**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**.

Created a new Java project named **SingletonPatternExample** on Eclipse IDE.

1. **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.

// Logger.java

// A singleton class representing a logging utility.

public class Logger {

// Private static instance of the Logger class

private static Logger instance;

// Private constructor to prevent instantiation from outside

private Logger() {}

// Public static method to get the instance of the Logger class

public static Logger getInstance() {

if (instance == null) {

// Create a new instance if it doesn't exist

instance = new Logger();

}

return instance;

}

// Example logging method

public void log(String message) {

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

}

}

1. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.

// Thread-safe getInstance() method using double-checked locking

public static Logger getInstance() {

if (instance == null) {

synchronized (Logger.class) {

if (instance == null) {

instance = new Logger();

}

}

}

return instance;

}

4. **Test the Singleton Implementation:**

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

// LoggerTest.java

public class LoggerTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

// Both logger1 and logger2 should refer to the same instance

System.out.println(logger1 == logger2); // Expected output: true

logger1.log("Hello, World!"); // Should print "Logged: Hello, World!"

}

}

**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**.

Created a new Java project named **FactoryMethodPatternExample** on Eclipse IDE.

1. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.

// Document.java

// Interface representing a document.

public interface Document {

void open();

void save();

void close();

}

1. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.

// WordDocument.java

// Represents a Word document.

public class WordDocument implements Document {

@Override

public void open() {

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

}

@Override

public void save() {

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

}

@Override

public void close() {

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

} }

// PdfDocument.java

// Represents a PDF document.

public class PdfDocument implements Document {

@Override

public void open() {

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

}

@Override

public void save() {

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

}

@Override

public void close() {

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

}

}

// ExcelDocument.java

// Represents an Excel document.

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

@Override

public void save() {

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

}

@Override

public void close() {

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

} }

1. **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.

// DocumentFactory.java

// Abstract factory class for creating documents.

public abstract class DocumentFactory {

public abstract Document createDocument();

}

// WordDocumentFactory.java

// Factory class for creating Word documents.

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

// PdfDocumentFactory.java

// Factory class for creating PDF documents.

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

// ExcelDocumentFactory.java

// Factory class for creating Excel documents.

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument(); } }

1. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

// DocumentFactoryTest.java

public class DocumentFactoryTest {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDocument = wordFactory.createDocument();

wordDocument.open();

wordDocument.save();

wordDocument.close();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDocument = pdfFactory.createDocument();

pdfDocument.open();

pdfDocument.save();

pdfDocument.close();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDocument = excelFactory.createDocument();

excelDocument.open();

excelDocument.save();

excelDocument.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**.

Created a new Java project named **BuilderPatternExample** on Eclipse IDE.

1. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.

// Computer.java

// Represents a computer with various attributes.

public class Computer {

private String cpu;

private int ram;

private int storage;

private String graphicsCard;

private String operatingSystem;

// Private constructor to prevent direct instantiation

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.operatingSystem = builder.operatingSystem;

}

// toString() method for easy printing

@Override

public String toString() {

return "Computer{" +

"cpu='" + cpu + '\'' +

", ram=" + ram +

", storage=" + storage +

", graphicsCard='" + graphicsCard + '\'' +

", operatingSystem='" + operatingSystem + '\'' +

'}';

}

}

1. **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.

// Added this to the Computer.java file

public static class Builder {

private String cpu;

private int ram;

private int storage;

private String graphicsCard;

private String operatingSystem;

// Mandatory attribute

public Builder(String cpu) {

this.cpu = cpu;

}

// Optional attributes

public Builder withRam(int ram) {

this.ram = ram;

return this;

}

public Builder withStorage(int storage) {

this.storage = storage;

return this;

}

public Builder withGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder withOperatingSystem(String operatingSystem) {

this.operatingSystem = operatingSystem;

return this;

}

// build() method to create a Computer instance

public Computer build() {

return new Computer(this);

}

}

1. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.

This is already done in Step 2.

1. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

// ComputerBuilderTest.java

public class ComputerBuilderTest {

public static void main(String[] args) {

// Create a basic computer

Computer basicComputer = new Computer.Builder("Intel Core i3")

.withRam(8)

.withStorage(256)

.build();

System.out.println("Basic Computer: " + basicComputer);

// Create a gaming computer

Computer gamingComputer = new Computer.Builder("AMD Ryzen 9")

.withRam(16)

.withStorage(512)

.withGraphicsCard("NVIDIA GeForce RTX 3080")

.withOperatingSystem("Windows 11")

.build();

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

// Create a server computer

Computer serverComputer = new Computer.Builder("Intel Xeon")

.withRam(32)

.withStorage(1024)

.withOperatingSystem("Ubuntu Server")

.build();

System.out.println("Server Computer: " + serverComputer);

}

}

**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**.

Created a new Java project named **AdapterPatternExample** on Eclipse IDE.

1. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.

// PaymentProcessor.java

// Target interface for payment processing.

public interface PaymentProcessor {

void processPayment(double amount);

}

1. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.

// Paypal.java

// Represents the PayPal payment gateway.

public class Paypal {

private String email;

private String password;

public Paypal(String email, String password) {

this.email = email;

this.password = password;

}

public void pay(double amount) {

// Implementation to pay using PayPal

System.out.println("Paying " + amount + " using PayPal.");

}

}

// Stripe.java

// Represents the Stripe payment gateway.

public class Stripe {

private String apiKey;

public Stripe(String apiKey) {

this.apiKey = apiKey;

}

public void charge(double amount) {

// Implementation to charge using Stripe

System.out.println("Charging " + amount + " using Stripe.");

}

}

1. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.

// PaypalAdapter.java

// Adapter for PayPal payment gateway.

public class PaypalAdapter implements PaymentProcessor {

private Paypal paypal;

public PaypalAdapter(String email, String password) {

this.paypal = new Paypal(email, password);

}

@Override

public void processPayment(double amount) {

paypal.pay(amount);

}

}

// StripeAdapter.java

// Adapter for Stripe payment gateway.

public class StripeAdapter implements PaymentProcessor {

private Stripe stripe;

public StripeAdapter(String apiKey) {

this.stripe = new Stripe(apiKey);

}

@Override

public void processPayment(double amount) {

stripe.charge(amount);

}

}

1. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

// PaymentProcessorTest.java

public class PaymentProcessorTest {

public static void main(String[] args) {

// Pay using PayPal

PaymentProcessor paypalAdapter = new PaypalAdapter("john.doe@example.com", "secret");

paypalAdapter.processPayment(100);

// Pay using Stripe

PaymentProcessor stripeAdapter = new StripeAdapter("sk\_test\_12345");

stripeAdapter.processPayment(100);

}

}

**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**.

Created a new Java project named **DecoratorPatternExample** on Eclipse IDE.

1. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.

// Notifier.java

// Component interface for sending notifications.

public interface Notifier {

void send(String message);

}

1. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.

// EmailNotifier.java

// Concrete component for sending email notifications.

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

1. **Implement Decorator Classes:**
   * Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.

// NotifierDecorator.java

// Abstract decorator class for adding notification channels.

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message);

}

}

* + Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.

// SMSNotifierDecorator.java

// Concrete decorator for sending SMS notifications.

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 notification: " + message);

}

}

// SlackNotifierDecorator.java

// Concrete decorator for sending Slack notifications.

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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

} }

1. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

// NotifierTest.java

public class NotifierTest {

public static void main(String[] args) {

// Send notification via email only

Notifier emailNotifier = new EmailNotifier();

emailNotifier.send("Hello, this is an email notification.");

// Send notification via email and SMS

Notifier emailAndSMSNotifier = new SMSNotifierDecorator(emailNotifier);

emailAndSMSNotifier.send("Hello, this is an email and SMS notification.");

// Send notification via email, SMS, and Slack

Notifier emailSMSAndSlackNotifier = new SlackNotifierDecorator(emailAndSMSNotifier);

emailSMSAndSlackNotifier.send("Hello, this is an email, SMS, and Slack 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**.

Created a new Java project named **ProxyPatternExample** on Eclipse IDE.

1. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.

// Image.java

// Subject interface for displaying images.

public interface Image {

void display();

}

1. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.

// RealImage.java

// Real subject class that loads an image from a remote server.

public class RealImage implements Image {

private String url;

public RealImage(String url) {

this.url = url;

loadFromServer();

}

private void loadFromServer() {

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

// Simulate loading time

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

@Override

public void display() {

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

}

}

1. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.

