**Exercise 1: Implementing the Singleton Pattern**

Main.java -> main () calss

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

logger1.log("User login successful.");

Logger logger2 = Logger.*getInstance*();

logger2.log("Data saved to database.");

// Verify both logger1 and logger2 refer to the same instance

System.***out***.println("Are both loggers the same? " + (logger1 == logger2));

}

}

Logger.java -> calss

public class Logger {

// Step 1: Create a private static instance

private static Logger *instance*;

// Step 2: Make constructor private

private Logger() {

System.***out***.println("Logger initialized.");

}

// Step 3: Provide a public static method to get the instance

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger(); // Lazy initialization

}

return *instance*;

}

// Example logging method

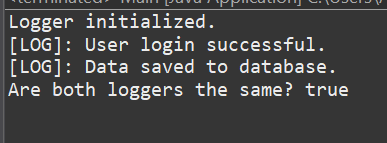
public void log(String message) {

System.***out***.println("[LOG]: " + message);

}

}

Output:



 **Private Static Instance Variable:**

* This holds the only instance of the Logger class.
* It is declared static so that it belongs to the class itself, not to any particular object.

 **Private Constructor:**

* Prevents external instantiation of the class using the new keyword.
* This ensures that no other class can create additional instances of Logger.

 **Public Static Access Method:**

* Provides a global access point to the singleton instance.
* This method checks if the instance already exists:
  + If **not**, it creates one.
  + If **yes**, it returns the existing instance.

**Exercise 2: Implementing the Factory Method Pattern**

WordDocument.java -> calss

public class WordDocument implements Document {

@Override

public void open() {

System.out.println("Opening a Word document.");

}

}

pdfDocument.java-> calss

public class PdfDocument implements Document {

@Override

public void open() {

System.out.println("Opening a PDF document.");

}

}

ExcelDocument.java

public class ExcelDocument implements Document {

@Override

public void open() {

System.out.println("Opening an Excel document.");

}

}

WordFactory.java

public class WordFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

PdfFactory.java

public class PdfFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

Main.java -> main() calss

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

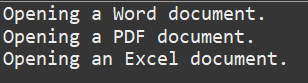
pdfDoc.open();

DocumentFactory excelFactory = new ExcelFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();}}

output:



**Document Management System** where users can create various types of documents—such as Word, PDF, and Excel documents. The goal is to use the **Factory Method Pattern** to decouple the client code from the specific document types, allowing easier scalability and maintenance.

**Exercise 3: Implementing the Builder Pattern**

**Computer.java -> calss**

public class Computer {

// Required attributes

private String CPU;

private String RAM;

// Optional attributes

private String storage;

private String graphicsCard;

private String keyboard;

private String monitor;

// ✅ Private constructor — only accessible from Builder

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.keyboard = builder.keyboard;

this.monitor = builder.monitor;

}

// ✅ Static nested Builder class

public static class Builder {

// Required attributes

private String CPU;

private String RAM;

// Optional attributes

private String storage;

private String graphicsCard;

private String keyboard;

private String monitor;

// Builder constructor for required fields

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder storage(String storage) {

this.storage = storage;

return this;

}

public Builder graphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder keyboard(String keyboard) {

this.keyboard = keyboard;

return this;

}

public Builder monitor(String monitor) {

this.monitor = monitor;

return this;

}

// 🔹 Build method returns the final Computer object

public Computer build() {

return new Computer(this);

}

}

*@Override*

public String toString() {

return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage +

", GraphicsCard=" + graphicsCard + ", Keyboard=" + keyboard + ", Monitor=" + monitor + "]";

}

}

Main.java –> main() calss

public class Main {

public static void main(String[] args) {

// 🔹 Basic configuration

Computer basicComputer = new Computer.Builder("Intel i3", "8GB")

.storage("256GB SSD")

.build();

// 🔹 Full configuration

Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")

.storage("2TB SSD")

.graphicsCard("NVIDIA RTX 4080")

.keyboard("Mechanical Keyboard")

.monitor("4K Monitor")

.build();

// 🔹 Print configurations

System.***out***.println("Basic Computer:");

System.***out***.println(basicComputer);

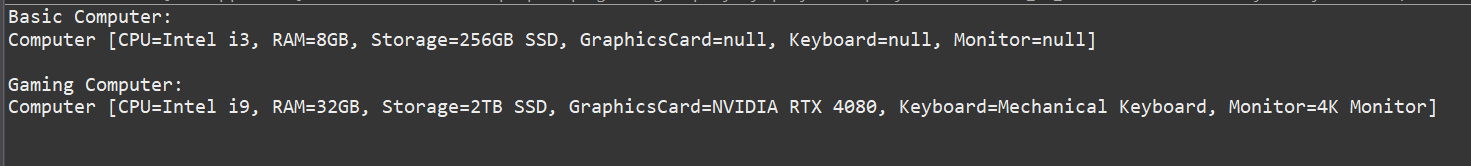
System.***out***.println("\nGaming Computer:");

System.***out***.println(gamingComputer);

}

}

Output:



**In the test:**

* Instantiate different Computer objects using the builder, such as:
  + A basic laptop with CPU and RAM only.
  + A gaming computer with CPU, RAM, GPU, and storage.
  + A developer workstation with all components configured.

