrayList(); // Warning: unchecked conversion
  ```

- **Unsafe Element Types:** Generics enforce that collections are homogeneous in type, reducing confusion and bugs in multi-type operations.

16.4.4 Practical Example: Generic List of Custom Objects

```
class Book {
    private String title;

    public Book(String title) {
        this.title = title;
    }
```

```
    public String getTitle() {
        return title;
    }
}

List<Book> library = new ArrayList<>();
library.add(new Book("Effective Java"));
library.add(new Book("Clean Code"));

for (Book book : library) {
    System.out.println(book.getTitle());
}
```

In this example, the type-safe `List<Book>` ensures you can only add `Book` objects. No need for casting, and the code is easier to maintain.

Full runnable code:

```
import java.util.ArrayList;
import java.util.List;

public class LibraryDemo {
    public static void main(String[] args) {
        class Book {
            private String title;

            public Book(String title) {
                this.title = title;
            }

            public String getTitle() {
                return title;
            }
        }

        List<Book> library = new ArrayList<>();
        library.add(new Book("Effective Java"));
        library.add(new Book("Clean Code"));

        for (Book book : library) {
            System.out.println(book.getTitle());
        }
    }
}
```

16.4.5 Conclusion

Generics bring structure and safety to the Java Collections Framework. By explicitly defining the types of elements stored in collections, developers avoid the pitfalls of runtime casting and gain the benefits of clearer, more expressive code. Whether managing a list of strings, mapping keys to values, or handling custom object types, generics ensure the right types are

used in the right places—reducing bugs and improving code reliability. For modern Java development, **leveraging generics in collections is not optional—it's essential**.

Chapter 17.

Enums, Records, and Sealed Classes

1. Using Enums in OOD
2. Java Records for Immutable Data Models
3. Sealed Classes and Controlled Inheritance

17 Enums, Records, and Sealed Classes

17.1 Using Enums in OOD

17.1.1 What Are Enums?

In Java, an **enum** (short for *enumeration*) is a special type used to define a **fixed set of constants**. Introduced in Java 5, enums provide a type-safe and object-oriented way to model a limited number of possible values, such as days of the week, directions, states, or operation types.

Instead of using arbitrary strings or integers, enums improve readability, enforce compile-time checking, and offer a structured way to encapsulate related behavior alongside the constants.

17.1.2 Defining and Using Enums

A basic enum looks like this:

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
```

You can use it in code as:

```
Day today = Day.MONDAY;

if (today == Day.MONDAY) {
    System.out.println("Start of the week!");
}
```

This approach ensures **type safety**—you can't assign a value outside the predefined set—and avoids typos or invalid constants often seen with string or integer constants.

Full runnable code:

```
public class EnumDemo {
    // Define the enum
    public enum Day {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
    }

    public static void main(String[] args) {
        Day today = Day.MONDAY;

        if (today == Day.MONDAY) {
            System.out.println("Start of the week!");
        } else {
            System.out.println("Not Monday");
```

```
        }
    }
}
```

17.1.3 Enums with Fields, Constructors, and Methods

Java enums are more powerful than simple constants. You can associate fields and behavior with each enum constant.

```
public enum Operation {
    ADD("+") {
        public double apply(double x, double y) {
            return x + y;
        }
    },
    SUBTRACT("-") {
        public double apply(double x, double y) {
            return x - y;
        }
    };

    private final String symbol;

    Operation(String symbol) {
        this.symbol = symbol;
    }

    public String getSymbol() {
        return symbol;
    }

    public abstract double apply(double x, double y);
}
```

Usage:

```
double result = Operation.ADD.apply(3, 5);
System.out.println("Result: " + result); // Output: Result: 8.0
```

This pattern allows enums to behave like classes, encapsulating logic related to each constant.

Full runnable code:

```
public class EnumOperationDemo {
    public enum Operation {
        ADD("+") {
            public double apply(double x, double y) {
                return x + y;
            }
        },
        SUBTRACT("-") {
```

```
            public double apply(double x, double y) {
                return x - y;
            }
        };

        private final String symbol;

        Operation(String symbol) {
            this.symbol = symbol;
        }

        public String getSymbol() {
            return symbol;
        }

        public abstract double apply(double x, double y);
    }

    public static void main(String[] args) {
        double result = Operation.ADD.apply(3, 5);
        System.out.println("Result: " + result); // Output: Result: 8.0
    }
}
```

17.1.4 Enums in Object-Oriented Design

Enums are often used in **object-oriented design** to model fixed categories of behavior or states. They are especially helpful in the following design contexts:

State Machines

You can represent finite states and transitions between them using enums.

```
enum TrafficLight {
    RED, GREEN, YELLOW;
}
```

Pairing this with switch statements or state-specific behavior methods can create clean, understandable models for systems with well-defined states.

Command Pattern

Enums can serve as lightweight implementations of the **Command pattern**, particularly when each constant represents a distinct action:

```
enum Command {
    START {
        public void execute() { System.out.println("Starting..."); }
    },
    STOP {
        public void execute() { System.out.println("Stopping..."); }
```

```
    };

    public abstract void execute();
}
```

This provides a concise and readable way to encapsulate commands without requiring separate classes for each.

Full runnable code:

```
public class CommandEnumDemo {
    enum Command {
        START {
            public void execute() { System.out.println("Starting..."); }
        },
        STOP {
            public void execute() { System.out.println("Stopping..."); }
        };

        public abstract void execute();
    }

    public static void main(String[] args) {
        Command.START.execute(); // Output: Starting...
        Command.STOP.execute();  // Output: Stopping...
    }
}
```

17.1.5 Advantages of Enums

- **Type Safety**: Only valid values can be used.
- **Readability**: Expressive names replace cryptic literals.
- **Namespace Management**: Constants are grouped together logically.
- **Encapsulation**: Behavior can be attached directly to enum constants.
- **Switch Support**: Works well with `switch` statements for branching logic.

17.1.6 Conclusion

Enums are a powerful tool in Java's object-oriented toolkit. They elevate constants into rich, expressive types that can carry behavior, state, and data. Whether modeling a fixed set of values, implementing state machines, or organizing command logic, enums improve both safety and clarity. When used thoughtfully, they reduce bugs, enhance maintainability, and align perfectly with the principles of clean, robust design.

17.2 Java Records for Immutable Data Models

17.2.1 Introduction to Java Records

Introduced in Java 14 as a preview and standardized in Java 16, **records** are a special kind of class in Java designed to model **immutable data**. They drastically reduce boilerplate by automatically generating constructors, accessors, `equals()`, `hashCode()`, and `toString()` methods for you.

Records are particularly useful when you need simple **data carriers**—also called *value objects*—that are defined more by the data they contain than the behavior they perform.

17.2.2 Declaring a Record

Here's how you declare a record:

```
public record Person(String name, int age) {}
```

Behind the scenes, Java automatically generates:

- A `final` class.
- A canonical constructor (`public Person(String name, int age)`).
- Getter-like accessor methods (`name()` and `age()`).
- `equals()`, `hashCode()`, and `toString()` methods.

Usage:

```
Person p = new Person("Alice", 30);
System.out.println(p.name()); // prints "Alice"
System.out.println(p); // prints "Person[name=Alice, age=30]"
```

This eliminates the need for verbose code often found in traditional POJOs.

17.2.3 Records vs Traditional POJOs

Consider a typical POJO:

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

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

```
    public String getName() { return name; }
    public int getAge() { return age; }

    @Override
    public boolean equals(Object o) { /* standard implementation */ }
    @Override
    public int hashCode() { /* standard implementation */ }
    @Override
    public String toString() { /* standard implementation */ }
}
```

Now compare this to:

```
public record Person(String name, int age) {}
```

Both provide the same core functionality, but the record version is cleaner, more concise, and easier to maintain.

17.2.4 Immutability and Finality

All fields in a record are **implicitly final**, and records themselves are **final classes**. This means:

- Once created, record instances cannot be modified.
- You cannot extend a record class.

This immutability makes records ideal for use in **functional programming**, **multi-threaded code**, or **data transfer objects (DTOs)** where consistency and thread safety are important.

17.2.5 Custom Behavior in Records

You can still add methods, static fields, and custom constructors:

```
public record Rectangle(int width, int height) {
    public int area() {
        return width * height;
    }
}
```

You can also customize the constructor to enforce invariants:

```
public record Rectangle(int width, int height) {
    public Rectangle {
        if (width <= 0 || height <= 0) {
            throw new IllegalArgumentException("Dimensions must be positive");
        }
```

```
    }
}
```

This flexibility lets records participate in robust design without losing their simplicity.

Full runnable code:

```
public record Rectangle(int width, int height) {
    // Custom constructor to enforce positive dimensions
    public Rectangle {
        if (width <= 0 || height <= 0) {
            throw new IllegalArgumentException("Dimensions must be positive");
        }
    }

    // Custom method to calculate area
    public int area() {
        return width * height;
    }

    public static void main(String[] args) {
        Rectangle rect = new Rectangle(5, 10);
        System.out.println("Area: " + rect.area()); // Output: Area: 50

        // Uncommenting the following line will throw IllegalArgumentException
        // Rectangle invalidRect = new Rectangle(-3, 4);
    }
}
```

17.2.6 Use Cases: Modeling Value Objects

Records are best suited for:

- **Value objects**: small, immutable data holders.
- **DTOs**: objects used for transferring data between layers or services.
- **Configuration settings**: where values don't change after creation.
- **Map keys or data grouping**: where `equals()` and `hashCode()` are important.

Example:

```
record Address(String city, String country) {}

record Customer(String name, Address address) {}
```

Here, both `Customer` and `Address` act as pure data containers, well-suited for serialization, database interaction, or API contracts.

17.2.7 Conclusion

Java records streamline the creation of simple, immutable data models, bringing clarity and conciseness to object-oriented design. By auto-generating common boilerplate and enforcing immutability, they encourage clean code and reduce opportunities for error. When designing classes whose identity is purely based on their data, records are a modern, elegant alternative to traditional POJOs.

17.3 Sealed Classes and Controlled Inheritance

17.3.1 What Are Sealed Classes?

In traditional Java inheritance, any class marked as `public` or `protected` can be extended by any other class—unless it's marked as `final`. This flexible model is powerful but sometimes too open. In cases where you want **explicit control over subclassing**, Java 15 (as a preview) and Java 17 (as a standard feature) introduced **sealed classes**.

A **sealed class** lets you define a **closed set of permitted subclasses**. It allows superclass authors to declare, "Only these specific classes can extend me," providing a safer and more predictable class hierarchy.

17.3.2 Syntax: `sealed`, `permits`, and `non-sealed`

Sealed class declarations involve three main keywords:

1. **`sealed`** – Marks the class whose inheritance is restricted.
2. **`permits`** – Explicitly lists the permitted subclasses.
3. **`non-sealed`** – Marks a subclass that removes sealing, allowing unrestricted subclassing.
4. **`final`** – A permitted subclass can still be marked `final` to prohibit further extension.

Here's a basic example:

```
public sealed class Vehicle permits Car, Truck {}

public final class Car extends Vehicle {}

public non-sealed class Truck extends Vehicle {}
```

- `Car` is `final` — no more inheritance allowed.
- `Truck` is `non-sealed` — others can extend `Truck`, but not `Vehicle`.
- Any other class not listed in `permits` will cause a compile-time error if it tries to extend `Vehicle`.

17.3.3 Practical Example: Modeling Payment Types

Let's model a controlled payment system:

```
public sealed class Payment permits CreditCard, PayPal, Crypto {}

public final class CreditCard extends Payment {
    // implementation
}

public final class PayPal extends Payment {
    // implementation
}

public non-sealed class Crypto extends Payment {
    // allows further extensions like Bitcoin or Ethereum
}
```

This ensures only `CreditCard`, `PayPal`, and `Crypto` are valid payment types. It avoids unauthorized extensions and keeps the class hierarchy tightly scoped.

17.3.4 Why Use Sealed Classes?

Enhanced Security and Predictability

By explicitly declaring permitted subclasses, sealed classes help ensure that your hierarchy cannot be arbitrarily extended. This is especially important in:

- **Security-sensitive domains**: Prevent third-party code from introducing unverified behavior.
- **Public APIs**: Maintain strict control over object models exposed to clients.

Better Maintainability

Knowing exactly which classes belong to a sealed hierarchy simplifies reasoning about the code. There's no need to anticipate unforeseen subclasses when applying logic across types.

For instance, consider a `switch` expression:

```
static String describe(Payment payment) {
    return switch (payment) {
        case CreditCard c -> "Paid by Credit Card";
        case PayPal p -> "Paid with PayPal";
        case Crypto crypto -> "Paid with Crypto";
    };
}
```

Because `Payment` is sealed and the compiler knows all its subtypes, the `switch` is **exhaustive**. If a new subtype is added, the compiler will require the switch to be updated, preventing silent omissions.

Full runnable code:

```
// Sealed class example demonstrating sealed, permits, non-sealed, and final

sealed class Vehicle permits Car, Truck {}

final class Car extends Vehicle {
    // Car is final - no further subclassing allowed
}

non-sealed class Truck extends Vehicle {
    // Truck allows unrestricted subclassing
}

class PickupTruck extends Truck {
    // This is allowed because Truck is non-sealed
}

public class Demo {
    static String describe(Vehicle vehicle) {
        return switch (vehicle) {
            case Car c -> "This is a Car";
            case Truck t -> "This is a Truck";
            default -> throw new IllegalArgumentException("Unexpected value: " + vehicle);
        };
    }

    public static void main(String[] args) {
        Vehicle car = new Car();
        Vehicle truck = new Truck();
        Vehicle pickup = new PickupTruck();

        System.out.println(describe(car));     // Output: This is a Car
        System.out.println(describe(truck));   // Output: This is a Truck
        System.out.println(describe(pickup));  // Output: This is a Truck
    }
}
```

Improved Pattern Matching Support

Java's recent enhancements in **pattern matching** work elegantly with sealed types. You can use sealed hierarchies to write concise, type-safe matching logic that the compiler can verify for completeness.

17.3.5 Design Considerations

- **Use sealed classes to define finite type sets**, especially for domain models like commands, tokens, events, or status types.
- Prefer `final` subclasses when you want to fully lock the hierarchy.
- Use `non-sealed` when extending the hierarchy is acceptable downstream—but in a controlled fashion.

17.3.6 Limitations and Rules

- All permitted subclasses **must reside in the same module** (or package, if modules aren't used).
- Permitted subclasses must explicitly **declare themselves as `final`, `sealed`, or `non-sealed`**.
- Sealed classes cannot be **anonymous** or **local**.

17.3.7 Conclusion

Sealed classes represent a powerful enhancement to Java's type system, allowing developers to create more robust and predictable class hierarchies. By limiting which classes can extend a given type, sealed classes provide greater control, improve pattern matching safety, and reduce the chance of misuse in large codebases. They're a natural choice when modeling domain-specific hierarchies where only certain subclasses make logical sense.

Chapter 18.

Lambda Expressions and Functional Interfaces

1. What is Functional Programming in Java?
2. Lambdas in OOP Design
3. `Function`, `Predicate`, `Consumer` Interfaces
4. Stream API and Composition

18 Lambda Expressions and Functional Interfaces

18.1 What is Functional Programming in Java?

18.1.1 Introduction to Functional Programming in Java

Functional programming (FP) is a paradigm that treats computation as the evaluation of mathematical functions and avoids changing state or mutable data. While Java has historically been an imperative, object-oriented language, Java 8 introduced powerful support for functional programming concepts, enabling developers to write more expressive, concise, and maintainable code.

This shift was made possible through the introduction of **lambda expressions**, **functional interfaces**, and **streams**, allowing Java to embrace many benefits of FP while remaining object-oriented at its core.

18.1.2 Core Concepts of Functional Programming

Functional programming focuses on a few key principles:

1. **Immutability** – Data is not changed after it is created. Instead, new data is derived from existing data.
2. **First-Class Functions** – Functions can be assigned to variables, passed as arguments, and returned from other functions.
3. **Higher-Order Functions** – Functions that take other functions as parameters or return them.
4. **Pure Functions** – Functions without side effects; the output depends only on the input.
5. **Declarative Style** – Code describes *what* should be done, not *how* to do it.

Java supports many of these features, especially from version 8 onward, making it easier to write clear and concise logic.

18.1.3 Javas Functional Programming Support

Java introduced the following key elements to support functional programming:

- **Lambda Expressions** – Compact syntax for writing anonymous functions.
- **Functional Interfaces** – Interfaces with a single abstract method, used as function types.
- **Stream API** – Enables processing of collections in a functional style.
- **Optional** – A container object used to avoid null values, encouraging safer and more

predictable code.

18.1.4 Imperative vs Functional: A Simple Comparison

To understand the impact of functional programming in Java, consider the task of filtering a list of names that start with "A".

Imperative style:

```
List<String> names = List.of("Alice", "Bob", "Alex", "Brian");
List<String> result = new ArrayList<>();

for (String name : names) {
    if (name.startsWith("A")) {
        result.add(name);
    }
}
System.out.println(result);
```

Functional style:

```
List<String> names = List.of("Alice", "Bob", "Alex", "Brian");
List<String> result = names.stream()
                           .filter(name -> name.startsWith("A"))
                           .collect(Collectors.toList());

System.out.println(result);
```

In the functional version, the code is more concise, expressive, and easier to reason about. It avoids mutability and abstracts away iteration logic.

18.1.5 Functions as First-Class Citizens

Java does not have true first-class functions like purely functional languages, but **functional interfaces and lambdas** simulate this capability effectively.

Consider a method that takes a function as a parameter:

```
public static int operate(int a, int b, BiFunction<Integer, Integer, Integer> operation) {
    return operation.apply(a, b);
}
```

We can pass different lambda expressions:

```
int sum = operate(5, 3, (x, y) -> x + y);
int product = operate(5, 3, (x, y) -> x * y);
```

```
System.out.println(sum);      // 8
System.out.println(product); // 15
```

Here, `BiFunction` is a functional interface, and the lambda expressions are passed as values—demonstrating functions as first-class citizens.