// ProxyImage.java

import java.util.HashMap;

import java.util.Map;

// Proxy class that implements lazy initialization and caching.

public class ProxyImage implements Image {

private String url;

private RealImage realImage;

private static Map<String, RealImage> cache = new HashMap<>();

public ProxyImage(String url) {

this.url = url;

}

@Override

public void display() {

if (realImage == null) {

realImage = cache.get(url);

if (realImage == null) {

realImage = new RealImage(url);

cache.put(url, realImage);

}

}

realImage.display();

}

}

1. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

// ImageTest.java

public class ImageTest {

public static void main(String[] args) {

// Load and display an image for the first time

Image image1 = new ProxyImage("https://example.com/image1.jpg");

image1.display();

// Load and display the same image again (cached)

Image image2 = new ProxyImage("https://example.com/image1.jpg");

image2.display();

// Load and display a different image

Image image3 = new ProxyImage("https://example.com/image2.jpg");

image3.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**.

Created a new Java project named **ObserverPatternExample** on Eclipse IDE.

1. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.

// Stock.java

// Subject interface for stock market.

public interface Stock {

void register(Observer observer);

void deregister(Observer observer);

void notifyObservers(String stockName, double newPrice);

}

1. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.

// StockMarket.java

import java.util.ArrayList;

import java.util.List;

// Concrete subject class that maintains a list of observers.

public class StockMarket implements Stock {

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

@Override

public void register(Observer observer) {

observers.add(observer);

}

@Override

public void deregister(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers(String stockName, double newPrice) {

for (Observer observer : observers) {

observer.update(stockName, newPrice);

}

}

public void setStockPrice(String stockName, double newPrice) {

System.out.println("Stock price updated: " + stockName + " = " + newPrice);

notifyObservers(stockName, newPrice);

}

}

1. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**

// Observer.java

// Observer interface for stock market observers.

public interface Observer {

void update(String stockName, double newPrice);

}

1. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.

// MobileApp.java

// Concrete observer class for mobile app.

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(String stockName, double newPrice) {

System.out.println(name + " received update: " + stockName + " = " + newPrice);

}

}

// WebApp.java

// Concrete observer class for web app.

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(String stockName, double newPrice) {

System.out.println(name + " received update: " + stockName + " = " + newPrice);

}

}

1. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

// ObserverTest.java

public class ObserverTest {

public static void main(String[] args) {

// Create a stock market

StockMarket stockMarket = new StockMarket();

// Create mobile and web apps

Observer mobileApp1 = new MobileApp("MobileApp1");

Observer mobileApp2 = new MobileApp("MobileApp2");

Observer webApp1 = new WebApp("WebApp1");

// Register mobile and web apps to the stock market

stockMarket.register(mobileApp1);

stockMarket.register(mobileApp2);

stockMarket.register(webApp1);

// Update stock prices

stockMarket.setStockPrice("GOOG", 1200.50);

stockMarket.setStockPrice("AAPL", 200.75);

// Unregister mobileApp1

stockMarket.deregister(mobileApp1);

// Update stock prices again

stockMarket.setStockPrice("GOOG", 1210.00);

stockMarket.setStockPrice("AAPL", 201.25);

}

}

**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**.

Created a new Java project named **StrategyPatternExample** on Eclipse IDE.

1. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.

// PaymentStrategy.java

// Strategy interface for payment methods.

public interface PaymentStrategy {

void pay(double amount);

}

1. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.

// CreditCardPayment.java

// Concrete strategy class for credit card payment.

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolderName;

public CreditCardPayment(String cardNumber, String cardHolderName) {

this.cardNumber = cardNumber;

this.cardHolderName = cardHolderName;

}

@Override

public void pay(double amount) {

System.out.println("Paying $" + amount + " using credit card " + cardNumber + " of " + cardHolderName);

}

}

// PayPalPayment.java

// Concrete strategy class for PayPal payment.

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

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

}

}

1. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.

// PaymentContext.java

// Context class for payment processing.

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public PaymentContext(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void executePayment(double amount) {

paymentStrategy.pay(amount);

}

}

1. **Test the Strategy Implementation:**
   * Create a test class to demonstrate selecting and using different payment strategies.

