**Questions for Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern(Mandatory)**

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**

Create a test class to verify that only one instance of Logger is created and used across the application

**Source Codes:**

***Logger.java:***

public class Logger {

    //Private static instance of Logger

    private static Logger instance;

    //Private constructor as per instruction

    private Logger() {

        System.out.println("Logger Initialized by Private Constructor!!");

    }

    //Public static method to return the same instance

    public static synchronized Logger getInstance() {

        /\*we have used synchronized here so that in case of multithreading,only one thread can access the function at a time and

        thus preventing creation of multiple instances if by chance two thread to call the method at one!!\*/

            if (instance == null)

            {

            instance = new Logger();

            }

        return instance;

    }

    // Example logging method to print message during creation of instances

    public void log(String message) {

        System.out.println("[LOG] " + message);

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        //Creating first instace to test

        Logger logger1 = Logger.getInstance();

        logger1.log("This is the first log message with hashcode:"+logger1.hashCode());

        //Creating second instance to test

        Logger logger2 = Logger.getInstance();

        logger2.log("This is the second log message with hashcode:"+logger2.hashCode());

        /\*Check if both instances are the same on basis of hashcode,We can directly

        check for equality too,but I have preferred to compare using hashCode that is unique to each\*/

        if (logger1.hashCode() == logger2.hashCode()) {

            System.out.println("As we can see that both instacnes have the same hashcode that is unique,Thus both logger instances are the same (Singleton works).");

        }

        else {

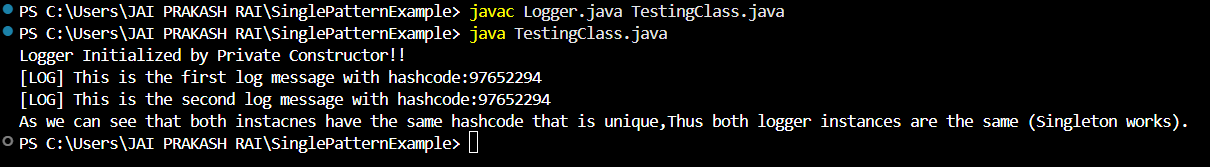
            System.out.println("Logger instances are different (Singleton failed).");

        }

    }

}

**Output:**



**Exercise 2: Implementing the Factory Method Pattern(Mandatory)**

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

**Source Code:**

***Document.java:***

public interface Document {

    //creating the interface with one method

    void create();

}

***WordDocument.java:***

public class WordDocument implements Document{

    //Concrete Class for Word document

    public void create(){

        System.out.println("Creating a Word Document");

    }

}

***PDFDocument.java:***

public class PDFDocument implements Document{

    //Concrete Class for PDF document

    public void create(){

        System.out.println("Creating a PDF Document");

    }

}

***ExcelDocument.java:***

public class ExcelDocument implements Document{

    //Concrete Class for Word document

    public void create(){

        System.out.println("Creating a Excel Document");

    }

}

***DocumentFactory.java:***

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

//Abstract class with an abstract method for concrete classes to implement

***WordDocumentFactory.java:***

public class WordDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }

}//Simple extended factory for Word Document

***PDFDocumentFactory.java:***

public class PDFDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new PDFDocument();

    }

}//Simple extended factory for PDF Document

***ExcelDocumentFactory.java:***

public class ExcelDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new ExcelDocument();

    }

}//Simple extended factory for Excel Document

***TestingClass.java:***

public class TestingClass{

    public static void main(String[] args) {

        //As we have already created individual factory for each type of document,thus we can just simply get the object through them.

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.create();

        DocumentFactory pdfFactory = new PDFDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.create();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.create();

    }

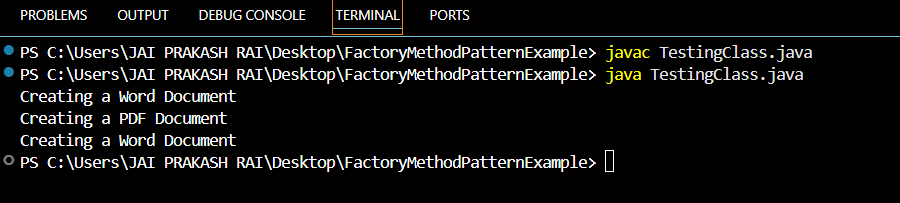
    /\*I could have preferred the way in which we just a single main factory

    that is responsible to make instances according to the need,but here in the question it

    is clearly mentioned that we need to form individual factories,so I have preferred this way.\*/