This demonstrates the flexibility and clarity provided by the Builder pattern

 Configuring a computer, car, or form where many optional settings are possible.

 Used widely in frameworks (e.g., StringBuilder, Lombok @Builder, or UI builders).

**Exercise 4: Implementing the Adapter Pattern**

PaymentProcessor.java -> calss

public interface PaymentProcessor {

void processPayment(double amount);

}

StripeGateway.java -> calss

public class StripeGateway {

public void makeStripePayment(double amount) {

System.out.println("Payment of ₹" + amount + " processed via Stripe.");

}

}

PayPalGateway.java-> calss

public class PayPalGateway {

public void sendPayment(double amountInRupees) {

System.out.println("Payment of ₹" + amountInRupees + " processed via PayPal.");

}

}

PayPalAdapter.java-> calss

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter() {

paypal = new PayPalGateway();

}

@Override

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

Main.java -> mani() calss

public class Main {

public static void main(String[] args) {

// Use Stripe through adapter

PaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.processPayment(1500.00);

// Use PayPal through adapter

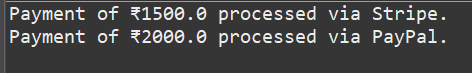
PaymentProcessor paypalProcessor = new PayPalAdapter();

paypalProcessor.processPayment(2000.00);

}

}

Output:

* **Interface Compatibility**: Bridges incompatible interfaces without modifying existing code.
* **Scalability**: Easily integrate new payment gateways by writing new adapter classes.
* **Decoupling**: Application code remains decoupled from third-party implementations.
* **Open/Closed Principle**: System is open to extension (new adapters) without changing existing code.

Integrating third-party APIs into a common platform

**Exercise 5: Implementing the Decorator Pattern**

Notifier.java -> component interface

public interface Notifier {

void send(String message);

}

EmailNotifier.java-> concorent component inetrface

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

System.out.println("Sending EMAIL: " + message);

}

}

NotifierDecorator.java -> Abstract decorator

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

@Override

public void send(String message) {

wrappee.send(message);

}

}

SMSNotifierDecorator.java -> **Decorator Implementation**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

System.out.println("Sending SMS: " + message);

}

}

Main.java-> mian() calss

public class Main {

public static void main(String[] args) {

// Base notifier: Email

Notifier notifier = new EmailNotifier();

// Add SMS functionality

notifier = new SMSNotifierDecorator(notifier);

// Add Slack functionality

notifier = new SlackNotifierDecorator(notifier);

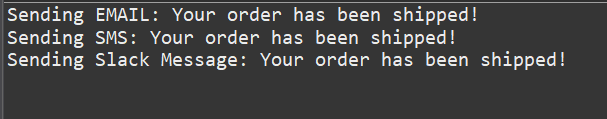
// Send notification via Email, SMS, and Slack

notifier.send("Your order has been shipped!");

}

}

Output:

* **Open/Closed Principle**: Add new notification types without modifying existing classes.
* **Flexible Composition**: Mix and match notification channels dynamically at runtime.
* **Avoids Inheritance Explosion**: No need for subclasses like EmailAndSMSNotifier, EmailAndSlackNotifier, etc.
* **Extensibility**: New decorators can be added with minimal impact.
* Notification and alert systems
* Input/output streams in Java (BufferedReader, DataInputStream)
* GUI toolkits where behavior needs to be layered (e.g., scrollable + resizable components)

**Exercise 6: Implementing the Proxy Pattern**

Image.java

public interface Image {

void display();

}

Realimage.java

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading image from remote server: " + filename);

// Simulate slow loading

try {

Thread.sleep(1000); // 1 second delay

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public void display() {

System.out.println("Displaying image: " + filename);

}

}

Proxyimage.java

public class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename); // Lazy initialization

} else {

System.out.println("Using cached image: " + filename);

}

realImage.display();

