Java Design Patterns

1. **Creational Patterns** — Object creation mechanisms
2. **Structural Patterns** — Composition of classes and objects
3. **Behavioral Patterns** — Object interaction and responsibility

## **✅ Creational Design Patterns**

1. **Singleton** – Ensures a class has only **one instance** and provides a global access point.
2. **Factory Method** – Lets subclasses decide **which concrete class to instantiate**.
3. **Abstract Factory** – Creates **families of related objects** without specifying their concrete classes.
4. **Builder** – Separates **complex object construction** from its representation, step by step.
5. **Prototype** – Creates new objects by **cloning existing ones**, avoiding the cost of new.

## ✅ **Structural Design Patterns**

1. **Adapter** – Converts the **interface of a class** into another interface clients expect (wrapper).
2. **Bridge** – **Decouples abstraction from implementation**, letting both vary independently.
3. **Composite** – Treats **individual objects and compositions** of objects uniformly (tree structure).
4. **Decorator** – Dynamically **adds new responsibilities/behavior** to objects without modifying their class.
5. **Facade** – Provides a **simple unified interface** to a complex subsystem.
6. **Flyweight** – Shares **common state** between many objects to save memory.
7. **Proxy** – Provides a **surrogate or placeholder** to control access to an object (lazy load, security).

## **✅ Behavioral Design Patterns**

1. **Chain of Responsibility** – Passes requests along a **chain of handlers** until one processes it.
2. **Command** – Encapsulates a **request as an object** so you can log, queue, or undo operations.
3. **Interpreter** – Defines a **grammar** and interprets sentences in that grammar.
4. **Iterator** – Provides a way to **traverse a collection** without exposing its internal structure.
5. **Mediator** – Centralizes **communication** between multiple objects to reduce dependencies.
6. **Memento** – **Captures and restores** an object’s internal state without exposing its details (undo).
7. **Observer** – Lets objects **subscribe** and get notified automatically of **state changes**.
8. **State** – Lets an object **change its behavior** when its internal state changes (avoids if/else).
9. **Strategy** – Defines a **family of algorithms** and makes them **interchangeable** at runtime.
10. **Template Method** – Defines the **skeleton of an algorithm**, letting subclasses fill in details.
11. **Visitor** – Lets you **add new operations** to a class hierarchy **without modifying existing classes**.
12. **Creational Patterns in Java**

These deal with **object creation mechanisms**. Instead of instantiating objects with new everywhere, we abstract that process.

**The goal:**

* Make your code **flexible** (easy to change what object is created).
* Hide **complex creation logic**.
* Follow **SOLID principles** (esp. Single Responsibility and Open/Closed).

Creational patterns in Java:

1. Singleton
2. Factory Method
3. Abstract Factory
4. Builder
5. Prototype

# ****1.1 Singleton Pattern****

### 📌 Intent

Ensure a class has **only one instance** and provide a **global access point** to it.  
Think of things like:

* A single configuration manager
* A single logging service
* A single database connection pool

### 📌 Real-World Analogy

Imagine the **Prime Minister of a country** — there can be only one official PM at a time. If you want to reach the PM, you go through the official office, not create a new one.

### 📌 Java Example (Thread-safe Lazy Singleton)

public class Singleton {

// volatile ensures visibility in multithreaded environments

private static volatile Singleton instance;

// private constructor prevents instantiation

private Singleton() {}

// global access point

public static Singleton getInstance() {

if (instance == null) { // first check (no locking for performance)

synchronized (Singleton.class) {

if (instance == null) { // second check (safety)

instance = new Singleton();

}

}

}

return instance;

}

public void showMessage() {

System.out.println("Hello from Singleton!");

}

}

class TestSingleton {

public static void main(String[] args) {

Singleton s1 = Singleton.getInstance();

Singleton s2 = Singleton.getInstance();

System.out.println(s1 == s2); // true → both point to same object

s1.showMessage();

}

}

### 📌 Key Points

* Private constructor blocks direct instantiation.
* volatile + **double-checked locking** → ensures thread safety.
* Suitable when **one object controls a global resource**.

# ****1.2. Factory Method Pattern****

### 📌 Intent

Define an interface for creating an object, but let **subclasses decide** which class to instantiate.  
The Factory Method makes a class defer instantiation to subclasses.

### 📌 Real-World Analogy

Imagine a **pizza store**. You order a “Veg Pizza” or a “Cheese Pizza”. The store’s makePizza() method decides which one to cook. You don’t worry about the actual cooking process.

### 📌 Java Example

// Product interface

interface Shape {

void draw();

}

// Concrete products

class Circle implements Shape {

public void draw() { System.out.println("Drawing Circle"); }

}

class Rectangle implements Shape {

public void draw() { System.out.println("Drawing Rectangle"); }

}

// Factory class

class ShapeFactory {

public Shape getShape(String type) {

return switch (type.toLowerCase()) {

case "circle" -> new Circle();

case "rectangle" -> new Rectangle();

default -> throw new IllegalArgumentException("Unknown shape: " + type);

};

}

}

class TestFactory {

public static void main(String[] args) {

ShapeFactory factory = new ShapeFactory();

Shape s1 = factory.getShape("circle");

Shape s2 = factory.getShape("rectangle");

s1.draw();

s2.draw();

}

}

### 📌 Key Points

* **Encapsulates object creation** → client doesn’t use new.
* Adding new Shape types is easy → just extend factory logic.
* Widely used in frameworks (e.g., JDBC DriverManager.getConnection() is a factory).

# ****1.3. Abstract Factory Pattern****

### 📌 Intent

Provide an **interface for creating families of related objects**, without specifying their concrete classes.  
So instead of one product (Shape), we deal with **related sets of products** (Button + Checkbox).

### 📌 Real-World Analogy

Think of **furniture factories**:

* A “Victorian Furniture Factory” makes **Victorian Chair + Victorian Sofa**.
* A “Modern Furniture Factory” makes **Modern Chair + Modern Sofa**.  
  You choose a factory, and it consistently gives you matching furniture.

### 📌 Java Example

// Product interfaces

interface Button { void paint(); }

interface Checkbox { void paint(); }

// Concrete Products - Windows

class WinButton implements Button {

public void paint() { System.out.println("Windows Button"); }

}

class WinCheckbox implements Checkbox {

public void paint() { System.out.println("Windows Checkbox"); }

}

// Concrete Products - Mac

class MacButton implements Button {

public void paint() { System.out.println("Mac Button"); }

}

class MacCheckbox implements Checkbox {

public void paint() { System.out.println("Mac Checkbox"); }

}

// Abstract Factory

interface GUIFactory {

Button createButton();

Checkbox createCheckbox();

}

// Concrete Factories

class WinFactory implements GUIFactory {

public Button createButton() { return new WinButton(); }

public Checkbox createCheckbox() { return new WinCheckbox(); }

}

class MacFactory implements GUIFactory {

public Button createButton() { return new MacButton(); }

public Checkbox createCheckbox() { return new MacCheckbox(); }

}

// Client

class Application {

private final Button button;

private final Checkbox checkbox;

public Application(GUIFactory factory) {

button = factory.createButton();

checkbox = factory.createCheckbox();

}

public void paint() {

button.paint();

checkbox.paint();

}

}

class TestAbstractFactory {

public static void main(String[] args) {

GUIFactory factory = new MacFactory(); // Could be chosen dynamically

Application app = new Application(factory);

app.paint();

}

}

### 📌 Key Points

* Provides **consistency** (Windows Button + Windows Checkbox, not mixed with Mac).
* Makes it easy to **switch families of products**.
* Used in cross-platform frameworks (Swing, AWT UI components).

