# WEEK-1

## **Module 1 - Design Patterns and Principles**

**NAME : SIDHARTH K**

**SUPERSET ID : 6430194**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

**Logger.java:**

public class Logger{

    private static Logger instance;

    private Logger(){

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

    }

    public static Logger getInstance(){

        if (instance == null){

            instance = new Logger();

        }

        return instance;

    }

    public void log(String msg){

        System.out.println(msg);

    }

}

**Main.java:**

public class Logger{

    private static Logger instance;

    private Logger(){

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

    }

    public static Logger getInstance(){

        if (instance == null){

            instance = new Logger();

        }

        return instance;

    }

    public void log(String msg){

        System.out.println(msg);

    }

}

**Output:**

**A black background with blue lines

AI-generated content may be incorrect.**

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

**Code:**

**Document.java:**

public interface Document {

    void open();

}

**WordDocument.java:**

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

}

**PdfDocument.java:**

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

}

**ExcelDocument.java:**

public class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

}

**DocumentFactory.java:**

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**WordDocumentFactory.java:**

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

**PdfDocumentFactory.java:**

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

**ExcelDocumentFactory.java:**

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**Main.java:**

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();

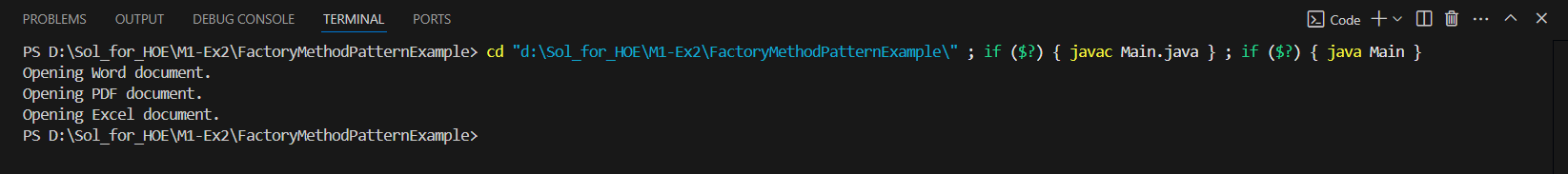
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**Output:**

****

**Exercise 3: Implementing the Builder Pattern**

**Code:**

**Computer.java:**

public class Computer {

    private final String cpu;

    private final String ram;

    private final String storage;

    private final String graphicsCard;

    private final 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 void showSpecs() {

        System.out.println("Computer Specs:");

        System.out.println("CPU: " + cpu);

        System.out.println("RAM: " + ram);

        System.out.println("Storage: " + (storage != null ? storage : "Not specified"));

        System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not specified"));

        System.out.println("OS: " + (os != null ? os : "Not specified"));

    }

    public static class Builder {

        private final String cpu;

        private final 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 setOS(String os) {

            this.os = os;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

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

                .build();

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

                .setStorage("1TB SSD")

                .setGraphicsCard("NVIDIA RTX 4080")

                .setOS("Windows 11")

                .build();

        System.out.println("\*\*\*\* Basic Computer \*\*\*\*");

        basicComputer.showSpecs();

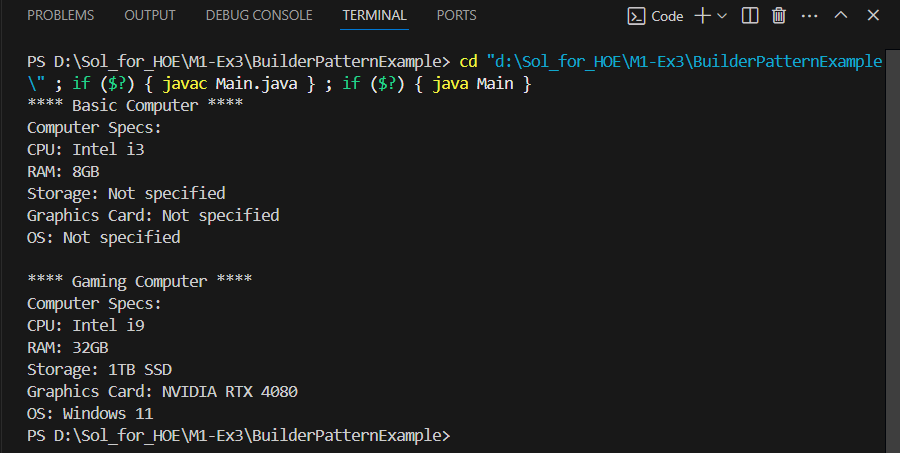
        System.out.println("\n\*\*\*\* Gaming Computer \*\*\*\*");

        gamingComputer.showSpecs();

    }

}

**Output:**



**Exercise 4: Implementing the Adapter Pattern**

**Code:**

**PaymentProcessor.java:**

public interface PaymentProcessor {

    void processPayment(double amount);

