**Exercise 1: Implementing the Singleton Pattern**

Code:-

public class Logger {

private static Logger singleInstance = null;

private Logger() {

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

}

public static Logger getInstance() {

if (singleInstance == null) {

singleInstance = new Logger();

}

return singleInstance;

}

public void log(String message) {

System.out.println("Log: " + message);

}

}

public class TestLogger {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

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

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

if (logger1 == logger2) {

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

} else {

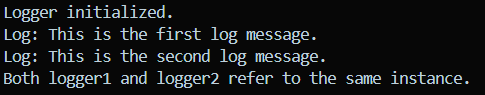
System.out.println("Different instances exist — Singleton failed.");

}

}

}

Output:-



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

Code:-

public interface Document {

void open();

}

public class WordDocument implements Document {

public void open() {

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

}

}

public class PdfDocument implements Document {

public void open() {

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

}

}

public class ExcelDocument implements Document {

public void open() {

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

}

}

public abstract class DocumentFactory {

public abstract Document createDocument();

}

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

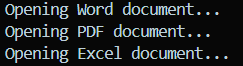
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

Output:-



**Exercise 3: Implementing the Builder Pattern**

Code:-

public class Computer {

private final String CPU;

private final String RAM;

private final String storage;

private final String graphicsCard;

private final String keyboard;

// Private constructor (only accessible via 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;

}

public static class Builder {

private final String CPU;

private final String RAM;

private String storage;

private String graphicsCard;

private String keyboard;

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setKeyboard(String keyboard) {

this.keyboard = keyboard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

@Override

public String toString() {

return "Computer Configuration:\n" +

"CPU: " + CPU + "\n" +

"RAM: " + RAM + "\n" +

"Storage: " + storage + "\n" +

"Graphics Card: " + graphicsCard + "\n" +

"Keyboard: " + keyboard;

}

}

public class Main {

public static void main(String[] args) {

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

.build();

Computer gamingComputer = new Computer.Builder("AMD Ryzen 7", "32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.setKeyboard("Mechanical RGB Keyboard")

.build();

System.out.println("=== Basic Computer ===");

System.out.println(basicComputer);

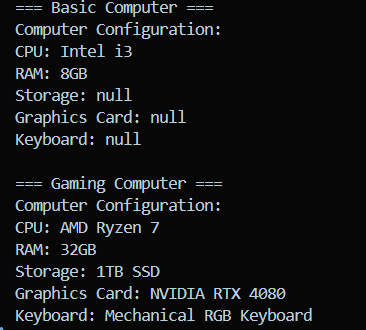
System.out.println("\n=== Gaming Computer ===");

System.out.println(gamingComputer);

}

}

Output:-



**Exercise 4: Implementing the Adapter Pattern**

Code:-

public interface PaymentProcessor {

void processPayment(double amount);

}

public class PayPalGateway {

public void makePayment(double amountInDollars) {

System.out.println("Paid $" + amountInDollars + " using PayPal.");

}

}

public class StripeGateway {

public void sendPayment(double amountInDollars) {

System.out.println("Paid $" + amountInDollars + " using Stripe.");

}

}

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.sendPayment(amount);

}

}

public class Main {

public static void main(String[] args) {

PayPalGateway paypal = new PayPalGateway();

PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);

paypalProcessor.processPayment(150.00);

StripeGateway stripe = new StripeGateway();

PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

stripeProcessor.processPayment(200.00);

}

}

Output:-



**Exercise 5: Implementing the Decorator Pattern**

Code:-

public interface Notifier {

void send(String message);

}

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message);

}

}

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);

}

}

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

Notifier baseNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

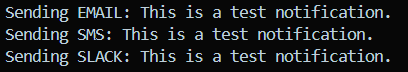
Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

slackNotifier.send("This is a test notification.");

}

}

Output:-



**Exercise 6: Implementing the Proxy Pattern**

Code:-

public interface Image {

void display();

}

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

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

}

@Override

public void display() {

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

}

}

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);

}

realImage.display();

}

}

public class ProxyPatternTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

System.out.println("--- First display ---");

image1.display();

image2.display();

