**DESIGN PATTERN AND PRINCIPLES WEEK 1**

**Exercise 1: Implementing the Singleton Pattern**

**CODE FOR LOGGER 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 log(String message) {

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

    }

}

**TEST CLASS FOR LOGGERS**

public class LoggerTest {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        logger1.log("Good Morning");

        logger2.log("Hello World");

        if (logger1 == logger2) {

            System.out.println("Both logger instances are the same. Singleton works!");

        } else {

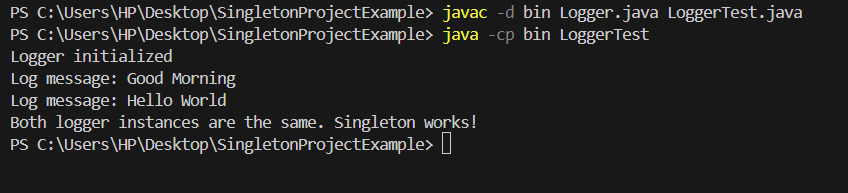
            System.out.println("Different logger instances found. Singleton failed!");

        }

    }

}

OUTPUT



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

* DOCUMENT INTERFACE:

public interface Document {

    void open();

}

* CREATING DOCUMENT CLASSES

public class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

}

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

}

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

}

* CREATING ABSTRACT CLASS DocumentFactory

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

* CREATING FACTORY CLASSES

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

* CREATING A TEST CLASS

public class FactoryMethodTest {

    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();

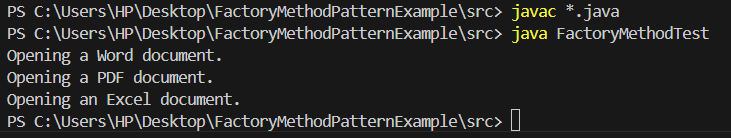
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

* OUTPUT



**Exercise 3: Implementing the Builder Pattern**

* CODE FOR Computer.java

public class Computer {

    private String CPU;

    private String RAM;

    private String Storage;

    private String GraphicsCard;

    private String OS;

    private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.Storage = builder.Storage;

        this.GraphicsCard = builder.GraphicsCard;

        this.OS = builder.OS;

    }

    public static class Builder {

        private String CPU;

        private String RAM;

        private String Storage;

        private String GraphicsCard;

        private String OS;

        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 setOperatingSystem(String operatingSystem) {

            this.OS = operatingSystem;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

    @Override

    public String toString() {

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

            ", Storage=" + Storage + ", GraphicsCard=" + GraphicsCard +

            ", OS=" + OS + "]";

    }

}

* CODE FOR Test.java

public class Test {

    public static void main(String[] args) {

        Computer basicComputer = new Computer.Builder("Intel i3", "4GB").build();

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

                .setStorage("2TB SSD")

                .setGraphicsCard("NVIDIA RTX 4080")

                .setOperatingSystem("Windows 11")

                .build();

        Computer officePC = new Computer.Builder("Apple MD", "16GB")

                .setStorage("512GB SSD")

                .setOperatingSystem("macOS Ventura")

                .build();

        System.out.println("Basic Computer: " + basicComputer);

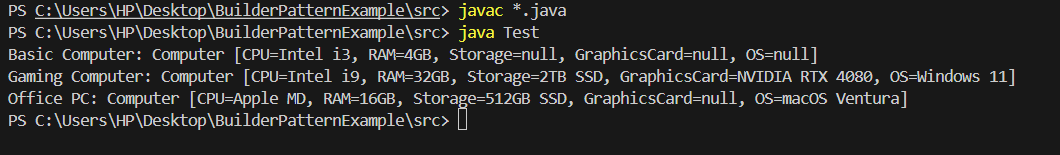
        System.out.println("Gaming Computer: " + gamingComputer);

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

    }

}

* OUTPUT



**Exercise 4: Implementing the Adapter Pattern**

* CODE FOR PaymentProcessor.java

public interface PaymentProcessor {

    void processPayment(double amount);

}

* CODES FOR ADAPTEE CLASSES

public class GooglePay {

    public void makePayment(double amount) {

        System.out.println("Paid Rs." + amount + " using Google Pay.");

    }

}

public class PhonePe {

    public void executePayment(double value) {

        System.out.println("Paid Rs." + value + " using PhonePe.");

    }

}

* CODES FOR ADAPTER CLASSES

public class GooglePayAdapter implements PaymentProcessor {

    private GooglePay googlePay;

    public GooglePayAdapter(GooglePay googlePay) {

        this.googlePay = googlePay;

    }

    @Override

    public void processPayment(double amount) {

        googlePay.makePayment(amount);

    }

}

public class PhonePeAdapter implements PaymentProcessor {

    private PhonePe phonePe;

    public PhonePeAdapter(PhonePe phonePe) {

        this.phonePe = phonePe;

    }

    @Override

    public void processPayment(double amount) {

        phonePe.executePayment(amount);

    }

}

* CODE FOR TEST CLASS

public class Test {

    public static void main(String[] args) {

        PaymentProcessor googlePayPayment = new GooglePayAdapter(new GooglePay());

        googlePayPayment.processPayment(1000.00);

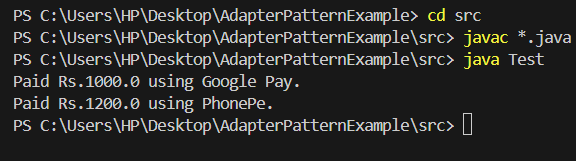
        PaymentProcessor phonePePayment = new PhonePeAdapter(new PhonePe());

        phonePePayment.processPayment(1200.00);

    }

}

* OUTPUT



**Exercise 5: Implementing the Decorator Pattern**

* CODE FOR Notifier.java

public interface Notifier {

    void send(String message);

}

* CODE FOR CONCRETE COMPONENT

public class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

* CODE FOR DECORATORS

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); // Call previous send

        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); // Call previous send

        sendSlack(message);

    }

    private void sendSlack(String message) {

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

    }

}

* CODE FOR Test.java

public class Test {

    public static void main(String[] args) {

        Notifier baseNotifier = new EmailNotifier();

        Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

        Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

        System.out.println("Notification through Email + SMS + Slack:");

        fullNotifier.send("System maintenance scheduled at 9 PM.");

    }

}

* OUTPUT

