**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

public class Logger {

    private static Logger instance;

    private Logger() {

    }

    public static Logger getInstance(String userName) {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void logDetails(String msg) {

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

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        Logger log1 = Logger.getInstance("Alice");

        Logger log2 = Logger.getInstance("Bob");

        if (log1.hashCode() == log2.hashCode()) {

            log1.logDetails("Hello");

            log2.logDetails("World");

        }

    }

}

**Output:**



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

**Document.java**

package FactoryMethodPattern;

public interface Document {

    void open();

    void close();

}

**ExcelDocument.java**

package FactoryMethodPattern;

public class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing Excel Document");

    }

}

**PdfDocument.java**

package FactoryMethodPattern;

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing PDF Document");

    }

}

**WordDocument.java**

package FactoryMethodPattern;

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing Word Document");

    }

}

**DocumentFactory.java**

package FactoryMethodPattern;

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**ExcelFactory.java**

package FactoryMethodPattern;

public class ExcelFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**PdfFactory.java**

package FactoryMethodPattern;

public class PdfFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

**WordFactory.java**

package FactoryMethodPattern;

public class WordFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

**Main.java**

import java.util.Scanner;

import FactoryMethod.\*;

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the type of document to create (Word/Excel/Pdf): ");

        String docType = sc.nextLine();

        DocumentFactory factory;

        switch (docType.toLowerCase()) {

            case "word":

                factory = new WordFactory();

                break;

            case "excel":

                factory = new ExcelFactory();

                break;

            case "pdf":

                factory = new PdfFactory();

                break;

            default:

                System.out.println("Invalid document type.");

                return;

        }

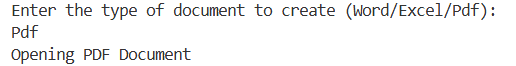
        Document document = factory.createDocument();

        document.open();

    }

}

**Output**

****

**Exercise 3: Implementing the Builder Pattern**

**Computer.java**

public class Computer{

    private final String cpu;

    private final String ram;

    private final String storage;

    private final String gpu;

    public Computer(Builder builder){

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

        this.gpu = builder.gpu;

    }

    public static class Builder{

        private String cpu;

        private String ram;

        private String storage;

        private String gpu;

        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 Builder setGpu(String gpu){

            this.gpu = gpu;

            return this;

        }

        public Computer build(){

            return new Computer(this);

        }

    }

    public String getCpu() {

        return cpu;

    }

    public String getRam() {

        return ram;

    }

    public String getStorage() {

        return storage;

    }

    public String getGpu() {

        return gpu;

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        Computer computer = new Computer.Builder()

                .setCpu("Intel Core i9")

                .setRam("32GB")

                .setStorage("1TB SSD")

                .setGpu("NVIDIA RTX 3080")

                .build();

        System.out.println("Computer built with the following specifications:");

        System.out.println("CPU: " + computer.getCpu());

        System.out.println("RAM: " + computer.getRam());

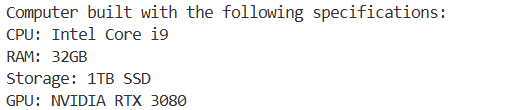
        System.out.println("Storage: " + computer.getStorage());

        System.out.println("GPU: " + computer.getGpu());

    }

}

**Output.java**

****

**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

public interface PaymentProcessor {

    void processPayment(double amount);

    String getPaymentGateway();

}

**Gpay.java**

public class GPay {

    public void makePayment(double amount) {

        System.out.println("Processing payment of " + amount + " through GPay.");

    }

}

**Paypal.java**

public class PayPal {

    public void makePayment(double amount) {

        System.out.println("Processing payment of " + amount + " through PayPal.");

    }

}

**GPayAdapter.java**

public class GpayAdapter implements PaymentProcessor {

    private final GPay gPay;

    public GpayAdapter(GPay gPay) {

        this.gPay = gPay;

    }

    @Override

    public void processPayment(double amount) {

        gPay.makePayment(amount);

    }

    @Override

    public String getPaymentGateway() {

        return "GPay";

    }

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

    private final PayPal payPal;

    public PayPalAdapter(PayPal payPal) {

        this.payPal = payPal;

    }

    public void processPayment(double amount) {

        payPal.makePayment(amount);

    }

    public String getPaymentGateway() {

        return "PayPal";

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        PaymentProcessor paymentProcessor = new GpayAdapter(new GPay());

        paymentProcessor.processPayment(100.0);

        System.out.println("Using payment gateway: " + paymentProcessor.getPaymentGateway());

        PaymentProcessor paymentProcessor2 = new PayPalAdapter(new PayPal());

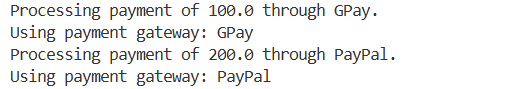
        paymentProcessor2.processPayment(200.0);

        System.out.println("Using payment gateway: " + paymentProcessor2.getPaymentGateway());

    }

}

**Output**

****

**Exercise 5: Implementing the Decorator Pattern**

**Notifier.java**

public interface Notifier {

    void send(String message);

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    public void send(String message) {

        notifier.send(message);

    }

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        super.send(message);

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

    }

}

**SlackNotifier.java**

public 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);

    }

}

**Test.java**

public class Test {

        public static void main(String[] args) {

        Notifier base = new EmailNotifier();

        Notifier sms = new SMSNotifierDecorator(base);

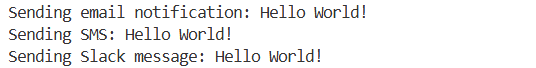
        Notifier slack = new SlackNotifierDecorator(sms);

        slack.send("Hello World!");

    }

}

**Output**

****