}

**Output:**



**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

**Source Code:**

***Computer.java:***

public class Computer {

    // COnsidering Required

    private final String cpu;

    // Optional that is the main point of using builder

    private final String ram;

    private final String storage;

    private final String graphicsCard;

    private final String operatingSystem;

    // Private constructor

    private Computer(Builder builder) {

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

        this.operatingSystem = builder.operatingSystem;

    }

    // Builder Static Nested Class

    public static class Builder {

        private final String cpu; // Required

        private String ram;

        private String storage;

        private String graphicsCard;

        private String operatingSystem;

        public Builder(String cpu) {

            this.cpu = cpu;

        }

        public Builder setRam(String ram) {

            this.ram = ram;

            return this;

        }

        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.operatingSystem = operatingSystem;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

    @Override

    public String toString() {

        return "Computer Specs:\n" +

               "CPU: " + cpu + "\n" +

               "RAM: " + (ram != null ? ram : "N/A") + "\n" +

               "Storage: " + (storage != null ? storage : "N/A") + "\n" +

               "Graphics Card: " + (graphicsCard != null ? graphicsCard : "N/A") + "\n" +

               "Operating System: " + (operatingSystem != null ? operatingSystem : "N/A");

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        // Full configuration for Gaming

        Computer gamingPC = new Computer.Builder("Intel i9")

                .setRam("32GB")

                .setStorage("2TB SSD")

                .setGraphicsCard("NVIDIA RTX 4080")

                .setOperatingSystem("Windows 11 Pro")

                .build();

        // Partial configuration to show use with optional

        Computer officePC = new Computer.Builder("Intel i5")

                .setRam("8GB")

                .setStorage("512GB SSD")

                .build();

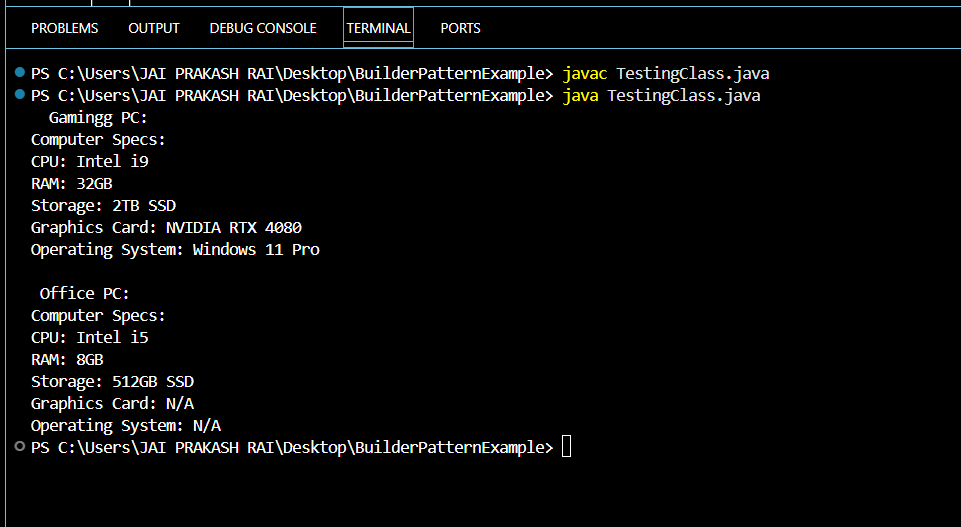
        System.out.println("  Gamingg PC:\n" + gamingPC);

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

    }

}

***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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

**Source Code:**

***PaymentProcessor.java:***

public interface PaymentProcessor {

    void processPayment(double amount);

}

***PayPalGateway.java:***

//Adaptee!

public class PayPalGateway {

    public void makePayment(double amountInDollars) {

        System.out.println("Paid ₹" + amountInDollars + " using PayPal.");

    }

}

***RazorPayGateway.java:***

//Adaptee

public class RazorPayGateway {

    public void pay(double money) {

        System.out.println("Charged ₹" + money + " through Stripe.");

    }

}

***PayPalAdapter.java:***

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPal;

    //Constructor

    public PayPalAdapter(PayPalGateway payPal) {

        this.payPal = payPal;

    }

    @Override

    public void processPayment(double amount) {

        payPal.makePayment(amount);

    }

}

***RazorPayAdapter.java:***

public class RazorPayAdapter implements PaymentProcessor {

    private RazorPayGateway stripe;

    //Constructor

    public RazorPayAdapter(RazorPayGateway stripe) {

        this.stripe = stripe;

    }

    @Override

    public void processPayment(double amount) {

        stripe.pay(amount);

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        // Using PayPal

        PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

        paypalProcessor.processPayment(250.00);

        // Using RazorPay

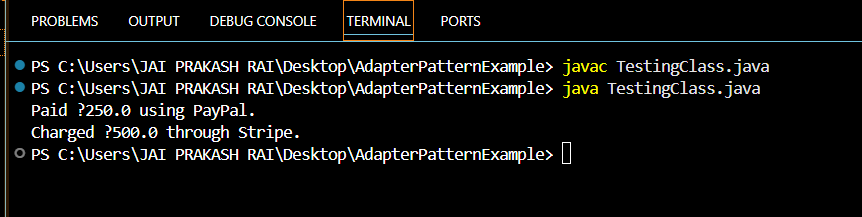
        PaymentProcessor stripeProcessor = new RazorPayAdapter(new RazorPayGateway());

        stripeProcessor.processPayment(500.00);

    }

}

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   * Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   * Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

**Source 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 SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        // Basic notifier

        Notifier baseNotifier = new EmailNotifier();

        // Decorate with SMS

        Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

        // Further decorate with Slack

        Notifier multiNotifier = new SlackNotifierDecorator(smsNotifier);

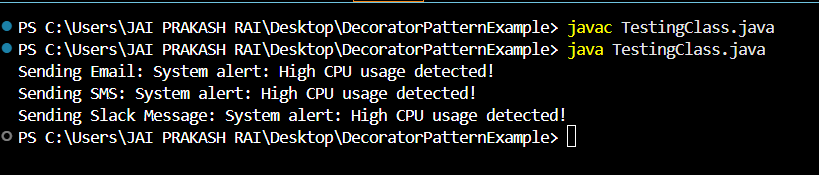
        // Send message through all channels

        multiNotifier.send("System alert: High CPU usage detected!");

    }

}

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

**Source 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;

        loadFromRemoteServer();

    }

    private void loadFromRemoteServer() {

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

        try {

            Thread.sleep(1000); // Simulate 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) {

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

        } else {

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

        }

        realImage.display();

    }

}

***TestingClass.java:***

public class TestingClass{

    public static void main(String[] args) {

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

        System.out.println("First display call:");

        image1.display(); // Loads from remote

        System.out.println("\nSecond display call:");

        image1.display(); // Uses cached

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

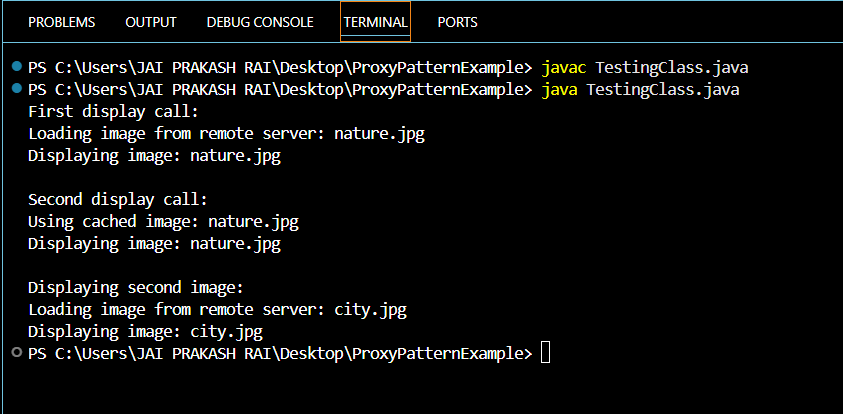
        System.out.println("\nDisplaying second image:");

        image2.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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

**Source Code:**

***Stock.java:***

public interface Stock {

    void registerObserver(Observer observer);

    void removeObserver(Observer observer);

    void notifyObservers();

}

***StockMarket.java:***

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

    private double stockPrice;

    public void setStockPrice(double newPrice) {

        this.stockPrice = newPrice;

        System.out.println("\n[StockMarket] Stock price updated to $" + newPrice);

        notifyObservers();

    }

    @Override

    public void registerObserver(Observer observer) {

        observers.add(observer);

        System.out.println("[StockMarket] Registered " + observer.getClass().getSimpleName());

