**Exercise 1: Implementing the Singleton Pattern**

**1. Create a New Java Project**

**2. Define a Singleton Class**

public class Logger {

    private static Logger instance;

    private Logger(){

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

    }

    public static Logger getInstance(){

        if(instance==null){

            instance=new Logger();

        }

        return instance;

    }

    public void testFunc(){

        System.out.println("Logger is working fine!!!");

    }

}

**3. Implement the Singleton Pattern**

**4. Test the Singleton Implementation**

public class TestLogger {

    public static void main(String args[]) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        logger1.testFunc();

        logger2.testFunc();

        if (logger1 == logger2) {

            System.out.println("Both instances are same");

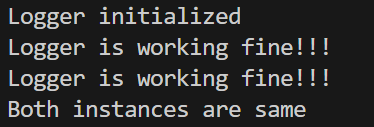
        } else {

            System.out.println("Both instances are different");

        }

    }

}



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

**1. Create a New Java Project**

**2. Define Document Classes**

**3. Create Concrete Document Classes:**

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");

    }

}

**4. Implement the Factory Method**

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();

    }

}

**5. Test the Factory Method Implementation**

public class FactoryTest {

    public static void main(String[] args) {

        DocumentFactory wordDoc=new WordDocumentFactory();

        Document wd=wordDoc.createDocument();

        wd.open();

        DocumentFactory pdfDoc=new PDFDocumentFactory();

        Document pd=pdfDoc.createDocument();

        pd.open();

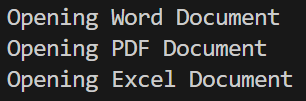
        DocumentFactory excelDoc=new ExcelDocumentFactory();

        Document ed=excelDoc.createDocument();

        ed.open();

    }

}



**Exercise 3: Implementing the Builder Pattern**

**1. Create a New Java Project**

**2. Define a Product Class**

**3. Implement the Builder Class**

**4. Implement the Builder Pattern**

public class Computer{

    private String cpu;

    private String ram;

    private String storage;

    private Computer(Builder builder) {

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

    }

    public static class Builder{

        private String cpu;

        private String ram;

        private String storage;

        public Builder setCPU(String cpu){

            this.cpu=cpu;

            return this;

        }

        public Builder setRAM(String ram){

            this.ram=ram;

            return this;

        }

        public Builder setStorage(String storage){

            this.storage=storage;

            return this;

        }

        public Computer build(){

            return new Computer(this);

        }

    }

    public String toString() {

        return "Computer Configuration:\n" +

               "CPU: " + cpu + "\n" +

               "RAM: " + ram + "\n" +

               "Storage: " + storage;

    }

}

**5. Test the Builder Implementation**

public class TestBuilder {

    public static void main(String[] args) {

        Computer pc=new Computer.Builder().setCPU("i5 12th gen").setRAM("16 gb").setStorage("1tb ssd").build();

        System.out.println(pc);

    }

}

**Exercise 3: Implementing the Builder Pattern**

1,2,3,4.

public class Computer {

    private String CPU;

    private String RAM;

    private String storage;

    private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.storage = builder.storage;

    }

    public static class Builder {

        private String CPU;

        private String RAM;

        private String storage;

        public Builder setCPU(String CPU) {

            this.CPU = CPU;

            return this;

        }

        public Builder setRAM(String RAM) {

            this.RAM = RAM;

            return this;

        }

        public Builder setStorage(String storage) {

            this.storage = storage;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

    public String toString() {

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

    }

}

5.

public class TestBuilder {

    public static void main(String[] args) {

        Computer comp = new Computer.Builder()

                .setCPU("Intel i5")

                .setRAM("8GB")

                .setStorage("2TB SSD")

                .build();

        System.out.println(comp);

    }

}



**Exercise 4: Implementing the Adapter Pattern**

1,2,3,4.

public interface PaymentProcessor {

    void processPayment(double amount);

    void refundPayment(double amount);

}

public class PayPalGateway {

    public void sendPayment(double amount) {

        System.out.println("PayPal payment of Rs " + amount +" is successful.");

    }

    public void makeRefund(double amount) {

        System.out.println("PayPal refund of Rs " + amount +" is successful.");

    }

}

public class StripeGateway {

    public void stripePay(double amount) {

        System.out.println("Stripe payment of Rs " + amount + " successful.");

    }

    public void stripeRefund(double amount) {

        System.out.println("Stripe refund of Rs " + amount + " successful.");

    }

}

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway gateway = new PayPalGateway();

    public void processPayment(double amount) {

        gateway.sendPayment(amount);

    }

    @Override

    public void refundPayment(double amount) {

        gateway.makeRefund(amount);

    }

}

public class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripe = new StripeGateway();

    @Override

    public void processPayment(double amount) {

        stripe.stripePay(amount);

    }

    @Override

    public void refundPayment(double amount) {

        stripe.stripeRefund(amount);

    }

}

5.

public class TestAdapter {

    public static void main(String[] args) {

        PaymentProcessor processor = new PayPalAdapter();

        processor.processPayment(180.00);

        processor.refundPayment(80.0);

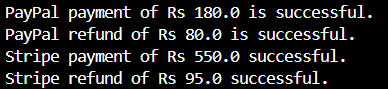
        PaymentProcessor stripe = new StripeAdapter();

        stripe.processPayment(550.0);

        stripe.refundPayment(95.0);

    }

}



**Exercise 5: Implementing the Decorator Pattern**

1,2,3,4.

public interface Notifier {

    void send(String message);