18.1.6 Benefits of Functional Style in Java

- **Concise Code** – Lambdas and streams reduce boilerplate, especially when working with collections.
- **Improved Readability** – Declarative style describes intent rather than control flow.
- **Safer Code** – Immutability reduces bugs related to shared mutable state.
- **Parallelization** – Stream API makes it easier to parallelize data operations safely.

18.1.7 Caution and Balance

Functional programming is powerful, but overusing it in Java can lead to less readable code, especially when chaining complex lambda expressions. It's important to balance FP with OOP principles and not force FP where traditional OOP solutions are more intuitive.

18.1.8 Conclusion

Functional programming in Java empowers developers with new ways to write clearer, more maintainable code. By leveraging lambdas, functional interfaces, and streams, Java code can become more expressive while maintaining the strengths of object-oriented design. As you continue through this chapter, you'll see how these features integrate with traditional Java architecture to enhance modularity, flexibility, and clarity in software design.

18.2 Lambdas in OOP Design

18.2.1 Introduction

Lambda expressions, introduced in Java 8, brought functional programming concepts to the object-oriented Java language. They offer a concise way to represent instances of functional interfaces—interfaces with a single abstract method—allowing developers to treat behavior as a first-class value. In object-oriented design (OOD), lambdas enhance expressiveness,

reduce boilerplate code, and improve modularity when behavior needs to be passed around dynamically.

This section explores how lambda expressions integrate with OOP practices, simplify common patterns such as callbacks and event handling, and improve code readability and maintainability.

18.2.2 Syntax and Structure of Lambda Expressions

Lambda expressions have a compact syntax:

```
(parameters) -> expression
```

Or with a block body:

```
(parameters) -> {
    // statements
    return result;
}
```

If the lambda has one parameter, parentheses are optional:

```
x -> x * x
```

For example, using a `Comparator<String>` to sort a list alphabetically in reverse order:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Collections.sort(names, (a, b) -> b.compareTo(a));
```

This replaces the more verbose anonymous inner class:

```
Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

The lambda expression eliminates unnecessary boilerplate and focuses on the logic itself.

18.2.3 Lambdas and Anonymous Classes

Before Java 8, implementing short-lived behaviors like event handlers required anonymous inner classes. Lambdas offer a simpler, more expressive alternative.

Anonymous Class Example (pre-Java 8):

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});
```

With Lambda Expression:

```
button.addActionListener(e -> System.out.println("Button clicked!"));
```

This reduces verbosity while keeping the intent clear. Lambdas are particularly useful in GUI event handling, stream processing, and anywhere a functional interface is expected.

18.2.4 Integrating Lambdas with Object-Oriented Principles

Lambdas don't replace OOP—they complement it. In fact, they allow better modularity and encapsulation by enabling behavior injection. This aligns well with design principles such as the **Strategy Pattern**, where a behavior is selected at runtime.

Without Lambda (Strategy via Interface):

```
interface PaymentStrategy {
    void pay(double amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " with credit card");
    }
}
```

With Lambda:

```
PaymentStrategy creditCard = amount ->
    System.out.println("Paid $" + amount + " with credit card");
```

By defining the strategy behavior inline, you avoid creating a dedicated class unless the behavior is reused. This encourages simpler, more flexible designs while adhering to polymorphism.

18.2.5 Practical Example: Filtering with Lambdas

Consider a list of products. You want to filter those under a certain price:

```
List<Product> products = getInventory();
List<Product> cheapProducts = products.stream()
    .filter(p -> p.getPrice() < 50)
    .collect(Collectors.toList());
```

Here, the `filter` method accepts a `Predicate<Product>`—a functional interface. The lambda `p -> p.getPrice() < 50` provides that behavior. The same task with anonymous classes would require several lines of boilerplate code.

18.2.6 Readability and Maintainability

Lambda expressions often make code easier to read and maintain by expressing intent more directly and reducing clutter. They focus on the "what" rather than the "how." This leads to:

- **Shorter, cleaner code**: Especially beneficial for inline operations like sorting or filtering.
- **Improved modularity**: Lambdas make it easy to pass behavior without creating separate classes.
- **Reduced duplication**: Less boilerplate leads to more focused, DRY-compliant code.

However, overuse or misuse—especially with deeply nested or complex lambdas—can hurt readability. When a lambda grows beyond a few lines, it's often better to extract it into a method or named class.

18.2.7 Conclusion

Lambda expressions provide a bridge between object-oriented and functional paradigms in Java. They simplify event handling, enhance behavioral design patterns, and reduce verbosity without sacrificing the clarity of OOP principles. When used thoughtfully, lambdas lead to more expressive, maintainable, and modular Java applications—especially in systems where behavior needs to be passed, varied, or executed in response to events.

18.3 `Function`, `Predicate`, `Consumer` Interfaces

18.3.1 Introduction

Java 8 introduced a set of standard functional interfaces in the `java.util.function` package to support lambda expressions and method references. Among the most commonly used are `Function<T, R>`, `Predicate<T>`, and `Consumer<T>`. These interfaces form the foundation for functional-style operations in Java, particularly when working with collections, streams, or behavior injection.

This section explores these interfaces in detail, showing how they help developers pass logic as parameters, compose operations, and write concise, expressive code.

18.3.2 `Function<T, R>` Mapping and Transformation

The `Function` interface represents a function that accepts one argument and returns a result:

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Example: Transforming Strings to Their Length

```
import java.util.function.Function;

public class FunctionExample {
    public static void main(String[] args) {
        Function<String, Integer> stringLength = s -> s.length();

        System.out.println(stringLength.apply("Hello"));  // Output: 5
    }
}
```

`Function` is widely used in mapping operations, such as transforming data from one type to another in `Stream.map()`.

Composition with `andThen()` and `compose()`

You can chain `Function` instances to build pipelines:

```
Function<String, String> trim = String::trim;
Function<String, Integer> toLength = String::length;
Function<String, Integer> trimThenLength = trim.andThen(toLength);

System.out.println(trimThenLength.apply("  Java  ")); // Output: 4
```

`compose()` performs the opposite order of `andThen()`.

18.3.3 `PredicateT` Testing and Filtering

The `Predicate` interface represents a boolean-valued function of one argument:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Example: Filtering Even Numbers

```
import java.util.function.Predicate;
import java.util.Arrays;
import java.util.List;

public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;

        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
        numbers.stream()
               .filter(isEven)
               .forEach(System.out::println); // Output: 2 4 6
    }
}
```

Chaining with `and()`, `or()`, and `negate()`

Predicates can be composed for complex logic:

```
Predicate<String> nonEmpty = s -> !s.isEmpty();
Predicate<String> startsWithA = s -> s.startsWith("A");
Predicate<String> complexCheck = nonEmpty.and(startsWithA);

System.out.println(complexCheck.test("Apple"));  // true
System.out.println(complexCheck.test(""));       // false
```

This makes `Predicate` especially useful for filtering collections.

18.3.4 `ConsumerT` Performing Actions

The `Consumer` interface represents an operation that accepts a single input and returns no result:

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

It is typically used to perform actions such as printing or modifying objects.

Example: Printing Elements

```
import java.util.function.Consumer;
import java.util.Arrays;
import java.util.List;

public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> printUpperCase = s -> System.out.println(s.toUpperCase());

        List<String> names = Arrays.asList("alice", "bob", "charlie");
        names.forEach(printUpperCase);
        // Output: ALICE BOB CHARLIE
    }
}
```

Chaining with `andThen()`

Multiple actions can be chained using `andThen()`:

```
Consumer<String> greet = s -> System.out.print("Hello, ");
Consumer<String> printName = s -> System.out.println(s);
Consumer<String> greetAndPrint = greet.andThen(printName);

greetAndPrint.accept("Sam"); // Output: Hello, Sam
```

18.3.5 Composition and Use in Streams

These functional interfaces are often used together in streams to create readable and concise code pipelines:

```
import java.util.function.*;
import java.util.*;

public class StreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "", "Bob", "Anna");

        Predicate<String> nonEmpty = s -> !s.isEmpty();
        Function<String, Integer> length = String::length;
        Consumer<Integer> print = System.out::println;

        names.stream()
             .filter(nonEmpty)
             .map(length)
             .forEach(print);
        // Output: 5 3 4
    }
}
```

This demonstrates the combined power of `Predicate`, `Function`, and `Consumer` in a processing pipeline.

18.3.6 Conclusion

`Function<T, R>`, `Predicate<T>`, and `Consumer<T>` are essential tools in Java's functional programming toolbox. They allow behavior to be abstracted, composed, and reused, leading to cleaner, more expressive, and modular code. By understanding how to apply and combine these interfaces, developers can write Java that's both object-oriented and functionally fluent, enhancing code readability, flexibility, and testability across modern applications.

18.4 Stream API and Composition

18.4.1 Introduction to the Stream API

Java's Stream API, introduced in Java 8, revolutionized how developers process collections by providing a functional, declarative approach to data manipulation. Instead of writing verbose, imperative loops and conditional code, streams enable a pipeline-style syntax to describe **what** should be done, not **how** it is done.

Streams process sequences of elements (like collections, arrays, or I/O channels) and support **functional-style operations** such as filtering, mapping, and reduction. This leads to more readable, maintainable code and opens up possibilities for parallel execution and lazy evaluation.

18.4.2 Declarative, Parallel, and Efficient Processing

Declarative Approach

With streams, you express operations on collections declaratively. For example, rather than manually iterating through a list to filter and transform elements, you describe the pipeline of operations succinctly:

```
List<String> names = List.of("Alice", "Bob", "Charlie", "David");

List<String> filteredNames = names.stream()
                                  .filter(name -> name.startsWith("A"))
                                  .map(String::toUpperCase)
                                  .collect(Collectors.toList());

System.out.println(filteredNames); // Output: [ALICE]
```

The code states **what** is done — filter names starting with "A" and convert them to uppercase — without specifying iteration mechanics.

Parallel Processing

Streams make it simple to leverage multi-core processors by switching to parallel mode:

```
List<String> largeList = /* large dataset */;
long count = largeList.parallelStream()
                      .filter(s -> s.length() > 5)
                      .count();
```

This parallelizes operations internally, improving performance on large datasets without additional threading code.

Lazy Evaluation

Stream operations are **lazy** — intermediate operations (like `filter` and `map`) are not executed until a terminal operation (like `collect` or `reduce`) is invoked. This enables optimizations such as short-circuiting and efficient chaining.

18.4.3 Core Stream Operations

The Stream API provides a rich set of operations classified as **intermediate** and **terminal**:

- **Intermediate operations** return a new stream, allowing chaining (e.g., `filter`, `map`, `sorted`).
- **Terminal operations** produce a result or side effect and end the stream pipeline (e.g., `collect`, `forEach`, `reduce`).

Example: Filtering, Mapping, and Reducing

Let's explore a pipeline that filters, transforms, and aggregates:

```
import java.util.Arrays;
import java.util.List;

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(2, 3, 5, 8, 10, 12);

        // Sum of squares of even numbers
        int sum = numbers.stream()
                         .filter(n -> n % 2 == 0)      // Keep even numbers
                         .map(n -> n * n)              // Square each number
                         .reduce(0, Integer::sum);     // Sum the squares

        System.out.println("Sum of squares of even numbers: " + sum);
```

```
        // Output: 4 + 64 + 100 + 144 = 312
    }
}
```

- **`filter`**: selects elements based on a predicate.
- **`map`**: transforms elements via a function.
- **`reduce`**: aggregates elements into a single result.

This pipeline illustrates a clear, concise way to express data processing.

18.4.4 Composing Streams with Lambdas and Functional Interfaces

Streams integrate seamlessly with lambda expressions and functional interfaces like `Predicate`, `Function`, and `Consumer`. This combination promotes a **clean separation of concerns** and **reusability**.

For example, you can extract predicates and functions as reusable components:

```
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.List;
import java.util.stream.Collectors;

public class StreamComposition {
    public static void main(String[] args) {
        List<String> fruits = List.of("apple", "banana", "pear", "avocado", "blueberry");

        Predicate<String> startsWithA = s -> s.startsWith("a");
        Function<String, String> capitalize = s -> s.substring(0, 1).toUpperCase() + s.substring(1);

        List<String> result = fruits.stream()
                                    .filter(startsWithA)
                                    .map(capitalize)
                                    .collect(Collectors.toList());

        System.out.println(result); // Output: [Apple, Avocado]
    }
}
```

By composing streams with well-defined functional interfaces, you keep each operation modular, readable, and testable.

18.4.5 Advanced Composition: Chaining and Lazy Evaluation

You can chain multiple operations for complex pipelines without clutter:

```
numbers.stream()
       .filter(n -> n > 5)
       .map(n -> n * 3)
       .sorted()
       .forEach(System.out::println);
```

Each intermediate operation produces a new stream, enabling flexible combinations.

Since streams are lazy, no computation happens until a terminal operation (like `forEach`) triggers execution. This means you can compose intricate pipelines without performance penalties.

18.4.6 Benefits for Clean Design

- **Readability**: Declarative pipelines closely match business logic and domain language.
- **Maintainability**: Modular, composable lambdas and streams simplify understanding and modifications.
- **Reusability**: Functional interfaces extracted from stream operations can be reused and tested independently.
- **Parallelism**: Easy switch to parallel streams improves scalability.
- **Reduced Boilerplate**: No explicit loops or temporary collections needed.

18.4.7 Summary

The Stream API in Java provides a powerful, expressive way to process data collections by combining functional interfaces, lambda expressions, and declarative pipelines. Its support for lazy and parallel evaluation enhances performance and scalability while maintaining clean, readable code.

By mastering stream composition and common operations like `filter`, `map`, and `reduce`, Java developers can write concise, maintainable, and efficient code that aligns perfectly with modern object-oriented and functional programming principles.

Chapter 19.

Case Study 1 Designing a Library System

1. Requirements Analysis
2. Domain Modeling
3. Code Implementation

19 Case Study 1 Designing a Library System

19.1 Requirements Analysis

19.1.1 The Importance of Requirements Analysis

Before diving into design or coding, thoroughly understanding and defining the system requirements is essential. Requirements analysis serves as the foundation for successful software development, ensuring that the final product meets user needs and business goals. It helps clarify what the system must do, sets realistic expectations, and guides architectural and design decisions.

Poorly gathered or ambiguous requirements often lead to costly rework, missed deadlines, and systems that don't fulfill their intended purpose. In the context of a library system, this phase involves close collaboration with stakeholders such as librarians, patrons, and administrators to capture a clear and comprehensive picture of the system's desired features and constraints.

19.1.2 Typical Functional Requirements of a Library System

A library system, while seemingly straightforward, involves various functions that support daily operations and user interactions. Key functional requirements generally include:

- **Book Catalog Management**: Ability to add, update, and remove books from the catalog, including tracking book details like title, author, ISBN, genre, and availability status.
- **User Management**: Managing user profiles such as library members and staff, including registration, authentication, and role-based access controls.
- **Borrowing and Returning**: Support for checking out books to users, recording loan dates, due dates, and handling returns.
- **Reservation System**: Allowing users to place holds or reservations on books that are currently checked out.
- **Search and Browse**: Enabling users to search the catalog by various criteria (title, author, keywords) and browse categories or genres.
- **Overdue and Fines**: Tracking overdue items and calculating fines for late returns.
- **Notifications**: Sending reminders or alerts to users about due dates, reservations ready for pickup, or account issues.
- **Reporting**: Generating reports on book circulation, user activity, inventory status, etc.

These functional requirements define the core capabilities that the system must deliver to support library operations effectively.

19.1.3 Non-Functional Requirements

While functional requirements describe **what** the system should do, non-functional requirements specify **how** the system performs and behaves under various conditions. They address quality attributes critical for system acceptance:

- **Performance**: The system should respond to user queries (e.g., search results) within a reasonable time, even with large catalogs or many concurrent users.
- **Usability**: The user interface must be intuitive for both staff and patrons, minimizing the learning curve and errors.
- **Security**: User authentication and authorization mechanisms must protect sensitive data and prevent unauthorized access.
- **Scalability**: The system should accommodate growth in the number of users and catalog size without degradation.
- **Reliability**: The system must be robust, ensuring data integrity and availability, even in the event of failures.
- **Maintainability**: The design should allow easy updates, bug fixes, and feature additions.

Capturing these non-functional aspects early guides technology choices and architectural patterns that support long-term system viability.

19.1.4 Translating Requirements into Use Cases and User Stories

A practical step after gathering requirements is to convert them into use cases or user stories, which describe interactions between users and the system in concrete scenarios. This approach helps clarify requirements from an end-user perspective and facilitates communication among stakeholders and developers.

Use Case Example: *"Borrow Book"*

- **Actor**: Library Member
- **Precondition**: Member is registered and logged in; the book is available.
- **Basic Flow**:
    1. Member searches for a book.
    2. Selects the book and requests to borrow it.
    3. System checks availability and user's borrowing limits.
    4. System records the loan with due date.
    5. Member receives confirmation.
- **Postcondition**: Book status updated to loaned, loan details stored.

User Story Example: *As a library member, I want to reserve a book that is currently checked out so that I can borrow it once it becomes available.*

Breaking down requirements into such scenarios helps in designing classes, interfaces, and workflows that directly support user needs.

19.1.5 Sample Requirements Checklist for a Library System

Requirement ID	Description	Priority	Notes
FR-001	Add, update, delete books	High	CRUD operations on catalog
FR-002	Register and authenticate users	High	Support roles: member, staff
FR-003	Borrow and return books	High	Enforce borrowing rules
FR-004	Search catalog	High	Support multiple filters
FR-005	Reserve books	Medium	Notify users on availability
FR-006	Calculate overdue fines	Medium	Fine structure configurable
NFR-001	Response time < 2 seconds	High	Under 1000 concurrent users
NFR-002	Secure login with encrypted passwords	High	Use industry-standard hashing
NFR-003	Mobile-friendly UI	Low	Future enhancement

19.1.6 Summary

Effective requirements analysis is a critical first step in designing any software system, including a library management system. Clear functional and non-functional requirements, expressed through use cases or user stories, provide a solid foundation for architecture and implementation. This structured approach minimizes ambiguity, aligns stakeholder expectations, and enables designing a maintainable, scalable, and user-friendly system.

19.2 Domain Modeling

19.2.1 Introduction to Domain Modeling

After gathering clear requirements, the next crucial step in designing a software system is **domain modeling**. Domain modeling bridges the gap between real-world concepts and their software representations. It involves identifying key entities, their attributes, and the relationships between them, mapping the problem domain into a conceptual model that guides design and implementation.

This process helps developers and stakeholders build a shared understanding of the system's structure and behavior. By visually representing entities and their connections, domain modeling lays a foundation for creating robust, maintainable, and extensible software.

19.2.2 Visualizing the Domain: UML Class Diagrams

Unified Modeling Language (UML) class diagrams are the most common tool used to depict domain models. These diagrams illustrate classes (entities), their attributes, methods (optional at this stage), and relationships such as associations, aggregations, compositions, and inheritance hierarchies.

A UML class diagram provides a bird's-eye view of the domain structure, making it easier to analyze and communicate design decisions. It acts as a blueprint for implementing classes in code, ensuring that the design aligns with requirements.

19.2.3 Key Domain Entities for a Library System

For our library system, several core domain entities naturally emerge from the requirements analysis. These include:

- **Book** Represents the physical or digital book that the library manages. *Attributes*: `isbn`, `title`, `author`, `publisher`, `publicationYear`, `genre`, `copiesAvailable` The Book entity encapsulates all essential data about library materials.
- **Member** Represents a library user who borrows and returns books. *Attributes*: `memberId`, `name`, `email`, `phoneNumber`, `membershipDate`, `maxBooksAllowed` Members have borrowing privileges and account details.
- **Loan** Represents the borrowing of a specific book copy by a member. *Attributes*: `loanId`, `borrowDate`, `dueDate`, `returnDate`, `status` This entity tracks the lifecycle of book loans.
- **Librarian** Represents staff who manage catalog and member services. *Attributes*: `employeeId`, `name`, `role`, `workSchedule` Librarians have permissions distinct from regular members.

Additional supporting entities might include **Reservation**, **Fine**, or **Catalog**, depending on the scope, but the above cover the core functionality.

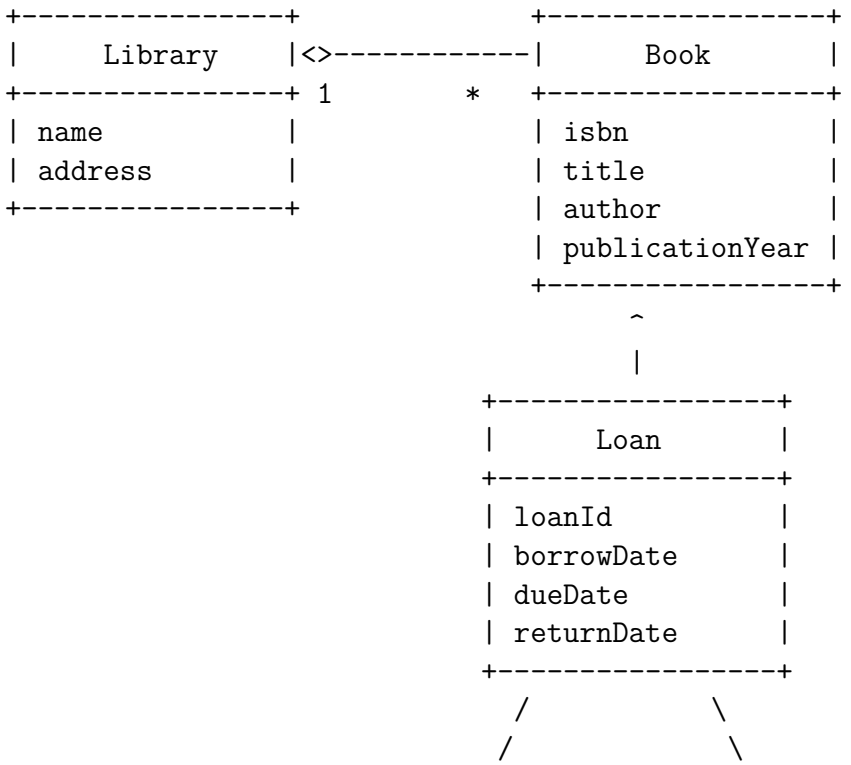
19.2.4 Defining Relationships Among Entities

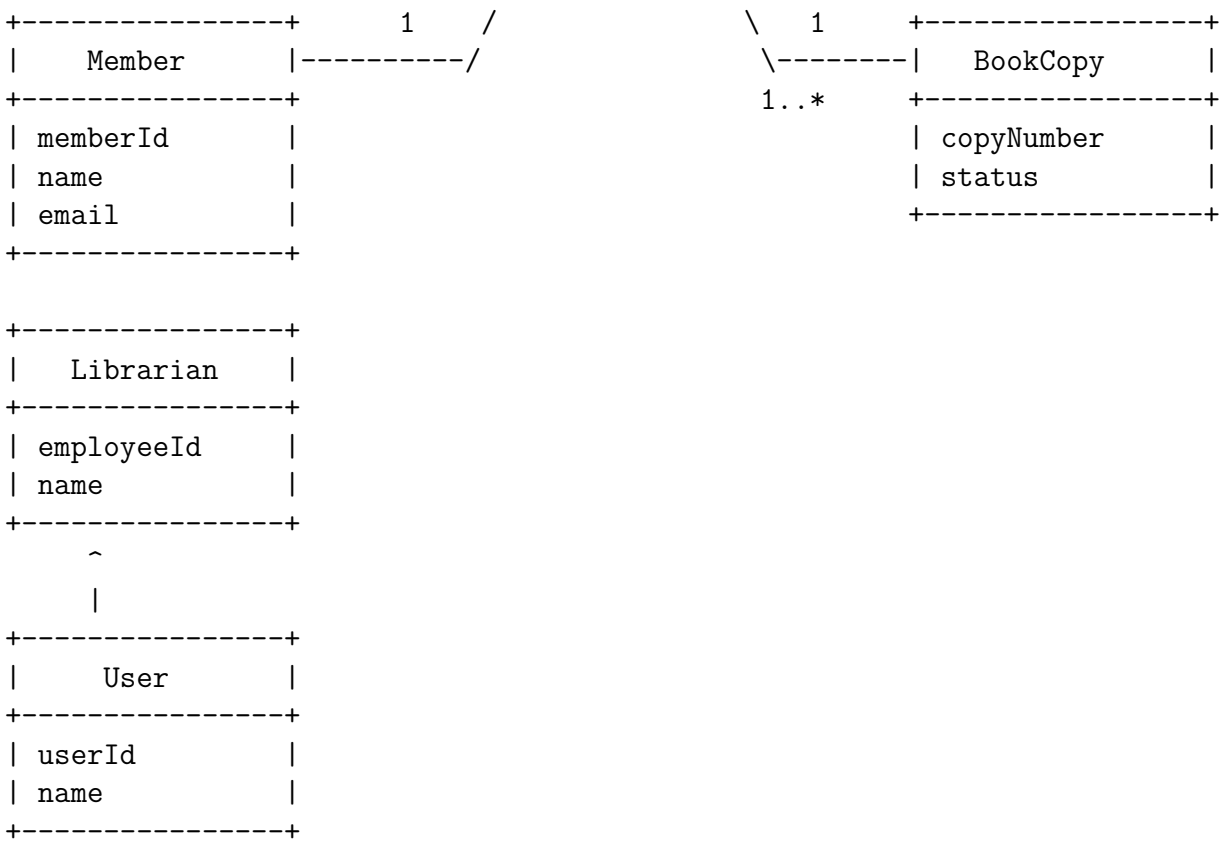
With entities identified, it's important to define **how they relate**. Key types of relationships include:

- **Association**: A basic link between classes indicating usage or connection. For example, a Member *has many* Loans; a Loan *is associated with* one Book.
- **Aggregation**: A "whole-part" relationship where the part can exist independently of the whole. For instance, a Library *aggregates* many Books, but a Book can exist outside a specific Library (in another system).
- **Composition**: A stronger "whole-part" relationship where the part's lifecycle depends on the whole. This might apply if, say, a Library Branch *composes* its physical Sections, which cannot exist separately.
- **Inheritance**: Modeling "is-a" relationships where subclasses extend base classes to share and override behavior. For example, Librarian and Member could both inherit from a common User superclass capturing shared attributes.

19.2.5 Example Domain Model Diagram

Below is a conceptual UML class diagram illustrating these entities and relationships:





19.2.6 Justification of Modeling Decisions

- **Library to Book (Aggregation)**: The Library contains many books, but each Book exists independently (conceptually) and can be shared or referenced elsewhere. Hence aggregation fits better than composition.
- **Member to Loan (Association)**: Members have zero or more Loans, tracking which books they currently borrow or have borrowed. Loans link to a specific BookCopy, reflecting that multiple copies of the same book can exist.
- **Book to BookCopy**: To handle multiple physical copies of the same title, we separate `Book` (conceptual book info) from `BookCopy` (individual physical copies with unique IDs and statuses like Available, Borrowed).
- **User, Member, and Librarian (Inheritance)**: Both members and librarians share common user attributes, so inheriting from a base `User` class reduces duplication and clarifies roles.

This structure supports scalability and future extension. For example, adding reservations or fines can build on top of this foundation.

19.2.7 Conclusion

Domain modeling is a vital step in object-oriented design that maps real-world concepts to software structures. By identifying key entities like Book, Member, Loan, and Librarian, defining their attributes, and establishing clear relationships through UML diagrams, developers create a blueprint that guides implementation. Thoughtful modeling decisions around aggregation, association, and inheritance improve system clarity, maintainability, and extensibility—paving the way for a robust library system.

19.3 Code Implementation

19.3.1 Translating Domain Model into Java Code

With the domain model established, the next step is to translate it into concrete Java classes that encapsulate data and behavior. This involves defining fields for attributes, constructors for object creation, and methods to represent key actions and enforce rules. Proper encapsulation ensures that internal state is protected and modified only through well-defined interfaces.

19.3.2 Implementing Key Entities

Below are core classes from the domain model with examples of fields, constructors, methods, and interactions.

The `Book` Class

```
public class Book {
    private final String isbn;
    private final String title;
    private final String author;
    private final int publicationYear;
    private int totalCopies;
    private int copiesAvailable;

    public Book(String isbn, String title, String author, int publicationYear, int totalCopies) {
        if (isbn == null || isbn.isEmpty()) {
            throw new IllegalArgumentException("ISBN must not be empty");
        }
        if (totalCopies < 0) {
            throw new IllegalArgumentException("Total copies cannot be negative");
        }
        this.isbn = isbn;
        this.title = title;
        this.author = author;
```

```
        this.publicationYear = publicationYear;
        this.totalCopies = totalCopies;
        this.copiesAvailable = totalCopies;
    }

    public String getIsbn() { return isbn; }
    public String getTitle() { return title; }
    public String getAuthor() { return author; }
    public int getPublicationYear() { return publicationYear; }

    public int getCopiesAvailable() { return copiesAvailable; }

    public boolean checkoutCopy() {
        if (copiesAvailable > 0) {
            copiesAvailable--;
            return true;
        }
        return false;
    }

    public void returnCopy() {
        if (copiesAvailable < totalCopies) {
            copiesAvailable++;
        } else {
            throw new IllegalStateException("All copies are already in library");
        }
    }

    @Override
    public String toString() {
        return String.format("%s by %s (%d) - Available: %d/%d",
            title, author, publicationYear, copiesAvailable, totalCopies);
    }
}
```

- **Validation** in constructor ensures `isbn` is not empty and copies are non-negative.
- `checkoutCopy()` decreases availability if possible.
- `returnCopy()` increases availability, with checks against over-returning.

The `Member` Class

```
import java.util.ArrayList;
import java.util.List;

public class Member {
    private final String memberId;
    private final String name;
    private final List<Loan> loans = new ArrayList<>();
    private final int maxBooksAllowed;

    public Member(String memberId, String name, int maxBooksAllowed) {
        if (memberId == null || memberId.isEmpty()) {
            throw new IllegalArgumentException("Member ID required");
        }
        this.memberId = memberId;
        this.name = name;
```

```
        this.maxBooksAllowed = maxBooksAllowed;
    }

    public String getMemberId() { return memberId; }
    public String getName() { return name; }
    public List<Loan> getLoans() { return new ArrayList<>(loans); } // Defensive copy

    public boolean canBorrow() {
        return loans.size() < maxBooksAllowed;
    }

    public void borrowBook(Book book) throws IllegalStateException {
        if (!canBorrow()) {
            throw new IllegalStateException("Max book limit reached");
        }
        if (book.checkoutCopy()) {
            Loan loan = new Loan(this, book);
            loans.add(loan);
            System.out.println(name + " borrowed " + book.getTitle());
        } else {
            throw new IllegalStateException("No copies available to borrow");
        }
    }

    public void returnBook(Loan loan) {
        if (loans.remove(loan)) {
            loan.returnBook();
            System.out.println(name + " returned " + loan.getBook().getTitle());
        } else {
            throw new IllegalArgumentException("Loan not found for member");
        }
    }
}
```

- Keeps track of loans and borrowing limits.
- Throws exceptions when attempting invalid operations, e.g., borrowing over limit.
- Demonstrates **encapsulation** by not exposing the internal loan list directly.

The Loan Class

```
import java.time.LocalDate;

public class Loan {
    private static final int LOAN_PERIOD_DAYS = 14;

    private final Member member;
    private final Book book;
    private final LocalDate borrowDate;
    private LocalDate returnDate;

    public Loan(Member member, Book book) {
        this.member = member;
        this.book = book;
        this.borrowDate = LocalDate.now();
        this.returnDate = null;
    }
```

```
    public Member getMember() { return member; }
    public Book getBook() { return book; }
    public LocalDate getBorrowDate() { return borrowDate; }
    public LocalDate getReturnDate() { return returnDate; }

    public boolean isReturned() {
        return returnDate != null;
    }

    public void returnBook() {
        if (isReturned()) {
            throw new IllegalStateException("Book already returned");
        }
        this.returnDate = LocalDate.now();
        book.returnCopy();
    }

    public boolean isOverdue() {
        if (isReturned()) {
            return false;
        }
        return LocalDate.now().isAfter(borrowDate.plusDays(LOAN_PERIOD_DAYS));
    }

    @Override
    public String toString() {
        return String.format("Loan of '%s' by %s on %s%s",
            book.getTitle(),
            member.getName(),
            borrowDate,
            isReturned() ? " (Returned)" : "");
    }
}
```

- Captures the loan lifecycle and status.
- Manages book return, throwing exceptions if double return attempted.
- Tracks overdue loans.

19.3.3 Encapsulation and Constructor Best Practices

Each class uses **private fields** and exposes accessors where needed, protecting internal state. Constructors validate inputs to prevent illegal states. For example, the `Book` class ensures that `isbn` is not empty, and copies are non-negative. Such defensive programming reduces runtime errors and enforces invariants.

19.3.4 Handling Exceptions and Input Validation

The above classes illustrate how exceptions help guard against invalid operations:

- Borrowing when no copies are available triggers an `IllegalStateException`.
- Returning a book that wasn't borrowed results in an `IllegalArgumentException`.
- Defensive checks in constructors prevent creation of invalid objects.

This explicit failure signaling aids debugging and promotes robust, predictable behavior.

19.3.5 Incremental Development and Testing

Building a library system benefits from an **incremental approach**:

1. **Implement core entities** (`Book`, `Member`, `Loan`) first.
2. Write **unit tests** verifying constructors, getters, and key methods like `borrowBook()`.
3. Extend functionality with supporting classes (`Librarian`, `Catalog`, `Reservation`).
4. Add exception handling and edge case tests.
5. Integrate components gradually to test interactions.

Writing automated tests at each step ensures the system behaves as expected and allows safe refactoring later.

19.3.6 Sample Usage Scenario

```
public class LibraryDemo {
    public static void main(String[] args) {
        Book book = new Book("978-0134685991", "Effective Java", "Joshua Bloch", 2018, 3);
        Member alice = new Member("M001", "Alice Johnson", 5);

        try {
            alice.borrowBook(book);
            alice.borrowBook(book); // Borrow second copy
            System.out.println(book);

            // Return one copy
            Loan loan = alice.getLoans().get(0);
            alice.returnBook(loan);
            System.out.println(book);

        } catch (IllegalStateException | IllegalArgumentException ex) {
            System.err.println("Operation failed: " + ex.getMessage());
        }
    }
}
```

Output:

```
Alice Johnson borrowed Effective Java
```

```
Alice Johnson borrowed Effective Java
Effective Java by Joshua Bloch (2018) - Available: 1/3
Alice Johnson returned Effective Java
Effective Java by Joshua Bloch (2018) - Available: 2/3
```

Full runnable code:

```java
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

// Book class
class Book {
    private final String isbn;
    private final String title;
    private final String author;
    private final int publicationYear;
    private int totalCopies;
    private int copiesAvailable;

    public Book(String isbn, String title, String author, int publicationYear, int totalCopies) {
        if (isbn == null || isbn.isEmpty()) {
            throw new IllegalArgumentException("ISBN must not be empty");
        }
        if (totalCopies < 0) {
            throw new IllegalArgumentException("Total copies cannot be negative");
        }
        this.isbn = isbn;
        this.title = title;
        this.author = author;
        this.publicationYear = publicationYear;
        this.totalCopies = totalCopies;
        this.copiesAvailable = totalCopies;
    }

    public String getIsbn() { return isbn; }
    public String getTitle() { return title; }
    public String getAuthor() { return author; }
    public int getPublicationYear() { return publicationYear; }
    public int getCopiesAvailable() { return copiesAvailable; }

    public boolean checkoutCopy() {
        if (copiesAvailable > 0) {
            copiesAvailable--;
            return true;
        }
        return false;
    }

    public void returnCopy() {
        if (copiesAvailable < totalCopies) {
            copiesAvailable++;
        } else {
            throw new IllegalStateException("All copies are already in library");
        }
    }
```

