**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.

**Source Code:**

//Logger.java

package SingletonPattern;

public class Logger {

// Step 1: Create a private static instance of the same class

private static Logger *instance*;

// Step 2: Make the constructor private so it cannot be instantiated from outside

private Logger() {

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

}

// Step 3: Provide a public static method to get the single instance

public static Logger getInstance() {

if (*instance* == null) {

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

}

return *instance*;

}

// Logger method to print a log message

public void log(String message) {

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

}

}

//TestSingleton.java

package SingletonPattern;

public class TestSingleton {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

Logger logger2 = Logger.*getInstance*();

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

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

// Verify both references point to the same object

if (logger1 == logger2) {

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

} else {

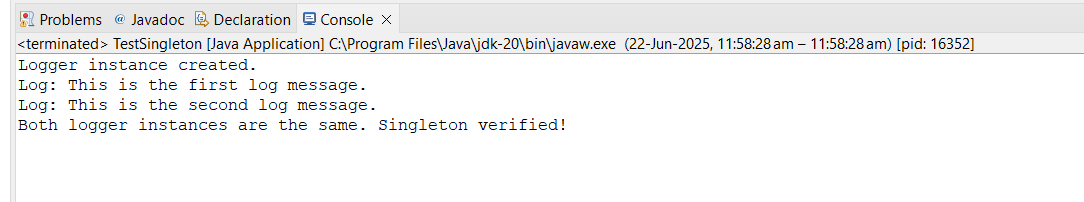
System.***out***.println("Different instances exist. Singleton failed.");

}

}

}

**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.

**Source Code:**

// Document.java

public interface Document {

void open();

}

// WordDocument.java

public class WordDocument implements Document {

public void open() {

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

}

}

// PdfDocument.java

public class PdfDocument implements Document {

public void open() {

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

}

}

// ExcelDocument.java

public class ExcelDocument implements Document {

public void open() {

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

}

}

// DocumentFactory.java

public abstract class DocumentFactory {

public abstract Document createDocument();

}

// WordDocumentFactory.java

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

// PdfDocumentFactory.java

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

// ExcelDocumentFactory.java

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// DocumentFactoryTest.java

public class DocumentFactoryTest {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

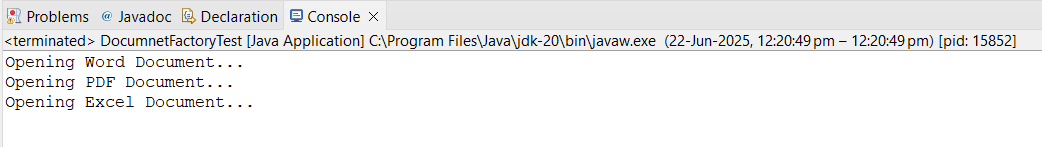
Document excelDoc = excelFactory.createDocument();

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.

**Source Code:**

// Computer.java

public class Computer {

// Required attributes

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean hasWifi;

private boolean hasBluetooth;

// Private constructor

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);

}

}

// Display computer configuration

public void displayConfig() {

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);

}

}

// BuilderPatternTest.java

public class BuilderPatternTest {

public static void main(String[] args) {

// High-end Computer

Computer gamingComputer = new Computer.Builder()

.setCPU("Intel Core i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4090")

.setWifi(true)

.setBluetooth(true)

.build();

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

gamingComputer.displayConfig();

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

// Budget Computer

Computer officeComputer = new Computer.Builder()

.setCPU("Intel Core i5")

.setRAM("8GB")

.setStorage("512GB HDD")

.setWifi(true)

.setBluetooth(false)

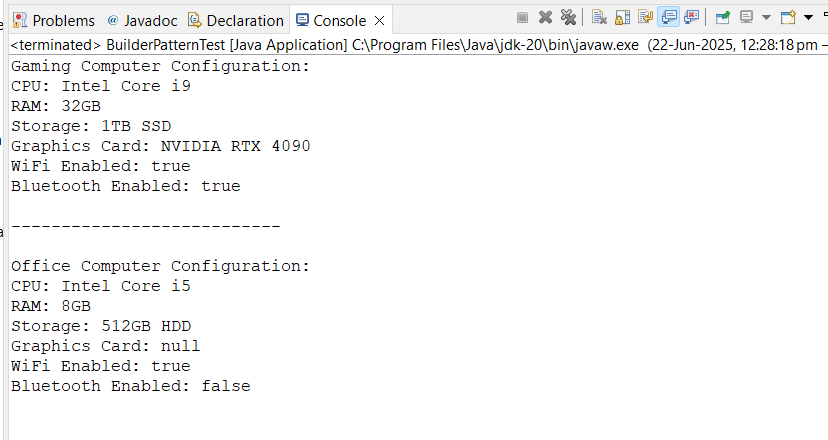
.build();

