**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

public class Logger {

// Private static instance of Logger

private static Logger instance;

// Private constructor to prevent external instantiation

private Logger() {

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

}

// Public method to return the single instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); // Create instance if it doesn't exist

}

return instance;

}

// Logging method

public void log(String message) {

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

}

}

**TestLogger.java**

public class TestLogger {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("This is the first log message.");

Logger logger2 = Logger.getInstance();

logger2.log("This is the second log message.");

// Check if both references point to the same object

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 refer to the same instance.");

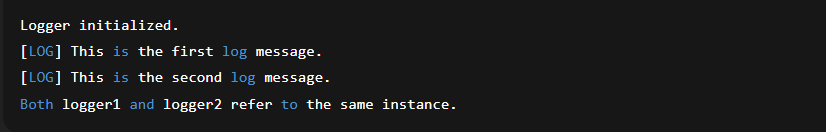
} else {

System.out.println("Different instances! Singleton failed.");

}

}

}



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

**1.Document.java**

public interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

}

**2. DocumentFactory.java**

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**Main.java**

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();

}

}



**Exercise 3: Implementing the Builder Pattern**

**Computer.java**

public class Computer {

// Required parameters

private final String cpu;

private final int ram;

// Optional parameters

private final int storage;

private final boolean hasGraphicsCard;

// ✅ Private constructor

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.hasGraphicsCard = builder.hasGraphicsCard;

}

// ✅ Static nested Builder class

public static class Builder {

// Required

private final String cpu;

private final int ram;

// Optional

private int storage = 256; // Default

private boolean hasGraphicsCard = false;

public Builder(String cpu, int ram) {

this.cpu = cpu;

this.ram = ram;

}

public Builder setStorage(int storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(boolean hasGraphicsCard) {

this.hasGraphicsCard = hasGraphicsCard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public String toString() {

return "Computer[CPU=" + cpu + ", RAM=" + ram + "GB, Storage=" + storage +

"GB, GraphicsCard=" + hasGraphicsCard + "]";

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

// Basic computer

Computer basicPC = new Computer.Builder("Intel i3", 8).build();

// Gaming PC

Computer gamingPC = new Computer.Builder("AMD Ryzen 7", 16)

.setStorage(1000)

.setGraphicsCard(true)

.build();

// Office PC

Computer officePC = new Computer.Builder("Intel i5", 16)

.setStorage(512)

.build();

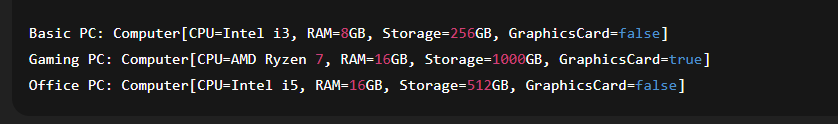
System.out.println("Basic PC: " + basicPC);

System.out.println("Gaming PC: " + gamingPC);

System.out.println("Office PC: " + officePC);

}

}



**Exercise 4: Implementing the Adapter Pattern**

**1. PaymentProcessor.java**

// Target Interface

public interface PaymentProcessor {

void processPayment(double amount);

}

// Adapter for PayPal

class PayPalAdapter implements PaymentProcessor {

private final PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.makePayPalPayment(amount);

}

}

// Adapter for Stripe

class StripeAdapter implements PaymentProcessor {

private final StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.sendStripePayment(amount);

}

}

**2. ThirdPartyGateways.java**

// Third-party PayPal gateway

class PayPalGateway {

public void makePayPalPayment(double amount) {

System.out.println("Processing payment via PayPal: ₹" + amount);

}

}

// Third-party Stripe gateway

class StripeGateway {

public void sendStripePayment(double amount) {

System.out.println("Processing payment via Stripe: ₹" + amount);

}

}

**3. Main.java**

public class Main {

public static void main(String[] args) {

// Using PayPal

PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

paypalProcessor.processPayment(2500.0);

// Using Stripe

PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

stripeProcessor.processPayment(4500.0);

}

}



**Exercise 5: Implementing the Decorator Pattern**

**1. Notifier.java**

// Component Interface

public interface Notifier {

void send(String message);

}

// Concrete Component

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**2. NotificationDecorators.java**

// Abstract Decorator

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

// Concrete SMS Decorator

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

// Concrete Slack Decorator

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

**3. Main.java**

public class Main {

public static void main(String[] args) {

// Base email notifier

Notifier notifier = new EmailNotifier();

// Add SMS and Slack using decorators

Notifier multiNotifier = new SlackNotifierDecorator(

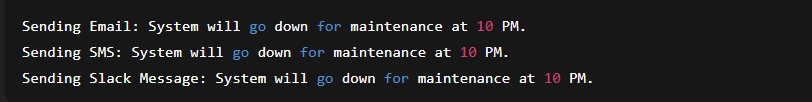
new SMSNotifierDecorator(notifier));

// Send the notification

multiNotifier.send("System will go down for maintenance at 10 PM.");

}

}



**Exercise 6: Implementing the Proxy Pattern**

**1.image.java**

public interface Image {

void display();

}

**2. RealImage.java**

public class RealImage implements Image {

private final String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

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

}

public void display() {

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

}

}

**3. ProxyImage.java**

public class ProxyImage implements Image {

private final String fileName;

private RealImage realImage;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realImage == null) {