# ****1.4. Builder Pattern****

### 📌 Intent

Separate **construction of a complex object** from its representation.  
This lets you build objects **step by step** with a fluent API.

### 📌 Real-World Analogy

Ordering a **customized Subway sandwich**:

* Choose bread → add cheese → add veggies → add sauces.  
  At the end, you get your sandwich (the final object).

### 📌 Java Example

class House {

private String walls;

private String roof;

private String garage;

private House() {} // private constructor

@Override

public String toString() {

return "House [walls=" + walls + ", roof=" + roof + ", garage=" + garage + "]";

}

// Builder as inner static class

public static class Builder {

private final House house = new House();

public Builder buildWalls(String walls) {

house.walls = walls;

return this;

}

public Builder buildRoof(String roof) {

house.roof = roof;

return this;

}

public Builder buildGarage(String garage) {

house.garage = garage;

return this;

}

public House build() {

return house;

}

}

}

class TestBuilder {

public static void main(String[] args) {

House house = new House.Builder()

.buildWalls("Brick Walls")

.buildRoof("Tile Roof")

.buildGarage("Two Car Garage")

.build();

System.out.println(house);

}

}

### 📌 Key Points

* Avoids “telescoping constructors” (e.g., 5+ parameter constructors).
* Makes object construction **readable and flexible**.
* Used in StringBuilder, Stream.Builder in Java.

# ****1.5. Prototype Pattern****

### 📌 Intent

Instead of creating objects from scratch, **clone an existing object**.  
This is useful when:

* Object creation is costly.
* You want a copy with small modifications.

### 📌 Real-World Analogy

Think of **photocopying a document** instead of retyping it from scratch.

### 📌 Java Example

class Document implements Cloneable {

private String content;

public Document(String content) {

this.content = content;

}

@Override

public Document clone() {

try {

return (Document) super.clone(); // shallow copy

} catch (CloneNotSupportedException e) {

throw new RuntimeException(e);

}

}

public void setContent(String content) { this.content = content; }

public String getContent() { return content; }

}

class TestPrototype {

public static void main(String[] args) {

Document doc1 = new Document("Original Content");

Document doc2 = doc1.clone(); // clone

doc2.setContent("Modified Copy");

System.out.println("Doc1: " + doc1.getContent());

System.out.println("Doc2: " + doc2.getContent());

}

}

### 📌 Key Points

* Cloning avoids re-creation of heavy objects.
* Must decide between **shallow copy vs deep copy**.
* Used in Java: Object.clone() (but not widely recommended because of pitfalls).

✅ That’s the **complete set of Creational Patterns** with explanations + code.

# ****Structural Patterns in Java****

These deal with **object composition** — how classes and objects are combined to form larger structures.  
They focus on:

* Reusing objects.
* Making code more **flexible and extensible**.
* Defining **relationships between entities** without tightly coupling them.

The goal:

* Organize classes and objects in a way that makes the system **scalable**.
* Promote **low coupling and high cohesion**.

Structural Patterns in Java:

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Facade
6. Flyweight
7. Proxy

## **2.1. Adapter Pattern**

**Intent:**  
Convert the interface of a class into another interface that clients expect.  
Adapter lets classes work together that could not otherwise because of incompatible interfaces.

**Real-world analogy:**  
Think of a **power adapter**:

* Your laptop charger plug doesn’t fit the socket in another country.
* An adapter converts the socket format without changing either your laptop or the wall socket.

**When to use:**

* When you have an existing class but its interface doesn’t match the required one.
* To make old code work with new systems without rewriting.

**Java Example:**

// Target interface (what client expects)

interface MediaPlayer {

void play(String audioType, String fileName);

}

// Adaptee (incompatible interface)

class AdvancedMediaPlayer {

void playVlc(String fileName) {

System.out.println("Playing vlc file: " + fileName);

}

void playMp4(String fileName) {

System.out.println("Playing mp4 file: " + fileName);

}

}

// Adapter

class MediaAdapter implements MediaPlayer {

private AdvancedMediaPlayer advancedMusicPlayer = new AdvancedMediaPlayer();

@Override

public void play(String audioType, String fileName) {

if (audioType.equalsIgnoreCase("vlc")) {

advancedMusicPlayer.playVlc(fileName);

} else if (audioType.equalsIgnoreCase("mp4")) {

advancedMusicPlayer.playMp4(fileName);

}

}

}

// Client

class AudioPlayer implements MediaPlayer {

private MediaAdapter mediaAdapter;

@Override

public void play(String audioType, String fileName) {

if (audioType.equalsIgnoreCase("mp3")) {

System.out.println("Playing mp3 file: " + fileName);

} else {

mediaAdapter = new MediaAdapter();

mediaAdapter.play(audioType, fileName);

}

}

}

// Demo

public class AdapterPatternDemo {

public static void main(String[] args) {

AudioPlayer audioPlayer = new AudioPlayer();

audioPlayer.play("mp3", "song.mp3");

audioPlayer.play("mp4", "movie.mp4");

audioPlayer.play("vlc", "video.vlc");

}

}

## **2.2. Bridge Pattern**

**Intent:**  
Decouple an abstraction from its implementation so that the two can vary independently.

**Real-world analogy:**  
Think of a **TV remote control**:

* The remote (abstraction) works with different TV brands (implementations).
* You can add new remotes or new TV models independently.

**When to use:**

* When you want to avoid a permanent binding between abstraction and implementation.
* When both abstractions and implementations should be extendable independently.