System.out.println("Office Computer Configuration:");

officeComputer.displayConfig();

}

}

****

**Output:**

**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Steps:**

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

**Source Code:**

// PaymentProcessor.java

public interface PaymentProcessor {

void processPayment(double amount);

}

// PayPalGateway.java

public class PayPalGateway {

public void makePayment(double amount) {

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

}

}

// StripeGateway.java

public class StripeGateway {

public void pay(double amount) {

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

}

}

// RazorpayGateway.java

public class RazorpayGateway {

public void sendPayment(double amount) {

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

}

}

// PayPalAdapter.java

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

// StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.pay(amount);

}

}

// RazorpayAdapter.java

public class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpay;

public RazorpayAdapter(RazorpayGateway razorpay) {

this.razorpay = razorpay;

}

public void processPayment(double amount) {

razorpay.sendPayment(amount);

}

}

// AdapterPatternTest.java

public class AdapterPatternTest {

public static void main(String[] args) {

// Using PayPal

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

paypalProcessor.processPayment(500.0);

// Using Stripe

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

stripeProcessor.processPayment(1200.5);

// Using Razorpay

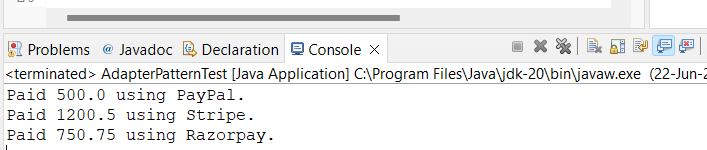
PaymentProcessor razorpayProcessor = new RazorpayAdapter(new RazorpayGateway());

razorpayProcessor.processPayment(750.75);

}

}

**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.

**Source Code:**

// Notifier.java

public interface Notifier {

void send(String message);

}

// EmailNotifier.java

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// NotifierDecorator.java

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

// SMSNotifierDecorator.java

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

// SlackNotifierDecorator.java

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

// DecoratorPatternTest.java

public class DecoratorPatternTest {

public static void main(String[] args) {

// Base notifier - only Email

Notifier notifier = new EmailNotifier();

System.out.println("=== Email Only ===");

notifier.send("Hello, Email user!");

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

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

smsNotifier.send("Hello, Multi-channel user!");

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

Notifier fullNotifier = new SlackNotifierDecorator(

new SMSNotifierDecorator(

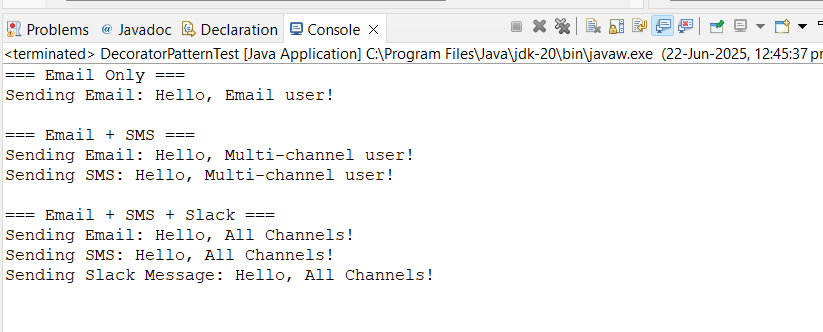
new EmailNotifier()));

fullNotifier.send("Hello, All Channels!");

}

}

**Output:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Steps:**

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

**Source Code:**

// Image.java

public interface Image {

void display();

}

// RealImage.java

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer(); // Simulate image loading

}

private void loadFromRemoteServer() {

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

}

public void display() {

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

}

}

// ProxyImage.java

public class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

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

}

realImage.display(); // Cached display

}

}

// ProxyPatternTest.java

public class ProxyPatternTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("cat.png");