}

**StripeGateway.java:**

public class StripeGateway {

    public void makeStripePayment(double amountInDollars) {

        System.out.println("Stripe processed payment of $" + amountInDollars);

    }

}

**PayPalGateway.java:**

public class PayPalGateway {

    public void sendPayment(double usdAmount) {

        System.out.println("PayPal processed payment of $" + usdAmount);

    }

}

**StripeAdapter.java:**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.makeStripePayment(amount);

}

}

**PayPalAdapter.java:**

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway paypal;

    public PayPalAdapter(PayPalGateway paypal) {

        this.paypal = paypal;

    }

    @Override

    public void processPayment(double amount) {

        paypal.sendPayment(amount);

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

        StripeGateway stripe = new StripeGateway();

        PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

        stripeProcessor.processPayment(150.0);

        PayPalGateway paypal = new PayPalGateway();

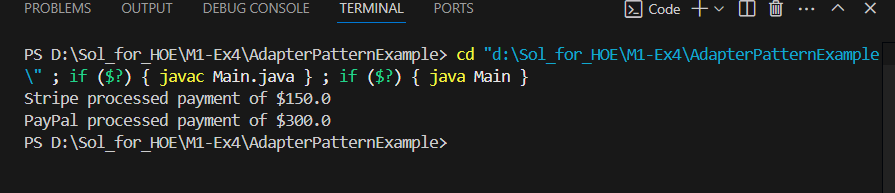
        PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);

        paypalProcessor.processPayment(300.0);

    }

}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**Code:**

**Notifier.java:**

public interface Notifier {

    void send(String message);

}

**EmailNotifier.java:**

public class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

**NotifierDecorator.java:**

public abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier notifier) {

        this.wrappedNotifier = notifier;

    }

    @Override

    public void send(String message) {

        wrappedNotifier.send(message);

    }

}

**SMSNotifierDecorator.java:**

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

**SlackNotifierDecorator.java:**

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message); // Call base notifier

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

    }

}

**Main.java:**

public class Main {

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

Notifier multiChannelNotifier = new SlackNotifierDecorator(smsNotifier);

multiChannelNotifier.send("System alert: CPU usage at 95%");

}

}

**Output:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Exercise 6: Implementing the Proxy Pattern**

**Code:**

**Image.java:**

public interface Image {

    void display();

}

**RealImage.java:**

public class RealImage implements Image {

    private String fileName;

    public RealImage(String fileName) {

        this.fileName = fileName;

        loadFromServer();

    }

    private void loadFromServer() {

        System.out.println("Loading image from remote server: " + fileName);

        try {

            Thread.sleep(2000); // Simulate network delay

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    @Override

    public void display() {

        System.out.println("Displaying image: " + fileName);

    }

}

**ProxyImage.java:**

public class ProxyImage implements Image {

    private String fileName;

    private RealImage realImage;

    public ProxyImage(String fileName) {

        this.fileName = fileName;

    }

    @Override

    public void display() {

        if (realImage == null) {

            System.out.println("Creating RealImage object for: " + fileName);

            realImage = new RealImage(fileName); // Lazy initialization

        } else {

            System.out.println("Using cached RealImage for: " + fileName);

        }

        realImage.display();

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

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

        image1.display();

        System.out.println();

        image1.display();

        System.out.println();

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

        image2.display();

    }

}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Exercise 7: Implementing the Observer Pattern**

**Code:**

**Stock.java:**

public interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();

}

**Observer.java:**

public interface Observer {

    void update(String stockName, double newPrice);

}

**StockMarket.java:**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

    private String stockName;

    private double price;

    public StockMarket(String stockName, double initialPrice) {

        this.stockName = stockName;

        this.price = initialPrice;

    }

    public void setPrice(double newPrice) {

        System.out.println("\nUpdating " + stockName + " stock price to RS." + newPrice);

        this.price = newPrice;

        notifyObservers();

    }

    @Override

    public void registerObserver(Observer o) {

        observers.add(o);

        System.out.println("Observer registered: " + o.getClass().getSimpleName());

    }

    @Override

    public void removeObserver(Observer o) {

        observers.remove(o);

        System.out.println("Observer removed: " + o.getClass().getSimpleName());

    }

    @Override

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(stockName, price);

        }

    }

}

**MobileApp.java:**

public class MobileApp implements Observer {

    @Override

    public void update(String stockName, double newPrice) {

        System.out.println("[MobileApp] " + stockName + " updated to Rs." + newPrice);

    }

}

**WebApp.java:**

public class WebApp implements Observer {

    @Override

    public void update(String stockName, double newPrice) {

        System.out.println("[WebApp] " + stockName + " updated to RS." + newPrice);

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

        StockMarket appleStock = new StockMarket("AAPL", 150.0);

        Observer mobileApp = new MobileApp();