    }

    @Override

    public void removeObserver(Observer observer) {

        observers.remove(observer);

        System.out.println("[StockMarket] Removed " + observer.getClass().getSimpleName());

    }

    @Override

    public void notifyObservers() {

        for (Observer obs : observers) {

            obs.update(stockPrice);

        }

    }

}

***Observer.java:***

public interface Observer {

    void update(double price);

}

***MobileApp.java:***

public class MobileApp implements Observer {

@Override

public void update(double price) {

System.out.println("[MobileApp] Stock price updated to ₹" + price);

}

}

***WebApp.java:***

public class WebApp implements Observer {

    @Override

    public void update(double price) {

        System.out.println("[WebApp] Stock price updated to ₹" + price);

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobile = new MobileApp();

        Observer web = new WebApp();

        stockMarket.registerObserver(mobile);

        stockMarket.registerObserver(web);

        stockMarket.setStockPrice(100.50);

        stockMarket.setStockPrice(102.75);

        // Remove mobile app to show how we can simple deregister

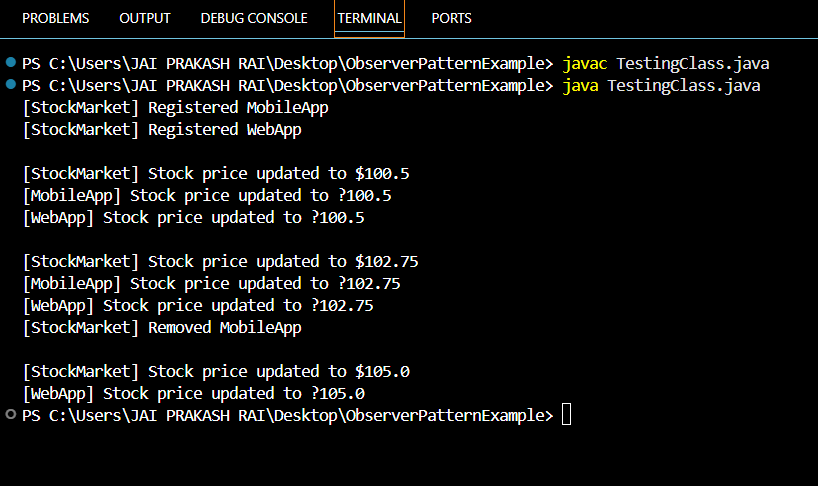
        stockMarket.removeObserver(mobile);

        stockMarket.setStockPrice(105.00);

    }

}

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**

* Create a test class to demonstrate selecting and using different payment strategies.

**Source 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 ₹" + amount + " using Credit Card [" + cardNumber + "]");

    }

}

***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 ₹" + amount + " using PayPal account [" + email + "]");

    }

}

***PaymentContext.java:***

public class PaymentContext {

    private PaymentStrategy strategy;

    // Set strategy dynamically

    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    // Execute selected strategy

    public void payAmount(double amount) {

        if (strategy == null) {

            System.out.println("Please set a payment method.");

        } else {

            strategy.pay(amount);

        }

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        // Pay using Credit Card

        context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));

        context.payAmount(150.00);

        // Switch to PayPal

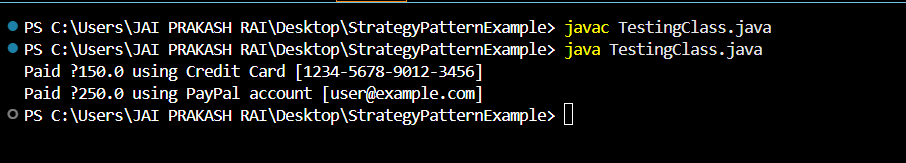
        context.setPaymentStrategy(new PayPalPayment("user@example.com"));

        context.payAmount(250.00);

    }

}

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

**Source Code:-**

***Command.java:***

public interface Command {

    void execute();

}

***Light.java:***

public class Light {

    public void turnOn() {

        System.out.println("💡 Light is ON");

    }

    public void turnOff() {

        System.out.println("💡 Light is OFF");

    }

}

***LightOffCommand.java:***

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

***LightOnCommand.java:***

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

***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 set.");

        }

    }

}

***TestingClass.java:***

public class TestingClass{

    public static void main(String[] args) {

        // Receiver

        Light livingRoomLight = new Light();