}

}  
Main.java

public class Main {

public static void main(String[] args) {

Image image1 = new ProxyImage("pic1.jpg");

Image image2 = new ProxyImage("pic2.jpg");

// First call - loads from remote

image1.display();

// Second call - uses cached RealImage

image1.display();

// First call for new image

image2.display();

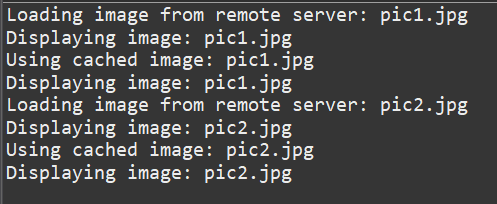
// Cached again

image2.display();

}

}

Output:



**Exercise 7: Implementing the Observer Pattern**

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

// Step 4: Observer Interface

interface Observer {

void update(String stockName, double price);

}

// Step 3: Concrete Subject

class StockMarket implements Stock {

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

private String stockName;

private double price;

public void setStockData(String stockName, double price) {

this.stockName = stockName;

this.price = price;

notifyObservers();

}

*@Override*

public void registerObserver(Observer o) {

observers.add(o);

System.***out***.println("Observer registered: " + o.getClass().getSimpleName());

}

*@Override*

public void removeObserver(Observer o) {

observers.remove(o);

System.***out***.println("Observer removed: " + o.getClass().getSimpleName());

}

*@Override*

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, price);

}

}

}

// Step 5: Concrete Observers

class MobileApp implements Observer {

*@Override*

public void update(String stockName, double price) {

System.***out***.println("MobileApp: " + stockName + " price updated to ₹" + price);

}

}

class WebApp implements Observer {

*@Override*

public void update(String stockName, double price) {

System.***out***.println("WebApp: " + stockName + " price updated to ₹" + price);

}

}

import java.util.\*;

public class ObserverPatternExample {

public static void main(String[] args) {

StockMarket market = new StockMarket();

~~Observer~~ mobile = new MobileApp();

~~Observer~~ web = new WebApp();

market.registerObserver(mobile);

market.registerObserver(web);

System.***out***.println("\n--- Stock Update 1 ---");

market.setStockData("TCS", 3750.50);

System.***out***.println("\n--- Stock Update 2 ---");

market.setStockData("Infosys", 1420.75);

// Remove mobile app

market.removeObserver(mobile);

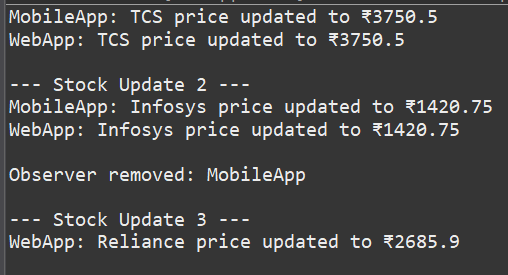
System.***out***.println("\n--- Stock Update 3 ---");

market.setStockData("Reliance", 2685.90);

}

}

Output:



* **Decoupling**: Subject doesn’t know the concrete implementation of observers.
* **Dynamic Relationships**: Observers can register/unregister at runtime.
* **Real-time Updates**: Immediate communication when the state changes.
* **Scalability**: Easily extend by adding new observer types.
* **Stock tracking apps**
* **Social media notifications**
* **Chat/messaging systems**
* **News feed or RSS updates**

**Exercise 8: Implementing the Strategy Pattern**

PaymentStrategy.java

public interface PaymentStrategy {

void pay(double amount);

}

CreditCardPayment.java

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card [" + cardNumber + "]");

}

}

PayPalPayment.java

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using PayPal account [" + email + "]");

}

}

PaymentContext.java

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void payAmount(double amount) {

if (strategy == null) {

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

} else {

strategy.pay(amount);

}

}}

Main.java -> main() calss

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay using Credit Card

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));

context.payAmount(2500.00);

// Pay using PayPal

context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.payAmount(4200.75);

}

}

Output:



* **Open/Closed Principle**: New payment methods can be added without modifying existing code.
* **Flexibility**: Allows switching strategies at runtime.
* **Encapsulation**: Each strategy contains its own logic, improving readability and maintainability.
* **Testability**: Easy to unit test each strategy independently.
* Payment gateways
* Sorting algorithms selection
* Compression format strategies (ZIP, RAR, GZIP)
* Routing or pathfinding strategies

**Exercise 9: Implementing the Command Pattern**

public class CommandPatternExample {

// Step 2: Command Interface

interface Command {

void execute();

}

// Step 5: Receiver Class

static class Light {

private String name;

public Light(String name) {

this.name = name;

