**WEEK-1 ( DESIGN PATTERNS AND PRINCIPLES HANDS-ON )**

**(MANDATORY HANDS-ON)**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Code :**

public class Logger {

    private static Logger instance;

    private Logger(){}

    public static Logger getInstance(){

        if (instance == null){

            instance = new Logger();

        }

        return instance;

    }

}

class SingletonPatternTest {

    public static void main(String args[]){

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        if (logger1 == logger2){

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

        }

        else {

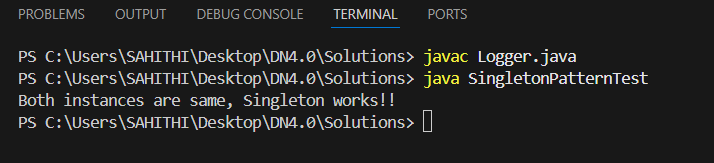
            System.out.println("Both instances are diffferent, Singleton Failed!!");

        }

    }

}

**OUTPUT :**

****

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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Code :**

interface Document{

   void documentViewer();  //interface for Documents

}

class WordDocument implements Document{

    public void documentViewer(){

        System.out.println("This is a Word Document");

    }

}

class PdfDocument implements Document{

    public void documentViewer(){

        System.out.println("This is a Pdf Document");

    }

}

class ExcelDocument implements Document{

    public void documentViewer(){

        System.out.println("This is an Excel Document");

    }

}

abstract class DocumentFactory {

   protected abstract Document createDocument();

    public void makeDocument(){

        Document doc = createDocument();

        doc.documentViewer();

    }

}

class WordDocumentCreator extends DocumentFactory{

    @Override

    protected Document createDocument(){

        return new WordDocument();

    }

}

class PdfDocumentCreator extends DocumentFactory{

    @Override

    protected Document createDocument(){

        return new PdfDocument();

    }

}

class ExcelDocumentCreator extends DocumentFactory{

    @Override

    protected Document createDocument(){

        return new ExcelDocument();

    }

}

 public class FactoryMethodPatternTest{

    public static void main(String[] args){

        DocumentFactory wordDoc = new WordDocumentCreator();

        wordDoc.makeDocument();

        DocumentFactory pdfDoc = new PdfDocumentCreator();

        pdfDoc.makeDocument();

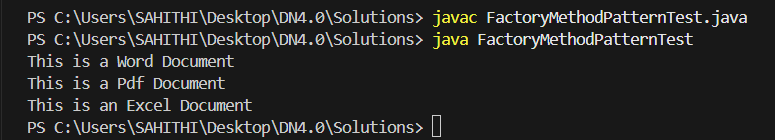
        DocumentFactory excelDoc = new ExcelDocumentCreator();

        excelDoc.makeDocument();

    }

}

**OUTPUT :**

****

**( ADDITIONAL HANDS-ON )**

**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**CODE :**

class Computer{

    String CPU;

    int RAM;

    int Storage;

    String OS;

    String Brand;

    public Computer(ComputerBuilder builder){

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.Storage = builder.Storage;

        this.OS = builder.OS;

        this.Brand = builder.Brand;

    }

    public String getDescription(){

        return "This computer contains a " + CPU + " processor, " + RAM + " GB RAM, " + Storage + " GB of storage, runs on " + OS + ", and is manufactured by " + Brand + ".";

    }

    public static class ComputerBuilder{

        private String CPU;

        private int RAM;

        private int Storage;

        private String OS;

        private String Brand;

        public ComputerBuilder setCPU(String CpuName){

            this.CPU = CpuName;

            return this;

        }

        public ComputerBuilder setRAM(int ram){

            this.RAM = ram;

            return this;

        }

        public ComputerBuilder setStorage(int storage){

            this.Storage = storage;

            return this;

        }

         public ComputerBuilder setOS(String os){

            this.OS = os;

            return this;

        }

         public ComputerBuilder setBrand(String brand){

            this.Brand = brand;

            return this;

        }

        public Computer build(){

            return new Computer(this);

