**Design Pattern & Principles**

**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.

**Code:**

class Logger {

// Private static instance

private static Logger *instance*;

// Private constructor

private Logger() {

System.***out***.println("Logger instance created.");

}

// Public static method to get the instance

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger(); // Lazy initialization

}

return *instance*;

}

public void log(String message) {

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

}

}

// Test class with main method

public class LoggerTest {

public static void main(String[] args) {

System.***out***.println("Getting Logger instance 1...");

Logger logger1 = Logger.*getInstance*();

System.***out***.println("Getting Logger instance 2...");

Logger logger2 = Logger.*getInstance*();

logger1.log("This is the first log message.");

logger2.log("This is the second log message.");

if (logger1 == logger2) {

System.***out***.println("Both logger1 and logger2 are the same instance.");

} else {

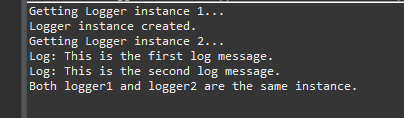
System.***out***.println("Different instances! Singleton failed.");

}

}

}

**Output:**

****

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

**Scenario:**

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

**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.

**Code:**

// Define Document interface

interface Document {

void open();

}

// Create Concrete Document Classes

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

}

// Abstract Factory Class

abstract class DocumentFactory {

public abstract Document createDocument();

}

// Concrete Factory Classes

class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// Test Class

public class FactoryTest {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**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.

**Code:**

// Define Product class

class Computer {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean hasWiFi;

private boolean hasBluetooth;

// Private constructor accepts Builder

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.hasWiFi = builder.hasWiFi;

this.hasBluetooth = builder.hasBluetooth;

}

// Static Nested Builder class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean hasWiFi;

private boolean hasBluetooth;

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 setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setWiFi(boolean hasWiFi) {

this.hasWiFi = hasWiFi;

return this;

}

public Builder setBluetooth(boolean hasBluetooth) {

this.hasBluetooth = hasBluetooth;

return this;

}

public Computer build() {

return new Computer(this);

}

}

// Method to display configuration

public void displayConfiguration() {

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

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

System.***out***.println("Storage: " + storage);

System.***out***.println("Graphics Card: " + graphicsCard);

System.***out***.println("WiFi Enabled: " + hasWiFi);

System.***out***.println("Bluetooth Enabled: " + hasBluetooth);

}

}

// Test class

public class ComputerBuilderTest {

public static void main(String[] args) {

// Build a gaming computer

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.setWiFi(true)

.setBluetooth(true)

.build();

System.***out***.println("Gaming PC Configuration:");

gamingPC.displayConfiguration();

System.***out***.println("\nOffice PC Configuration:");

// Build an office computer with minimal specs

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("512GB SSD")

.setWiFi(true)

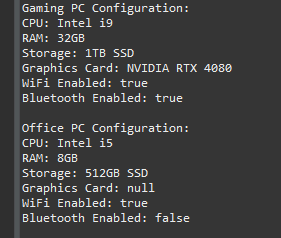
.build();

officePC.displayConfiguration();

}

}

**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.

**Code:**

// Target Interface

interface PaymentProcessor {

void processPayment(double amount);

}

// Adaptee Classes (Existing incompatible interfaces)

class PayPalGateway {

public void sendPayment(double amount) {

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

}

}

class StripeGateway {

public void makePayment(double amount) {

System.***out***.println("Stripe: Processing payment of $" + amount);

}

}

class RazorpayGateway {

public void pay(double amountInRupees) {

System.***out***.println("Razorpay: Processing payment of ₹" + amountInRupees);

}

}

// Adapter Classes

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter(PayPalGateway payPal) {

this.payPal = payPal;

}

public void processPayment(double amount) {

payPal.sendPayment(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpay;

public RazorpayAdapter(RazorpayGateway razorpay) {

this.razorpay = razorpay;

}

public void processPayment(double amount) {

razorpay.pay(amount);

}

}

// Test Class

public class PaymentTest {

public static void main(String[] args) {

System.***out***.println("Using PayPal:");

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

paypal.processPayment(100.00);

System.***out***.println("\nUsing Stripe:");

PaymentProcessor stripe = new StripeAdapter(new StripeGateway());

stripe.processPayment(250.50);

System.***out***.println("\nUsing Razorpay:");