}

public void turnOn() {

System.***out***.println(name + " light is ON");

}

public void turnOff() {

System.***out***.println(name + " light is OFF");

}

}

// Step 3: Concrete Commands

static class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// Step 4: Invoker Class

static class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

if (command != null) {

command.execute();

} else {

System.***out***.println("No command set.");

}

}

}

// Step 6: Test in main()

public static void main(String[] args) {

// Receiver

Light livingRoomLight = new Light("Living Room");

// Concrete Commands

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

// Invoker

RemoteControl remote = new RemoteControl();

// Execute ON command

remote.setCommand(lightOn);

remote.pressButton();

// Execute OFF command

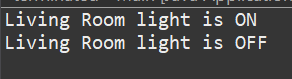
remote.setCommand(lightOff);

remote.pressButton();

}

}

Output:



* **Encapsulation**: Encapsulates a request as an object.
* **Decoupling**: Separates the invoker from the receiver.
* **Flexibility**: Supports undo/redo, macro commands, and logging.
* **Extensibility**: Easy to add new commands without changing existing code.
* Remote controls and GUI buttons
* Transaction and undo mechanisms
* Macro recording and playback

**Exercise 10: Implementing the MVC Pattern**

public class MVCPatternExample {

// Step 2: Model

static class Student {

private String name;

private String id;

private String grade;

public Student(String name, String id, String grade) {

this.name = name;

this.id = id;

this.grade = grade;

}

// Getters and Setters

public String getName() { return name; }

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

public String getId() { return id; }

public void setId(String id) { this.id = id; }

public String getGrade() { return grade; }

public void setGrade(String grade) { this.grade = grade; }

}

// Step 3: View

static class StudentView {

public void displayStudentDetails(String name, String id, String grade) {

System.***out***.println("Student Details:");

System.***out***.println("Name : " + name);

System.***out***.println("ID : " + id);

System.***out***.println("Grade : " + grade);

System.***out***.println();

}

}

// Step 4: Controller

static class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Update student name

public void setStudentName(String name) {

model.setName(name);

}

public String getStudentName() {

return model.getName();

}

public void setStudentId(String id) {

model.setId(id);

}

public String getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());

}

}

// Step 5: Main

public static void main(String[] args) {

// Model

Student student = new Student("Ravi Kumar", "S1234", "A");

// View

StudentView view = new StudentView();

// Controller

StudentController controller = new StudentController(student, view);

// Initial display

controller.updateView();

// Update model data

controller.setStudentName("Ravi Teja");

controller.setStudentGrade("A+");

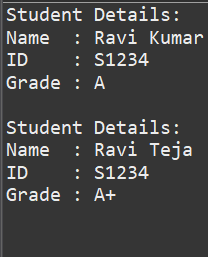
// Display updated data

controller.updateView();

}

}

Output:



* **Separation of Concerns**: Each layer has its own responsibility.
* **Scalability**: Easy to expand with more views (e.g., web, desktop, mobile).
* **Maintainability**: View and model can change independently.
* **Testability**: Business logic is isolated from the user interface.
* Web applications using frameworks like **Spring MVC**, **Struts**, **JSF**
* GUI apps with **JavaFX**, **Swing**, **Android (MVC-like MVP/MVVM)**
* Enterprise and microservices-based applications with **controller-service-repo** layering

**Exercise 11: Implementing Dependency Injection**

public class DependencyInjectionExample {

// Step 2: Repository Interface

interface CustomerRepository {

String findCustomerById(String id);

}

// Step 3: Concrete Repository Implementation

static class CustomerRepositoryImpl implements CustomerRepository {

*@Override*

public String findCustomerById(String id) {

// Simulating DB fetch

return "Customer[ID=" + id + ", Name=Ravi Kumar]";

}

}

// Step 4: Service Class

static class CustomerService {

private CustomerRepository repository;

// Step 5: Constructor-based Dependency Injection

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

String details = repository.findCustomerById(id);

System.***out***.println("Retrieved: " + details);

}

}

// Step 6: Main Class

public static void main(String[] args) {

// Manually injecting the dependency

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomerDetails("C102");

}

}

Output



* **Loose Coupling**: Classes depend on abstractions, not implementations.
* **Better Testability**: Easily inject mock dependencies in unit tests.
* **Cleaner Code**: Reduces boilerplate and improves structure.
* **Flexibility**: Swap implementations without modifying the dependent class.
* Used extensively in **Spring Framework** (@Autowired, constructor injection).
* Applied in **Guice**, **Dagger**, and **Jakarta EE**.
* Common in **unit testing** scenarios with mocked repositories.