**Java Example:**

// Implementor

interface Device {

void turnOn();

void turnOff();

void setVolume(int level);

}

// Concrete Implementors

class TV implements Device {

public void turnOn() { System.out.println("TV is ON"); }

public void turnOff() { System.out.println("TV is OFF"); }

public void setVolume(int level) { System.out.println("TV volume set to " + level); }

}

class Radio implements Device {

public void turnOn() { System.out.println("Radio is ON"); }

public void turnOff() { System.out.println("Radio is OFF"); }

public void setVolume(int level) { System.out.println("Radio volume set to " + level); }

}

// Abstraction

abstract class RemoteControl {

protected Device device;

public RemoteControl(Device device) {

this.device = device;

}

public abstract void togglePower();

public abstract void volumeUp();

public abstract void volumeDown();

}

// Refined Abstraction

class AdvancedRemote extends RemoteControl {

private boolean power = false;

private int volume = 0;

public AdvancedRemote(Device device) {

super(device);

}

public void togglePower() {

if (power) {

device.turnOff();

} else {

device.turnOn();

}

power = !power;

}

public void volumeUp() {

volume++;

device.setVolume(volume);

}

public void volumeDown() {

volume--;

device.setVolume(volume);

}

}

// Demo

public class BridgePatternDemo {

public static void main(String[] args) {

Device tv = new TV();

RemoteControl remote = new AdvancedRemote(tv);

remote.togglePower();

remote.volumeUp();

remote.volumeDown();

Device radio = new Radio();

remote = new AdvancedRemote(radio);

remote.togglePower();

remote.volumeUp();

}

}

## **2.3. Composite Pattern**

**Intent:**  
Compose objects into **tree structures** to represent part-whole hierarchies.  
Composite lets clients treat individual objects and compositions of objects **uniformly**.

**Real-world analogy:**  
Think of a **file system**:

* A file is a leaf (cannot contain other files).
* A folder can contain files or other folders.
* You interact with both files and folders in a similar way (e.g., open, delete).

**When to use:**

* When you have to represent a hierarchy of objects.
* When you want clients to treat individual objects and groups of objects in the same way.

**Java Example:**

import java.util.\*;

// Component

interface Employee {

void showDetails();

}

// Leaf

class Developer implements Employee {

private String name;

private String role;

public Developer(String name, String role) {

this.name = name;

this.role = role;

}

public void showDetails() {

System.out.println("Developer: " + name + ", Role: " + role);

}

}

// Leaf

class Designer implements Employee {

private String name;

private String role;

public Designer(String name, String role) {

this.name = name;

this.role = role;

}

public void showDetails() {

System.out.println("Designer: " + name + ", Role: " + role);

}

}

// Composite

class Manager implements Employee {

private String name;

private List<Employee> subordinates = new ArrayList<>();

public Manager(String name) {

this.name = name;

}

public void addEmployee(Employee e) {

subordinates.add(e);

}

public void removeEmployee(Employee e) {

subordinates.remove(e);

}

public void showDetails() {

System.out.println("Manager: " + name);

for (Employee e : subordinates) {

e.showDetails();

}

}

}

// Demo

public class CompositePatternDemo {

public static void main(String[] args) {

Employee dev1 = new Developer("Alice", "Backend Dev");

Employee dev2 = new Developer("Bob", "Frontend Dev");

Employee designer = new Designer("Charlie", "UI/UX Designer");

Manager manager = new Manager("David");

manager.addEmployee(dev1);

manager.addEmployee(dev2);

manager.addEmployee(designer);

manager.showDetails(); // prints hierarchy

}

}

👉 This allows you to handle **individual employees** and **teams of employees** the same way.

## **2.4. Decorator Pattern**

**Intent:**  
Attach additional responsibilities to an object **dynamically**.  
It provides a flexible alternative to subclassing for extending functionality.

**Real-world analogy:**  
Think of a **coffee order system**:

* You start with a plain coffee.
* You can add milk, sugar, or whipped cream as decorators.
* You don’t create hundreds of coffee subclasses (CoffeeWithMilk, CoffeeWithSugar, etc.) — you just **wrap** the base object dynamically.

**When to use:**

* To add behavior to individual objects, without affecting other objects of the same class.
* When subclassing would create too many classes.

**Java Example:**

// Component

interface Coffee {

String getDescription();

double getCost();

}

// Concrete Component

class SimpleCoffee implements Coffee {

public String getDescription() {

return "Simple Coffee";

}

public double getCost() {

return 5.0;

}

}

// Decorator

abstract class CoffeeDecorator implements Coffee {

protected Coffee decoratedCoffee;

public CoffeeDecorator(Coffee coffee) {

this.decoratedCoffee = coffee;

}

public String getDescription() {

return decoratedCoffee.getDescription();

}

public double getCost() {

return decoratedCoffee.getCost();

}

}

// Concrete Decorators

class MilkDecorator extends CoffeeDecorator {

public MilkDecorator(Coffee coffee) {

super(coffee);

}

public String getDescription() {

return decoratedCoffee.getDescription() + ", Milk";

}

public double getCost() {

return decoratedCoffee.getCost() + 1.5;

}

}

class SugarDecorator extends CoffeeDecorator {

public SugarDecorator(Coffee coffee) {

super(coffee);

}

public String getDescription() {

return decoratedCoffee.getDescription() + ", Sugar";

}

public double getCost() {

return decoratedCoffee.getCost() + 0.5;

}

}

// Demo

public class DecoratorPatternDemo {

public static void main(String[] args) {

Coffee coffee = new SimpleCoffee();

System.out.println(coffee.getDescription() + " -> $" + coffee.getCost());

coffee = new MilkDecorator(coffee);

System.out.println(coffee.getDescription() + " -> $" + coffee.getCost());

coffee = new SugarDecorator(coffee);

System.out.println(coffee.getDescription() + " -> $" + coffee.getCost());

}

}

👉 This way, you can **dynamically layer functionality** (add milk, sugar, etc.) without subclass explosion.

## **2.5. Facade Pattern**

**Intent:**  
Provide a **unified, simplified interface** to a set of interfaces in a subsystem.  
The Facade defines a higher-level interface that makes the subsystem easier to use.

**Real-world analogy:**  
Think of a **hotel receptionist**:

* Instead of directly interacting with housekeeping, room service, and maintenance, you just call the receptionist.
* The receptionist (facade) delegates your request to the right department.

**When to use:**

* When you want to simplify a complex subsystem.
* To provide a single entry point for a subsystem.

**Java Example:**

// Subsystems

class CPU {

void start() { System.out.println("CPU started"); }

void execute() { System.out.println("CPU executing instructions"); }

void shutdown() { System.out.println("CPU shutdown"); }

}

class Memory {

void load() { System.out.println("Memory loaded"); }

void clear() { System.out.println("Memory cleared"); }

}

class HardDrive {

void read() { System.out.println("Reading data from HardDrive"); }

void write() { System.out.println("Writing data to HardDrive"); }

}

// Facade

class ComputerFacade {

private CPU cpu;

private Memory memory;

private HardDrive hardDrive;

public ComputerFacade() {

this.cpu = new CPU();

this.memory = new Memory();

this.hardDrive = new HardDrive();

}

public void startComputer() {

System.out.println("Starting computer...");

cpu.start();

memory.load();

hardDrive.read();

cpu.execute();

}

public void shutDownComputer() {

System.out.println("Shutting down computer...");

cpu.shutdown();

memory.clear();

hardDrive.write();

}

}

// Demo

public class FacadePatternDemo {

public static void main(String[] args) {

ComputerFacade computer = new ComputerFacade();

computer.startComputer();

computer.shutDownComputer();

}

}

👉 Clients only use the **facade** (ComputerFacade) instead of dealing with CPU, Memory, and HardDrive directly.