        }

    }

}

public class BuilderPatternTest {

    public static void main(String[] args) {

        Computer basicComputer = new Computer.ComputerBuilder()

                .setCPU("Intel Core i5-1135G7")

                .setRAM(8)

                .setStorage(256)

                .setOS("Windows 10")

                .setBrand("Dell")

                .build();

        System.out.println(basicComputer.getDescription());

        Computer minimalComputer = new Computer.ComputerBuilder()

                .setCPU("Apple M1")

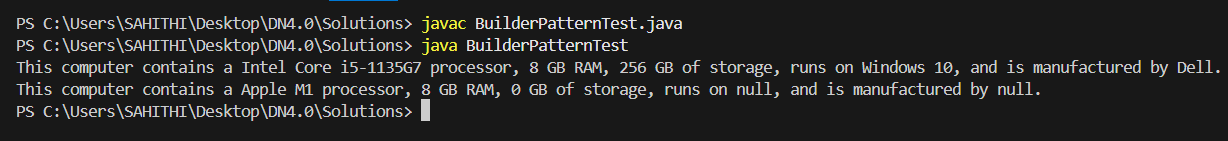
                .setRAM(8)

                .build();

        System.out.println(minimalComputer.getDescription());

    }

}

**OUTPUT :** 

**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**CODE :**

interface PaymentProcessor{

   void processPayment();

}

 class PayPalGateway {

    public void sendPayment(double amount) {

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

    }

}

 class StripeGateway {

    public void makeStripePayment(String currency, double amount) {

        System.out.println("Processing " + currency + " " + amount + " via Stripe.");

    }

}

 class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPalGateway;

    private double amount;

    public PayPalAdapter(PayPalGateway gateway, double amount) {

        this.payPalGateway = gateway;

        this.amount = amount;

    }

    @Override

    public void processPayment() {

        payPalGateway.sendPayment(amount);

    }

}

 class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripeGateway;

    private double amount;

    private String currency;

    public StripeAdapter(StripeGateway gateway, String currency, double amount) {

        this.stripeGateway = gateway;

        this.currency = currency;

        this.amount = amount;

    }

    @Override

    public void processPayment() {

        stripeGateway.makeStripePayment(currency, amount);

    }

}

public class AdapterTest {

    public static void main(String[] args) {

        PayPalGateway payPal = new PayPalGateway();

        PaymentProcessor payPalProcessor = new PayPalAdapter(payPal, 100.0);

        payPalProcessor.processPayment();

        StripeGateway stripe = new StripeGateway();

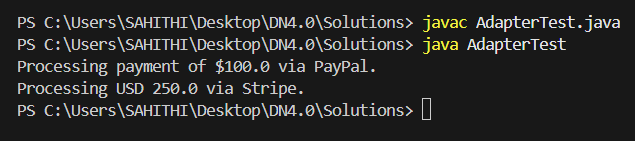
        PaymentProcessor stripeProcessor = new StripeAdapter(stripe, "USD", 250.0);

        stripeProcessor.processPayment();

    }

}

**OUTPUT :**

****

**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**CODE :**

interface Notifier {

    void send(String message);

}

class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappee;

    public NotifierDecorator(Notifier notifier) {

        this.wrappee = notifier;

    }

    @Override

    public void send(String message) {

        wrappee.send(message);

    }

}

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);

    }

}

class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

public class DecorativePatternTest {

    public static void main(String[] args) {

        // Base notifier

        Notifier notifier = new EmailNotifier();

        // Add SMS and Slack notifications

        Notifier multiChannelNotifier = new SlackNotifierDecorator(

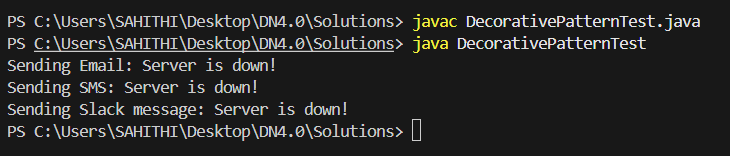
                                           new SMSNotifierDecorator(notifier));

        multiChannelNotifier.send("Server is down!");