```
    @Override
    public String toString() {
        return String.format("%s by %s (%d) - Available: %d/%d",
            title, author, publicationYear, copiesAvailable, totalCopies);
    }
}

// Loan class
class Loan {
    private static final int LOAN_PERIOD_DAYS = 14;

    private final Member member;
    private final Book book;
    private final LocalDate borrowDate;
    private LocalDate returnDate;

    public Loan(Member member, Book book) {
        this.member = member;
        this.book = book;
        this.borrowDate = LocalDate.now();
        this.returnDate = null;
    }

    public Member getMember() { return member; }
    public Book getBook() { return book; }
    public LocalDate getBorrowDate() { return borrowDate; }
    public LocalDate getReturnDate() { return returnDate; }

    public boolean isReturned() {
        return returnDate != null;
    }

    public void returnBook() {
        if (isReturned()) {
            throw new IllegalStateException("Book already returned");
        }
        this.returnDate = LocalDate.now();
        book.returnCopy();
    }

    public boolean isOverdue() {
        if (isReturned()) {
            return false;
        }
        return LocalDate.now().isAfter(borrowDate.plusDays(LOAN_PERIOD_DAYS));
    }

    @Override
    public String toString() {
        return String.format("Loan of '%s' by %s on %s%s",
            book.getTitle(),
            member.getName(),
            borrowDate,
            isReturned() ? " (Returned)" : "");
    }
}

// Member class
```

```
class Member {
    private final String memberId;
    private final String name;
    private final List<Loan> loans = new ArrayList<>();
    private final int maxBooksAllowed;

    public Member(String memberId, String name, int maxBooksAllowed) {
        if (memberId == null || memberId.isEmpty()) {
            throw new IllegalArgumentException("Member ID required");
        }
        this.memberId = memberId;
        this.name = name;
        this.maxBooksAllowed = maxBooksAllowed;
    }

    public String getMemberId() { return memberId; }
    public String getName() { return name; }
    public List<Loan> getLoans() { return new ArrayList<>(loans); } // Defensive copy

    public boolean canBorrow() {
        return loans.size() < maxBooksAllowed;
    }

    public void borrowBook(Book book) {
        if (!canBorrow()) {
            throw new IllegalStateException("Max book limit reached");
        }
        if (book.checkoutCopy()) {
            Loan loan = new Loan(this, book);
            loans.add(loan);
            System.out.println(name + " borrowed " + book.getTitle());
        } else {
            throw new IllegalStateException("No copies available to borrow");
        }
    }

    public void returnBook(Loan loan) {
        if (loans.remove(loan)) {
            loan.returnBook();
            System.out.println(name + " returned " + loan.getBook().getTitle());
        } else {
            throw new IllegalArgumentException("Loan not found for member");
        }
    }
}

// Demo main class
public class LibraryDemo {
    public static void main(String[] args) {
        Book book = new Book("978-0134685991", "Effective Java", "Joshua Bloch", 2018, 3);
        Member alice = new Member("M001", "Alice Johnson", 5);

        try {
            alice.borrowBook(book);
            alice.borrowBook(book); // Borrow second copy
            System.out.println(book);

            // Return one copy
```

```
            Loan loan = alice.getLoans().get(0);
            alice.returnBook(loan);
            System.out.println(book);

        } catch (IllegalStateException | IllegalArgumentException ex) {
            System.err.println("Operation failed: " + ex.getMessage());
        }
    }
}
```

19.3.7 Summary and Best Practices

- **Translate domain entities** into classes with attributes and methods that encapsulate behavior.
- **Validate inputs** and protect invariants in constructors and setters.
- **Use exceptions** to handle illegal states or invalid user operations clearly.
- **Encapsulate internal data**, expose only necessary methods.
- Adopt an **incremental development cycle**, with unit tests to verify correctness and simplify debugging.
- Keep methods **focused and cohesive**, for readability and maintainability.

This approach ensures your library system remains flexible to future changes, easier to understand, and reliable in operation.

Chapter 20.

Case Study 2 Online Shopping Cart System

1. Managing Products and Inventory
2. User and Cart Design
3. Applying Patterns and Principles
4. Testing and Validation

20 Case Study 2 Online Shopping Cart System

20.1 Managing Products and Inventory

20.1.1 Core Components for Product and Inventory Management

In an online shopping cart system, managing products and inventory forms the backbone of the platform. These components ensure customers can browse available products, view accurate pricing, and purchase items that are in stock. At its core, product and inventory management encompasses:

- **Product Catalog:** The collection of all products available for sale.
- **Product Categories:** Grouping products logically for easier navigation and management.
- **Stock Levels:** Tracking quantities available to avoid overselling.
- **Price Management:** Maintaining current and promotional pricing accurately.

Modeling these elements thoughtfully in your design enables a robust, scalable system that supports day-to-day operations and growth.

20.1.2 Modeling Products, Categories, Stock, and Prices

Product

The `Product` class represents an individual item for sale. Key attributes typically include:

- `id` — unique identifier
- `name` — product name
- `description` — textual description
- `category` — reference to a `Category` object
- `price` — current price
- `stockQuantity` — number of units available

Additional fields might include SKU, weight, or supplier info depending on complexity.

Category

Categories organize products into logical groups such as Electronics, Clothing, or Books. The `Category` class might have:

- `id` — unique identifier
- `name` — category name
- `parentCategory` — for hierarchical categories (optional)
- A list of products belonging to the category (aggregation)

This structure supports browsing and filtering in the user interface.

Inventory Management

Inventory tracks the stock quantity of each product. This can be represented simply within the `Product` class or as a separate `InventoryItem` class if more detail is needed (e.g., location-based stock).

Stock levels must be carefully managed to avoid overselling, especially in concurrent purchase scenarios.

20.1.3 Java Class Examples

Below is a simplified example of the core classes modeling these concepts:

```
// Category.java
public class Category {
    private final String id;
    private final String name;
    private Category parentCategory;

    public Category(String id, String name) {
        this.id = id;
        this.name = name;
    }

    // Getters and setters omitted for brevity
}

// Product.java
public class Product {
    private final String id;
    private String name;
    private String description;
    private Category category;
    private double price;
    private int stockQuantity;

    public Product(String id, String name, String description, Category category, double price, int sto
        this.id = id;
        this.name = name;
        this.description = description;
        this.category = category;
        this.price = price;
        this.stockQuantity = stockQuantity;
    }

    public synchronized boolean purchase(int quantity) {
        if (quantity <= 0) {
            throw new IllegalArgumentException("Purchase quantity must be positive");
        }
        if (stockQuantity >= quantity) {
            stockQuantity -= quantity;
            return true;
        }
```

```
        return false; // Not enough stock
    }

    public synchronized void restock(int quantity) {
        if (quantity <= 0) {
            throw new IllegalArgumentException("Restock quantity must be positive");
        }
        stockQuantity += quantity;
    }

    // Getters and setters omitted for brevity
}
```

This design encapsulates stock management within the `Product` class. The `purchase` method ensures stock is decremented only if sufficient units exist, and `restock` allows replenishing inventory.

20.1.4 Inventory Updates: Purchases and Restocking

When customers place orders, the system must update stock levels accordingly:

- **Purchase:** The `purchase` method attempts to reduce `stockQuantity` by the requested amount. If insufficient stock exists, the operation fails gracefully.
- **Restocking:** The system adds new stock either manually (via admin interface) or automatically (from suppliers) using the `restock` method.

The synchronization in these methods (using `synchronized`) helps prevent race conditions when multiple purchase or restock operations happen concurrently in a multi-threaded environment, a common real-world challenge.

20.1.5 Real-World Considerations: Concurrency and Consistency

In practice, inventory management must account for:

- **Concurrency:** Multiple customers may attempt to buy the same product simultaneously. Proper locking or transactional mechanisms are required to maintain consistent stock levels and prevent overselling.
- **Distributed Systems:** In large-scale e-commerce, inventory data might be replicated across services or data centers. Ensuring eventual consistency or strong consistency can be a complex design decision.
- **Performance:** Locking strategies need balance to avoid bottlenecks under heavy load.
- **Integration:** Inventory often connects with external systems like suppliers or warehouse management software, adding complexity to synchronization.

These considerations often push designs toward robust transactional support, event-driven

updates, or use of message queues to maintain inventory accuracy and scalability.

20.1.6 Summary

Managing products and inventory in an online shopping cart system requires carefully modeled classes reflecting core domain concepts: products, categories, prices, and stock levels. Encapsulation of inventory updates within `Product` ensures atomic and safe stock modifications.

While the presented code offers a clear starting point, real-world applications must address concurrency and consistency challenges through additional architectural layers and infrastructure support.

With a strong domain model and understanding of operational nuances, the system will be well-positioned for reliability, extensibility, and smooth customer experience.

20.2 User and Cart Design

20.2.1 Modeling Users and Shopping Carts

In an online shopping cart system, managing users and their carts is essential for delivering a smooth purchasing experience. Users represent the customers interacting with the platform, while shopping carts hold the collection of products that users intend to purchase.

When designing these components, focus on clearly modeling attributes and behaviors for each, keeping in mind that users may be either **guest users** or **registered users**. The shopping cart must support common operations like adding/removing items, updating quantities, and calculating the total price.

20.2.2 User Model: Customers and Sessions

The `User` class encapsulates data and behaviors for customers. Important attributes include:

- `id`: Unique identifier (UUID or database ID)
- `name`: User's full name
- `email`: Contact email (mandatory for registered users)
- `isRegistered`: Flag indicating if the user is registered or a guest
- `cart`: The active shopping cart associated with the user session

The design should allow for extensibility to accommodate different user types. For example,

a `GuestUser` subclass might omit authentication details, while a `RegisteredUser` might include address and payment info.

20.2.3 Shopping Cart Model

The `ShoppingCart` class maintains a collection of `CartItem` instances, each representing a product and quantity selected by the user. Key responsibilities include:

- Adding products with quantities
- Removing products
- Updating quantities
- Calculating total price of items
- Clearing the cart

This design promotes encapsulation by centralizing cart logic and interactions.

20.2.4 Java Code Examples

Below is a runnable example illustrating the user and cart design, including basic cart operations.

```
import java.util.*;

// User.java
public class User {
    private final String id;
    private final String name;
    private final String email;
    private final boolean isRegistered;
    private ShoppingCart cart;

    public User(String id, String name, String email, boolean isRegistered) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.isRegistered = isRegistered;
        this.cart = new ShoppingCart();
    }

    public ShoppingCart getCart() {
        return cart;
    }

    // Additional getters and user-specific methods here
}

// CartItem.java
class CartItem {
```

```
    private final Product product;
    private int quantity;

    public CartItem(Product product, int quantity) {
        this.product = product;
        this.quantity = quantity;
    }

    public Product getProduct() { return product; }
    public int getQuantity() { return quantity; }
    public void setQuantity(int quantity) { this.quantity = quantity; }

    public double getTotalPrice() {
        return product.getPrice() * quantity;
    }
}

// ShoppingCart.java
class ShoppingCart {
    private final Map<String, CartItem> items = new HashMap<>();

    public void addProduct(Product product, int quantity) {
        if (quantity <= 0) throw new IllegalArgumentException("Quantity must be positive");
        CartItem item = items.get(product.getId());
        if (item == null) {
            items.put(product.getId(), new CartItem(product, quantity));
        } else {
            item.setQuantity(item.getQuantity() + quantity);
        }
    }

    public void removeProduct(String productId) {
        items.remove(productId);
    }

    public void updateProductQuantity(String productId, int quantity) {
        if (quantity <= 0) {
            removeProduct(productId);
        } else {
            CartItem item = items.get(productId);
            if (item != null) {
                item.setQuantity(quantity);
            }
        }
    }

    public double calculateTotal() {
        return items.values().stream()
            .mapToDouble(CartItem::getTotalPrice)
            .sum();
    }

    public void clear() {
        items.clear();
    }

    public void printCartContents() {
        if (items.isEmpty()) {
```

```
            System.out.println("Shopping cart is empty.");
            return;
        }
        System.out.println("Shopping Cart Contents:");
        for (CartItem item : items.values()) {
            System.out.printf("- %s (x%d): $%.2f%n",
                item.getProduct().getName(),
                item.getQuantity(),
                item.getTotalPrice());
        }
        System.out.printf("Total: $%.2f%n", calculateTotal());
    }
}

// Product.java (simplified for demo)
class Product {
    private final String id;
    private final String name;
    private final double price;

    public Product(String id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public String getId() { return id; }
    public String getName() { return name; }
    public double getPrice() { return price; }
}

// Demo usage
public class ShoppingCartDemo {
    public static void main(String[] args) {
        Product p1 = new Product("P001", "Wireless Mouse", 25.99);
        Product p2 = new Product("P002", "Mechanical Keyboard", 79.99);

        User user = new User("U1001", "Alice Smith", "alice@example.com", true);

        ShoppingCart cart = user.getCart();
        cart.addProduct(p1, 2);
        cart.addProduct(p2, 1);
        cart.printCartContents();

        cart.updateProductQuantity("P001", 1);
        cart.printCartContents();

        cart.removeProduct("P002");
        cart.printCartContents();
    }
}
```

This example illustrates key behaviors: adding products to the cart, updating quantities, removing items, and calculating totals. The `User` holds a reference to their `ShoppingCart`, modeling the user-session relationship.

20.2.5 Design for Extensibility: Guest vs Registered Users

In a real system, users fall into different categories with varying privileges and data needs:

- **Guest Users:** Usually have temporary carts and limited personalization.
- **Registered Users:** Can save carts, track orders, and manage payment options.

To support this, consider subclassing `User`:

```
public class GuestUser extends User {
    public GuestUser() {
        super(UUID.randomUUID().toString(), "Guest", null, false);
    }
}

public class RegisteredUser extends User {
    private String address;
    // Additional attributes

    public RegisteredUser(String id, String name, String email, String address) {
        super(id, name, email, true);
        this.address = address;
    }

    // Getters and setters for extended info
}
```

This design enables differentiated behavior while reusing common user-cart functionality.

20.2.6 Managing User Sessions and Cart Persistence

In web applications, user sessions manage state between requests. The shopping cart often persists in the user session or a database. Design considerations include:

- **Session Management:** Attach a cart to a user session object, restoring it upon login or continuing as a guest.
- **Persistence:** Store cart contents in a database or cache to survive server restarts or multi-device access.
- **Synchronization:** For logged-in users, synchronize cart data across devices.

Though these involve infrastructure beyond plain OOP, designing clear cart and user abstractions helps integrate with session and persistence mechanisms easily.

20.2.7 Summary

Modeling users and shopping carts requires careful attention to data encapsulation and behaviors such as adding/removing products and calculating totals. By designing flexible

user hierarchies, the system can cleanly handle guest and registered users alike.

The sample code demonstrates core operations in a straightforward, maintainable way. Future extensions might include coupon handling, inventory checks at add-to-cart time, or cart expiration policies.

Effective design here enhances user experience and supports evolving business requirements.

20.3 Applying Patterns and Principles

20.3.1 Applying SOLID Principles and Design Patterns

In designing an online shopping cart system, adhering to established object-oriented principles and leveraging design patterns is key to building flexible, maintainable, and extensible software. Throughout the system, SOLID principles guide the structure, while classic design patterns solve common challenges elegantly.

20.3.2 Where SOLID Principles Apply

- **Single Responsibility Principle (SRP):** Each class should have a focused responsibility. For example, the `ShoppingCart` manages cart operations, while `PaymentProcessor` handles payment logic.
- **Open/Closed Principle (OCP):** The system should be open for extension but closed for modification. When supporting multiple payment methods, rather than modifying core code, new payment types extend a `PaymentStrategy` interface.
- **Liskov Substitution Principle (LSP):** Subtypes like `RegisteredUser` and `GuestUser` should be substitutable without breaking functionality.
- **Interface Segregation Principle (ISP):** Interfaces such as `Notifier` (for notifications) are kept minimal, so implementing classes aren't forced to define unnecessary methods.
- **Dependency Inversion Principle (DIP):** High-level modules depend on abstractions (interfaces), not concrete implementations, enabling easy swapping of components like notification or payment processors.

20.3.3 Strategy Pattern: Flexible Payment Processing

One classic example is the **Strategy** pattern to encapsulate payment algorithms. The system defines a `PaymentStrategy` interface:

```java
public interface PaymentStrategy {
    void pay(double amount);
}
```

Concrete implementations for different payment methods—credit card, PayPal, or gift card—implement this interface:

```java
public class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        // process credit card payment
        System.out.println("Paid $" + amount + " with credit card.");
    }
}

public class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) {
        // process PayPal payment
        System.out.println("Paid $" + amount + " via PayPal.");
    }
}
```

The shopping cart or checkout class can then accept any `PaymentStrategy` instance, allowing seamless addition of new payment methods without modifying existing code:

```java
public class CheckoutService {
    private PaymentStrategy paymentStrategy;

    public CheckoutService(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(double amount) {
        paymentStrategy.pay(amount);
    }
}
```

Full runnable code:

```java
// PaymentStrategy interface
public interface PaymentStrategy {
    void pay(double amount);
}

// Concrete strategy: Credit Card payment
public class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " with credit card.");
    }
}

// Concrete strategy: PayPal payment
public class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " via PayPal.");
    }
```

```
}

// Checkout service using a PaymentStrategy
public class CheckoutService {
    private PaymentStrategy paymentStrategy;

    public CheckoutService(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(double amount) {
        paymentStrategy.pay(amount);
    }
}

