**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.

**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.

**1.Create a New Java Project:**

Create a new Java project named SingletonPatternExample in your preferred IDE

**2.Define a Singleton Class:**

public class Logger

{

private static Logger instance;

private Logger()

{ }

**3. Implement the Singleton Pattern:**

public static Logger getInstance()

{

if (instance == null)

{

synchronized (Logger.class)

{

if (instance == null)

{

instance = new Logger();

}

}

}

return instance;

}

public void log(String message)

{

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

}

}

4.**Test the Singleton Implementation:**

public class SingletonPatternTest

{

public static void main(String[] args)

{

Logger logger1 = Logger.getInstance();

logger1.log(" hi, this is first message");

Logger logger2 = Logger.getInstance();

logger2.log(" hi, this is second message");

if (logger1 == logger2)

{

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

} else {

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

}

}

}

**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.

**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.

**1.Create a New Java Project:**

Create a new Java project named FactoryMethodPatternExample.

**2.Define Document Classes:**

abstract class Document

{

public abstract void open();

public abstract void close();

public abstract void save();

}

**3.Create Concrete Document Classes:**

class WordDocument extends Document {

@Override

public void open() {

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

}

@Override

public void close() {

System.out.println("Closing Word document...");

}

@Override

public void save() {

System.out.println("Saving Word document...");

}

}

class PdfDocument extends Document

{

@Override

public void open()

{

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

}

@Override

public void close()

{

System.out.println("Closing PDF document...");

}

@Override

public void save()

{

System.out.println("Saving PDF document...");

}

}

class ExcelDocument extends Document {

@Override

public void open() {

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

}

@Override

public void close() {

System.out.println("Closing Excel document...");

}

@Override

public void save() {

System.out.println("Saving Excel document...");

}

}

**4.Implement the Factory Method:**

abstract class DocumentFactory

{

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory

{

@Override

public Document createDocument()

{

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory

{

@Override

public Document createDocument()

{

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory

{

@Override

public Document createDocument()

{

return new ExcelDocument();

}

}

**5.Test the Factory Method Implementation:**

public class FactoryMethodPatternExample

{

public static void main(String[] args)

{

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

wordDoc.save();

wordDoc.close();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

pdfDoc.save();

pdfDoc.close();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

excelDoc.save();

excelDoc.close();

}

}

**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.

**STEP1: Create a New Java Project:**

Create a new Java project named **BuilderPatternExample**.

**STEP2: Define a Product Class:**

**Class Computer:**

package Builder\_example;

public class Computer {

    private String cpu;

    private String ram;

    private String storage;

    private String operatingsystem;

    public Computer(Builder build) {

        this.cpu=build.cpu;

        this.ram=build.ram;

        this.storage=build.storage;

        this.operatingsystem=build.operatingsystem;

    }

    public String setCPU() {

        return cpu;

    }

    public String setram() {

        return ram;

    }

    public String setStorage() {

        return storage;

    }

    public String setOS() {

        return operatingsystem;

    }

    public String toString() {

        return "Computer [CPU=" + cpu + ", RAM=" + ram + ", storage=" + storage +",OS="+operatingsystem+"]";

    }

**STEP3: Implement the Builder Class:**

public static class Builder {

    private String cpu;

    private String ram;

    private String storage;

    private String operatingsystem;

    public Builder() {

    }

    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 Builder setOS(String os) {

        this.operatingsystem=os;

        return this;

    }

    public Computer build() {

        return new Computer(this);

    }

}

}

**STEP4: Implement the Builder Pattern:**

    public Computer(Builder build) {

        this.cpu=build.cpu;

        this.ram=build.ram;

        this.storage=build.storage;

        this.operatingsystem=build.operatingsystem;

    }

**STEP5: Test the Builder Implementation:**

**Test\_builder Class:**

package Builder\_example;

public class Test\_builder {

    public static void main(String[] args) {

        Computer gamingCp = new Computer.Builder()

                .setCPU("Intel Core i9")

                .setRAM("32GB")

                .setStorage("1TB SSD")

                .setOS("Windows 11")

                .build();

        Computer officeCp = new Computer.Builder()

                .setCPU("Intel Core i5")

                .setRAM("16GB")

                .setStorage("512GB SSD")

                .setOS("Windows 10")

                .build();

        Computer budgetCp = new Computer.Builder()

                .setCPU("AMD Ryzen 3")

                .setRAM("8GB")

                .setStorage("256GB SSD")

                .setOS("Linux")

                .build();

        // Print the computer built

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

        System.out.println("Office Computer: " + officeCp);

        System.out.println("Budget Computer: " + budgetCp);

    }

}

**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.

**Solution**

**Step 1: Create a New Java Project**

Create a new Java project named AdapterPatternExample.