## **2.6. Flyweight Pattern**

**Intent:**  
Minimize memory usage by **sharing objects** that are similar, instead of creating new ones for every use.

**Real-world analogy:**  
Think of a **text editor**:

* You may have thousands of characters in a document.
* Instead of storing a separate font object for each character, you store one font object and reuse it for all instances.

**When to use:**

* When you need to create a large number of similar objects.
* When memory cost is high due to object duplication.

**Java Example:**

import java.util.HashMap;

import java.util.Map;

// Flyweight

interface Shape {

void draw(int x, int y);

}

// Concrete Flyweight

class Circle implements Shape {

private String color; // intrinsic property (shared)

public Circle(String color) {

this.color = color;

}

public void draw(int x, int y) {

System.out.println("Drawing " + color + " circle at (" + x + "," + y + ")");

}

}

// Flyweight Factory

class ShapeFactory {

private static final Map<String, Shape> circleMap = new HashMap<>();

public static Shape getCircle(String color) {

Circle circle = (Circle) circleMap.get(color);

if (circle == null) {

circle = new Circle(color);

circleMap.put(color, circle);

System.out.println("Creating a new " + color + " circle");

}

return circle;

}

}

// Demo

public class FlyweightPatternDemo {

public static void main(String[] args) {

Shape redCircle1 = ShapeFactory.getCircle("Red");

redCircle1.draw(10, 20);

Shape redCircle2 = ShapeFactory.getCircle("Red");

redCircle2.draw(30, 40);

Shape blueCircle = ShapeFactory.getCircle("Blue");

blueCircle.draw(50, 60);

// Notice how redCircle1 and redCircle2 share the same object

}

}

👉 Instead of creating multiple Circle objects for the same color, we **reuse** them.

## **2.7. Proxy Pattern**

**Intent:**  
Provide a **surrogate or placeholder** for another object to control access to it.

**Real-world analogy:**  
Think of a **credit card**:

* It acts as a proxy to your bank account.
* Instead of directly accessing your bank balance (real object), you use a card (proxy) which adds control and security.

**Types of Proxy:**

* **Virtual Proxy** → Lazy initialization, e.g., loading images only when needed.
* **Protection Proxy** → Controls access (like permissions).
* **Remote Proxy** → Represent objects in different address spaces (like RMI).

**Java Example (Virtual Proxy):**

// Subject

interface Image {

void display();

}

// RealSubject

class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading " + fileName);

}

public void display() {

System.out.println("Displaying " + fileName);

}

}

// Proxy

class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName); // Lazy loading

}

realImage.display();

}

}

// Demo

public class ProxyPatternDemo {

public static void main(String[] args) {

Image image = new ProxyImage("photo.jpg");

// Image loaded only once

image.display();

image.display();

}

}

👉 The **proxy** (ProxyImage) loads the real object (RealImage) only when needed.

✅ Now we’ve covered all **7 Structural Patterns** in detail:

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Facade
6. Flyweight
7. Proxy

# ****Behavioral Patterns in Java****

These patterns are about **communication between objects**:

* How responsibilities are distributed.
* How objects interact without being tightly coupled.

The goal:

* Increase **flexibility in communication**.
* Make code **maintainable, reusable, and extensible**.

Behavioral Patterns in Java:

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template Method
11. Visitor

# ****3.1. Chain of Responsibility Pattern****

### **Intent**

* Allow multiple objects the chance to handle a request, without coupling the sender to a specific receiver.
* Each handler decides:
  + **Handle the request**, OR
  + **Pass it along** the chain.

### **Real-world analogy**

Think of **customer support escalation**:

* Level 1 support → Level 2 support → Level 3 support.
* If Level 1 can’t solve it, they pass it to the next level.

### **When to use**

* You want to **decouple request senders and receivers**.
* Multiple handlers could process the same request.
* You don’t want to hardcode which handler should handle what.

## **Java Example – Chain of Responsibility**

// Handler interface

abstract class Logger {

public static int INFO = 1;

public static int DEBUG = 2;

public static int ERROR = 3;

protected int level;

protected Logger nextLogger;

public void setNextLogger(Logger nextLogger) {

this.nextLogger = nextLogger;

}

public void logMessage(int level, String message) {

if (this.level <= level) {

write(message);

}

if (nextLogger != null) {

nextLogger.logMessage(level, message);

}

}

protected abstract void write(String message);

}

// Concrete Handlers

class ConsoleLogger extends Logger {

public ConsoleLogger(int level) {

this.level = level;

}

@Override

protected void write(String message) {

System.out.println("Console Logger: " + message);

}

}

class FileLogger extends Logger {

public FileLogger(int level) {

this.level = level;

}

@Override

protected void write(String message) {

System.out.println("File Logger: " + message);

}

}

class ErrorLogger extends Logger {

public ErrorLogger(int level) {

this.level = level;

}

@Override

protected void write(String message) {

System.out.println("Error Logger: " + message);

}

}

// Demo

public class ChainOfResponsibilityDemo {

private static Logger getChainOfLoggers() {

Logger errorLogger = new ErrorLogger(Logger.ERROR);

Logger fileLogger = new FileLogger(Logger.DEBUG);

Logger consoleLogger = new ConsoleLogger(Logger.INFO);

// Build the chain: Error -> File -> Console

errorLogger.setNextLogger(fileLogger);

fileLogger.setNextLogger(consoleLogger);

return errorLogger;

}

public static void main(String[] args) {

Logger loggerChain = getChainOfLoggers();

loggerChain.logMessage(Logger.INFO, "This is an information.");

loggerChain.logMessage(Logger.DEBUG, "This is a debug message.");

loggerChain.logMessage(Logger.ERROR, "This is an error message.");

}

}

### **Explanation of the Code**

1. **Logger (abstract class)**
   * Defines the **interface** for handling requests.
   * Contains reference to the **next handler** (nextLogger).
   * Method logMessage() checks if the current logger should handle the message, then forwards to the next handler.
2. **Concrete Handlers (ConsoleLogger, FileLogger, ErrorLogger)**
   * Each one knows how to handle messages of its own level.
   * Example:
     + ErrorLogger handles errors.
     + FileLogger handles debug.
     + ConsoleLogger handles info.
3. **Chain Building (getChainOfLoggers)**
   * We link handlers: **Error → File → Console**.
   * So requests go down the chain until someone handles it.
4. **Execution**
   * INFO → handled by ConsoleLogger.
   * DEBUG → handled by FileLogger (then ConsoleLogger).
   * ERROR → handled by ErrorLogger (then passed down).

✅ This pattern makes logging extensible:

* Tomorrow you add a DatabaseLogger → just link it in the chain.
* No need to change client code.

# ****3.2. Command Pattern****

### **Intent**

* Encapsulate a **request** as an object, so you can:
  + Parameterize clients with different requests.
  + Queue, log, or undo operations.