// Demo usage
public class StrategyDemo {
    public static void main(String[] args) {
        CheckoutService checkout1 = new CheckoutService(new CreditCardPayment());
        checkout1.checkout(100.00);

        CheckoutService checkout2 = new CheckoutService(new PayPalPayment());
        checkout2.checkout(55.50);
    }
}
```

Benefits:

- The system is open for new payment types without changing existing classes (OCP).
- The payment logic is cleanly separated, supporting SRP.
- Swapping payment methods at runtime is simple.

20.3.4 Observer Pattern: Event-Driven Notifications

The **Observer** pattern supports sending notifications—such as order confirmation emails or SMS alerts—when certain events occur (e.g., successful payment or item shipment).

Define an abstract `Notifier` interface:

```
public interface Notifier {
    void update(String message);
}
```

Concrete observers implement this interface:

```
public class EmailNotifier implements Notifier {
    public void update(String message) {
        System.out.println("Sending email: " + message);
    }
}
```

```java
public class SMSNotifier implements Notifier {
    public void update(String message) {
        System.out.println("Sending SMS: " + message);
    }
}
```

The subject class (e.g., `Order`) maintains a list of observers and notifies them on events:

```java
import java.util.*;

public class Order {
    private List<Notifier> notifiers = new ArrayList<>();

    public void addNotifier(Notifier notifier) {
        notifiers.add(notifier);
    }

    public void completeOrder() {
        // Order completion logic
        notifyAllObservers("Order completed successfully.");
    }

    private void notifyAllObservers(String message) {
        for (Notifier notifier : notifiers) {
            notifier.update(message);
        }
    }
}
```

Full runnable code:

```java
import java.util.*;

// Observer interface
public interface Notifier {
    void update(String message);
}

// Concrete observer: Email notification
public class EmailNotifier implements Notifier {
    public void update(String message) {
        System.out.println("Sending email: " + message);
    }
}

// Concrete observer: SMS notification
public class SMSNotifier implements Notifier {
    public void update(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

// Subject class
public class Order {
    private List<Notifier> notifiers = new ArrayList<>();

    public void addNotifier(Notifier notifier) {
```

```
        notifiers.add(notifier);
    }

    public void completeOrder() {
        // Order completion logic
        notifyAllObservers("Order completed successfully.");
    }

    private void notifyAllObservers(String message) {
        for (Notifier notifier : notifiers) {
            notifier.update(message);
        }
    }
}

// Demo usage
public class ObserverDemo {
    public static void main(String[] args) {
        Order order = new Order();
        order.addNotifier(new EmailNotifier());
        order.addNotifier(new SMSNotifier());

        order.completeOrder();
    }
}
```

Benefits:

- New notification channels can be added without modifying the `Order` class.
- Decouples order logic from notification mechanisms, improving maintainability.

20.3.5 Factory Pattern: Simplifying Object Creation

When creating products or users, the **Factory** pattern can encapsulate complex instantiation logic:

```
public class UserFactory {
    public static User createUser(String type, String name, String email) {
        if ("guest".equalsIgnoreCase(type)) {
            return new GuestUser();
        } else if ("registered".equalsIgnoreCase(type)) {
            return new RegisteredUser(name, email);
        }
        throw new IllegalArgumentException("Unknown user type");
    }
}
```

This approach centralizes creation logic, reducing duplication and adhering to SRP.

20.3.6 Decorator Pattern: Enhancing Cart Features Dynamically

The **Decorator** pattern allows dynamically adding responsibilities to objects. For example, you might add a feature to apply discounts to a cart without modifying the original `ShoppingCart` class:

```
public interface Cart {
    double calculateTotal();
}

public class BasicCart implements Cart {
    // existing cart implementation
}

public class DiscountedCart implements Cart {
    private Cart wrappedCart;
    private double discountRate;

    public DiscountedCart(Cart cart, double discountRate) {
        this.wrappedCart = cart;
        this.discountRate = discountRate;
    }

    @Override
    public double calculateTotal() {
        double baseTotal = wrappedCart.calculateTotal();
        return baseTotal * (1 - discountRate);
    }
}
```

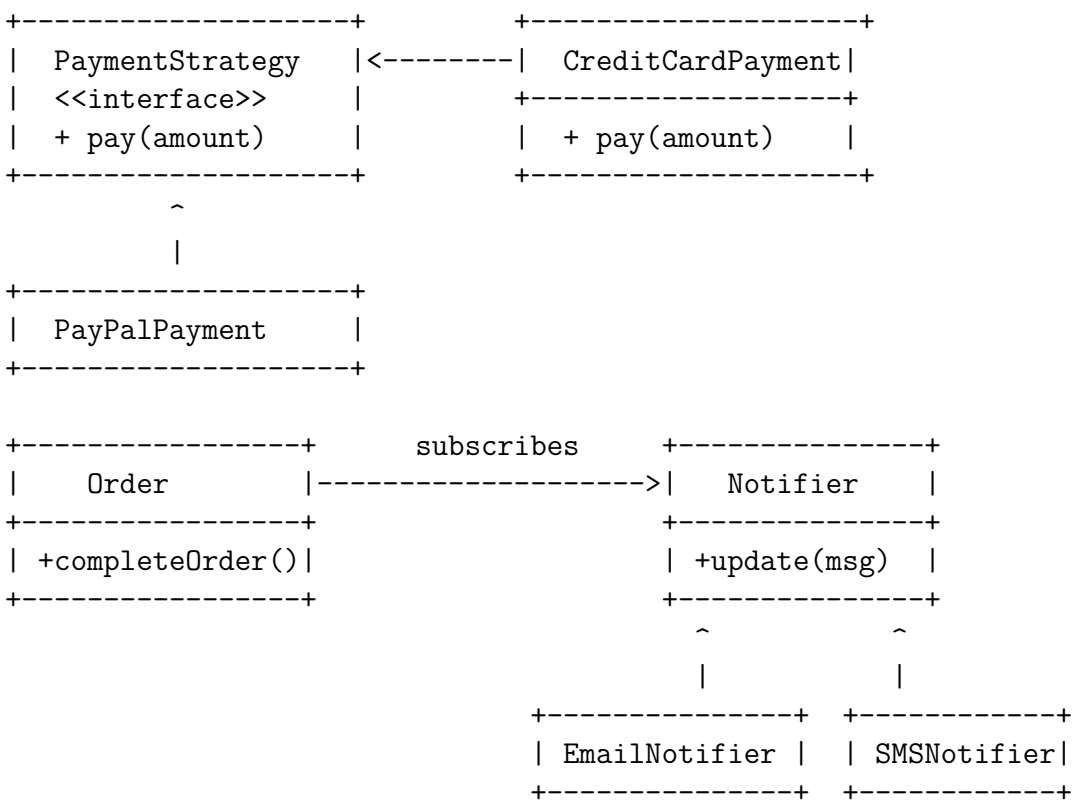
Benefits:

- New behaviors can be layered transparently.
- Avoids monolithic classes and keeps concerns separated.

20.3.7 How These Patterns Improve the System

- **Flexibility:** New features or behaviors can be added with minimal code changes, often just new classes.
- **Maintainability:** Clear separation of concerns makes it easier to locate and fix bugs.
- **Extensibility:** The system can evolve gracefully as requirements change.
- **Testability:** Isolated components and use of interfaces enable easier unit testing and mocking.

20.3.8 Simplified UML Diagram



20.3.9 Summary

By applying SOLID principles and design patterns like Strategy, Observer, Factory, and Decorator, the shopping cart system achieves modularity and adaptability. These patterns allow for clean separation of concerns, ease of extension, and clearer communication among developers — all essential qualities for robust software development.

20.4 Testing and Validation

Testing is a critical activity in software development, ensuring that a system functions correctly, reliably, and as intended. For an online shopping cart system, thorough testing is essential because it directly impacts user experience, business operations, and trust.

20.4.1 Importance of Testing

The complexity of a shopping cart system — involving user sessions, inventory updates, payment processing, and more — makes it vulnerable to errors. Testing reduces the risk of bugs, prevents regressions during changes, and guarantees that the system meets its functional and non-functional requirements. Automated tests accelerate development by providing quick feedback, enabling safer refactoring, and improving code quality.

20.4.2 Unit Testing Core Components

Unit tests focus on small, isolated pieces of code, such as classes or methods, verifying their behavior independently. In the shopping cart system, key components to unit test include:

- **Shopping Cart operations:** Adding/removing items, calculating totals, applying discounts.
- **Inventory management:** Stock updates, out-of-stock handling.
- **User management:** Validating user registration, session handling, permissions.

Unit testing frameworks like **JUnit** are widely used in Java to write and run such tests.

20.4.3 Example JUnit Test Case

Consider testing the `ShoppingCart` class's ability to add products and calculate the total price:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class ShoppingCartTest {

    private ShoppingCart cart;
    private Product apple;
    private Product orange;

    @BeforeEach
    public void setup() {
        cart = new ShoppingCart();
        apple = new Product("Apple", 1.0);
        orange = new Product("Orange", 1.5);
    }

    @Test
    public void testAddItemAndCalculateTotal() {
        cart.addItem(apple, 3); // 3 apples
        cart.addItem(orange, 2); // 2 oranges
```

```
        double expectedTotal = 3 * 1.0 + 2 * 1.5;
        assertEquals(expectedTotal, cart.calculateTotal());
    }

    @Test
    public void testRemoveItem() {
        cart.addItem(apple, 2);
        cart.removeItem(apple);
        assertEquals(0, cart.calculateTotal());
    }
}
```

This example uses the JUnit 5 framework, demonstrating setup, test annotations, and assertions to verify behavior.

20.4.4 Validation Strategies

Validating inputs and business rules is equally important. Validation helps catch errors early and enforce correct usage. Some common validation tasks in the shopping cart system include:

- **Input correctness:** Ensuring quantities are positive integers, product IDs exist, and user details meet format requirements.
- **Business rules:** Preventing checkout when the cart is empty, enforcing stock availability, and validating payment information.

Validation can be implemented using explicit checks in methods or using validation frameworks like **Hibernate Validator** (JSR-380).

Example validation snippet:

```
public void addItem(Product product, int quantity) {
    if (quantity <= 0) {
        throw new IllegalArgumentException("Quantity must be positive.");
    }
    if (!inventory.isInStock(product, quantity)) {
        throw new IllegalStateException("Insufficient stock.");
    }
    // add item to cart
}
```

20.4.5 Best Practices and Tools

- **Write tests early and often:** Follow Test-Driven Development (TDD) to write tests before implementation.
- **Keep tests isolated:** Avoid dependencies between tests to ensure reliability.

- **Use mocking frameworks:** Tools like **Mockito** help isolate unit tests by mocking dependencies (e.g., database or external services).
- **Automate testing:** Integrate tests into continuous integration pipelines to catch issues quickly.
- **Code coverage:** Measure what percentage of your code is tested and aim for high coverage, but prioritize meaningful tests over coverage metrics alone.

Popular tools in Java ecosystem include:

- **JUnit**: The de facto unit testing framework.
- **Mockito**: For mocking dependencies.
- **AssertJ**: Fluent assertions for readable tests.
- **JaCoCo**: Code coverage analysis.

20.4.6 Summary

Testing and validation form the backbone of a reliable online shopping cart system. By writing comprehensive unit tests, validating inputs rigorously, and adopting best practices, developers ensure the system behaves correctly and is robust against changes. This not only improves user trust but also eases ongoing maintenance and feature enhancement.

Chapter 21.

Case Study 3 Task Manager App with JavaFX

1. MVC in Object-Oriented GUI Design
2. Layered Architecture
3. Integrating User Interface and Business Logic

21 Case Study 3 Task Manager App with JavaFX

21.1 MVC in Object-Oriented GUI Design

Designing graphical user interfaces (GUIs) can quickly become complex as applications grow in functionality. To manage this complexity and produce maintainable, testable, and scalable applications, software architects often rely on design patterns. One of the most influential and widely adopted patterns for GUI design is **Model-View-Controller (MVC)**.

21.1.1 Understanding the MVC Pattern

MVC is a **separation of concerns** architectural pattern that divides an application into three interconnected components:

- **Model**: Represents the core data and business logic. It manages the state of the application and notifies other components when data changes.
- **View**: Handles all UI-related elements and displays data from the Model. It also captures user inputs and forwards them for processing.
- **Controller**: Acts as an intermediary between the Model and the View. It handles user inputs, updates the Model, and directs the View to refresh.

This separation isolates concerns, allowing developers to work on UI design, business logic, and input handling independently. It also supports parallel development, easier testing, and more flexible maintenance.

21.1.2 MVC in the Task Manager App Context

Consider building a **Task Manager app** using JavaFX—a popular Java framework for rich client applications. The app allows users to create, edit, delete, and mark tasks as complete. Mapping this functionality into the MVC components helps keep the design clean and manageable.

Model: Task Data and Business Logic

The **Model** contains the classes representing tasks and their states. For example, a `Task` class encapsulates properties like:

- `title` (String)
- `description` (String)
- `dueDate` (LocalDate)
- `completed` (boolean)

The Model manages how tasks are stored, validated, and modified. It might also include business rules, such as preventing a task's due date from being set in the past.

The Model is independent of the UI. It can notify observers about state changes (e.g., using JavaFX's `ObservableList` or property bindings), allowing the View to update automatically.

View: User Interface Components

The **View** is responsible for displaying the task list, task details, and controls (buttons, text fields). In JavaFX, this typically involves FXML files or programmatically constructed UI nodes:

- A `ListView<Task>` to display the list of tasks.
- Text fields for entering or editing task titles and descriptions.
- Date picker controls for setting due dates.
- Checkboxes for marking tasks complete.

The View binds to the Model's data using JavaFX's property bindings, providing a responsive interface that reflects the current state.

Controller: Event Handling and Interaction Logic

The **Controller** processes user interactions such as button clicks or list selections. It reads input from the View, validates it if necessary, updates the Model, and triggers UI updates.

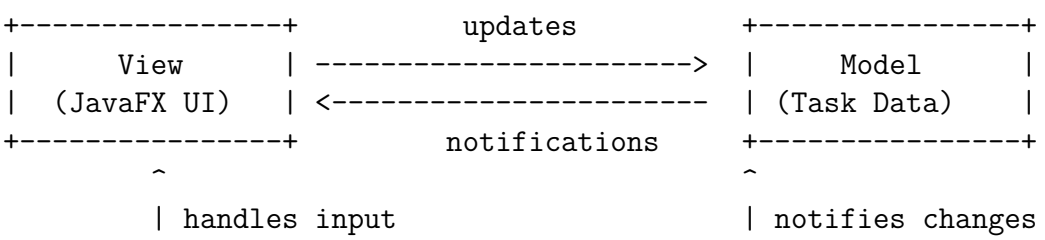
For example:

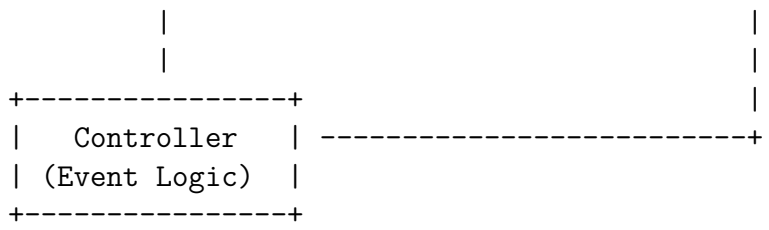
- When a user clicks "Add Task," the Controller reads the entered task data, creates a new `Task` instance, and adds it to the Model.
- When a task is marked complete, the Controller updates the Model's corresponding `Task` object.
- The Controller may also handle navigation between UI screens or dialogs.

The Controller acts as a bridge that interprets user actions and applies them to the application state.

21.1.3 MVC Structure Diagram

To visualize the flow and relationships, consider this simplified UML-like diagram illustrating MVC for the Task Manager app:





- The View sends user inputs to the Controller.
- The Controller updates the Model.
- The Model notifies the View of state changes.
- The View refreshes to reflect the current state.

21.1.4 JavaFX Example Demonstrating MVC Separation

Below is a simplified code snippet illustrating MVC separation in JavaFX for adding a task:

Model (Task.java)

```
public class Task {
    private final StringProperty title = new SimpleStringProperty();
    private final BooleanProperty completed = new SimpleBooleanProperty(false);

    public Task(String title) {
        this.title.set(title);
    }

    public String getTitle() { return title.get(); }
    public void setTitle(String value) { title.set(value); }
    public StringProperty titleProperty() { return title; }

    public boolean isCompleted() { return completed.get(); }
    public void setCompleted(boolean value) { completed.set(value); }
    public BooleanProperty completedProperty() { return completed; }
}
```

View + Controller (TaskManagerController.java)

```
public class TaskManagerController {

    @FXML private TextField taskInputField;
    @FXML private ListView<Task> taskListView;
    private ObservableList<Task> tasks = FXCollections.observableArrayList();

    @FXML
    public void initialize() {
        taskListView.setItems(tasks);
        taskListView.setCellFactory(list -> new ListCell<>() {
            @Override
            protected void updateItem(Task task, boolean empty) {
                super.updateItem(task, empty);
```

```
                setText(empty || task == null ? "" : task.getTitle());
            }
        });
    }

    @FXML
    public void handleAddTask() {
        String title = taskInputField.getText();
        if (title != null && !title.trim().isEmpty()) {
            Task newTask = new Task(title.trim());
            tasks.add(newTask);
            taskInputField.clear();
        }
    }
}
```

Here, the Controller handles input and updates the Model (`tasks` list), while the View reflects those changes automatically. The use of property bindings and observable lists helps keep the UI in sync without manual refresh logic.

21.1.5 Benefits of MVC in JavaFX GUI Design

- **Separation of concerns**: Each component has a clear role, reducing complexity.
- **Reusability**: Models can be reused across different Views or UI technologies.
- **Testability**: Business logic in the Model can be unit tested independently of the UI.
- **Maintainability**: Changes in UI or business logic tend to have localized impacts.
- **Scalability**: MVC supports incremental development as new features or UI changes arise.