System.out.println("\n--- Second display (uses cache) ---");

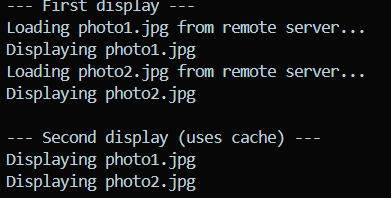
image1.display();

image2.display();

}

}

Output:-



**Exercise 7: Implementing the Observer Pattern**

Code:-

import java.util.ArrayList;

import java.util.List;

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

public class StockMarket implements Stock {

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

private double stockPrice;

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockPrice);

}

}

public void setStockPrice(double price) {

this.stockPrice = price;

notifyObservers();

}

}

public interface Observer {

void update(double stockPrice);

}

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("MobileApp " + name + " received stock update: ₹" + stockPrice);

}

}

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("WebApp " + name + " received stock update: ₹" + stockPrice);

}

}

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobile1 = new MobileApp("Groww");

Observer mobile2 = new MobileApp("Zerodha");

Observer web1 = new WebApp("NSE Website");

stockMarket.registerObserver(mobile1);

stockMarket.registerObserver(mobile2);

stockMarket.registerObserver(web1);

System.out.println("Updating stock price to ₹1550...");

stockMarket.setStockPrice(1550);

System.out.println("\nRemoving Zerodha MobileApp...");

stockMarket.removeObserver(mobile2);

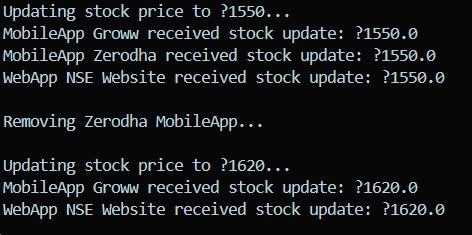
System.out.println("\nUpdating stock price to ₹1620...");

stockMarket.setStockPrice(1620);

}

}

Output:-



**Exercise 8: Implementing the Strategy Pattern**

Code:-

public interface PaymentStrategy {

void pay(double amount);

}

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolderName;

public CreditCardPayment(String cardNumber, String cardHolderName) {

this.cardNumber = cardNumber;

this.cardHolderName = cardHolderName;

}

@Override

public void pay(double amount) {

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

}

}

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);

}

}

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.paymentStrategy = strategy;

}

public void payAmount(double amount) {

if (paymentStrategy == null) {

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

} else {

paymentStrategy.pay(amount);

}

}

}

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

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

context.payAmount(500.0);

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

context.payAmount(300.0);

}

}

Output:-



**Exercise 9: Implementing the Command Pattern**

Code:-

public interface Command {

void execute();

}

public class Light {

public void turnOn() {

System.out.println("The light is ON.");

}

public void turnOff() {

System.out.println("The light is OFF.");

}

}

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

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();

remote.setCommand(lightOn);

remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton();

}

}

Output:-



**Exercise 10: Implementing the MVC Pattern**

Code:-

public class Student {

private String id;

private String name;

private String grade;

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

this.id = id;

this.name = name;

this.grade = grade;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

public class StudentView {

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

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

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

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

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

}

}

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public void setStudentId(String id) {

model.setId(id);

}

public String getStudentName() {

return model.getName();

}

public String getStudentGrade() {

return model.getGrade();

}

public String getStudentId() {

return model.getId();

}

public void updateView() {

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

}

}

public class MVCPatternExample {

public static void main(String[] args) {

Student student = new Student("101", "Alice", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Bob");

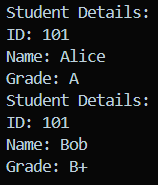
controller.setStudentGrade("B+");

controller.updateView();

}

}

Output:-



**Exercise 11: Implementing Dependency Injection**

Code:-

public interface CustomerRepository {

String findCustomerById(int id);

}

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(int id) {

return "Customer #" + id + " - John Doe";

}

}

public class CustomerService {

private final CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void displayCustomer(int id) {

String customer = customerRepository.findCustomerById(id);

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

}

}

public class Main {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.displayCustomer(101);

}

}

Output:-