### **Real-world analogy**

Think of a **TV remote control**:

* Each button (command) can be mapped to an action (turn on/off, change channel).
* The remote (invoker) doesn’t know the details of how the TV works, it just triggers commands.

### **When to use**

* You want to **decouple sender and receiver** of requests.
* You need **undo/redo functionality**.
* You want to **queue or log operations**.

## **Java Example – Command Pattern**

// Command interface

interface Command {

void execute();

}

// Receiver

class Light {

public void turnOn() {

System.out.println("Light is ON");

}

public void turnOff() {

System.out.println("Light is OFF");

}

}

// Concrete Commands

class TurnOnLightCommand implements Command {

private Light light;

public TurnOnLightCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

class TurnOffLightCommand implements Command {

private Light light;

public TurnOffLightCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

// Invoker

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

// Demo

public class CommandPatternDemo {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightsOn = new TurnOnLightCommand(livingRoomLight);

Command lightsOff = new TurnOffLightCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

// Turn the light on

remote.setCommand(lightsOn);

remote.pressButton();

// Turn the light off

remote.setCommand(lightsOff);

remote.pressButton();

}

}

### **Explanation of the Code**

1. **Command interface**
   * Declares a single method execute().
   * All commands must implement this.
2. **Light (Receiver)**
   * The actual object that knows how to perform operations (turn on/off).
3. **Concrete Commands (TurnOnLightCommand, TurnOffLightCommand)**
   * Wrap specific actions of the receiver inside a command.
   * Example: TurnOnLightCommand calls light.turnOn().
4. **RemoteControl (Invoker)**
   * Doesn’t know the details of what the command does.
   * It just calls command.execute().
   * Allows dynamic assignment of commands.
5. **Execution**
   * Client creates Light and corresponding commands.
   * Remote is configured with a command → pressButton() executes it.
   * Easily switch behavior by setting a different command.

✅ **Why it’s powerful**

* You can add more commands (e.g., DimLightCommand, ChangeColorCommand) without changing RemoteControl.
* Commands can be **stored in a list**, **executed later**, or even **undone**.

# ****3.3. Interpreter Pattern****

### **Intent**

* Define a **grammar** for a language and provide an **interpreter** to evaluate sentences in that language.
* Each rule in the grammar is represented by a **class**.

### **Real-world analogy**

Think of a **calculator** that interprets math expressions:

* Expression: 5 AND 3
* Interpreter translates it into logic and evaluates it.

Another example: SQL engines interpret SQL queries into execution steps.

### **When to use**

* When you have a **simple language/grammar** to interpret.
* When expressions need to be evaluated repeatedly (e.g., rules, filters).
* Not good for complex grammars → use real parsers instead.

## **Java Example – Interpreter Pattern**

// Expression interface

interface Expression {

boolean interpret(String context);

}

// Terminal Expression

class TerminalExpression implements Expression {

private String data;

public TerminalExpression(String data) {

this.data = data;

}

@Override

public boolean interpret(String context) {

return context.contains(data);

}

}

// Or Expression

class OrExpression implements Expression {

private Expression expr1;

private Expression expr2;

public OrExpression(Expression expr1, Expression expr2) {

this.expr1 = expr1;

this.expr2 = expr2;

}

@Override

public boolean interpret(String context) {

return expr1.interpret(context) || expr2.interpret(context);

}

}

// And Expression

class AndExpression implements Expression {

private Expression expr1;

private Expression expr2;

public AndExpression(Expression expr1, Expression expr2) {

this.expr1 = expr1;

this.expr2 = expr2;

}

@Override

public boolean interpret(String context) {

return expr1.interpret(context) && expr2.interpret(context);

}

}

// Demo

public class InterpreterPatternDemo {

// Rule: "John" AND "Married"

public static Expression getMarriedJohnExpression() {

Expression john = new TerminalExpression("John");

Expression married = new TerminalExpression("Married");

return new AndExpression(john, married);

}

// Rule: "Robert" OR "John"

public static Expression getMaleExpression() {

Expression robert = new TerminalExpression("Robert");

Expression john = new TerminalExpression("John");

return new OrExpression(robert, john);

}

public static void main(String[] args) {

Expression isMale = getMaleExpression();

Expression isMarriedJohn = getMarriedJohnExpression();

System.out.println("John is male? " + isMale.interpret("John"));

System.out.println("Robert is male? " + isMale.interpret("Robert"));

System.out.println("Married John? " + isMarriedJohn.interpret("Married John"));

System.out.println("Single John? " + isMarriedJohn.interpret("Single John"));

}

}

### **Explanation of the Code**

1. **Expression interface**
   * Defines the interpret(String context) method.
   * Every grammar rule must implement this.
2. **TerminalExpression**
   * Represents leaf nodes in grammar (like words John, Married).
   * Checks if the context string contains that word.
3. **OrExpression & AndExpression**
   * Non-terminal expressions.
   * Combine other expressions using OR/AND logic.
4. **Demo Setup**
   * getMaleExpression() = Robert OR John.
   * getMarriedJohnExpression() = John AND Married.
5. **Execution**
   * "John" → true for male.
   * "Robert" → true for male.
   * "Married John" → true for married John rule.
   * "Single John" → false because “Married” is missing.

✅ **Why it’s useful**

* You can build complex rules dynamically using a **tree of expressions**.
* Useful in scenarios like:
  + Rule engines (e.g., eligibility rules).
  + Simple DSLs (domain-specific languages).
  + Text parsers.

# ****3.4. Iterator Pattern****

### **Intent**

* Provide a way to **access elements of a collection sequentially** without exposing its internal structure.
* Lets you traverse a collection **uniformly**, whether it’s an array, list, or custom collection.

### **Real-world analogy**

Think of a **TV remote’s channel button**:

* You can go to the next channel without knowing how the TV internally stores channels.

### **When to use**

* When you want to **traverse a collection** without exposing how it’s stored.
* When you want to provide **different ways of iterating** (forward, backward, filtered).

## **Java Example – Iterator Pattern**

// Iterator interface

interface Iterator<T> {

boolean hasNext();

T next();

}

// Aggregate interface

interface Container<T> {

Iterator<T> getIterator();

}

// Concrete Aggregate

class NameRepository implements Container<String> {

private String[] names = {"Alice", "Bob", "Charlie", "David"};

@Override

public Iterator<String> getIterator() {

return new NameIterator();

}

// Concrete Iterator

private class NameIterator implements Iterator<String> {

int index = 0;

@Override

public boolean hasNext() {

return index < names.length;

}

@Override

public String next() {

if (hasNext()) {

return names[index++];

}

return null;

}

}

}

// Demo

public class IteratorPatternDemo {

public static void main(String[] args) {

NameRepository nameRepository = new NameRepository();

Iterator<String> iterator = nameRepository.getIterator();

while (iterator.hasNext()) {

System.out.println("Name: " + iterator.next());

}

}

}

### **Explanation of the Code**

1. **Iterator<T> interface**
   * Defines hasNext() to check if more elements exist.
   * Defines next() to get the next element.
2. **Container<T> interface**
   * Declares getIterator() so clients can get an iterator without knowing the internal structure.
3. **NameRepository (Concrete Aggregate)**
   * Stores the actual collection (names[]).
   * Returns an instance of its **inner iterator**.
4. **NameIterator (Concrete Iterator)**
   * Maintains the **current index**.
   * hasNext() checks if index < array length.
   * next() returns the current element and increments the index.
5. **Execution Flow**
   * Client calls getIterator() → gets NameIterator.
   * Calls hasNext() → true if there are more elements.
   * Calls next() → retrieves elements sequentially.
   * Iteration stops when hasNext() returns false.