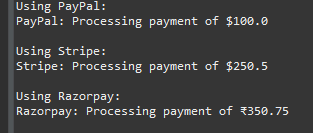
PaymentProcessor razorpay = new RazorpayAdapter(new RazorpayGateway());

razorpay.processPayment(350.75);

}

}

**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.

**Code:**

// Component Interface

interface Notifier {

void send(String message);

}

// Concrete Component

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// Abstract Decorator

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

// Concrete Decorators

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

// Test Class

public class NotificationTest {

public static void main(String[] args) {

// Basic Email Notification

Notifier emailOnly = new EmailNotifier();

System.***out***.println("== Single Channel ==");

emailOnly.send("Meeting at 3 PM");

// Email + SMS

Notifier emailAndSMS = new SMSNotifierDecorator(new EmailNotifier());

System.***out***.println("\n== Email + SMS ==");

emailAndSMS.send("Your OTP is 123456");

// Email + SMS + Slack

Notifier allChannels = new SlackNotifierDecorator(

new SMSNotifierDecorator(

new EmailNotifier()));

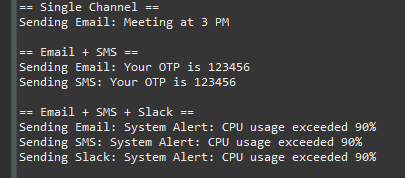
System.***out***.println("\n== Email + SMS + Slack ==");

allChannels.send("System Alert: CPU usage exceeded 90%");

}

}

**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.

**Code:**

// Subject Interface

interface Image {

void display();

}

// Real Subject

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);

}

public void display() {

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

}

}

// Proxy Class

class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realImage == null) {

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

} else {

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

}

realImage.display();

}

}

// Test Class

public class ImageViewerTest {

public static void main(String[] args) {

Image img1 = new ProxyImage("nature.jpg");

Image img2 = new ProxyImage("cityscape.png");

// First load - real loading from remote

System.***out***.println("\n--- First time viewing nature.jpg ---");

img1.display();

// Second time - should use cache

System.***out***.println("\n--- Second time viewing nature.jpg ---");

img1.display();

// Viewing different image

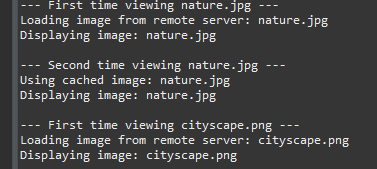
System.***out***.println("\n--- First time viewing cityscape.png ---");

img2.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.

**Code:**

import java.util.\*;

// Subject Interface

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

// Observer Interface

interface Observer {

void update(String stockName, double price);

}

// Concrete Subject

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStockPrice(String stockName, double price) {

this.stockName = stockName;

this.stockPrice = price;

System.***out***.println("\n[StockMarket] Price updated for " + stockName + ": ₹" + price);

notifyObservers();

}

public void registerObserver(Observer o) {

observers.add(o);

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

}

public void removeObserver(Observer o) {

observers.remove(o);

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

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

// Concrete Observers

class MobileApp implements Observer {

public void update(String stockName, double price) {

System.***out***.println("[MobileApp] " + stockName + " is now ₹" + price);

}

}

class WebApp implements Observer {

public void update(String stockName, double price) {

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

}

}

// Test Class

public class StockObserverTest {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobileApp = new MobileApp();

Observer webApp = new WebApp();

market.registerObserver(mobileApp);

market.registerObserver(webApp);

market.setStockPrice("TCS", 3850.50);

market.setStockPrice("INFY", 1540.75);

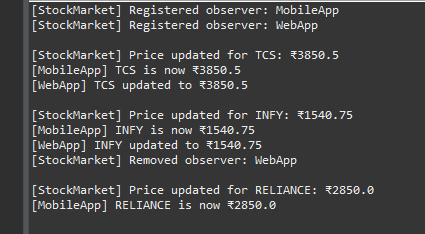
market.removeObserver(webApp);

market.setStockPrice("RELIANCE", 2850.00);

}

}

**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.

**Code:**

import java.util.Scanner;

// Strategy Interface

interface PaymentStrategy {

void pay(double amount);

}

// Concrete Strategies

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

public void pay(double amount) {

System.***out***.println("Paid ₹" + amount + " using Credit Card ending with " + cardNumber.substring(cardNumber.length() - 4));

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

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

}

}

// Context Class

class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void processPayment(double amount) {

if (paymentStrategy == null) {

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

} else {

paymentStrategy.pay(amount);