    }

}

**OUTPUT :**



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**CODE :**

interface Image {

    void display();

}

class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadFromRemoteServer();

    }

    private void loadFromRemoteServer() {

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

    }

    @Override

    public void display() {

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

    }

}

class ProxyImage implements Image {

    private String filename;

    private RealImage realImage;

    public ProxyImage(String filename) {

        this.filename = filename;

    }

    @Override

    public void display() {

        if (realImage == null) {

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

        } else {

            System.out.println("Image already loaded, using cached image.");

        }

        realImage.display();

    }

}

public class ProxyPatternTest {

    public static void main(String[] args) {

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

        image1.display();

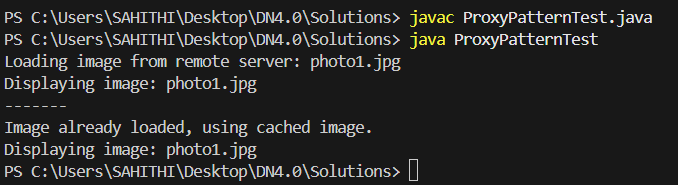
        System.out.println("-------");

        image1.display();

    }

}

**OUTPUT :**

****

**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**CODE :**

import java.util.ArrayList;

import java.util.List;

interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();

}

interface Observer {

    void update(String stockName, double price);

}

class StockMarket implements Stock {

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

    private String stockName;

    private double stockPrice;

    public void setStock(String stockName, double stockPrice) {

        this.stockName = stockName;

        this.stockPrice = stockPrice;

        notifyObservers();

    }

    @Override

    public void registerObserver(Observer o) {

        observers.add(o);

    }

    @Override

    public void removeObserver(Observer o) {

        observers.remove(o);

    }

    @Override

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockName, stockPrice);

        }

    }

}

class MobileApp implements Observer {

    private String user;

    public MobileApp(String user) {

        this.user = user;

    }

    @Override

    public void update(String stockName, double price) {

        System.out.println("MobileApp [" + user + "]: " + stockName + " updated to $" + price);

    }

}

class WebApp implements Observer {

    private String user;

    public WebApp(String user) {

        this.user = user;

    }

    @Override

    public void update(String stockName, double price) {

        System.out.println("WebApp [" + user + "]: " + stockName + " updated to $" + price);

    }

}

public class ObserverPatternTest {

    public static void main(String[] args) {

        StockMarket market = new StockMarket();

        Observer aliceMobile = new MobileApp("Alice");

        Observer bobWeb = new WebApp("Bob");

        market.registerObserver(aliceMobile);

        market.registerObserver(bobWeb);

        market.setStock("AAPL", 190.5);

        System.out.println("-----");

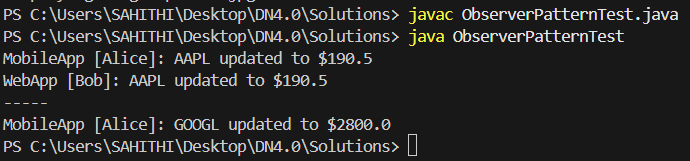
        market.removeObserver(bobWeb);

        market.setStock("GOOGL", 2800.0);

    }

}

**OUTPUT :**

****

**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**CODE :**

interface PaymentStrategy {

    void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    public CreditCardPayment(String cardNumber) {

        this.cardNumber = cardNumber;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid $" + amount + " using Credit Card: " + cardNumber);

    }

}

class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid $" + amount + " using PayPal account: " + email);

    }

}

class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentStrategy = paymentStrategy;

    }

    public void pay(double amount) {

        if (paymentStrategy == null) {

            System.out.println("No payment strategy selected.");

        } else {

            paymentStrategy.pay(amount);

        }

    }

}

public class StrategyPatternTest {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));

        context.pay(150.0);

        System.out.println("-----");