**Step 2: Define Target Interface**

// PaymentProcessor.java

public interface PaymentProcessor {

void processPayment(double amount);

}

**Step 3: Implement Adaptee Classes**

// Paypal.java

public class Paypal {

public void sendPayment(double amount) {

System.out.println("Paypal: Sending payment of Rs." + amount);

}

}

// Stripe.java

public class Stripe {

public void makePayment(double amount) {

System.out.println("Stripe: Making payment of Rs." + amount);

}

}

// Square.java

public class Square {

public void executePayment(double amount) {

System.out.println("Square: Making payment of Rs." + amount);

}

}

**Step 4: Implement the Adapter Classes**

// PaypalAdapter.java

public class PaypalAdapter implements PaymentProcessor {

private Paypal paypal;

public PaypalAdapter(Paypal paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

// StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private Stripe stripe;

public StripeAdapter(Stripe stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

// SquareAdapter.java

public class SquareAdapter implements PaymentProcessor {

private Square square;

public SquareAdapter(Square square) {

this.square = square;

}

@Override

public void processPayment(double amount) {

square.executePayment(amount);

}

}

**Step 5: Create AdapterPatternExample Class**

// AdapterPatternExample.java

public class AdapterPatternExample {

public static void main(String[] args) {

AdapterPatternExample example = new AdapterPatternExample();

example.run();

}

public void run() {

Paypal paypal = new Paypal();

PaymentProcessor paypalProcessor = new PaypalAdapter(paypal);

processPayment(paypalProcessor, 150.00);

Stripe stripe = new Stripe();

PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

processPayment(stripeProcessor, 250.00);

Square square = new Square();

PaymentProcessor squareProcessor = new SquareAdapter(square);

processPayment(squareProcessor, 350.00);

}

private void processPayment(PaymentProcessor processor, double amount) {

processor.processPayment(amount);

}

}

**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**.

1. **Define Component Interface:**

Create an interface **Notifier** with a method **send()**.

1. **Implement Concrete Component:**

Create a class **EmailNotifier** that implements Notifier.

1. **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**.

1. **Test the Decorator Implementation:**

Create a test class to demonstrate sending notifications via multiple channels using decorators.

**Code:**

**Step 1: Create a New Java Project**

Create a new Java project named `DecoratorPatternPgm`.

**Step 2: Define Component Interface**

public interface Notifier {

void send(String message);

}

**Step 3:** Implement Concrete Component

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

**Step 4:** Implement Decorator Classes

//Abstract Decorator Class

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);

}

}

//Concrete Decorator Classes

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

**Step 5:** Test the Decorator Implementation

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

// Wrap the base notifier with SMS notifier

notifier = new SMSNotifierDecorator(notifier);

notifier = new SlackNotifierDecorator(notifier);

notifier.send("Hello! This is a test notification.");

}

}

**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 image.

1. **Create a New Java Project:**

Create a new Java project named **ProxyPatternExample**.

**2.Define Subject Interface:**

// Image.java

public interface Image {

void display();

}

**3.Implement Real Subject Class:**

// RealImage.java

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk(fileName);

}

private void loadFromDisk(String fileName) {

System.out.println("Loading " + fileName);

// Simulate a delay in loading the image from a remote server

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public void display() {

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

}

}

**4. Implement Proxy Class:**

// ProxyImage.java

public class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName);

}

realImage.display();

}

}

5. **Test the Proxy Implementation:**

// ProxyPatternDemo.java

public class ProxyPatternDemo {

public static void main(String[] args) {

Image image1 = new ProxyImage("test\_1.jpg");

Image image2 = new ProxyImage("test\_2.jpg");

// Image will be loaded from disk

image1.display();

System.out.println("");

// Image will not be loaded from disk

image1.display();

System.out.println("");

// Image will be loaded from disk

image2.display();

System.out.println("");

// Image will not be loaded from disk

image2.display();

}

}

**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

1. **Create a New Java Project:**

Create a new Java project named **ObserverPatternExample**.

1. **Define Subject Interface:**

public interface Stock {

void registerObserver(Observer o);

void deregisterObserver(Observer o);

void notifyObservers();

}

1. **Implement Concrete Subject:**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers;

private String stockName;

private double stockPrice;

public StockMarket() {

observers = new ArrayList<>();

}

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 deregisterObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

1. **Define Observer Interface:**

public interface Observer {

void update(String stockName, double stockPrice);

}

1. **Implement Concrete Observers:**

// MobileApp.java

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double stockPrice) {

System.out.println(appName + " received update: " + stockName + " is now $" + stockPrice);

}

}

//WebApp.java

public class WebApp implements Observer {

private String appName;

public WebApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double stockPrice) {

System.out.println(appName + " received update: " + stockName + " is now $" + stockPrice);

}

}

1. **Test the Observer Implementation:**

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp("MobileApp");

Observer webApp = new WebApp("WebApp");

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

stockMarket.setStock("Apple", 150.50);

stockMarket.setStock("Google", 2800.75);

stockMarket.deregisterObserver(mobileApp);

stockMarket.setStock("Amazon", 3400.10);

}

}

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

**1.Create a New Java Project:**

* 1. Create a new Java project named StrategyPatternExample.