        Observer webApp = new WebApp();

        appleStock.registerObserver(mobileApp);

        appleStock.registerObserver(webApp);

        appleStock.setPrice(155.0);

        appleStock.setPrice(160.5);

        appleStock.removeObserver(webApp);

        appleStock.setPrice(162.75);

    }

}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Exercise 8: Implementing the Strategy Pattern**

**Code:**

**PaymentStrategy.java:**

public interface PaymentStrategy {

    void pay(double amount);

}

**CreditCardPayment.java:**

public class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    public CreditCardPayment(String cardNumber) {

        this.cardNumber = cardNumber;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid RS." + amount + " using Credit Card ending with " +

                cardNumber.substring(cardNumber.length() - 4));

    }

}

**PayPalPayment.java:**

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 RS." + amount + " via PayPal account: " + email);

    }

}

**PaymentContext.java:**

public class PaymentContext {

    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void processPayment(double amount) {

        if (strategy == null) {

            System.out.println("Payment strategy not set.");

        } else {

            strategy.pay(amount);

        }

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9876-5432");

        context.setStrategy(creditCard);

        context.processPayment(25000.00);

        PaymentStrategy paypal = new PayPalPayment("user@example.com");

        context.setStrategy(paypal);

        context.processPayment(18000.50);

    }

}

**Output:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Exercise 9: Implementing the Command Pattern**

**Code:**

**Command.java:**

public interface Command {

    void execute();

}

**LightOnCommand.java:**

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

**LightOffCommand.java:**

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff(); } }

**RemoteControl.java:**

public class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void pressButton() {

        if (command != null) {

            command.execute();

        } else {

            System.out.println("No command assigned.");

        }

    }

}

**Light.java:**

public class Light {

    public void turnOn() {

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

    }

    public void turnOff() {

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

    }

}

**Main.java:**

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:**

A computer screen with text

AI-generated content may be incorrect.

**Exercise 10: Implementing the MVC Pattern**

**Code:**

**Student.java:**

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

    public String getId() { return id; }

    public void setId(String id) { this.id = id; }

    public String getGrade() { return grade; }

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

}

**StudentView.java:**

public class StudentView {

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

        System.out.println("\n--- Student Details ---");

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

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

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

    }

}

**StudentController.java:**

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 setStudentId(String id) {

        model.setId(id);

    }

    public void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    public void updateView() {

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

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

        Student student = new Student("Roshan", "S239", "B");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Kumar");

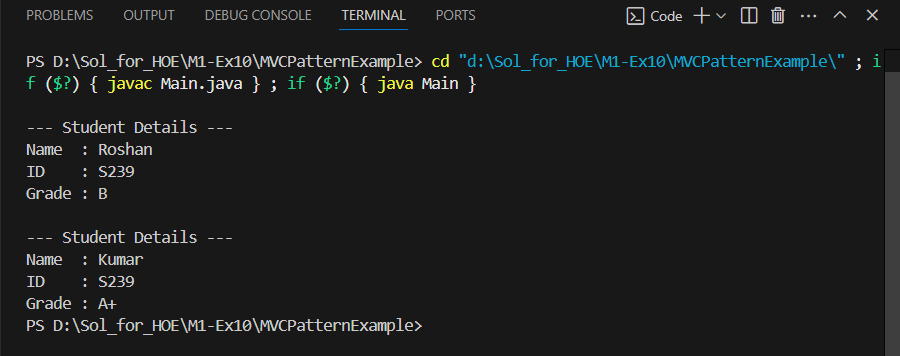
        controller.setStudentGrade("A+");

        controller.updateView();

    }

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Code:**

**Customer.java:**

public class Customer {

    private String id;

    private String name;

    public Customer(String id, String name) {

        this.id = id;

        this.name = name;

    }

    public String getId() { return id; }

    public String getName() { return name; }

    @Override

    public String toString() {

        return "Customer{id='" + id + "', name='" + name + "'}";

    }

}

**CustomerRepository.java:**

public interface CustomerRepository {

    Customer findCustomerById(String id);

}

**CustomerRepositoryImpl.java:**

import java.util.HashMap;

import java.util.Map;

public class CustomerRepositoryImpl implements CustomerRepository {

    private Map<String, Customer> customerDatabase = new HashMap<>();

    public CustomerRepositoryImpl() {

        // Simulate data

        customerDatabase.put("C531", new Customer("C531", "Sid"));

        customerDatabase.put("C202", new Customer("C202", "Raja"));

    }

    @Override

    public Customer findCustomerById(String id) {

        return customerDatabase.get(id);

    }

}

**CustomerService.java:**

public class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void showCustomerDetails(String id) {

        Customer customer = customerRepository.findCustomerById(id);

        if (customer != null) {

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

        } else {

            System.out.println("Customer not found for ID: " + id);

        }

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repository);

        service.showCustomerDetails("C531");

        service.showCustomerDetails("C742");

    }

}

**Output: A screen shot of a computer program

AI-generated content may be incorrect.**