21.1.6 Summary

The Model-View-Controller pattern is a cornerstone for building robust, maintainable JavaFX applications such as the Task Manager app. By clearly separating data, UI, and interaction logic, MVC facilitates clean code organization, easier testing, and scalable design. As you continue developing, remember that maintaining this separation helps keep your codebase flexible and responsive to change.

21.2 Layered Architecture

21.2.1 Layered Architecture

In software design, as applications grow in complexity, organizing code into distinct layers becomes critical for maintainability, modularity, and scalability. **Layered architecture** is a widely adopted approach that separates an application into logical layers, each with clearly defined responsibilities. This separation enables independent development, testing, and modification of each layer while reducing tight coupling between components.

21.2.2 Defining Layered Architecture

Layered architecture divides software into stacked layers, where each layer communicates primarily with the layer directly below or above it. The most common layers include:

- **Presentation Layer (UI)**: Handles user interactions and displays data.
- **Business Logic Layer (Service Layer)**: Contains the core domain logic, business rules, and workflows.
- **Data Access Layer (Persistence Layer)**: Manages data storage and retrieval, interacting with databases or external services.

By organizing code this way, each layer is responsible for one aspect of the application, adhering to the Single Responsibility Principle. Changes in one layer—such as updating the database schema—minimally impact other layers, making the system more resilient to change.

21.2.3 Layered Architecture in the Task Manager App

Let's contextualize layered architecture within our Task Manager app built using JavaFX. The app supports creating, updating, and viewing tasks, so structuring it into layers clarifies responsibilities and fosters maintainability.

Presentation Layer (UI)

This layer is where the JavaFX user interface components live—buttons, forms, lists, and other controls that interact directly with the user. The UI is responsible for rendering task data and capturing user input but contains minimal business logic. It delegates commands to the Business Logic Layer and reflects changes made there.

Example components:

- JavaFX controllers (`TaskManagerController`)
- FXML views defining UI layout

- UI event handlers wired to Controller methods

Business Logic Layer

The business logic layer processes user requests, applies business rules, and coordinates application workflows. It manipulates task data, validates inputs, enforces constraints (e.g., no overdue tasks without a due date), and provides services for managing tasks.

This layer acts as a bridge between the UI and data layers, encapsulating the core behavior of the application. It does not handle UI rendering or data storage directly.

Example:

- `TaskService` class with methods such as `addTask()`, `updateTask()`, `completeTask()`
- Validation logic for task attributes
- Notifications or events to update the UI on state changes

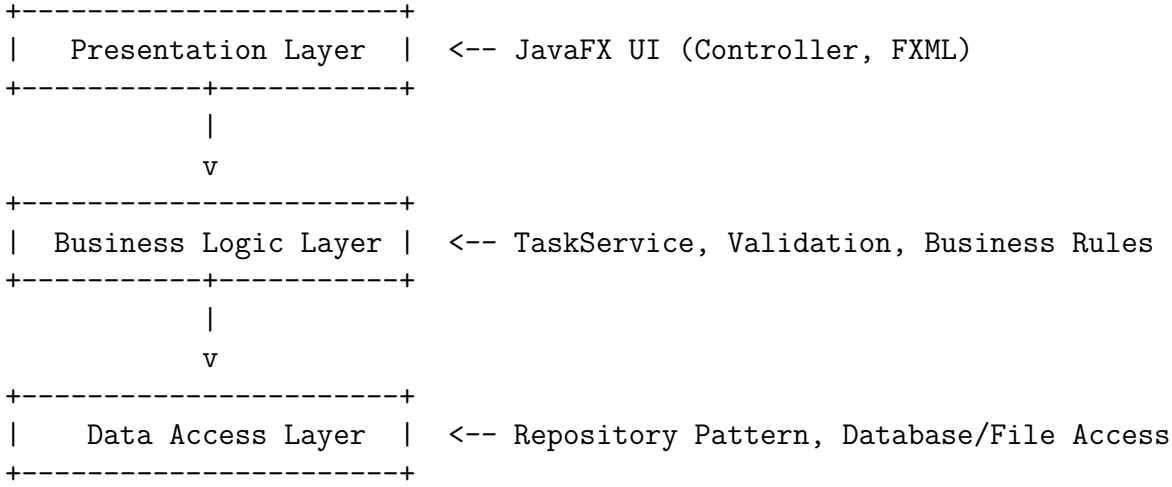
Data Access Layer

This layer interacts with the data store, whether it's a database, file system, or in-memory collection. It handles saving, retrieving, updating, and deleting task data. It abstracts data persistence details so that upper layers remain independent of the storage mechanism.

Example:

- `TaskRepository` interface with methods like `save(Task task)`, `findById()`, `delete(Task task)`
- Implementations for storage: `InMemoryTaskRepository`, `DatabaseTaskRepository`

21.2.4 Example Diagram of Layered Architecture



The UI calls services in the Business Logic Layer, which in turn calls repository methods in

the Data Access Layer. Data flows back up through the layers to the UI for display.

21.2.5 Example Code Snippet Showing Layer Interaction

Business Logic Layer (TaskService.java):

```
public class TaskService {
    private final TaskRepository repository;

    public TaskService(TaskRepository repository) {
        this.repository = repository;
    }

    public void addTask(Task task) {
        if (task.getTitle() == null || task.getTitle().isEmpty()) {
            throw new IllegalArgumentException("Task title cannot be empty");
        }
        repository.save(task);
    }

    public List<Task> getAllTasks() {
        return repository.findAll();
    }

    // Other business methods...
}
```

Data Access Layer (InMemoryTaskRepository.java):

```
public class InMemoryTaskRepository implements TaskRepository {
    private final Map<String, Task> taskStorage = new HashMap<>();

    @Override
    public void save(Task task) {
        taskStorage.put(task.getId(), task);
    }

    @Override
    public List<Task> findAll() {
        return new ArrayList<>(taskStorage.values());
    }

    // Other CRUD methods...
}
```

Presentation Layer (TaskManagerController.java):

```
public class TaskManagerController {
    private final TaskService taskService = new TaskService(new InMemoryTaskRepository());

    @FXML
    private TextField taskTitleInput;
```

```
    @FXML
    private ListView<Task> taskListView;

    @FXML
    public void handleAddTask() {
        String title = taskTitleInput.getText();
        try {
            Task newTask = new Task(title);
            taskService.addTask(newTask);
            taskListView.getItems().setAll(taskService.getAllTasks());
            taskTitleInput.clear();
        } catch (IllegalArgumentException e) {
            // Show error to user
            System.err.println(e.getMessage());
        }
    }
}
```

21.2.6 Benefits of Layered Architecture

- **Modularity**: Each layer encapsulates specific responsibilities, making the system easier to understand.
- **Maintainability**: Changes in one layer have limited impact on others. For example, switching from in-memory storage to a database requires only modifying the Data Access Layer.
- **Testability**: Layers can be tested independently. The business logic can be tested without the UI, using mock repositories.
- **Reusability**: Business logic and data access can be reused with different UI technologies (e.g., console app, web front-end).
- **Scalability**: Layering supports incremental development and the addition of new features without large rewrites.

21.2.7 Supporting Testing and Future Enhancements

With clear layering, you can write unit tests for the business logic using mock implementations of the repository, ensuring your core logic behaves correctly without needing the UI or database. Similarly, UI tests focus only on presentation concerns.

Looking forward, this architecture supports enhancements like:

- Adding authentication in a separate Security Layer.
- Introducing caching or transactional layers between business logic and data access.
- Replacing the data access implementation with a remote web service without impacting UI or business rules.

21.2.8 Summary

Layered architecture is a foundational approach to organizing the Task Manager app into distinct modules, each focusing on a single responsibility. This organization enhances modularity, maintainability, and testability, making it easier to develop, extend, and manage the application over time. By separating the UI, business logic, and data access, developers gain flexibility and resilience, key factors for successful, real-world software systems.

21.3 Integrating User Interface and Business Logic

21.3.1 Integrating User Interface and Business Logic

A key challenge in designing rich desktop applications like a Task Manager is the seamless integration between the **user interface (UI)** and the **business logic**. In JavaFX applications, the UI layer handles user interactions and visual presentation, while the business logic layer manages the core operations such as task creation, deletion, and updates. Clean integration between these layers ensures a responsive, maintainable, and scalable app.

This section will illustrate how JavaFX UI components interact with the underlying business logic classes, show runnable examples linking UI events to task operations, and discuss best practices and concurrency considerations.

21.3.2 Linking UI Events to Business Logic

JavaFX provides event-driven programming through UI controls like buttons, lists, and text fields. These components fire events—such as button clicks or list selections—that your controller class listens to and responds by invoking business logic methods.

Here's a typical flow for integrating UI with business logic:

- User performs an action (e.g., clicks "Add Task").
- The JavaFX controller captures the event.
- Controller calls business logic service methods to update the task data.
- Business logic modifies underlying data models.
- Controller updates UI components to reflect changes.

21.3.3 Example: Adding and Removing Tasks

Suppose we have a simple `TaskService` that manages tasks, and a JavaFX controller class `TaskManagerController` managing the UI.

Business Logic Class (TaskService.java):

```
import java.util.ArrayList;
import java.util.List;

public class TaskService {
    private final List<Task> tasks = new ArrayList<>();

    public void addTask(Task task) {
        tasks.add(task);
    }

    public void removeTask(Task task) {
        tasks.remove(task);
    }

    public List<Task> getAllTasks() {
        return new ArrayList<>(tasks);
    }
}
```

Model Class (Task.java):

```
public class Task {
    private String title;

    public Task(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    @Override
    public String toString() {
        return title;
    }
}
```

JavaFX Controller (TaskManagerController.java):

```
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;

public class TaskManagerController {
    @FXML
    private TextField taskInput;

    @FXML
    private Button addButton;
```

```
    @FXML
    private Button deleteButton;

    @FXML
    private ListView<Task> taskListView;

    private final TaskService taskService = new TaskService();
    private final ObservableList<Task> taskObservableList = FXCollections.observableArrayList();

    @FXML
    public void initialize() {
        taskListView.setItems(taskObservableList);

        addButton.setOnAction(event -> {
            String title = taskInput.getText().trim();
            if (!title.isEmpty()) {
                Task newTask = new Task(title);
                taskService.addTask(newTask);
                refreshTaskList();
                taskInput.clear();
            }
        });

        deleteButton.setOnAction(event -> {
            Task selected = taskListView.getSelectionModel().getSelectedItem();
            if (selected != null) {
                taskService.removeTask(selected);
                refreshTaskList();
            }
        });
    }

    private void refreshTaskList() {
        taskObservableList.setAll(taskService.getAllTasks());
    }
}
```

In this example:

- The UI text field and buttons trigger events.
- The controller handles these events, forwarding the add or remove commands to the `TaskService`.
- The observable list, linked to the UI list view, updates to reflect the current state of tasks.

21.3.4 Handling Updates and Task Modification

Extending this pattern, updating a task can follow a similar approach. For example, selecting a task from the list loads its details in editable fields. Upon user modification and clicking "Save," the controller updates the model through the service.

21.3.5 Concurrency and Responsiveness

JavaFX runs all UI updates on a special thread called the **JavaFX Application Thread**. Long-running operations on this thread block the UI, causing it to freeze and become unresponsive. To maintain a smooth user experience, any expensive business logic—such as database access or network calls—should run on background threads.

JavaFX provides concurrency utilities like `Task` and `Service` to run background tasks and update the UI safely once complete.

Example: Running a long-running save operation in the background

```
Task<Void> saveTask = new Task<>() {
    @Override
    protected Void call() throws Exception {
        taskService.saveToDatabase();
        return null;
    }
};

saveTask.setOnSucceeded(event -> {
    // Update UI after save completes
    refreshTaskList();
});

new Thread(saveTask).start();
```

This pattern ensures the UI remains responsive during data processing, improving user experience.

21.3.6 Best Practices for Clean UI-Logic Integration

1. **Keep UI Controllers Thin**: Avoid placing complex business logic in controllers. Delegate to service classes to promote separation of concerns.
2. **Use Observable Collections**: Bind UI components like `ListView` or `TableView` to observable collections to automatically reflect changes in the underlying data.
3. **Decouple with Interfaces**: Use interfaces for services and repositories, allowing easy substitution for testing or different implementations.
4. **Handle Validation Gracefully**: Validate user input before calling business logic and provide clear feedback via the UI.
5. **Employ MVVM or MVP Patterns**: For more complex apps, consider design patterns like Model-View-ViewModel (MVVM) or Model-View-Presenter (MVP) to further isolate UI and logic.

21.3.7 Challenges in Integration

- **Threading Issues**: Improper use of threads can lead to race conditions or UI exceptions. Always update UI components on the JavaFX Application Thread.
- **Event Handling Complexity**: Large applications can have tangled event logic. Centralizing event handling or using event buses can help.
- **State Synchronization**: Keeping UI state synchronized with underlying data models is crucial and can get complicated as the app grows.

21.3.8 Summary

Integrating JavaFX UI components with business logic classes involves handling UI events, delegating operations to service classes, and updating the UI based on changes. Using observable collections, background threads, and clear separation between UI and business logic leads to maintainable, responsive applications.

By following best practices, developers can build rich JavaFX applications where the UI and logic coexist cleanly, offering users a smooth experience and developers an organized codebase.

Chapter 22.

Unit Testing and Design with JUnit

1. Writing Unit Tests for OOP Code
2. Test-Driven Development (TDD)
3. Mocking and Dependency Injection

22 Unit Testing and Design with JUnit

22.1 Writing Unit Tests for OOP Code

Unit testing is a foundational practice in modern software development, especially critical in object-oriented design (OOD). It involves writing automated tests that verify the correctness of individual units—typically classes or methods—of your application. This practice promotes higher code quality, facilitates refactoring, and helps catch defects early in the development cycle.

22.1.1 Purpose and Benefits of Unit Testing in OOD

In OOD, systems are composed of interacting objects encapsulating state and behavior. Unit tests ensure that each object behaves as expected in isolation. The benefits are multifold:

- **Improved Code Reliability:** Tests confirm that methods return correct results and handle edge cases properly.
- **Safe Refactoring:** Developers can confidently improve or modify code knowing tests will catch regressions.
- **Clearer Design:** Writing tests encourages developers to design classes with testability in mind, often leading to better separation of concerns and cleaner interfaces.
- **Documentation:** Tests serve as executable specifications that illustrate how classes should be used.

22.1.2 Introduction to JUnit Basics

JUnit is the most widely used testing framework for Java, offering annotations and assertion methods that simplify writing and running tests.

Key Components:

- **Test Methods:** Annotated with `@Test`, these methods contain test logic.
- **Assertions:** Provided by `org.junit.jupiter.api.Assertions` (in JUnit 5), such as `assertEquals`, `assertTrue`, and `assertThrows`, to verify expected outcomes.
- **Test Lifecycle Annotations:**
  - `@BeforeEach` and `@AfterEach` run before and after each test, for setup and cleanup.
  - `@BeforeAll` and `@AfterAll` run once per test class for shared initialization.

22.1.3 Practical Example: Testing a Simple Class

Consider a basic `Calculator` class:

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

    public int divide(int numerator, int denominator) {
        if (denominator == 0) {
            throw new IllegalArgumentException("Denominator cannot be zero");
        }
        return numerator / denominator;
    }
}
```

A corresponding JUnit 5 test class might look like this:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class CalculatorTest {
    private Calculator calculator;

    @BeforeEach
    void setup() {
        calculator = new Calculator();
    }

    @Test
    void testAdd() {
        assertEquals(5, calculator.add(2, 3), "2 + 3 should equal 5");
        assertEquals(0, calculator.add(-2, 2), "Adding negative and positive should work");
    }

    @Test
    void testDivide() {
        assertEquals(2, calculator.divide(6, 3), "6 / 3 should equal 2");
    }

    @Test
    void testDivideByZero() {
        Exception exception = assertThrows(IllegalArgumentException.class, () -> {
            calculator.divide(5, 0);
        });
        assertEquals("Denominator cannot be zero", exception.getMessage());
    }
}
```

Here:

- `@BeforeEach` ensures a fresh `Calculator` before every test.
- Assertions verify normal and exceptional behaviors.
- Tests are isolated and repeatable.

22.1.4 Testing Classes with More Complexity

For classes that manage collections or collaborate with other objects, tests should verify state changes and interactions.

Example: Testing a `TaskManager` that adds/removes tasks:

```
public class TaskManager {
    private final List<String> tasks = new ArrayList<>();

    public void addTask(String task) {
        if (task == null || task.isEmpty()) {
            throw new IllegalArgumentException("Task cannot be empty");
        }
        tasks.add(task);
    }

    public boolean removeTask(String task) {
        return tasks.remove(task);
    }

    public List<String> getTasks() {
        return new ArrayList<>(tasks);
    }
}
```

Test example:

```
@Test
void testAddAndRemoveTasks() {
    TaskManager manager = new TaskManager();
    manager.addTask("Write tests");
    assertTrue(manager.getTasks().contains("Write tests"));

    assertTrue(manager.removeTask("Write tests"));
    assertFalse(manager.getTasks().contains("Write tests"));
}

@Test
void testAddEmptyTaskThrows() {
    TaskManager manager = new TaskManager();
    assertThrows(IllegalArgumentException.class, () -> manager.addTask(""));
}
```

22.1.5 Best Practices for Writing Effective Unit Tests

1. **Isolate Tests:** Each test should test one behavior and run independently.
2. **Use Descriptive Names:** Test methods like `testDivideByZeroThrowsException()` communicate intent clearly.
3. **Arrange-Act-Assert:** Structure tests into setup (Arrange), action (Act), and verification (Assert) phases.

4. **Avoid Test Interdependence:** Don't share mutable state across tests to prevent flaky results.
5. **Test Both Happy and Edge Cases:** Cover typical use, boundary conditions, and error scenarios.
6. **Keep Tests Fast:** Quick tests encourage frequent runs and early feedback.

22.1.6 Common Pitfalls and How to Avoid Them

- **Over-Mocking:** Excessive use of mocks can make tests brittle and less meaningful. Mock only external dependencies.
- **Testing Implementation Details:** Tests should verify observable behavior, not internal code structure.
- **Ignoring Exceptions:** Always test for error handling and exceptional cases.
- **Large Test Methods:** Break complex tests into smaller focused ones to improve clarity.
- **Skipping Setup/Cleanup:** Use lifecycle annotations to avoid repetitive boilerplate.

22.1.7 Summary

Writing unit tests is an essential skill in object-oriented design, ensuring code correctness and maintainability. JUnit provides a straightforward way to create effective tests, from simple method checks to complex object interactions. By following best practices and avoiding common pitfalls, developers can produce a robust test suite that facilitates confident development and easier refactoring.

22.2 Test-Driven Development (TDD)

Test-Driven Development (TDD) is a disciplined software development practice that reverses the traditional order of coding and testing: instead of writing code first and then tests, TDD emphasizes writing tests before the implementation code. This approach has transformed how developers think about design and quality by making tests the starting point for all development.

22.2.1 The TDD Cycle: Red-Green-Refactor

TDD follows a simple but powerful iterative cycle often summarized as **Red-Green-Refactor**:

1. **Red:** Write a failing test that defines a new piece of functionality or behavior. At this point, the test fails because the feature is not yet implemented.
2. **Green:** Write the minimum amount of production code necessary to make the failing test pass. This code does not have to be perfect—just enough to satisfy the test.
3. **Refactor:** Improve the newly written code, cleaning up duplication and enhancing design without changing its behavior. The tests serve as a safety net ensuring that the refactoring preserves correctness.

This cycle repeats continuously for each small feature or behavior.

22.2.2 How TDD Guides Design and Improves Code Quality

By forcing developers to write tests first, TDD naturally encourages thoughtful design:

- **Clear Requirements:** Writing a test requires understanding what the feature should do, promoting clarity in requirements.
- **Small, Focused Units:** Tests often cover small units of behavior, leading to finer-grained, modular code.
- **Better API Design:** Developers create APIs from the client's perspective (the test), resulting in more intuitive and usable interfaces.
- **Early Bug Detection:** Bugs are caught immediately as tests run with every change.
- **Documentation:** Tests serve as living documentation that describe expected behaviors.
- **Confident Refactoring:** Because tests cover expected behaviors, developers can refactor and improve the design safely.

22.2.3 Step-by-Step Example: Developing a Simple `BankAccount` Feature Using TDD

Suppose we need to implement a `deposit` method on a `BankAccount` class. Here's how TDD would proceed:

Step 1: Write the failing test (Red)

```
@Test
void depositIncreasesBalance() {
    BankAccount account = new BankAccount();
    account.deposit(100);
    assertEquals(100, account.getBalance());
}
```

This test expects that after depositing 100 units, the balance should reflect the deposit. Since `BankAccount` doesn't exist or `deposit` is not implemented yet, this test will fail.

Step 2: Write minimal code to pass the test (Green)

```java
public class BankAccount {
    private int balance = 0;

    public void deposit(int amount) {
        balance += amount;
    }

    public int getBalance() {
        return balance;
    }
}
```

Now, running the test passes successfully.

Step 3: Refactor if needed (Refactor)

The code is simple and clean, so no changes are needed. But if there were duplication or unclear naming, now is the time to fix that.

22.2.4 Benefits of Adopting TDD

- **Improved Code Quality:** The continuous verification of behavior ensures fewer bugs and higher confidence.
- **Better Design:** Small increments and tests lead to more maintainable and loosely coupled code.
- **Reduced Debugging Time:** Problems are identified close to their origin.
- **Enhanced Collaboration:** Tests clarify expected behavior, easing communication among team members.
- **Faster Feedback:** Developers get immediate validation that changes work as intended.

22.2.5 Challenges of TDD in Real Projects

Despite its benefits, TDD can present challenges:

- **Initial Learning Curve:** Developers new to TDD may find the process slow or awkward at first.
- **Requires Discipline:** Writing tests first can feel counterintuitive and requires a mindset shift.
- **Complex Integration Testing:** TDD focuses on unit tests, so integration and system-level testing need additional strategies.
- **Time Investment:** Writing tests upfront takes time, which can be difficult in tight deadlines.

However, many teams find that the long-term gains in reliability and maintainability outweigh these challenges.

22.2.6 TDD and Object-Oriented Principles

TDD complements key OOP principles perfectly:

- **Encapsulation:** Testing encourages small, self-contained units with clear responsibilities.
- **Single Responsibility Principle:** Writing focused tests often highlights when a class or method does too much.
- **Open/Closed Principle:** Tests make extending behavior safer, as regressions are caught early.
- **Polymorphism and Abstraction:** Tests written to interfaces or abstract classes enable flexible, maintainable designs.
- **Design by Contract:** Tests effectively become executable contracts that specify expected behavior.

In essence, TDD is not just a testing technique—it is a design technique that ensures code is both correct and well-structured.

22.2.7 Summary

Test-Driven Development reshapes how developers write code by making tests the foundation of the design process. Through its Red-Green-Refactor cycle, TDD fosters cleaner code, better design decisions, and higher confidence in software quality. While adopting TDD can be challenging, the benefits for maintainability, extensibility, and bug reduction are significant, making it a powerful practice for any object-oriented software project.

22.3 Mocking and Dependency Injection

Unit testing is most effective when tests isolate the behavior of the class under test from external dependencies, such as databases, web services, or other classes. This isolation ensures tests are reliable, fast, and focused on a single unit of behavior. However, in realistic systems, classes rarely work alone—they collaborate with other components. To test a class in isolation, we often need to **mock** or simulate these dependencies.

22.3.1 Why Mock Dependencies?

Imagine you are testing a `PaymentService` class that depends on an external `PaymentGateway` interface to process transactions. If your tests call the real payment gateway, they might:

- Be slow due to network latency.

- Fail unpredictably if the external service is down.
- Produce inconsistent results due to external state.
- Make real transactions, causing unintended side effects.

To avoid these issues, **mocking** replaces real collaborators with controlled fake implementations that simulate expected behavior. This way, you test only the logic inside `PaymentService` while assuming collaborators behave as specified.

22.3.2 Introduction to Mocking Frameworks

Manually writing mocks is tedious and error-prone. Mocking frameworks simplify this by automatically creating mock objects and specifying their behavior. One of the most popular frameworks in Java is **Mockito**.

Mockito allows you to:

- Create mock instances of interfaces or classes.
- Define what happens when methods on mocks are called.
- Verify interactions with mocks.
- Reset mocks to clear behavior or recorded calls.

22.3.3 Basic Mockito Usage Example

Suppose we have this interface and class:

```
public interface PaymentGateway {
    boolean processPayment(double amount);
}

public class PaymentService {
    private PaymentGateway gateway;

    public PaymentService(PaymentGateway gateway) {
        this.gateway = gateway;
    }

    public boolean pay(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
        return gateway.processPayment(amount);
    }
}
```

Here's a unit test for `PaymentService` using Mockito:

```
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class PaymentServiceTest {

    @Test
    void testSuccessfulPayment() {
        // Create a mock PaymentGateway
        PaymentGateway mockGateway = mock(PaymentGateway.class);

        // Define behavior: any amount returns true
        when(mockGateway.processPayment(anyDouble())).thenReturn(true);

        PaymentService service = new PaymentService(mockGateway);
        boolean result = service.pay(100);

        // Verify result and interaction
        assertTrue(result);
        verify(mockGateway).processPayment(100);
    }
}
```

In this test, the actual payment processing is never performed. Instead, the `PaymentGateway` is mocked to always return `true`, allowing us to test `PaymentService` in isolation.

Full runnable code:

```
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.ArgumentMatchers.anyDouble;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;

class PaymentServiceTest {

    @Test
    void testSuccessfulPayment() {
        // Create a mock PaymentGateway
        PaymentGateway mockGateway = mock(PaymentGateway.class);

        // Define behavior: any amount returns true
        when(mockGateway.processPayment(anyDouble())).thenReturn(true);

        PaymentService service = new PaymentService(mockGateway);
        boolean result = service.pay(100);

        // Verify result and interaction
        assertTrue(result);
        verify(mockGateway).processPayment(100);
    }
}

interface PaymentGateway {
    boolean processPayment(double amount);
```

```
}

class PaymentService {
    private PaymentGateway gateway;

    public PaymentService(PaymentGateway gateway) {
        this.gateway = gateway;
    }

    public boolean pay(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
        return gateway.processPayment(amount);
    }
}
```

22.3.4 Dependency Injection: Facilitating Mocking and Decoupling

To effectively mock dependencies, you must be able to **inject** them into the class under test. Dependency Injection (DI) is a design technique where an object's dependencies are supplied externally rather than hard-coded inside the class. DI promotes **loose coupling** and **testability**.

There are several forms of dependency injection:

- **Constructor Injection:** Dependencies are provided through a class constructor.
- **Setter Injection:** Dependencies are set via setter methods.
- **Interface Injection:** Dependencies are passed through interface methods (less common).

Constructor injection is the most popular and recommended for mandatory dependencies because it ensures immutability and clear dependencies.

22.3.5 Simple Dependency Injection Example

Revisiting the `PaymentService` class, note that the `PaymentGateway` is injected via the constructor:

```
public PaymentService(PaymentGateway gateway) {
    this.gateway = gateway;
}
```

This simple pattern enables:

- Easy substitution of real or mock implementations.

- Clear visibility of dependencies.
- Improved modularity.

In tests, as shown above, we can inject a mock `PaymentGateway`. In production, the real implementation is injected.

22.3.6 Benefits of Dependency Injection in Testing and Design

- **Isolated Testing:** Injected mocks isolate the class under test from collaborators.
- **Flexibility:** Swapping implementations becomes trivial without changing client code.
- **Better Design:** Classes depend on abstractions (interfaces) rather than concrete classes, adhering to the Dependency Inversion Principle.
- **Reusability:** Components are more reusable since dependencies are externally managed.
- **Easier Maintenance:** Clear dependencies simplify understanding and modifying code.

22.3.7 Summary and Best Practices

Mocking and dependency injection are powerful techniques that complement each other in improving unit test quality and software design:

- Use **mocking frameworks** like Mockito to create and control mock collaborators effortlessly.
- Design classes to receive dependencies externally, preferably through **constructor injection**.
- Keep mocks simple and focused on behavior relevant to the test.
- Avoid over-mocking; test with real implementations when feasible, especially for value objects or utilities.
- Combine dependency injection with inversion of control (IoC) containers like Spring for more scalable applications.

By embracing these practices, developers can write reliable, maintainable, and testable code, ultimately leading to higher quality software.

Chapter 23.

Clean Code and Maintainable Design

1. Naming, Formatting, and Readability
2. Reducing Coupling and Increasing Cohesion
3. Reusability and Extensibility

23 Clean Code and Maintainable Design

23.1 Naming, Formatting, and Readability

Clean, maintainable code is the cornerstone of professional software development. Among the many factors that contribute to code quality, **naming**, **formatting**, and overall **readability** play a pivotal role. These aspects might seem trivial compared to complex algorithms or architectural decisions, but in reality, they dramatically influence how easily developers can understand, maintain, and extend software over time.

23.1.1 The Importance of Clear and Consistent Naming

Naming is often described as the "*window into the programmer's mind.*" Well-chosen names communicate intent, making code self-explanatory. Conversely, poor names increase cognitive load, forcing readers to guess what variables, methods, or classes represent.

In Object-Oriented Programming (OOP), clear naming conventions become even more crucial because names describe not only data but also behaviors and relationships. For example:

- Class names should be **nouns** or noun phrases, clearly describing the object's role.
- Method names should be **verbs** or verb phrases, expressing actions or queries.
- Variable names should describe the **purpose or content** clearly and concisely.

Example: Good vs Poor Naming

```
// Poor Naming
int d; // What does 'd' represent?
String nm; // Ambiguous abbreviation

// Good Naming
int durationInSeconds;
String customerName;
```

Poorly named variables like `d` or `nm` force the reader to look elsewhere for clues. Clear names like `durationInSeconds` or `customerName` immediately convey meaning.

Similarly, method names should reflect their intent:

```
// Poor Naming
void calc(); // What is being calculated?

// Good Naming
void calculateInvoiceTotal();
```

In this example, the improved method name specifies the action and context, aiding comprehension.

23.1.2 Formatting: Indentation, Spacing, and Line Length

Formatting affects how quickly one can scan and understand code. Consistent indentation, appropriate spacing, and line length contribute to a clean visual structure.

- **Indentation:** Properly indent nested blocks, such as inside methods, loops, or conditionals, to visually separate logical units.
- **Spacing:** Use spaces around operators and after commas to improve readability.
- **Line length:** Aim to keep lines within 80-120 characters to avoid horizontal scrolling and improve side-by-side code comparison.

Example: Poor vs Good Formatting

```
// Poor formatting
public void process(){if(flag){doSomething();}else{doSomethingElse();}}

// Good formatting
public void process() {
    if (flag) {
        doSomething();
    } else {
        doSomethingElse();
    }
}
```

The formatted version is easier to read, debug, and modify. Consistent formatting prevents misunderstandings and reduces the chance of errors introduced by misaligned blocks.

23.1.3 Tools and Techniques to Enforce Code Style

Maintaining naming and formatting consistency in a team is challenging without automation. Fortunately, modern development environments and tools can enforce style guidelines and flag deviations early.

- **Integrated Development Environments (IDEs):** Most Java IDEs like IntelliJ IDEA and Eclipse provide automatic formatting and naming inspections.
- **Static Analysis Tools:** Tools such as Checkstyle, PMD, and SonarQube analyze codebases for style violations and potential bugs.
- **Code Formatters:** Tools like Google's Java Format or Eclipse's formatter enforce uniform style automatically.
- **Pre-commit Hooks:** Integrate style checks into the version control workflow to prevent poorly formatted code from entering the repository.

Using these tools helps ensure all team members adhere to agreed standards, reducing "style noise" in code reviews and focusing attention on design and logic.

23.1.4 Readabilitys Impact on Collaboration and Maintenance

Readable code is easier to review, debug, and extend. It lowers the barrier for new team members to onboard quickly, reducing costly misunderstandings and miscommunication. When code is self-explanatory, developers spend less time deciphering what it does and more time improving features or fixing bugs.

Moreover, readable code supports better **refactoring**, a continuous process essential for keeping the codebase healthy and adaptive to changing requirements.

In contrast, poorly named variables and inconsistent formatting contribute to **technical debt**—the accumulated cost of shortcuts and messy code that slows down development and increases the likelihood of defects.

23.1.5 Summary

- Use **clear, descriptive names** for classes, methods, and variables to convey intent.
- Follow **consistent formatting guidelines** for indentation, spacing, and line length to enhance visual clarity.
- Employ **tools and automated checks** to enforce coding standards in a team environment.
- Recognize that readability directly improves **collaboration, maintenance, and code quality**.

By prioritizing naming, formatting, and readability, developers create code that not only works but also communicates clearly—an invaluable asset in any software project.

23.2 Reducing Coupling and Increasing Cohesion

Two of the most critical attributes of maintainable object-oriented software are **low coupling** and **high cohesion**. These qualities directly affect a system's **modularity**, **understandability**, and **resilience to change**. While they are often discussed together, each addresses a distinct aspect of class and module design.

23.2.1 What Are Coupling and Cohesion?

Coupling refers to the degree of **interdependence** between software modules or classes. When two classes are tightly coupled, a change in one often requires changes in the other. This makes systems brittle and harder to evolve.

Cohesion refers to how **strongly related** and **focused** the responsibilities of a single module or class are. A highly cohesive class does one thing well. Low cohesion results in classes doing too many unrelated things, making them harder to reuse, test, and maintain.

Example of Tight Coupling and Low Cohesion:

```
public class ReportManager {
    private DatabaseConnection db;
    private FilePrinter printer;

    public void generateAndPrintReport() {
        db.connect();
        String data = db.fetchData();
        printer.print(data);
    }
}
```

Here, `ReportManager` is tightly coupled to both `DatabaseConnection` and `FilePrinter`, and it mixes concerns—data access and printing—violating the **Single Responsibility Principle (SRP)**.

23.2.2 Why Low Coupling and High Cohesion Matter

Reducing coupling:

- Makes components more **modular** and **testable**.
- Allows components to change **independently**.
- Supports **dependency inversion** and **interface-driven design**.

Increasing cohesion:

- Improves **clarity** and **comprehension** of class responsibilities.
- Encourages better **encapsulation**.
- Makes **reuse** and **refactoring** easier.

Together, they enable codebases to evolve gracefully as requirements change.

23.2.3 Refactoring for Better Coupling and Cohesion

Let's improve the previous example by decoupling the components using **interfaces** and separating responsibilities.

```
public interface DataFetcher {
    String fetchData();
}
```

```java
public interface OutputDevice {
    void output(String content);
}
```

```java
public class ReportGenerator {
    private final DataFetcher fetcher;

    public ReportGenerator(DataFetcher fetcher) {
        this.fetcher = fetcher;
    }

    public String generateReport() {
        return fetcher.fetchData();
    }
}

public class ReportPrinter {
    private final OutputDevice device;

    public ReportPrinter(OutputDevice device) {
        this.device = device;
    }

    public void print(String report) {
        device.output(report);
    }
}
```

Now, `ReportGenerator` and `ReportPrinter` are each highly cohesive and have no direct dependency on specific implementations. This design enables easier testing and supports new data sources or output formats with minimal changes.

23.2.4 Patterns and Principles That Support These Goals

Several design principles and patterns explicitly aim to reduce coupling and improve cohesion:

- **Single Responsibility Principle (SRP):** Encourages cohesive classes by limiting each class to one reason to change.
- **Dependency Inversion Principle (DIP):** Promotes low coupling by depending on abstractions rather than concrete implementations.
- **Strategy Pattern:** Encapsulates interchangeable behaviors, reducing direct coupling between clients and algorithms.
- **Observer Pattern:** Reduces coupling between event sources and listeners, allowing independent evolution.