}

public class EmailNotifier implements Notifier {

    public void send(String message) {

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

    }

}

public abstract class NotifierDecorator implements Notifier {

    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    public void send(String message) {

        notifier.send(message);

    }

}

public class SMSNotifier extends NotifierDecorator {

    public SMSNotifier(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        super.send(message);

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

    }

}

public class SlackNotifier extends NotifierDecorator {

    public SlackNotifier(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

5.

public class TestDecorator {

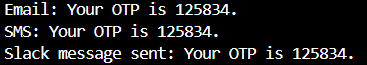
    public static void main(String[] args) {

        Notifier notifier = new SlackNotifier(new SMSNotifier(new EmailNotifier()));

        notifier.send("Your OTP is 125834.");

    }

}



**Exercise 6: Implementing the Proxy Pattern**

1,2,3,4.

public interface Image {

    void display();

}

public 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);

    }

}

public 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);

        }

        realImage.display();

    }

}

5.

public class TestProxy {

    public static void main(String[] args) {

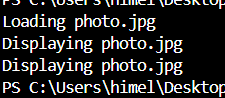
        Image img = new ProxyImage("photo.jpg");

        img.display(); // Loads and displays

        img.display(); // Only displays

    }

}



**Exercise 7: Implementing the Observer Pattern**

1,2,3,4.

public interface Stock {

    void register(Observer o);

    void deregister(Observer o);

    void notifyObservers();

}

import java.util.\*;

public class StockMarket implements Stock {

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

    private float price;

    public void register(Observer o) { observers.add(o); }

    public void deregister(Observer o) { observers.remove(o); }

    public void setPrice(float price) {

        this.price = price;

        System.out.println("Stock price updated internally to: Rs " + price);

        notifyObservers();

    }

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(price);

        }

    }

}

5.

public class MobileApp implements Observer {

    public void update(float price) {

        System.out.println("Mobile App: Stock price updated to Rs " + price);

    }

}

public class WebApp implements Observer {

    public void update(float price) {

        System.out.println("Web App: Stock price updated to Rs " + price);

    }

}

6.

public class TestObserver {

    public static void main(String[] args) {

        StockMarket market = new StockMarket();

        Observer mobile = new MobileApp();

        Observer web = new WebApp();

        market.register(mobile);

        market.register(web);

        market.setPrice(90.0f);

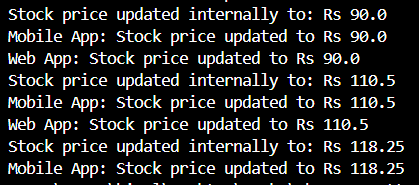
        market.setPrice(110.5f);

        market.deregister(web);

        market.setPrice(118.25f);

    }

}



**Exercise 8: Implementing the Strategy Pattern**

1,2,3,4.

public interface PaymentStrategy {

    void pay(double amount);

}

public class CreditCardPayment implements PaymentStrategy {

    public void pay(double amount) {

        System.out.println("Payment by Credit Card, Rs " + amount);

    }

}

public class PayPalPayment implements PaymentStrategy {

    public void pay(double amount) {

        System.out.println("Payment by PayPal, Rs " + amount);

    }

}

public class PaymentContext {

    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void pay(double amount) {

        strategy.pay(amount);

    }

}

5.

public class TestStrategy {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setStrategy(new CreditCardPayment());

        context.pay(800);

        context.setStrategy(new PayPalPayment());

        context.pay(150);

    }

}



**Exercise 9: Implementing the Command Pattern**

1,2,3,4.

public interface Command {

    void execute();

}

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.on();

    }

}

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.off();

    }

}

public class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void pressButton() {

        command.execute();

    }

}

5.

public class Light {

    public void on() {

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

    }

    public void off() {

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

    }

}

6.

public class TestCommand {

    public static void main(String[] args) {

        Light light = new Light();

        RemoteControl remote = new RemoteControl();

        remote.setCommand(new LightOnCommand(light));

        remote.pressButton();

        remote.setCommand(new LightOffCommand(light));

        remote.pressButton();

    }

}



**Exercise 10: Implementing the MVC Pattern**

1,2,3,4.

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;

    }

    public String getName() { return name; }

    public String getId() { return id; }

    public String getGrade() { return grade; }

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

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

}

public class StudentView {

    public void displayStudentDetails(Student student) {

        System.out.println("Student: " + student.getName() + ", ID: " + student.getId() + ", Grade: " + student.getGrade());

    }

}

public class StudentController {

    private Student student;

    private StudentView view;

    public StudentController(Student student, StudentView view) {

        this.student = student;

        this.view = view;

    }

    public void updateView() {

        view.displayStudentDetails(student);

    }

    public void setStudentName(String name) {

        student.setName(name);

    }

}

5.

public class TestMVC {

    public static void main(String[] args) {

        Student student = new Student("Allen", "110", "B");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Ayan");

        controller.updateView();

    }

}



**Exercise 11: Implementing Dependency Injection**

1,2,3,4,5.

public interface CustomerRepository {

    String findCustomerById(String id);

}

public class CustomerRepositoryImpl implements CustomerRepository {

    public String findCustomerById(String id) {

        return "Found Jay Ray, id is " + id;

    }

}

public class CustomerService {

    private CustomerRepository repo;

    public CustomerService(CustomerRepository repo) {

        this.repo = repo;

    }

    public void displayCustomer(String id) {

        System.out.println(repo.findCustomerById(id));

    }

}

6.

public class TestDI {

    public static void main(String[] args) {

        CustomerRepository repo = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repo);

        service.displayCustomer("74");

    }

}