// StrategyTest.java

public class StrategyTest {

public static void main(String[] args) {

// Create payment context

PaymentContext paymentContext = new PaymentContext(null);

// Create payment strategies

PaymentStrategy creditCardPayment = new CreditCardPayment("1234-5678-9012-3456", "John Doe");

PaymentStrategy payPalPayment = new PayPalPayment("johndoe@example.com");

// Set and execute credit card payment

paymentContext.setPaymentStrategy(creditCardPayment);

paymentContext.executePayment(100.00);

// Set and execute PayPal payment

paymentContext.setPaymentStrategy(payPalPayment);

paymentContext.executePayment(200.00);

}

}

**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**.

Created a new Java project named **CommandPatternExample** on Eclipse IDE.

1. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.

// Command.java

// Command interface for home automation commands.

public interface Command {

void execute();

}

1. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.

// LightOnCommand.java

// Concrete command class for turning the light on.

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

// LightOffCommand.java

// Concrete command class for turning the light off.

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

1. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.

// RemoteControl.java

// Invoker class for issuing commands.

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

1. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.

// Light.java

// Receiver class for the light device.

public class Light {

public void turnOn() {

System.out.println("Light is on");

}

public void turnOff() {

System.out.println("Light is off");

}

}

1. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

// CommandTest.java

public class CommandTest {

public static void main(String[] args) {

// Create light device

Light light = new Light();

// Create commands

Command lightOnCommand = new LightOnCommand(light);

Command lightOffCommand = new LightOffCommand(light);

// Create remote control

RemoteControl remoteControl = new RemoteControl();

// Set and execute light on command

remoteControl.setCommand(lightOnCommand);

remoteControl.pressButton();

// Set and execute light off command

remoteControl.setCommand(lightOffCommand);

remoteControl.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**.

Created a new Java project named **MVCPatternExample** on Eclipse IDE.

1. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.

// Student.java

public class Student {

private String name;

private int id;

private double grade;

public Student(String name, int id, double 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 double getGrade() {

return grade;

}

public void setGrade(double grade) {

this.grade = grade;

}

}

1. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.

// StudentView.java

public class StudentView {

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

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

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

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

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

}

}

1. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.

// 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(int id) {

model.setId(id);

}

public void setStudentGrade(double grade) {

model.setGrade(grade);

}

public void updateView() {

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

} }

1. **Test the MVC Implementation:**
   * Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

// MVCTest.java

public class MVCTest {

public static void main(String[] args) {

// Create a student

Student model = new Student("Sujata Kumari", 123, 85.0);

// Create a student view

StudentView view = new StudentView();

// Create a student controller

StudentController controller = new StudentController(model, view);

// Update student details

controller.setStudentName("Sujata Dey");

controller.setStudentGrade(90.0);

// Display student details

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**.

Created a new Java project named **DependencyInjectionExample** on Eclipse IDE.

1. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.

// CustomerRepository.java

public interface CustomerRepository {

Customer findCustomerById(int id); }

1. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.

// CustomerRepositoryImpl.java

import java.util.HashMap;

import java.util.Map;

public class CustomerRepositoryImpl implements CustomerRepository {

private Map<Integer, Customer> customers;

public CustomerRepositoryImpl() {

customers = new HashMap<>();

customers.put(1, new Customer(1, "John Doe"));

customers.put(2, new Customer(2, "Jane Doe"));

}

@Override

public Customer findCustomerById(int id) {

return customers.get(id);

}

}

1. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.

// CustomerService.java

public class CustomerService {

private CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public Customer findCustomerById(int id) {

return customerRepository.findCustomerById(id);

}

}

1. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.

In the **CustomerService** class, the **CustomerRepository** is injected through the constructor. This allows you to decouple the **CustomerService** from the concrete implementation of **CustomerRepository**.

1. **Test the Dependency Injection Implementation:**
   * Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

// DependencyInjectionTest.java

public class DependencyInjectionTest {

public static void main(String[] args) {

// Create a customer repository

CustomerRepository customerRepository = new CustomerRepositoryImpl();

// Create a customer service with the customer repository

CustomerService customerService = new CustomerService(customerRepository);

// Find a customer by ID

Customer customer = customerService.findCustomerById(1);

// Print the customer details

if (customer != null) {

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

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

} else {

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

}

}

}