By leveraging these ideas, systems can remain flexible and easy to extend.

23.2.5 Measuring and Balancing

Although subjective, coupling and cohesion can be estimated through indicators:

- **High coupling signs:** Lots of imports, direct instantiations of concrete classes, and ripple effects when making changes.
- **Low cohesion signs:** Classes or methods doing multiple unrelated things, long methods with unrelated branches.

Tooling like static analyzers (SonarQube, IntelliJ inspections) can offer metrics like **class coupling** or **method cohesion** scores.

Balance is key. Over-engineering to reduce coupling (e.g., excessive use of interfaces or factories) can introduce unnecessary complexity. Similarly, extreme cohesion might result in many tiny classes that are hard to manage.

23.2.6 Conclusion

Low coupling and high cohesion are foundational to writing clean, maintainable Java code. They enhance clarity, modularity, and the system's ability to evolve over time. Applying SOLID principles, using appropriate design patterns, and refactoring regularly are effective strategies for achieving these qualities.

As you design and review your code, ask:

- Is this class doing only one thing?
- Can I change this class without affecting others?
- Would this design be easier to test if I broke it up?

These questions help guide your code toward greater cohesion and lower coupling—hallmarks of sustainable software architecture.

23.3 Reusability and Extensibility

In object-oriented programming (OOP), **reusability** and **extensibility** are essential attributes of maintainable, scalable systems. Reusability allows components to be used in multiple contexts with minimal modification, while extensibility ensures that systems can adapt to new requirements without significant rework. Together, they empower developers to build software that grows organically and remains cost-effective over time.

23.3.1 What Makes Code Reusable?

Reusable code is **modular**, **decoupled**, and **generalized** enough to be applicable in different situations. This often means writing code that solves a broader problem than a single use case demands—without becoming abstract to the point of confusion.

For example, rather than hardcoding business logic into a specific report format, a reusable report generator might accept different data sources and output formats via interfaces:

```
public interface ReportDataSource {
    String getData();
}

public interface ReportFormatter {
    String format(String data);
}

public class ReportGenerator {
    private final ReportDataSource dataSource;
    private final ReportFormatter formatter;

    public ReportGenerator(ReportDataSource dataSource, ReportFormatter formatter) {
        this.dataSource = dataSource;
        this.formatter = formatter;
    }

    public String generateReport() {
        return formatter.format(dataSource.getData());
    }
}
```

This structure allows the same `ReportGenerator` to be reused across various domains by plugging in different implementations.

23.3.2 What Makes Code Extensible?

Extensible code is designed with **change in mind**. It anticipates that new behaviors, types, or rules may be added later. The **Open/Closed Principle**—classes should be open for extension but closed for modification—is foundational to extensibility.

Abstract classes and interfaces are key mechanisms:

```
public abstract class PaymentProcessor {
    public abstract void processPayment(double amount);
}

public class CreditCardProcessor extends PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        // logic for credit card
    }
```

```
}
```

Later, if a new payment method is needed, a subclass can be introduced without changing existing code.

23.3.3 Strategies to Promote Reuse and Extension

1. **Modular Design**: Break systems into small, focused modules with clear responsibilities. This improves reusability across contexts.
2. **Composition Over Inheritance**: Instead of inheriting behavior, use composition to assemble it. This enables more flexible reuse.

```
public class NotificationService {
    private final MessageSender sender;

    public NotificationService(MessageSender sender) {
        this.sender = sender;
    }

    public void notify(String message) {
        sender.send(message);
    }
}
```

By injecting different `MessageSender` implementations (e.g., SMS, Email), the `NotificationService` becomes reusable and extensible.

3. **Interface-Oriented Design**: Depending on interfaces rather than concrete classes reduces coupling and increases adaptability.
4. **Parameterization with Generics**: Generic classes and methods enhance reuse by allowing the same logic to operate on different types.

```
public class Pair<T, U> {
    private T first;
    private U second;
    // constructors, getters
}
```

23.3.4 Trade-Offs: Flexibility vs Complexity

While flexibility is powerful, **overengineering** can backfire. Adding too many interfaces or extension points may introduce **unnecessary complexity**, especially when the added flexibility is speculative rather than based on real needs.

For example, creating a plugin system for a simple text editor might be premature unless extensibility is a core requirement. It's crucial to balance current requirements with future-proofing.

Follow the **YAGNI** ("You Aren't Gonna Need It") principle to avoid designing for imaginary needs.

23.3.5 Pitfalls That Hinder Reuse

1. **Tight Coupling**: Components that rely on specific implementations are harder to reuse elsewhere.
2. **Low Cohesion**: Classes doing too much are harder to extract and reuse.
3. **Scattered Logic**: Business logic split across multiple unrelated classes makes reuse harder.
4. **Hidden Dependencies**: Code that relies on global state or hardcoded values makes reuse fragile and error-prone.

23.3.6 Conclusion

Reusability and extensibility are not accidental—they result from thoughtful design. By applying OOP principles such as modularization, abstraction, and composition, you can build software that not only works today but remains adaptable tomorrow. Always strive to write code that is **as simple as possible but no simpler**, focusing on real needs while keeping the door open to future evolution.

Chapter 24.

Designing for Change: Flexibility and Scalability

1. Open-Ended Design with Interfaces
2. Strategy for Extensible Systems
3. Designing APIs

24 Designing for Change: Flexibility and Scalability

24.1 Open-Ended Design with Interfaces

24.1.1 Open-Ended Design with Interfaces

Designing software that stands the test of time requires more than just solving today's problem—it demands anticipating tomorrow's changes. In Java and object-oriented design, **interfaces** play a critical role in achieving this flexibility. They form the backbone of **open-ended design**, enabling polymorphism, decoupling, and extension without modifying existing code.

24.1.2 The Power of Interfaces

An interface in Java defines a contract: a set of method signatures without implementation. By programming to interfaces rather than implementations, we decouple **what** a component does from **how** it does it. This decoupling is key to **extensibility** and **testability**.

Consider a payment processing example:

```
public interface PaymentMethod {
    void pay(double amount);
}

public class CreditCardPayment implements PaymentMethod {
    public void pay(double amount) {
        // credit card processing logic
    }
}

public class PayPalPayment implements PaymentMethod {
    public void pay(double amount) {
        // PayPal processing logic
    }
}

public class CheckoutService {
    private final PaymentMethod paymentMethod;

    public CheckoutService(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    public void completePurchase(double total) {
        paymentMethod.pay(total);
    }
}
```

This design allows new payment types to be added (e.g., cryptocurrency) without modifying

the `CheckoutService` class, adhering to the **Open/Closed Principle**.

24.1.3 Enabling Polymorphism and Decoupling

Interfaces promote **polymorphism**—different classes implementing the same interface can be treated uniformly. This enables interchangeable components, reducing dependencies between system parts.

In the example above, `CheckoutService` doesn't care whether it's dealing with `CreditCardPayment` or `PayPalPayment`; both conform to `PaymentMethod`. This level of abstraction allows business logic to evolve independently from concrete implementations.

This also encourages **dependency inversion**, where high-level modules depend on abstractions rather than concretions. This is fundamental to building flexible, testable architectures.

24.1.4 Interface Segregation and Versioning

Interfaces should remain focused and minimal to avoid forcing clients to depend on methods they don't use. This is the essence of the **Interface Segregation Principle** from SOLID. Smaller, role-specific interfaces improve flexibility and reduce the risk of breaking changes when evolving systems.

For example:

```
public interface Readable {
    String read();
}

public interface Writable {
    void write(String data);
}
```

Compared to a fat interface like:

```
public interface FileHandler {
    String read();
    void write(String data);
    void delete();
}
```

Splitting large interfaces allows components to implement only what they need. A logging service may only require `Writable`, while a configuration loader may need just `Readable`.

Versioning is another consideration. Once an interface is published, changing its method signatures can break clients. One solution is to define new interfaces for extended capabilities:

```
public interface AdvancedPaymentMethod extends PaymentMethod {
    boolean validatePayment();
}
```

This approach allows older code to continue functioning while newer code benefits from extended features.

24.1.5 Supporting Evolving Requirements

Requirements rarely remain static. Interfaces enable you to accommodate new functionality without invasive changes. For instance, suppose your original system supported only file-based logging:

```
public class FileLogger implements Logger {
    public void log(String message) {
        // write to file
    }
}
```

Later, you might introduce a `DatabaseLogger` or `RemoteLogger`. Because the system depends on the `Logger` interface, the underlying implementations can evolve freely. This adaptability is critical for scalability and maintainability in long-lived systems.

Interfaces also foster **testing and mocking**. A unit test can inject a mock implementation of an interface to isolate and test logic in isolation:

```
@Test
void testCheckoutWithMockPayment() {
    PaymentMethod mockPayment = amount -> System.out.println("Mock pay: " + amount);
    CheckoutService service = new CheckoutService(mockPayment);
    service.completePurchase(50.0);
}
```

24.1.6 Conclusion

Interfaces are essential tools for designing software that accommodates change. By abstracting behavior, promoting polymorphism, and enforcing decoupling, they enable developers to build systems that grow and evolve over time. Keeping interfaces small and cohesive, avoiding unnecessary commitments to implementation details, and planning for backward compatibility are all key practices in open-ended design. When used thoughtfully, interfaces empower codebases to remain agile, testable, and scalable as requirements inevitably change.

24.2 Strategy for Extensible Systems

24.2.1 Strategy for Extensible Systems

In modern software development, requirements evolve—sometimes unpredictably. Designing systems that are flexible and easy to extend is essential for long-term sustainability. Extensible design doesn't mean predicting every possible future feature; instead, it means building systems with **well-defined extension points** and **pluggable behavior** that allow new functionality to be integrated without disrupting existing code.

This section explores key strategies and patterns that help achieve extensibility, such as the **Strategy Pattern**, **Plugin Architecture**, and the use of **interfaces and extension points**.

24.2.2 Designing for Change

Extensibility is the ability of a system to grow or be enhanced with minimal code changes. Instead of modifying core logic, developers should be able to add new behavior through composition or configuration. Designing for extension involves:

- Programming to interfaces.
- Separating core behavior from variant behavior.
- Avoiding tight coupling between modules.
- Encapsulating change behind abstractions.

Let's explore practical strategies to achieve these goals.

24.2.3 Strategy Pattern: Swappable Behavior

The **Strategy Pattern** encapsulates a family of algorithms or behaviors, allowing them to be selected at runtime. This supports extensibility by enabling new strategies to be added without altering existing classes.

```
public interface DiscountStrategy {
    double applyDiscount(double price);
}

public class NoDiscount implements DiscountStrategy {
    public double applyDiscount(double price) {
        return price;
    }
}

public class PercentageDiscount implements DiscountStrategy {
    private final double percent;
```

```
    public PercentageDiscount(double percent) {
        this.percent = percent;
    }
    public double applyDiscount(double price) {
        return price * (1 - percent);
    }
}

public class ShoppingCart {
    private DiscountStrategy discountStrategy;

    public void setDiscountStrategy(DiscountStrategy strategy) {
        this.discountStrategy = strategy;
    }

    public double checkout(double total) {
        return discountStrategy.applyDiscount(total);
    }
}
```

New discount strategies can be added without touching `ShoppingCart`, making the system flexible and future-proof.

24.2.4 Plugin Architecture: Dynamic Extensions

The **Plugin Pattern** enables runtime discovery and loading of modules. A plugin system allows new components—such as file format readers, authentication providers, or payment processors—to be added without modifying the core application.

This architecture typically defines:

- A **plugin interface**.
- A **loader or registry**.
- A mechanism to discover implementations (e.g., Java's `ServiceLoader`).

Example:

```
public interface ReportPlugin {
    String generateReport();
}
```

Each plugin class implements this interface, and the application uses `ServiceLoader` to dynamically load them:

```
ServiceLoader<ReportPlugin> loader = ServiceLoader.load(ReportPlugin.class);
for (ReportPlugin plugin : loader) {
    System.out.println(plugin.generateReport());
}
```

Adding a new plugin requires no changes to core logic—only a new JAR or class with a

proper metadata file. This is common in IDEs, browsers, and enterprise systems.

24.2.5 Extension Points: Interfaces and Hooks

Another technique is to define **extension points** in the codebase where developers can inject behavior. These often appear as abstract classes, interfaces, or callbacks.

For example, in a document editor:

```
public interface ExportFormat {
    void export(Document doc);
}
```

Rather than hardcoding export types (PDF, Word, HTML), this interface allows new formats to be supported later without changing core functionality.

Frameworks like Spring and Eclipse use this approach extensively, offering points where developers can extend functionality declaratively.

24.2.6 Testing and Maintenance Implications

Extensible design often improves **testability**. Since components are decoupled and communicate via interfaces, mock implementations can be injected for unit tests.

```
@Test
void testDiscountStrategy() {
    DiscountStrategy mockStrategy = price -> 0.0; // always free
    ShoppingCart cart = new ShoppingCart();
    cart.setDiscountStrategy(mockStrategy);
    assertEquals(0.0, cart.checkout(100.0));
}
```

However, extensibility introduces **complexity**. You must clearly document extension points, version interfaces carefully, and ensure changes don't break existing contracts.

Best practices include:

- Using **interface segregation** to avoid bloated APIs.
- Applying **Open/Closed Principle** to separate stable code from evolving behavior.
- Providing default implementations or fallbacks for optional extensions.

24.2.7 Conclusion

Extensible systems are not only easier to enhance—they're more resilient to change. By leveraging patterns like Strategy, Plugin, and defining thoughtful extension points, developers can evolve systems without the fragility of constant rewrites. Careful design up front, combined with a modular architecture, helps create systems that adapt gracefully as requirements shift.

24.3 Designing APIs

In software development, an **Application Programming Interface (API)** is more than just a set of methods—it's a contract between the component and its users. A well-designed API promotes usability, maintainability, and flexibility, serving as the foundation for extensible and scalable systems. Whether you're exposing a library, framework, or service interface, careful API design greatly influences how easily clients adopt and integrate with your code.

24.3.1 Principles of Good API Design

A good API should be:

- **Clear**: Easy to read, understand, and use correctly.
- **Consistent**: Follows naming and structural conventions.
- **Minimal**: Exposes only what is necessary; no leakage of internal details.
- **Stable**: Designed to evolve without breaking existing users.
- **Composable**: Works well with other APIs and supports extension or customization.

APIs should follow the **"principle of least astonishment"**—users should not be surprised by its behavior. This minimizes the learning curve and reduces errors.

24.3.2 Example: A Simple Inventory API

Consider an API for managing product inventory:

```
public interface InventoryService {
    void addProduct(Product product);
    boolean removeProduct(String productId);
    Product findProduct(String productId);
    List<Product> listProducts();
}
```

This interface is intuitive and communicates intent clearly. It uses common types (`String`, `List`, and a domain-specific `Product` class), avoids exposing implementation details, and

provides a cohesive set of operations. This API is easy to document and test, and it can be implemented or mocked in various ways.

24.3.3 Versioning and Backward Compatibility

As software evolves, so must APIs. However, careless changes can break client code. Preserving **backward compatibility**—ensuring older clients still function with new versions—is crucial for public APIs.

Avoid breaking changes, such as:

- Removing or renaming methods.
- Changing method signatures.
- Altering expected behavior or return types.

Instead, use strategies like:

- **Overloading**: Add new versions of methods with additional parameters.
- **Deprecation**: Mark old methods with `@Deprecated` and guide users to replacements.
- **Semantic versioning**: Adopt versioning schemes like MAJOR.MINOR.PATCH to signal compatibility expectations.

For example:

```
@Deprecated
void addProduct(String name); // old method

void addProduct(Product product); // new preferred method
```

Internal systems can handle more aggressive changes, but public-facing APIs should evolve conservatively.

24.3.4 Documentation and Discoverability

An API is only as good as its documentation. This includes:

- **JavaDoc comments** explaining purpose, parameters, return values, and exceptions.
- **Usage examples** showing typical scenarios.
- **Edge cases** and limitations to watch for.

Good IDE integration and consistent naming also enhance **discoverability**. For instance, naming a method `findById()` instead of `lookup()` clarifies its intent immediately.

24.3.5 Impact on Client Code and Evolution

An API shapes how clients structure their code. A poor API can force users to write boilerplate or make incorrect assumptions, increasing coupling and reducing flexibility.

Consider this rigid design:

```
public class ProductManager {
    public void manageInventory(List<Product> inventory);
}
```

Here, the client has no idea what "manageInventory" does. A better design would break down responsibilities and expose fine-grained control:

```
public interface InventoryService {
    void restockProduct(String productId, int quantity);
    boolean isInStock(String productId);
}
```

This structure allows client applications to evolve independently and compose functionality more effectively.

24.3.6 Balancing Simplicity and Functionality

API design requires trade-offs. A **simple** API is easy to learn and use but may lack power or flexibility. A **powerful** API may offer broad capabilities but become complex and hard to understand.

A good compromise involves:

- Providing a **simple core** API for common use cases.
- Offering **advanced hooks** or extensions for complex needs.

For example, Java's `Collections` API provides simple interfaces like `List` and `Map`, but also supports custom sorting with `Comparator`, custom iteration with `Spliterator`, and transformations via streams.

24.3.7 Conclusion

Designing clean, stable, and intuitive APIs is central to building flexible systems. A thoughtful API reduces coupling, guides users to correct usage, and adapts gracefully to change. By prioritizing clarity, versioning carefully, and balancing simplicity with extensibility, you create APIs that not only solve today's problems but are ready for tomorrow's growth.

Reflection Questions:

- Can your API be used correctly without reading the implementation?
- How would you add a new feature without breaking clients?
- Is each public method necessary, or does it expose internal concerns?

Design your APIs as if you'll support them for years—because you likely will.