✅ **Why it’s useful**

* The client doesn’t know how the data is stored (array, list, tree).
* You can create multiple iterators for the same collection (e.g., forward, backward).
* Promotes **low coupling** between collection and client.

# ****3.5. Mediator Pattern****

### **Intent**

* Define an object (mediator) that **encapsulates how a set of objects interact**.
* Promotes **loose coupling** by preventing objects from referring to each other directly.

### **Real-world analogy**

Think of an **air traffic control tower**:

* Planes (colleagues) don’t communicate directly with each other.
* The control tower (mediator) coordinates takeoffs, landings, and movements.

### **When to use**

* When many objects interact in complex ways.
* When you want to **centralize communication** to reduce dependencies between objects.

## **Java Example – Mediator Pattern**

// Mediator interface

interface ChatMediator {

void sendMessage(String message, User user);

void addUser(User user);

}

// Concrete Mediator

class ChatMediatorImpl implements ChatMediator {

private List<User> users;

public ChatMediatorImpl() {

this.users = new ArrayList<>();

}

@Override

public void addUser(User user) {

users.add(user);

}

@Override

public void sendMessage(String message, User sender) {

for (User user : users) {

// Message should not be received by the sender

if (user != sender) {

user.receive(message);

}

}

}

}

// Colleague

abstract class User {

protected ChatMediator mediator;

protected String name;

public User(ChatMediator mediator, String name) {

this.mediator = mediator;

this.name = name;

}

public abstract void send(String message);

public abstract void receive(String message);

}

// Concrete Colleague

class ChatUser extends User {

public ChatUser(ChatMediator mediator, String name) {

super(mediator, name);

}

@Override

public void send(String message) {

System.out.println(this.name + " sends: " + message);

mediator.sendMessage(message, this);

}

@Override

public void receive(String message) {

System.out.println(this.name + " receives: " + message);

}

}

// Demo

public class MediatorPatternDemo {

public static void main(String[] args) {

ChatMediator mediator = new ChatMediatorImpl();

User alice = new ChatUser(mediator, "Alice");

User bob = new ChatUser(mediator, "Bob");

User charlie = new ChatUser(mediator, "Charlie");

mediator.addUser(alice);

mediator.addUser(bob);

mediator.addUser(charlie);

alice.send("Hi everyone!");

bob.send("Hello Alice!");

}

}

### **Explanation of the Code**

1. **ChatMediator interface**
   * Defines sendMessage() to relay messages.
   * Defines addUser() to register participants.
2. **ChatMediatorImpl (Concrete Mediator)**
   * Maintains a list of users.
   * When sendMessage() is called, it forwards the message to all users **except the sender**.
3. **User (Colleague)**
   * Abstract class for chat participants.
   * Knows its mediator but not other users.
4. **ChatUser (Concrete Colleague)**
   * Implements send() → passes message to mediator.
   * Implements receive() → handles incoming message.
5. **Execution Flow**
   * Alice.send("Hi everyone!") → mediator receives message.
   * Mediator iterates over users → sends to Bob & Charlie.
   * Bob & Charlie call their receive() method to display the message.

✅ **Why it’s useful**

* Reduces **tight coupling**: users don’t communicate directly.
* Adding/removing participants is easy — no changes in other classes.
* Centralizes control → easier to maintain complex interactions.

## **3.6. Memento Pattern**

### **Intent**

* Capture an object’s **internal state** so it can be restored later, without exposing its internals.
* Useful for **undo/redo** or **rollback** functionality.

### **Real-world analogy**

Think of a **text editor**:

* You type something → save snapshot.
* Later you can undo → restore previous snapshot.

### **When to use**

* You need **undo/redo** support.
* You want to **save object state** without breaking encapsulation.

### **Java Example – Memento Pattern**

// Memento

class Memento {

private String state;

public Memento(String state) {

this.state = state;

}

public String getState() {

return state;

}

}

// Originator

class Originator {

private String state;

public void setState(String state) {

System.out.println("Setting state to: " + state);

this.state = state;

}

public String getState() {

return state;

}

public Memento saveStateToMemento() {

return new Memento(state);

}

public void restoreStateFromMemento(Memento memento) {

this.state = memento.getState();

System.out.println("Restored state to: " + state);

}

}

// Caretaker

class Caretaker {

private List<Memento> mementoList = new ArrayList<>();

public void add(Memento state) {

mementoList.add(state);

}

public Memento get(int index) {

return mementoList.get(index);

}

}

// Demo

public class MementoPatternDemo {

public static void main(String[] args) {

Originator originator = new Originator();

Caretaker caretaker = new Caretaker();

originator.setState("State #1");

originator.setState("State #2");

caretaker.add(originator.saveStateToMemento()); // Save snapshot

originator.setState("State #3");

caretaker.add(originator.saveStateToMemento()); // Save snapshot

originator.setState("State #4");

System.out.println("Current state: " + originator.getState());

// Undo to previous states

originator.restoreStateFromMemento(caretaker.get(0));

originator.restoreStateFromMemento(caretaker.get(1));

}

}

### **Explanation of the Code**

1. **Memento**
   * Stores the **internal state** (here just a String).
   * Only the originator can access and create mementos.
2. **Originator**
   * The object whose state we want to save/restore.
   * saveStateToMemento() → creates a snapshot.
   * restoreStateFromMemento() → loads a snapshot.
3. **Caretaker**
   * Holds the mementos but doesn’t modify their content.
   * Basically, an **undo history manager**.
4. **Execution flow**
   * Originator sets states: "State #1" → "State #2" → save snapshot.
   * Changes to "State #3" → save snapshot.
   * Changes to "State #4".
   * Restore snapshots to earlier states via caretaker.

✅ Output will show how we **rollback to previous states**.

### **Why it’s useful**

* Keeps **state saving** separate from the object’s logic.
* Perfect for **undo/redo** (text editors, games, IDEs).
* Protects encapsulation (no direct access to fields).

## **3.7. Observer Pattern**

### **Intent**

* Define a **one-to-many dependency** so that when one object changes state, all its dependents are **notified and updated automatically**.
* Also called **Publish–Subscribe (Pub–Sub)** pattern.