}

}

}

// Test Class

public class PaymentStrategyTest {

public static void main(String[] args) {

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

PaymentContext context = new PaymentContext();

System.***out***.println("=== Select Payment Method ===");

System.***out***.println("1. Credit Card");

System.***out***.println("2. PayPal");

System.***out***.print("Enter choice (1 or 2): ");

int choice = sc.nextInt();

sc.nextLine(); // consume newline

switch (choice) {

case 1:

System.***out***.print("Enter Credit Card Number: ");

String card = sc.nextLine();

context.setPaymentStrategy(new CreditCardPayment(card));

break;

case 2:

System.***out***.print("Enter PayPal Email: ");

String email = sc.nextLine();

context.setPaymentStrategy(new PayPalPayment(email));

break;

default:

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

return;

}

System.***out***.print("Enter amount to pay: ₹");

double amount = sc.nextDouble();

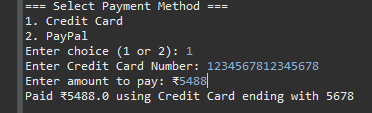
context.processPayment(amount);

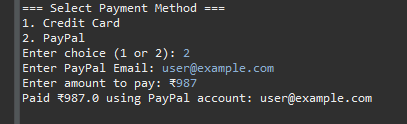
sc.close();

}

}

**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**.

**Code:**

// Command Interface

interface Command {

void execute();

}

// Receiver

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// Concrete Commands

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// Invoker

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.");

}

}

}

// Test Class

public class HomeAutomationTest {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

System.***out***.println("== Turning Light ON ==");

remote.setCommand(lightOn);

remote.pressButton();

System.***out***.println("\n== Turning Light OFF ==");

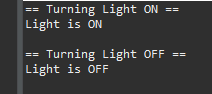
remote.setCommand(lightOff);

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**.

**Code:**

// Model

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 String getId() {

return id;

}

public String getGrade() {

return grade;

}

public void setName(String name) {

this.name = name;

}

public void setId(String id) {

this.id = id;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

// View

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);

}

}

// Controller

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Controller methods

public void updateView() {

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

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

}

// Main Test Class

public class MVCExampleTest {

public static void main(String[] args) {

// Create model

Student student = new Student("Ravi Kumar", "S101", "A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Initial data

System.***out***.println("=== Initial Student Record ===");

controller.updateView();

// Update data

controller.setStudentName("Ravi Raj");

controller.setStudentGrade("A+");

// Updated data

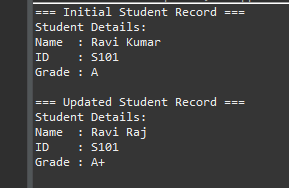
System.***out***.println("\n=== Updated Student Record ===");

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.

**Code:**

// Repository Interface

interface CustomerRepository {

String findCustomerById(String customerId);

}

// Concrete Repository Implementation

class CustomerRepositoryImpl implements CustomerRepository {

*@Override*

public String findCustomerById(String customerId) {

// Mocked data for demo

if (customerId.equals("C101")) {

return "Customer[ID=C101, Name=Ravi Kumar, Email=ravi@example.com]";

} else if (customerId.equals("C102")) {

return "Customer[ID=C102, Name=Anjali Sharma, Email=anjali@example.com]";

}

return "Customer not found";

}

}

// Service Class (Depends on Repository)

class CustomerService {

private CustomerRepository repository;

// Constructor-based Dependency Injection

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void showCustomer(String customerId) {

String customerData = repository.findCustomerById(customerId);

System.***out***.println(customerData);

}

}

// Main Class to Test

public class CustomerAppTest {

public static void main(String[] args) {

// Create repository (dependency)

CustomerRepository repo = new CustomerRepositoryImpl();

// Inject dependency via constructor

CustomerService service = new CustomerService(repo);

// Test finding customers

System.***out***.println("=== Fetching Customer C101 ===");

service.showCustomer("C101");

System.***out***.println("\n=== Fetching Customer C102 ===");

service.showCustomer("C102");

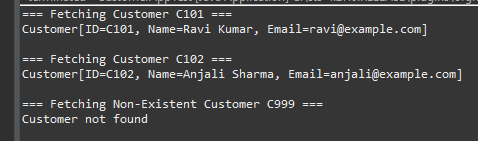
System.***out***.println("\n=== Fetching Non-Existent Customer C999 ===");

service.showCustomer("C999");

}

}

**OutPut:**

****