**2. Define Strategy Interface:**

**PaymentStrategy.java:**

public interface PaymentStrategy {

    void pay(double amount);

}

**3.Implement Concrete Strategies:**

**PayPalPayment,java**

public class PayPalPayment implements PaymentStrategy {

    String email;

    String password;

    PayPalPayment(String email, String password) {

        this.email = email;

        this.password = password;

    }

    public void pay(double amount) {

        System.out.println("Paid " + amount + " using Paypal");

    }

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

    String cardnumber;

    String cardHolderNAme;

    String cvv;

    String expiryDate;

    CreditCardPayment(String cardnumber, String cardHolderName, String cvv, String expiryDate) {

        this.cardnumber = cardnumber;

        this.cardHolderNAme = cardHolderName;

        this.cvv = cvv;

        this.expiryDate = expiryDate;

    }

    public void pay(double amount) {

        System.out.println("Paid " + amount + " using credit card");

    }

}

**4.Implement Context class:**

**PaymentContext.java**

public class PaymentContext {

    private PaymentStrategy paymentstrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentstrategy = paymentStrategy;

    }

    public void pay(double amount) {

        if (paymentstrategy != null) {

            paymentstrategy.pay(amount);

        } else {

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

        }

    }

}

1. **Test the Strategy Implementation:**

**Main.java**

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner scanner = **new** Scanner(System.in);

        PaymentContext paymentContext = **new** PaymentContext();

        try {

            System.out.println("Choose payment method (1: Credit Card, 2: PayPal):");

            int choice = scanner.nextInt();

            scanner.nextLine();

            System.out.println("Enter amount to pay:");

            double amount = scanner.nextDouble();

            scanner.nextLine();

            switch (choice) {

                case 1:

                    System.out.println("Enter card number:");

                    String cardNumber = scanner.nextLine();

                    System.out.println("Enter card holder name:");

                    String cardHolderName = scanner.nextLine();

                    System.out.println("Enter CVV:");

                    String cvv = scanner.nextLine();

                    System.out.println("Enter expiry date (MM/YY):");

                    String expiryDate = scanner.nextLine();

                    PaymentStrategy creditCardPayment = **new** CreditCardPayment(cardNumber, cardHolderName, cvv,

                            expiryDate);

                    paymentContext.setPaymentStrategy(creditCardPayment);

                    break;

                case 2:

                    System.out.println("Enter PayPal email:");

                    String email = scanner.nextLine();

                    System.out.println("Enter PayPal password:");

                    String password = scanner.nextLine();

                    PaymentStrategy payPalPayment = **new** PayPalPayment(email, password);

                    paymentContext.setPaymentStrategy(payPalPayment);

                    break;

                default:

                    System.out.println("Invalid choice");

                    return;

            }

            paymentContext.pay(amount);

        } finally {

            scanner.close();

        }

    }

}

**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**.

1. **Create a New Java Project:**

Create a new Java project named **CommandPatternExample**.

**2.Define Command Interface:**

// Command.java

interface Command {

void execute();

}

3. **Implement Concrete Commands:**

// LightOnCommand

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

// LightOffCommand

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

4. **Implement Invoker Class:**

// RemoteControl.java

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

5. **Implement Receiver Class:**

// Light

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

6.**Test the Command Implementation:**

public class CommandPatternExample {

public static void main(String[] args) {

Light livingRoomLight = new Light();

// Create concrete commands

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

// Create invoker

RemoteControl remote = new RemoteControl();

// Turn on the light

remote.setCommand(lightOn);

remote.pressButton();

// Turn off the light

remote.setCommand(lightOff);

remote.pressButton();

}

}

**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**.

**1. Create a New Java Project:**

Create a new Java project named **MVCPatternExample**.

**2.Define Model Class:**

class Student {

private String name;

private int id;

private String grade;

public Student(String name, int 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 int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

3. **Define View Class:**

class StudentView {

public void displayStudentDetails(String studentName, int studentId, String studentGrade) {

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

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

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

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

}

}

**4. Define Controller Class:**

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 String getStudentName() {

return model.getName();

}

public void setStudentId(int id) {

model.setId(id);

}

public int getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

**5. Test the MVC Implementation:**

public class Main {

public static void main(String[] args) {

Student student = new Student("John", 12, "A+");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Jane");

controller.setStudentId(34);

controller.setStudentGrade("A");

controller.updateView();

}

}

**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.

**1.Create a New Java Project:**

Create a new Java project named **DependencyInjectionExample**

**2. Define Repository Interface**

// CustomerRepository.java

public interface CustomerRepository {

Customer findCustomerById(int id);

}

3.**Implement Concrete Repository**

// CustomerRepositoryImpl.java

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(int id) {

return new Customer(id, "John Doe");

}

}

**4.Define Service Class**

// CustomerService.java

public class CustomerService {

private final CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public Customer getCustomerById(int id) {

return customerRepository.findCustomerById(id);

}

}

5.**Implement Dependency Injection:**

// Customer.java

public class Customer {

private int id;

private String name;

public Customer(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public String toString() {

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

}

}

**6.Test the Dependency Injection Implemtation**

// Main.java

public class Main {

public static void main(String[] args) {

CustomerRepository customerRepository = new CustomerRepositoryImpl();

CustomerService customerService = new CustomerService(customerRepository);

Customer customer = customerService.getCustomerById(1);

System.out.println(customer);

}

}