### **Real-world analogy**

* **YouTube channel** (Subject):
  + Viewers (Observers) subscribe.
  + When the channel uploads a video, all subscribers get notified.

### **When to use**

* You have an **event source** and multiple **listeners**.
* You want to **decouple** the object that changes from the objects that react.

### **Java Example – Observer Pattern**

import java.util.\*;

// Observer

interface Observer {

void update(String message);

}

// Concrete Observer

class Subscriber implements Observer {

private String name;

public Subscriber(String name) {

this.name = name;

}

@Override

public void update(String message) {

System.out.println(name + " received update: " + message);

}

}

// Subject

interface Subject {

void subscribe(Observer observer);

void unsubscribe(Observer observer);

void notifyObservers(String message);

}

// Concrete Subject

class YouTubeChannel implements Subject {

private List<Observer> observers = new ArrayList<>();

private String channelName;

public YouTubeChannel(String channelName) {

this.channelName = channelName;

}

@Override

public void subscribe(Observer observer) {

observers.add(observer);

}

@Override

public void unsubscribe(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers(String message) {

for (Observer obs : observers) {

obs.update(channelName + ": " + message);

}

}

public void uploadVideo(String videoTitle) {

System.out.println(channelName + " uploaded: " + videoTitle);

notifyObservers("New Video - " + videoTitle);

}

}

// Demo

public class ObserverPatternDemo {

public static void main(String[] args) {

YouTubeChannel techChannel = new YouTubeChannel("TechWorld");

Observer user1 = new Subscriber("Alice");

Observer user2 = new Subscriber("Bob");

Observer user3 = new Subscriber("Charlie");

techChannel.subscribe(user1);

techChannel.subscribe(user2);

techChannel.subscribe(user3);

techChannel.uploadVideo("Observer Pattern Explained");

System.out.println("--- Bob unsubscribes ---");

techChannel.unsubscribe(user2);

techChannel.uploadVideo("State Pattern Tutorial");

}

}

### **Explanation**

1. **Observer interface** → defines the update() method each subscriber implements.
2. **Subscriber** → concrete observer, prints updates.
3. **Subject interface** → defines how to add/remove observers and send updates.
4. **YouTubeChannel** → maintains a list of subscribers and notifies them when a new video is uploaded.
5. **Flow:**
   * We subscribe Alice, Bob, Charlie.
   * uploadVideo() calls notifyObservers().
   * Each observer’s update() method is called.
   * Bob unsubscribes; the next upload notifies only Alice and Charlie.

✅ **Output shows how notifications flow to all current subscribers.**

### **Where you see it in Java**

* java.util.Observer (legacy).
* EventListener in Swing/JavaFX.
* Reactive frameworks (RxJava, Spring Event Listeners).

## **3.8. State Pattern**

### **Intent**

* Allow an object to **change its behavior** when its internal **state changes**.
* Looks like the object **changed its class** at runtime.

### **Real-world analogy**

* A **Traffic light**:
  + Green → cars move.
  + Yellow → cars slow down.
  + Red → cars stop.
  + Same object, but behavior changes based on **current state**.

### **When to use**

* If you have many if/else or switch statements checking an object’s state.
* You want **cleaner, more maintainable state transitions**.

### **Java Example – State Pattern**

// State interface

interface State {

void handle();

}

// Concrete States

class GreenLight implements State {

@Override

public void handle() {

System.out.println("Green Light: Cars can go!");

}

}

class YellowLight implements State {

@Override

public void handle() {

System.out.println("Yellow Light: Cars should slow down!");

}

}

class RedLight implements State {

@Override

public void handle() {

System.out.println("Red Light: Cars must stop!");

}

}

// Context

class TrafficLight {

private State currentState;

public void setState(State state) {

this.currentState = state;

}

public void request() {

currentState.handle();

}

}

// Demo

public class StatePatternDemo {

public static void main(String[] args) {

TrafficLight light = new TrafficLight();

light.setState(new GreenLight());

light.request();

light.setState(new YellowLight());

light.request();

light.setState(new RedLight());

light.request();

}

}

### **Explanation**

1. **State interface** → defines handle() method that represents behavior.
2. **Concrete States (GreenLight, YellowLight, RedLight)** → each has its own implementation of handle().
3. **TrafficLight (Context)** → holds a reference to the current State.
   * setState() changes which behavior is active.
   * request() delegates to the current state’s handle().
4. **Flow:**
   * Set state to GreenLight → handle() prints “Cars can go!”.
   * Change to YellowLight → “Cars slow down!”.
   * Change to RedLight → “Cars must stop!”.

✅ No giant if/else — each state knows what to do.

### **Where you see it**

* Game character modes (walking, running, jumping).
* Media player states (Playing, Paused, Stopped).
* Workflow engines.

## **3.9. Strategy Pattern**

### **Intent**

* Define a **family of algorithms**, encapsulate each one, and make them **interchangeable**.
* Lets the algorithm vary independently from the clients that use it.

### **Real-world analogy**

* **Payment in an e-commerce site**:
  + You can pay by **Credit Card**, **PayPal**, **UPI**, etc.
  + The checkout process doesn’t care which payment method you use — it just asks for a **strategy**.

### **When to use**

* You have multiple ways to do something (e.g., sorting, payment, compression).
* You want to **switch algorithms at runtime**.
* Avoid huge if/else blocks deciding which method to call.

### **Java Example – Strategy Pattern**

// Strategy interface

interface PaymentStrategy {

void pay(int amount);

}

// Concrete strategies

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String name;

public CreditCardPayment(String name, String cardNumber) {

this.name = name;

this.cardNumber = cardNumber;

}

@Override

public void pay(int amount) {

System.out.println(amount + " paid with Credit Card (" + cardNumber + ")");

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(int amount) {

System.out.println(amount + " paid using PayPal account: " + email);

}

}

class UpiPayment implements PaymentStrategy {

private String upiId;

public UpiPayment(String upiId) {

this.upiId = upiId;

}

@Override

public void pay(int amount) {

System.out.println(amount + " paid using UPI ID: " + upiId);

}

}

// Context

class ShoppingCart {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void checkout(int amount) {

if (paymentStrategy == null) {

System.out.println("No payment method selected!");

return;

}

paymentStrategy.pay(amount);

}

}

// Demo

public class StrategyPatternDemo {

public static void main(String[] args) {

ShoppingCart cart = new ShoppingCart();

// Customer chooses to pay by credit card

cart.setPaymentStrategy(new CreditCardPayment("Alice", "1234-5678-9999"));

cart.checkout(2500);

// Customer switches to PayPal

cart.setPaymentStrategy(new PayPalPayment("alice@example.com"));

cart.checkout(1500);

// Customer switches to UPI

cart.setPaymentStrategy(new UpiPayment("alice@upi"));

cart.checkout(500);

}

}

### **Explanation**

1. **PaymentStrategy** → common interface for all payment methods.
2. **Concrete strategies** (CreditCardPayment, PayPalPayment, UpiPayment)
   * Each implements pay() in its own way.
3. **ShoppingCart (Context)**
   * Holds a reference to a PaymentStrategy.
   * Calls the selected strategy’s pay() during checkout.
4. **Flow:**
   * At runtime, we set the desired strategy (credit card → PayPal → UPI).
   * checkout() calls pay() on whichever strategy is set.
   * We can swap algorithms without touching ShoppingCart.