        // Concrete commands

        Command lightOn = new LightOnCommand(livingRoomLight);

        Command lightOff = new LightOffCommand(livingRoomLight);

        // Invoker

        RemoteControl remote = new RemoteControl();

        // Press ON

        remote.setCommand(lightOn);

        remote.pressButton();

        // Press OFF

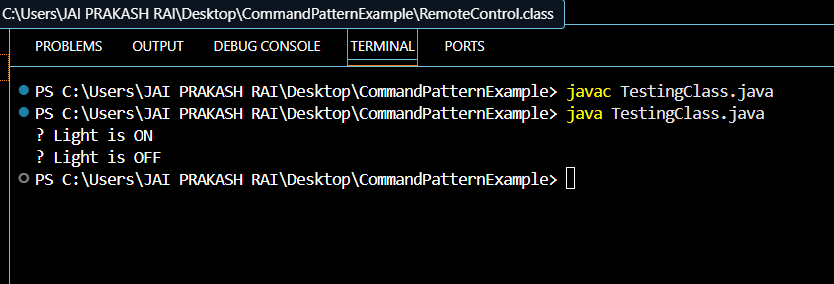
        remote.setCommand(lightOff);

        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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
   * Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

**Source 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;

    }

    // Getters and setters

    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("📄 Student Details:");

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

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

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

    }

}

***StudentController.java:***

public class StudentController {

    private Student student;

    private StudentView view;

    public StudentController(Student student, StudentView view) {

        this.student = student;

        this.view = view;

    }

    // Model getters and setters via Controller

    public void setStudentName(String name) {

        student.setName(name);

    }

    public void setStudentGrade(String grade) {

        student.setGrade(grade);

    }

    public String getStudentName() {

        return student.getName();

    }

    public String getStudentId() {

        return student.getId();

    }

    public String getStudentGrade() {

        return student.getGrade();

    }

    public void updateView() {

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

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        // Model

        Student student = new Student("Jai", "CSE52", "B+");

        // View

        StudentView view = new StudentView();

        // Controller

        StudentController controller = new StudentController(student, view);

        // Display initial data

        controller.updateView();

        System.out.println("\n🔁 Updating student grade to B+...");

        controller.setStudentGrade("A+");

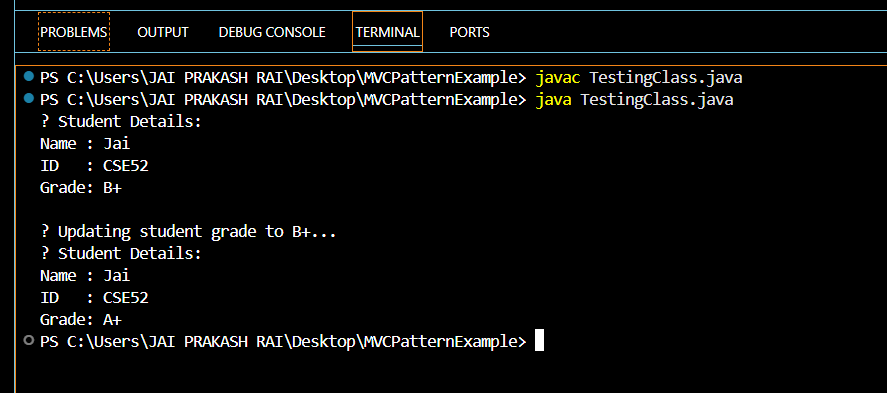
        // Display updated data

        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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   * Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

**Source 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; }

}

***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> customerDB = new HashMap<>();

    public CustomerRepositoryImpl() {

        // Simulate database

        customerDB.put("101", new Customer("101", "Shahid"));

        customerDB.put("102", new Customer("102", "Nakit"));

    }

    @Override

    public Customer findCustomerById(String id) {

        return customerDB.get(id);

    }

}

***CustomerService.java:***

public class CustomerService {

    private CustomerRepository repository;

    // Constructor Injection

    public CustomerService(CustomerRepository repository) {

        this.repository = repository;

    }

    public void displayCustomer(String id) {

        Customer customer = repository.findCustomerById(id);

        if (customer != null) {

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

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

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

        } else {

            System.out.println("Customer not found.");

        }

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        // Inject dependency

        CustomerRepository repository = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repository); // DI via constructor

        // Use service

        service.displayCustomer("101");

        service.displayCustomer("999"); // Invalid ID

    }

}

**Output:**