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

} else {

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

}

realImage.display();

}

}

**4. Main.java**

public class Main {

public static void main(String[] args) {

Image img1 = new ProxyImage("nature.jpg");

Image img2 = new ProxyImage("city.png");

// First call: loads from remote

img1.display();

System.out.println();

// Second call: uses cache

img1.display();

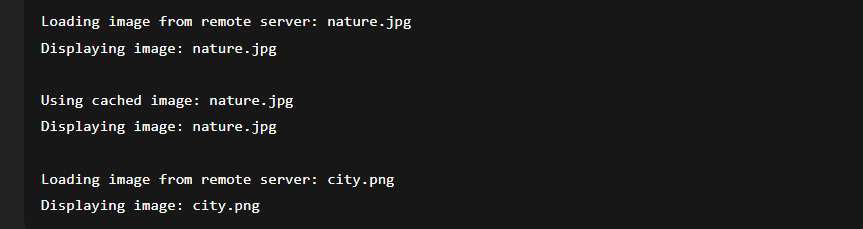
System.out.println();

// New image: loads from remote

img2.display();

}

}



**Exercise 7: Implementing the Observer Pattern**

**1. Stock.java**

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**2. StockMarket.java**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public StockMarket(String stockName) {

this.stockName = stockName;

}

public void setStockPrice(double price) {

this.stockPrice = price;

notifyObservers();

}

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

**3. Observer.java**

public interface Observer {

void update(String stockName, double stockPrice);

}

class MobileApp implements Observer {

public void update(String stockName, double stockPrice) {

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

}

}

class WebApp implements Observer {

public void update(String stockName, double stockPrice) {

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

}

}

**4. Main.java**

public class Main {

public static void main(String[] args) {

StockMarket reliance = new StockMarket("RELIANCE");

Observer mobileApp = new MobileApp();

Observer webApp = new WebApp();

reliance.registerObserver(mobileApp);

reliance.registerObserver(webApp);

// Price changes

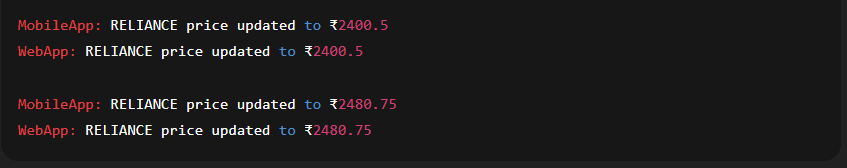
reliance.setStockPrice(2400.50);

System.out.println();

reliance.setStockPrice(2480.75);

}

}



**Exercise 8: Strategy Pattern – Payment System**

**1. PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

// Concrete Strategy: Credit Card

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

// Concrete Strategy: PayPal

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using PayPal.");

}

}

**2. PaymentContext.java**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy == null) {

System.out.println("Please select a payment method first.");

} else {

strategy.pay(amount);

}

}

}

**3. Main.java**

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// User selects PayPal

context.setPaymentStrategy(new PayPalPayment());

context.executePayment(1500.0);

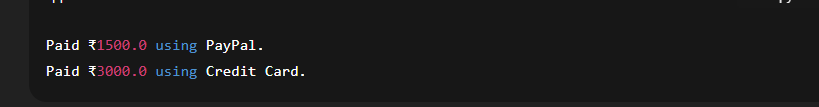
// User switches to Credit Card

context.setPaymentStrategy(new CreditCardPayment());

context.executePayment(3000.0);

}

}



**Exercise 9: Implementing the Command Pattern**

**1. Command.java**

public interface Command {

void execute();

}

**2. Light.java**

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**3. LightCommands.java**

// Turn ON Command

class LightOnCommand implements Command {

private final Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

// Turn OFF Command

class LightOffCommand implements Command {

private final Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

**4. RemoteControl.java**

public 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 assigned.");

}

}

}

**5. Main.java**

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

// Turn light ON

remote.setCommand(lightOn);

remote.pressButton();

// Turn light OFF

remote.setCommand(lightOff);

remote.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

**1. Student.java**

public 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; }

}

**2. StudentView.java**

public 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);

}

}

**3. StudentController.java**

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Update model

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

// Display using view

public void updateView() {

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

}

}

**4. Main.java**

public class Main {

public static void main(String[] args) {

// Model

Student student = new Student("Ananya", "S101", "A");

// View

StudentView view = new StudentView();

// Controller

StudentController controller = new StudentController(student, view);

// Display original data

controller.updateView();

System.out.println("\n-- Updating student info --\n");

// Update model via controller

controller.setStudentName("Ananya R");

controller.setStudentGrade("A+");

// Display updated data

controller.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

**1. CustomerRepository.java**

public interface CustomerRepository {

String findCustomerById(String id);

}

**2. CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

// Simulated DB lookup

return "Customer[ID=" + id + ", Name=John Doe]";

}

}

**3. CustomerService.java**

public class CustomerService {

private final CustomerRepository customerRepository;

// Constructor Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(String id) {

String customer = customerRepository.findCustomerById(id);

System.out.println("Customer Details: " + customer);

}

}

**4. Main.java**

public class Main {

public static void main(String[] args) {

// Create repository

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject repository into service

CustomerService service = new CustomerService(repository);

// Use service to fetch customer

service.getCustomerDetails("C123");

}

}