✅ Output shows payment method changing dynamically.

### **Why it’s useful**

* Eliminates if-else/switch clutter when choosing algorithms.
* Open/Closed Principle — new strategies can be added without changing the existing code.
* Makes testing easier (test each strategy separately).

### **Where you see it**

* Sorting (Comparator in Java is a strategy).
* Payment gateways.
* Compression libraries (ZIP, RAR, GZIP).
* Spring’s ResourceLoader or ViewResolver strategies.

## **3.10. Template Method Pattern**

### **Intent**

* Define the **skeleton** of an algorithm in a base class, but let **subclasses** redefine certain steps.
* This lets you keep the overall process the same while allowing customization of specific parts.

### **Real-world analogy**

* Think of **making tea or coffee**:
  1. Boil water
  2. Brew beverage (tea leaves / coffee)
  3. Pour in cup
  4. Add condiments
* Steps 1, 3 are fixed, but brewing and adding condiments differ.

### **When to use**

* You have an algorithm with a fixed structure but **steps that can change**.
* You want to **enforce a process** but allow subclasses to fill in details.

### **Java Example – Template Method Pattern**

// Abstract class with template method

abstract class BeverageMaker {

// Template method - final so subclasses can't change algorithm structure

public final void makeBeverage() {

boilWater();

brew();

pourInCup();

addCondiments();

}

private void boilWater() {

System.out.println("Boiling water...");

}

private void pourInCup() {

System.out.println("Pouring into cup...");

}

// Steps to be implemented by subclasses

protected abstract void brew();

protected abstract void addCondiments();

}

// Concrete class 1

class TeaMaker extends BeverageMaker {

@Override

protected void brew() {

System.out.println("Steeping the tea...");

}

@Override

protected void addCondiments() {

System.out.println("Adding lemon...");

}

}

// Concrete class 2

class CoffeeMaker extends BeverageMaker {

@Override

protected void brew() {

System.out.println("Dripping coffee through filter...");

}

@Override

protected void addCondiments() {

System.out.println("Adding sugar and milk...");

}

}

// Demo

public class TemplateMethodDemo {

public static void main(String[] args) {

System.out.println("Making Tea:");

BeverageMaker tea = new TeaMaker();

tea.makeBeverage();

System.out.println("\nMaking Coffee:");

BeverageMaker coffee = new CoffeeMaker();

coffee.makeBeverage();

}

}

### **Explanation**

1. **BeverageMaker** (abstract class)
   * Defines the **template method** makeBeverage() → it’s final so subclasses can’t alter the sequence.
   * Common steps (boilWater(), pourInCup()) are private & fixed.
   * Variable steps (brew(), addCondiments()) are abstract.
2. **Concrete classes** (TeaMaker, CoffeeMaker)
   * Each **implements the variable steps**.
3. **Flow:**
   * When tea.makeBeverage() is called:
     + boilWater() → brew() (tea) → pourInCup() → addCondiments() (lemon).
   * When coffee.makeBeverage() is called:
     + Same sequence but coffee-specific steps.

✅ The **overall recipe** is fixed; only details change.

### **Where you see it**

* **Java Servlet API**: HttpServlet.doGet() and doPost() (template is service()).
* **Spring Framework**: JdbcTemplate (fixed DB workflow, variable SQL parts).
* Build tools: Maven’s build lifecycle.

## **3.11. Visitor Pattern**

### **Intent**

* Represent an operation to be performed on elements of an object structure, **without changing the classes** of the elements.
* Useful when you have a **stable object structure** but want to **add new operations easily**.

### **Real-world analogy**

* A **tax auditor** visiting different types of companies:
  + Each company (Software, Retail) accepts the visitor.
  + The visitor performs a calculation depending on the company type.
* The companies don’t change; only the visitor adds behavior.

### **When to use**

* You have a **complex object structure** (e.g., AST, document, file system).
* You need to **add new operations frequently** without modifying element classes.

### **Java Example – Visitor Pattern**

// Visitor interface

interface Visitor {

void visit(Book book);

void visit(Fruit fruit);

}

// Concrete Visitor

class ShoppingCartVisitor implements Visitor {

@Override

public void visit(Book book) {

int cost = book.getPrice();

if (book.getPrice() > 500) {

cost -= 50; // discount

}

System.out.println("Book: " + book.getName() + " costs Rs." + cost);

}

@Override

public void visit(Fruit fruit) {

int cost = fruit.getPricePerKg() \* fruit.getWeight();

System.out.println(fruit.getName() + " costs Rs." + cost);

}

}

// Element interface

interface ItemElement {

void accept(Visitor visitor);

}

// Concrete Elements

class Book implements ItemElement {

private String name;

private int price;

public Book(String name, int price) {

this.name = name;

this.price = price;

}

public String getName() { return name; }

public int getPrice() { return price; }

@Override

public void accept(Visitor visitor) {

visitor.visit(this);

}

}

class Fruit implements ItemElement {

private String name;

private int pricePerKg;

private int weight;

public Fruit(String name, int pricePerKg, int weight) {

this.name = name;

this.pricePerKg = pricePerKg;

this.weight = weight;

}

public String getName() { return name; }

public int getPricePerKg() { return pricePerKg; }

public int getWeight() { return weight; }

@Override

public void accept(Visitor visitor) {

visitor.visit(this);

}

}

// Demo

public class VisitorPatternDemo {

public static void main(String[] args) {

ItemElement[] items = new ItemElement[] {

new Book("Design Patterns", 600),

new Book("Clean Code", 450),

new Fruit("Apple", 100, 2),

new Fruit("Banana", 60, 5)

};

Visitor shoppingCartVisitor = new ShoppingCartVisitor();

for (ItemElement item : items) {

item.accept(shoppingCartVisitor);

}

}

}

### **Explanation**

1. **Visitor interface** → declares visit() for each element type.
2. **ShoppingCartVisitor** → concrete visitor that calculates cost.
3. **ItemElement** → interface implemented by all elements.
4. **Book and Fruit** → concrete elements.
   * Implement accept() → pass themselves to the visitor.
5. **Flow:**
   * Each ItemElement calls visitor.visit(this) (double dispatch).
   * ShoppingCartVisitor runs correct logic based on the **actual type**.
   * We can add another visitor later (e.g., DiscountCalculatorVisitor) **without changing Book/Fruit**.

✅ **Decouples object structure from new operations**.

### **Where you see it**

* Compilers/Interpreters (AST traversal).
* Document processing (export to PDF/HTML).
* Tools like Eclipse/IntelliJ refactoring visitors.