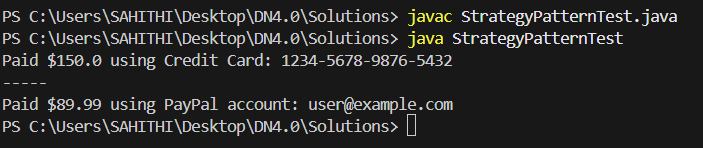
        context.setPaymentStrategy(new PayPalPayment("user@example.com"));

        context.pay(89.99);

    }

}

**OUTPUT :**

****

**Exercise 9: Implementing the Command Pattern**

**Scenario:**

You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**CODE :**

interface Command {

    void execute();

}

class Light {

    public void turnOn() {

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

    }

    public void turnOff() {

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

    }

}

class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

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 set.");

        }

    }

}

public class CommandPatternTest {

    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();

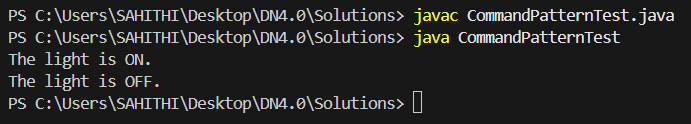
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**OUTPUT :**



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**CODE :**

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

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

}

class StudentView {

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

        System.out.println("Student Details:");

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

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

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

        System.out.println("------------------------");

    }

}

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 String getStudentName() {

        return model.getName();

    }

    public String getStudentId() {

        return model.getId();

    }

    public String getStudentGrade() {

        return model.getGrade();

    }

    public void updateView() {

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

    }

}

public class MVCPatternTest {

    public static void main(String[] args) {

        Student student = new Student("Saahi", "y22cs085", "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Rahul");

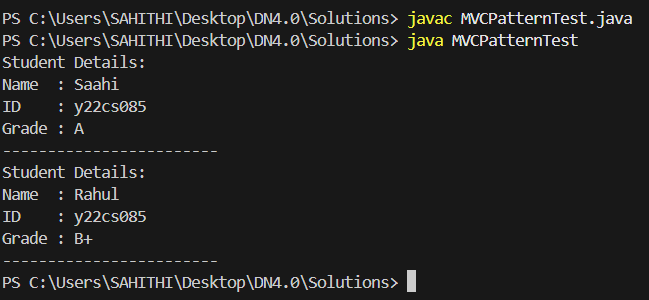
        controller.setStudentGrade("B+");

        controller.updateView();

    }

}

**OUTPUT :**

****

**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**CODE :**

import java.util.HashMap;

import java.util.Map;

interface CustomerRepository {

    Customer findCustomerById(String id);

}

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; }

}

class CustomerRepositoryImpl implements CustomerRepository {

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

    public CustomerRepositoryImpl() {

        customerDB.put("Y22CS001", new Customer("Y22CS001", "Gita"));

        customerDB.put("Y22CS002", new Customer("Y22CS002", "Mira"));

    }

    @Override

    public Customer findCustomerById(String id) {

        return customerDB.get(id);

    }

}

class CustomerService {

    private CustomerRepository customerRepository;

    // Constructor Injection

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void printCustomerDetails(String id) {

        Customer customer = customerRepository.findCustomerById(id);

        if (customer != null) {

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

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

        } else {

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

        }

    }

}

public class DependencyInjectionTest {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl(); // Create concrete repo

        CustomerService service = new CustomerService(repository);     // Inject into service

        // Use the service

        service.printCustomerDetails("Y22CS001");

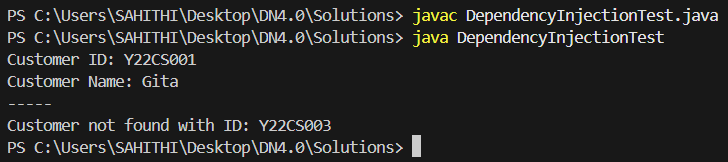
        System.out.println("-----");

        service.printCustomerDetails("Y22CS003"); // Non-existent ID

    }

}

**OUTPUT :**

****