Week 1

Design Patterns and Principles

**Exercise 1: Implementing the Singleton Pattern**

Code:

SingletonTest.java

class SingletonTest {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        logger1.log("First message");

        logger2.log("Second message");

        System.out.println(logger1 == logger2 ? "Same instance" : "Different instances");

    }

}

Logger.java

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;

    }

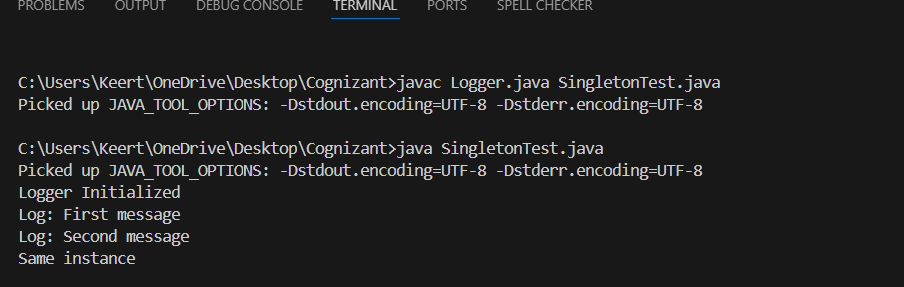
    public void log(String message) {

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

    }

}

Output:



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

Code:

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");

}

}

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();

}

}

class FactoryTest {

public static void main(String[] args) {

DocumentFactory factory = new PdfFactory();

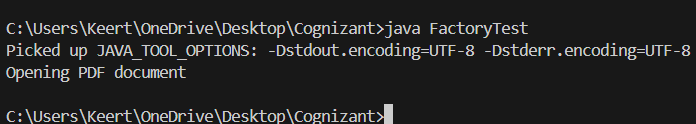
Document doc = factory.createDocument();

doc.open();

}

}

Output:



**Exercise 3: Implementing the Builder Pattern**

Code

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 String toString() {

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

    }

    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);

        }

    }

}

class BuilderTest {

    public static void main(String[] args) {

        Computer comp = new Computer.Builder()

                            .setCPU("Intel i7")

                            .setRAM("16GB")

                            .setStorage("1TB SSD")

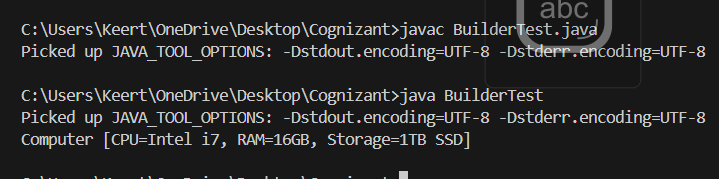
                            .build();

        System.out.println(comp);

    }

}

Output:



**Exercise 4: Implementing the Adapter Pattern**

Code:

interface PaymentProcessor {

    void processPayment(double amount);

}

class RazorpayGateway {

    public void makePayment(double amount) {

        System.out.println("Paid via Razorpay: Rs. " + amount);

    }

}

class StripeGateway {

    public void stripeCharge(double amount) {

        System.out.println("Paid via Stripe: Rs. " + amount);

    }

}

class RazorpayAdapter implements PaymentProcessor {

    private RazorpayGateway razorpay = new RazorpayGateway();

    public void processPayment(double amount) {

        razorpay.makePayment(amount);

    }

}

class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripe = new StripeGateway();

    public void processPayment(double amount) {

        stripe.stripeCharge(amount);

    }

}

class AdapterTest {

    public static void main(String[] args) {

        PaymentProcessor razor = new RazorpayAdapter();

        PaymentProcessor stripe = new StripeAdapter();

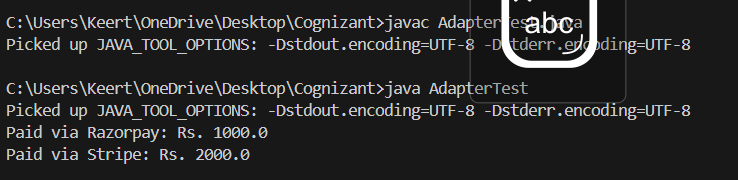
        razor.processPayment(1000);

        stripe.processPayment(2000);

    }

}

Output:



**Exercise 5: Implementing the Decorator Pattern**

Code:

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

public void send() {

System.out.println("Sending Email Notification");

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

notifier.send();

System.out.println("Sending SMS Notification");

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

notifier.send();

System.out.println("Sending Slack Notification");

}

}

class DecoratorTest {

public static void main(String[] args) {

Notifier email = new EmailNotifier();

Notifier sms = new SMSNotifierDecorator(email);

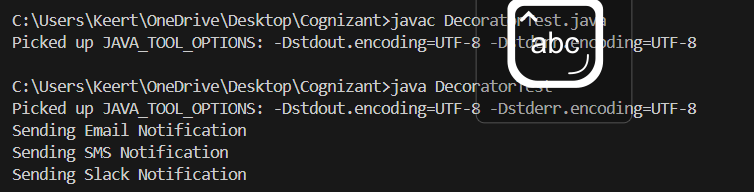
Notifier slack = new SlackNotifierDecorator(sms);

slack.send();

}

}

Output:



**Exercise 6: Implementing the Proxy Pattern**

Code:

interface Image {

    void display();

}

class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadFromDisk();

    }

    private void loadFromDisk() {

        System.out.println("Loading image: " + filename);

    }

    public void display() {

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

    }

}

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();

    }

}

class ProxyTest {

    public static void main(String[] args) {

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

        image.display(); // Loads and displays

        image.display(); // Just displays

    }

}

Output:

