**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.

**SOLUTION:**

**Logger.java**

**package** com.singleton.logger;

**public** **class** Logger {

**private** **static** Logger *instance*;

**private** Logger() {

System.***out***.println("Logger is initialized");

}

**public** **static** Logger getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** Logger();

}

**return** *instance*;

}

**public** **void** log(String msg) {

System.***out***.println("[LOG]: " + msg);

}

}

**LoggerTest.java**

**package** com.singleton.logger;

**public** **class** LoggerTest {

**public** **static** **void** main(String[] args) {

Logger l1 = Logger.*getInstance*();

l1.log("This is the First message");

Logger l2 = Logger.*getInstance*();

l2.log("This is the Second message");

**if** (l1 == l2) {

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

} **else** {

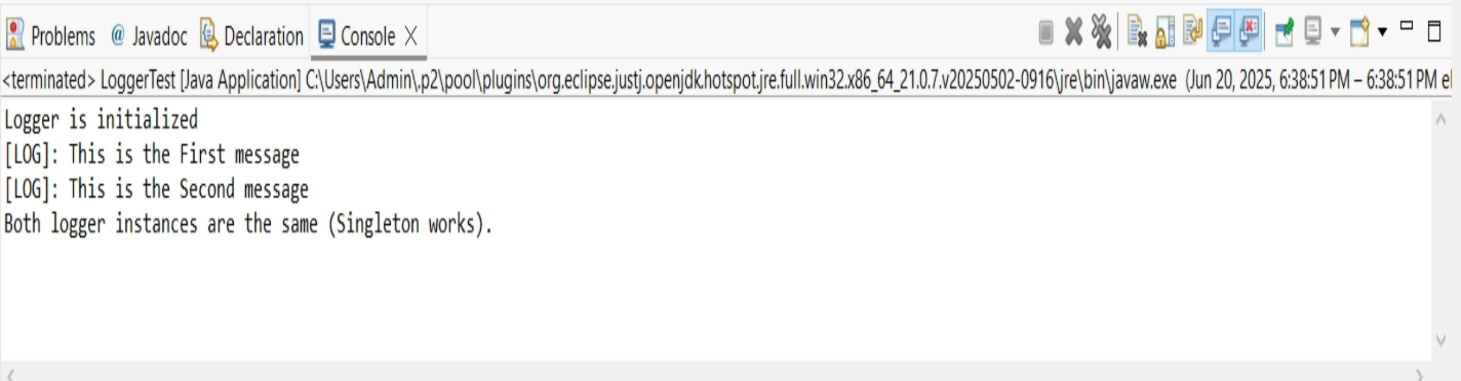
System.***out***.println("Logger instances are different (Singleton failed).");

}

}

}

**OUTPUT:**

****

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

**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.

**SOLUTION:**

**Document.java**

**package** com.factory.documents;

**public** **interface** Document {

**void** open();

}

**DocumentFactory.java**

**package** com.factory.documents;

**public** **abstract** **class** DocumentFactory {

**public** **abstract** Document createDocument();

}

**DocumentFactory.java**

**package** com.factory.documents;

**public** **class** DocumentFactoryTest {

**public** **static** **void** main(String[] args) {

DocumentFactory wordFactory = **new** WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = **new** PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = **new** ExcelDocumentFactory();

Document excel = excelFactory.createDocument();

excel.open();

}

}

**ExcelDocument.java**

**package** com.factory.documents;

**public** **class** ExcelDocument **implements** Document {

@Override

**public** **void** open() {

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

}

}

**ExcelDocumentFactory.java**

**package** com.factory.documents;

**public** **class** ExcelDocumentFactory **extends** DocumentFactory {

@Override

**public** Document createDocument() {

**return** **new** ExcelDocument();

}

}

**PdfDocument.java**

**package** com.factory.documents;

**public** **class** PdfDocument **implements** Document {

@Override

**public** **void** open() {

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

}

}

**PdfDocumentFactory.java**

**package** com.factory.documents;

**public** **class** PdfDocumentFactory **extends** DocumentFactory {

@Override

**public** Document createDocument() {

**return** **new** PdfDocument();

}

}

**WordDocument.java**

**package** com.factory.documents;

**public** **class** WordDocument **implements** Document {

@Override

**public** **void** open() {

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

}

}

**WordDocumentFactory.java**

**package** com.factory.documents;

**public** **class** WordDocumentFactory **extends** DocumentFactory {

@Override

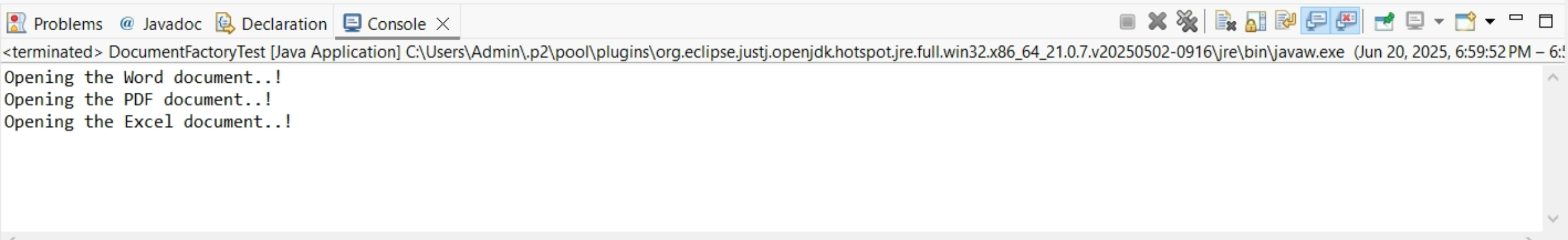
**public** Document createDocument() {

**return** **new** WordDocument();

}

}

**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.

**SOLUTION:**

**Computer.java**

**package** com.example.builder;

**public** **class** Computer {

**private** String cpu;

**private** String ram;

**private** String storage;

**private** **boolean** graphicsCard;

**private** **boolean** bluetooth;

**private** Computer(Builder builder) {

**this**.cpu = builder.cpu;

**this**.ram = builder.ram;

**this**.storage = builder.storage;

**this**.graphicsCard = builder.graphicsCard;

**this**.bluetooth = builder.bluetooth;

}

**public** **static** **class** Builder {

**private** String cpu;

**private** String ram;

**private** String storage;

**private** **boolean** graphicsCard;

**private** **boolean** bluetooth;

**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(**boolean** graphicsCard) {

**this**.graphicsCard = graphicsCard;

**return** **this**;

}

**public** Builder setBluetooth(**boolean** bluetooth) {

**this**.bluetooth = bluetooth;

**return** **this**;

}

**public** Computer build() {

**return** **new** Computer(**this**);

}

}

**public** **void** displaySpecs() {

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

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

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

System.***out***.println("Graphics Card: " + (graphicsCard ? "Yes" : "No"));

System.***out***.println("Bluetooth: " + (bluetooth ? "Yes" : "No"));

}

}

**BuilderPatternTest.java**

**package** com.example.builder;

**public** **class** BuilderPatternTest {

**public** **static** **void** main(String[] args) {

Computer basicComputer = **new** Computer.Builder()

.setCpu("Intel i3")

.setRam("8GB")

.setStorage("256GB SSD")

.build();

System.***out***.println("Basic Computer Configuration:");

basicComputer.displaySpecs();

System.***out***.println("\n------------------------\n");

Computer gamingComputer = **new** Computer.Builder()

.setCpu("Intel i5")

.setRam("64GB")

.setStorage("2TB SSD")

.setGraphicsCard(**true**)

.setBluetooth(**true**)

.build();

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

gamingComputer.displaySpecs();

}

}

**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.

**SOLUTION:**

**PaymentProcessor.java**

**package** com.example.adapter;

**public** **interface** PaymentProcessor {

**void** processPayment(**double** amount);

}

**PayPalGateway.java**

**package** com.example.adapter;

**public** **class** PayPalGateway {

**public** **void** sendPayment(**double** amount) {

System.***out***.println("Payment of $" + amount + " made using PayPal.");

}

}

**StripeGateway.java**

**package** com.example.adapter;

**public** **class** StripeGateway {

**public** **void** makeStripePayment(**double** amountInCents) {

System.***out***.println("Payment of $" + (amountInCents / 100) + " made using Stripe.");

}

}

**PayPalAdapter.java**

**package** com.example.adapter;

**public** **class** PayPalAdapter **implements** PaymentProcessor {

**private** PayPalGateway paypal;

**public** PayPalAdapter(PayPalGateway paypal) {

**this**.paypal = paypal;

}

@Override

**public** **void** processPayment(**double** amount) {

paypal.sendPayment(amount);

}

}

**StripeAdapter.java**

**package** com.example.adapter;

**public** **class** StripeAdapter **implements** PaymentProcessor {

**private** StripeGateway stripe;

**public** StripeAdapter(StripeGateway stripe) {

**this**.stripe = stripe;

}

@Override

**public** **void** processPayment(**double** amount) {

stripe.makeStripePayment(amount \* 100);

}

}

**PaymentSystemTest.java**

**package** com.example.adapter;

**public** **class** PaymentSystemTest {

**public** **static** **void** main(String[] args) {

PayPalGateway paypalGateway = **new** PayPalGateway();

PaymentProcessor paypalProcessor = **new** PayPalAdapter(paypalGateway);

paypalProcessor.processPayment(150.75);

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

StripeGateway stripeGateway = **new** StripeGateway();

PaymentProcessor stripeProcessor = **new** StripeAdapter(stripeGateway);

stripeProcessor.processPayment(89.50);

}

}

**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.

**SOLUTION:**

**Notifier.java**

**package** com.example.notifier;

**public** **interface** Notifier {

**void** send(String message);

}

**EmailNotifier.java**

**package** com.example.notifier;

**public** **class** EmailNotifier **implements** Notifier {

@Override

**public** **void** send(String message) {

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

}

}

**NotificationTest.java**

**package** com.example.notifier; **public** **class** NotificationTest {

**public** **static** **void** main(String[] args) {

System.***out***.println("Scenario 1: Basic Email Notification");

Notifier emailNotifier = **new** EmailNotifier();

emailNotifier.send("Hello World!");

System.***out***.println("\n-----------------------------------\n");

System.***out***.println("Scenario 2: Email and SMS Notification");

Notifier emailAndSMSNotifier = **new** SMSNotifierDecorator(**new** EmailNotifier());

emailAndSMSNotifier.send("Important update!");

System.***out***.println("\n-----------------------------------\n");

System.***out***.println("Scenario 3: Email and Slack Notification");

Notifier emailAndSlackNotifier = **new** SlackNotifierDecorator(**new** EmailNotifier());

emailAndSlackNotifier.send("Daily report is ready.");

System.***out***.println("\n-----------------------------------\n");

System.***out***.println("Scenario 4: Email, SMS, and Slack Notification");

Notifier allChannelsNotifier = **new** SlackNotifierDecorator(

**new** SMSNotifierDecorator(

**new** EmailNotifier()

)

);

allChannelsNotifier.send("Urgent alert! System compromised!");

System.***out***.println("\n-----------------------------------\n");

System.***out***.println("Scenario 5: SMS and Slack Notification (no email base)");

Notifier smsAndSlackNotifier = **new** SlackNotifierDecorator(**new** SMSNotifierDecorator(**new** Notifier() {

@Override

**public** **void** send(String message) {

System.***out***.println("Starting with a generic notifier (could be anything else)");

}

}));

smsAndSlackNotifier.send("Another test message.");

System.***out***.println("\n-----------------------------------\n");

}

}

**NotifierDecorator.java**

**package** com.example.notifier;

**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);

}

}

**SlackNotifierDecorator.java**

**package** com.example.notifier;

**public** **class** SlackNotifierDecorator **extends** NotifierDecorator {

**public** SlackNotifierDecorator(Notifier notifier) {

**super**(notifier);

}

@Override

**public** **void** send(String message) {

**super**.send(message);

sendSlackMessage(message);

}

**private** **void** sendSlackMessage(String message) {

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

}

}

**SMSNotifierDecorator.java**

**package** com.example.notifier;

**public** **class** SMSNotifierDecorator **extends** NotifierDecorator {

**public** SMSNotifierDecorator(Notifier notifier) {

**super**(notifier);

}

@Override

**public** **void** send(String message) {

**super**.send(message);

sendSMS(message);

}

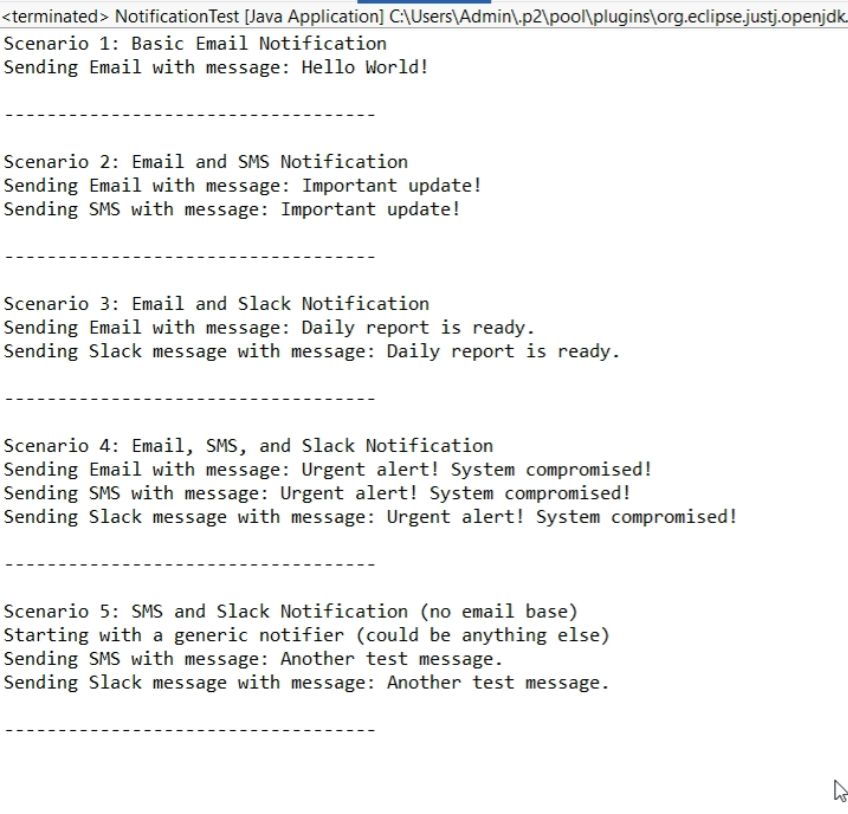
**private** **void** sendSMS(String message) {

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

}

}

**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.

**SOLUTION:**

**Image.java**

**package com.example.proxy.image;**

**public interface Image {**

**void display();**

**}**

**ImageClient.java**

**package** com.example.proxy.image;

**public** **class** ImageClient {

**public** **static** **void** main(String[] args) {

System.***out***.println("------ Creating ProxyImage for image1.jpg ------");

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

System.***out***.println("\n------ First call to display() for image1.jpg ------");

image1.display();

System.***out***.println("\n------ Second call to display() for image1.jpg ------");

image1.display();

System.***out***.println("\n------ Creating ProxyImage for image2.png ------");

Image image2 = **new** ProxyImage("image2.png");

System.***out***.println("\n------ First call to display() for image2.png ------");

image2.display();

System.***out***.println("\n------ Third call to display() for image1.jpg ------");

image1.display();

}

}

**RealImage.java**

**package** com.example.proxy.image;

**public** **class** RealImage **implements** Image {

**private** String fileName;

**public** RealImage(String fileName) {

**this**.fileName = fileName;

loadImageFromServer();

}

**private** **void** loadImageFromServer() {

System.***out***.println("Loading " + fileName + " from remote server...");

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

Thread.*currentThread*().interrupt();

System.***err***.println("Image loading interrupted: " + e.getMessage());

}

System.***out***.println("Finished loading " + fileName);

}

@Override

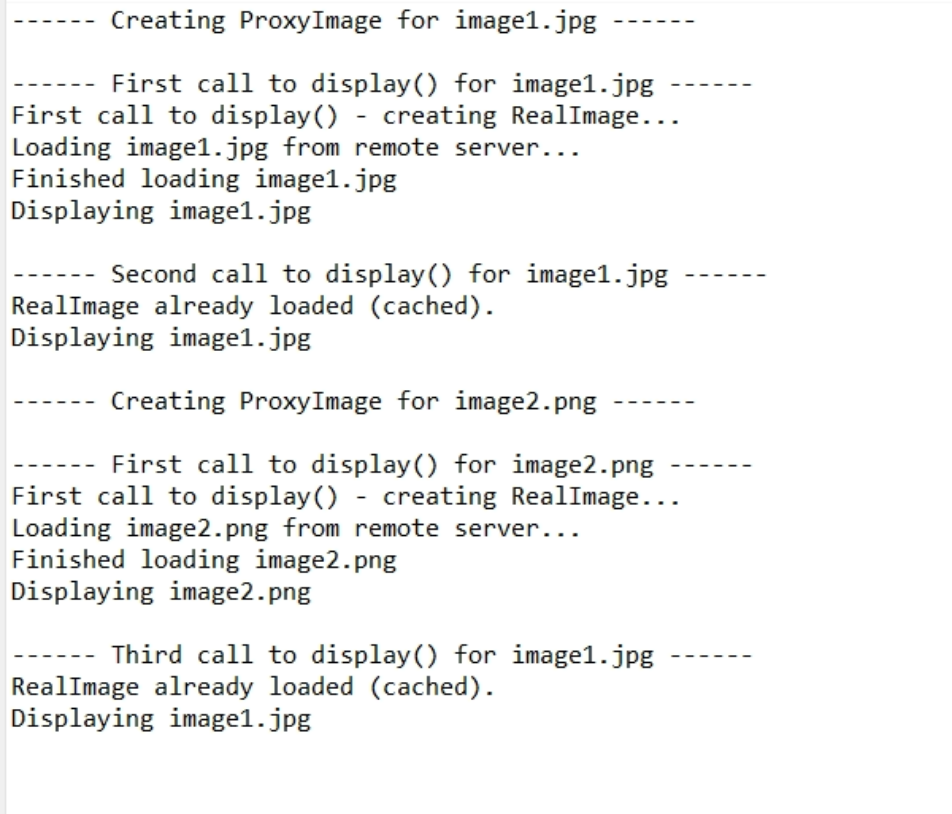
**public** **void** display() {

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

}

}

**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.

**SOLUTION:**

**Main.java**

**package** com.example.observerpattern;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

StockMarket market = **new** StockMarket();

Observer mobile = **new** MobileApp("Client A");

Observer web = **new** WebApp("Client B");

market.registerObserver(mobile);

market.registerObserver(web);

market.setStockPrice(150.0);

market.setStockPrice(155.5);

market.removeObserver(web);

market.setStockPrice(160.0);

}

}

**MobileApp.java**

**package** com.example.observerpattern;

**public** **class** MobileApp **implements** Observer {

**private** String clientName;

**public** MobileApp(String name) {

**this**.clientName = name;

}

@Override

**public** **void** update(**double** newPrice) {

System.***out***.println("[" + clientName + "] Mobile App - New stock price: " + newPrice);

}

}

**Observer.java**

**package** com.example.observerpattern;

**public** **interface** Observer {

**void** update(**double** newPrice);

}

**Stock.java**

**package** com.example.observerpattern;

**public** **interface** Stock {

**void** registerObserver(Observer o);

**void** removeObserver(Observer o);

**void** notifyObservers();

}

**StockMarket.java**

**package** com.example.observerpattern;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** StockMarket **implements** Stock {

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

**private** **double** stockPrice;

@Override

**public** **void** registerObserver(Observer o) {

observers.add(o);

}

@Override

**public** **void** removeObserver(Observer o) {

observers.remove(o);

}

@Override

**public** **void** notifyObservers() {

**for** (Observer o : observers) {

o.update(stockPrice);

}

}

**public** **void** setStockPrice(**double** newPrice) {

**this**.stockPrice = newPrice;

notifyObservers();

}

}

**WebApp.java**

**package** com.example.observerpattern;

**public** **class** WebApp **implements** Observer {

**private** String clientName;

**public** WebApp(String name) {

**this**.clientName = name;

}

@Override

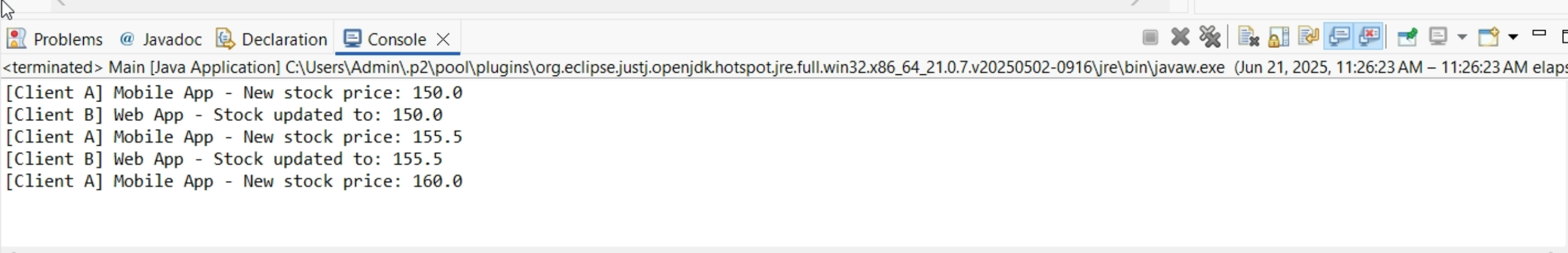
**public** **void** update(**double** newPrice) {

System.***out***.println("[" + clientName + "] Web App - Stock updated to: " + newPrice);

}

}

**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.

**SOLUTION:**

**CreditCardPayment.java**

**package** com.example.strategy;

**public** **class** CreditCardPayment **implements** PaymentStrategy {

**private** String cardNumber;

**private** String name;

**public** CreditCardPayment(String cardNumber, String name) {

**this**.cardNumber = cardNumber;

**this**.name = name;

}

@Override

**public** **void** pay(**double** amount) {

System.***out***.println("Paying $" + amount + " using Credit Card (Card No: " + cardNumber + ", Name: " + name + ")");

}

}

**PaymentContext.java**

**package** com.example.strategy;

**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) {

**if** (paymentStrategy == **null**) {

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

**return**;

}

paymentStrategy.pay(amount);

}

}

**PaymentStrategy.java**

**package** com.example.strategy;

**public** **interface** PaymentStrategy {

**void** pay(**double** amount);

}

**PaymentSystemTest.java**

**package** com.example.strategy;

**public** **class** PaymentSystemTest {

**public** **static** **void** main(String[] args) {

System.***out***.println("--- Credit Card Payment Scenario ---");

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

PaymentContext context = **new** PaymentContext(creditCard);

context.executePayment(100.50);

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

System.***out***.println("--- PayPal Payment Scenario ---");

PaymentStrategy payPal = **new** PayPalPayment("john.doe@example.com");

context.setPaymentStrategy(payPal);

context.executePayment(50.00);

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

System.***out***.println("--- Another Credit Card Payment Scenario ---");

PaymentStrategy anotherCreditCard = **new** CreditCardPayment("9876-5432-1098-7654", "Jane Smith");

context.setPaymentStrategy(anotherCreditCard);

context.executePayment(25.75);

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

System.***out***.println("--- No Strategy Set Scenario ---");

PaymentContext emptyContext = **new** PaymentContext(**null**);

emptyContext.executePayment(75.00);

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

}

}

**PayPalPayment.java**

**package** com.example.strategy;

**public** **class** PayPalPayment **implements** PaymentStrategy {

**private** String email;

**public** PayPalPayment(String email) {

**this**.email = email;

}

@Override

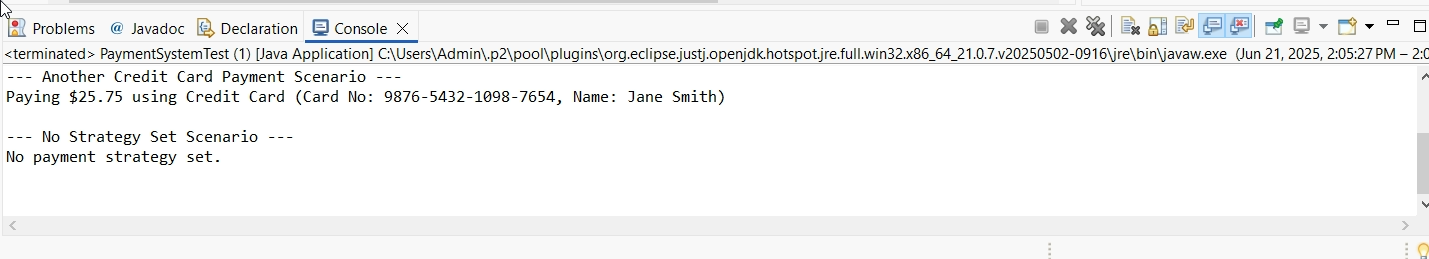
**public** **void** pay(**double** amount) {

System.***out***.println("Paying $" + amount + " using PayPal (Email: " + email + ")");

}

}

**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**.

**SOLUTION:**

**Command.java**

**package** com.example.command;

**public** **interface** Command {

**void** execute();

}

**Light.java**

**package** com.example.command;

**public** **class** Light {

**private** String location;

**public** Light(String location) {

**this**.location = location;

}

**public** **void** on() {

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

}

**public** **void** off() {

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

}

}

**LightOffCommand.java**

**package** com.example.command;

**public** **class** LightOffCommand **implements** Command {

**private** Light light;

**public** LightOffCommand(Light light) {

**this**.light = light;

}

@Override

**public** **void** execute() {

light.off();

}

}

**LightOnCommand.java**

**package** com.example.command;

**public** **class** LightOnCommand **implements** Command {

**private** Light light;

**public** LightOnCommand(Light light) {

**this**.light = light;

}

@Override

**public** **void** execute() {

light.on();

}

}

**RemoteControl.java**

**package** com.example.command;

**public** **class** RemoteControl {

**private** Command command;

**public** **void** setCommand(Command command) {

**this**.command = command;

}

**public** **void** pressButton() {

**if** (command != **null**) {

command.execute();

} **else** {

System.***out***.println("No command assigned to this button.");

}

}

}

**HomeAutomationTest.java**

**package** com.example.command;

**public** **class** HomeAutomationTest {

**public** **static** **void** main(String[] args) {

Light livingRoomLight = **new** Light("Living Room");

Light kitchenLight = **new** Light("Kitchen");

Command livingRoomLightOn = **new** LightOnCommand(livingRoomLight);

Command livingRoomLightOff = **new** LightOffCommand(livingRoomLight);

Command kitchenLightOn = **new** LightOnCommand(kitchenLight);

Command kitchenLightOff = **new** LightOffCommand(kitchenLight);

RemoteControl remote = **new** RemoteControl();

System.***out***.println("--- Scenario 1: Turning Living Room Light ON ---");

remote.setCommand(livingRoomLightOn);

remote.pressButton();

System.***out***.println("\n--- Scenario 2: Turning Living Room Light OFF ---");

remote.setCommand(livingRoomLightOff);

remote.pressButton();

System.***out***.println("\n--- Scenario 3: Turning Kitchen Light ON ---");

remote.setCommand(kitchenLightOn);

remote.pressButton();

System.***out***.println("\n--- Scenario 4: Attempting to press an unassigned button ---");

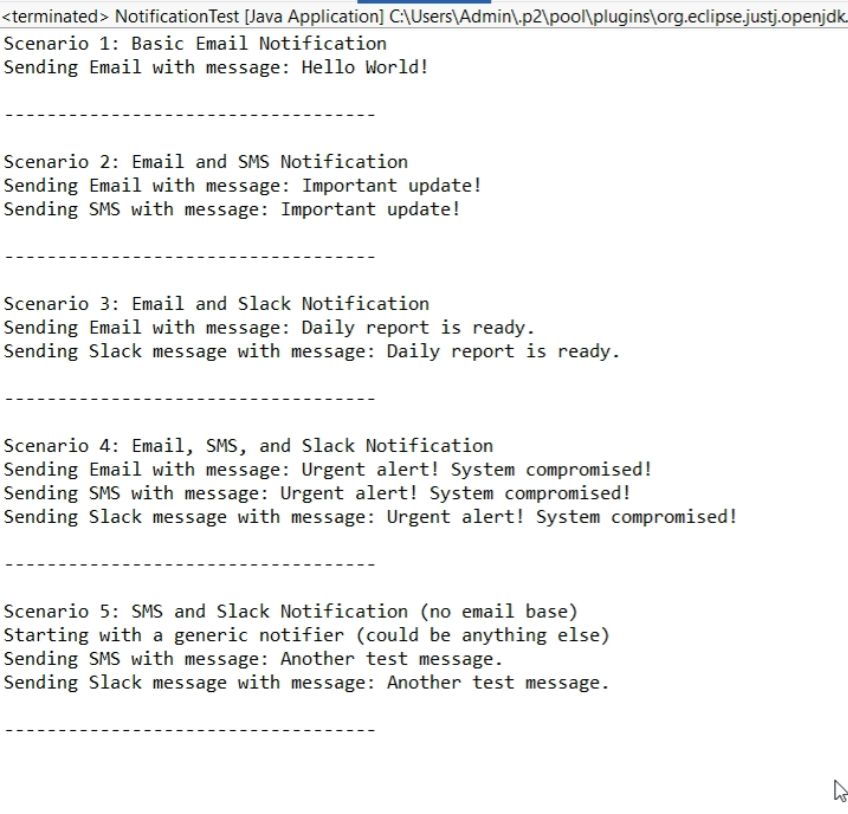
remote.setCommand(**null**);

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**.

**SOLUTION:**

**Student.java**

**package** com.example.mvc;

**public** **class** Student {

**private** String id;

**private** String name;

**private** String grade;

**public** Student(String id, String name, String grade) {

**this**.id = id;

**this**.name = name;

**this**.grade = grade;

}

**public** String getId() {

**return** id;

}

**public** String getName() {

**return** name;

}

**public** String getGrade() {

**return** grade;

}

**public** **void** setId(String id) {

**this**.id = id;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** setGrade(String grade) {

**this**.grade = grade;

}

}

**StudentController.java**

**package** com.example.mvc;

**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** String getStudentName() {

**return** model.getName();

}

**public** **void** setStudentId(String id) {

model.setId(id);

}

**public** String 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());

}

}

**StudentView.java**

**package** com.example.mvc;

**public** **class** StudentView {

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

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

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

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

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

}

}

**MVCTest.java**

**package** com.example.mvc;

**public** **class** MVCTest {

**public** **static** **void** main(String[] args) {

Student model = *retrieveStudentFromDatabase*();

StudentView view = **new** StudentView();

StudentController controller = **new** StudentController(model, view);

System.***out***.println("--- Initial Student Details ---");

controller.updateView();

System.***out***.println("\n-------------------------------\n");

System.***out***.println("--- Updating Student Name and Grade ---");

controller.setStudentName("Jane Doe");

controller.setStudentGrade("A+");

controller.updateView();

System.***out***.println("\n-------------------------------\n");

System.***out***.println("--- Updating Student ID ---");

controller.setStudentId("S102");

controller.updateView();

System.***out***.println("\n-------------------------------\n");

}

**private** **static** Student retrieveStudentFromDatabase() {

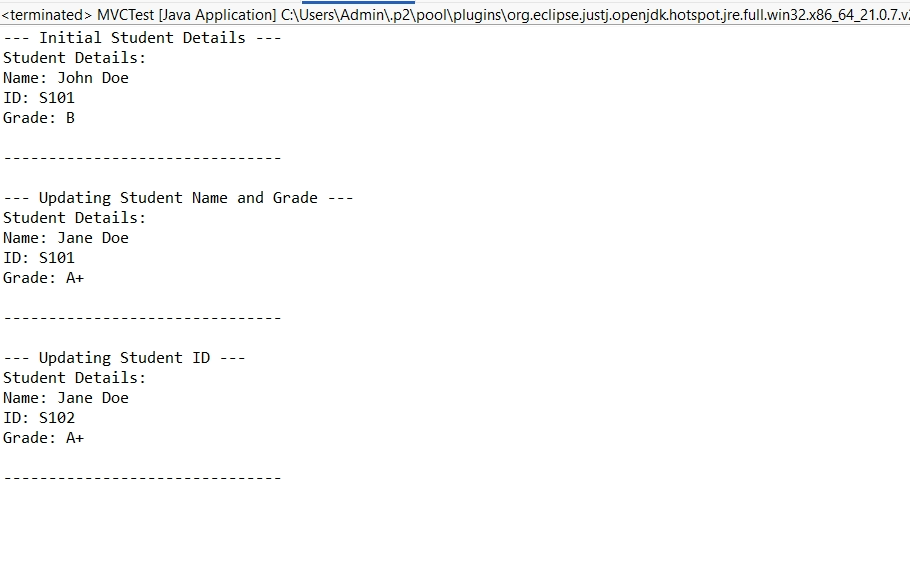
Student student = **new** Student("S101", "John Doe", "B");

**return** student;

}

}

**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.

**SOLUTION:**

**CustomerRepository.java**

**package** com.example.di;

**class** Customer {

**private** String id;

**private** String name;

**public** Customer(String id, String name) {

**this**.id = id;

**this**.name = name;

}

**public** String getId() { **return** id; }

**public** String getName() { **return** name; }

@Override

**public** String toString() {

**return** "Customer [ID=" + id + ", Name=" + name + "]";

}

}

**public** **interface** CustomerRepository {

Customer findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

**package** com.example.di;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** CustomerRepositoryImpl **implements** CustomerRepository {

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

**public** CustomerRepositoryImpl() {

customers.put("1", **new** Customer("1", "Alice Smith"));

customers.put("2", **new** Customer("2", "Bob Johnson"));

customers.put("3", **new** Customer("3", "Charlie Brown"));

}

@Override

**public** Customer findCustomerById(String id) {

System.***out***.println("CustomerRepositoryImpl: Fetching customer with ID: " + id);

**return** customers.get(id);

}

}

**CustomerService.java**

**package** com.example.di;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** CustomerRepositoryImpl **implements** CustomerRepository {

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

**public** CustomerRepositoryImpl() {

customers.put("1", **new** Customer("1", "Alice Smith"));

customers.put("2", **new** Customer("2", "Bob Johnson"));

customers.put("3", **new** Customer("3", "Charlie Brown"));

}

@Override

**public** Customer findCustomerById(String id) {

System.***out***.println("CustomerRepositoryImpl: Fetching customer with ID: " + id);

**return** customers.get(id);

}

}

**DITest.java**

**package** com.example.di;

**public** **class** DITest {

**public** **static** **void** main(String[] args) {

System.***out***.println("--- Starting Dependency Injection Demo ---");

CustomerRepository repository = **new** CustomerRepositoryImpl();

CustomerService service = **new** CustomerService(repository);

System.***out***.println("\n--- Fetching Customer 1 ---");

Customer customer1 = service.getCustomerDetails("1");

**if** (customer1 != **null**) {

System.***out***.println("Found: " + customer1);

} **else** {

System.***out***.println("Customer 1 not found.");

}

System.***out***.println("\n--- Fetching Customer 5 (non-existent) ---");

Customer customer5 = service.getCustomerDetails("5");

**if** (customer5 != **null**) {

System.***out***.println("Found: " + customer5);

} **else** {

System.***out***.println("Customer 5 not found.");

}

System.***out***.println("\n--- Dependency Injection Demo Finished ---");

}

}

**OUTPUT:**