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

// Image not loaded yet

System.out.println("Accessing cat.png...");

image1.display(); // Loads and displays

System.out.println("\nAccessing cat.png again...");

image1.display(); // Cached display

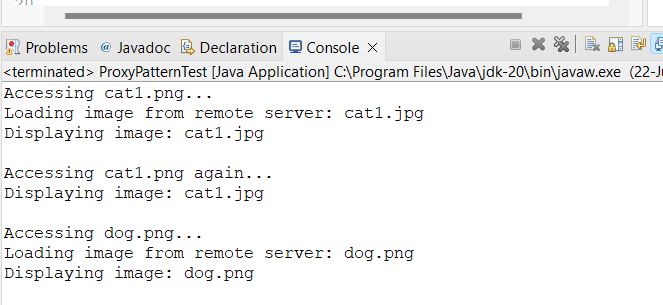
System.out.println("\nAccessing dog.png...");

image2.display(); // Loads and displays

}

}

**Output:**

****

**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

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

**Source Code:**

// Stock.java

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

// StockMarket.java

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private String stockName;

private double price;

public StockMarket(String stockName, double price) {

this.stockName = stockName;

this.price = price;

}

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, price);

}

}

public void setPrice(double newPrice) {

System.out.println("\nUpdating price of " + stockName + " to " + newPrice);

this.price = newPrice;

notifyObservers();

}

}

// Observer.java

public interface Observer {

void update(String stockName, double newPrice);

}

// MobileApp.java

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

public void update(String stockName, double newPrice) {

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

}

}

// WebApp.java

public class WebApp implements Observer {

private String appName;

public WebApp(String appName) {

this.appName = appName;

}

public void update(String stockName, double newPrice) {

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

}

}

// ObserverPatternTest.java

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket tcsStock = new StockMarket("TCS", 3800.00);

Observer mobileClient = new MobileApp("MyMobileApp");

Observer webClient = new WebApp("MyWebApp");

tcsStock.registerObserver(mobileClient);

tcsStock.registerObserver(webClient);

tcsStock.setPrice(3850.50); // Notify all observers

tcsStock.setPrice(3900.75); // Notify again

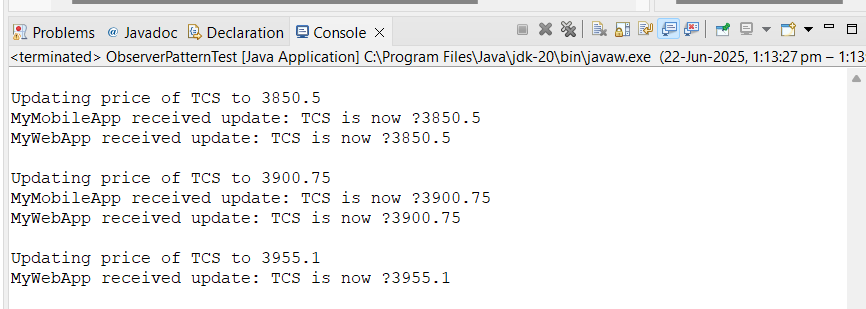
tcsStock.removeObserver(mobileClient);

tcsStock.setPrice(3955.10); // Only WebApp will be notified

}

}

**Output:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**Steps:**

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

**Source Code:**

// PaymentStrategy.java

public interface PaymentStrategy {

void pay(double amount);

}

// CreditCardPayment.java

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card: " + cardNumber);

}

}

// PayPalPayment.java

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

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

}

}

// PaymentContext.java

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void makePayment(double amount) {

if (strategy == null) {

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

} else {

strategy.pay(amount);

}

}

}

// StrategyPatternTest.java

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay using Credit Card

PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9876-5432");

context.setPaymentStrategy(creditCard);

context.makePayment(1500.75);

// Pay using PayPal

PaymentStrategy paypal = new PayPalPayment("user@example.com");

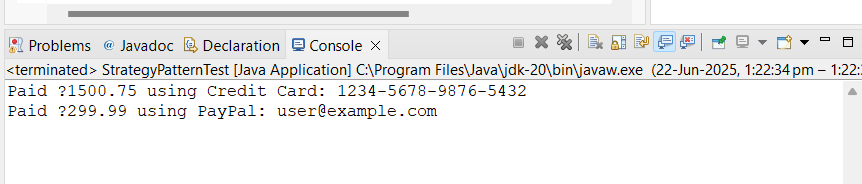
context.setPaymentStrategy(paypal);

context.makePayment(299.99);

}

}

**Output:**

****

**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Steps:**

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

**Source Code:**

// Command.java

public interface Command {

void execute();

}

// LightOnCommand.java

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

// LightOffCommand.java

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// Light.java

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// RemoteControl.java

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

if (command != null) {

command.execute();

} else {

System.out.println("No command set.");

}

}

}

// CommandPatternTest.java

public class CommandPatternTest {

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 ON the light:");

remote.setCommand(lightOn);

remote.pressButton();

System.out.println("Turning OFF the light:");

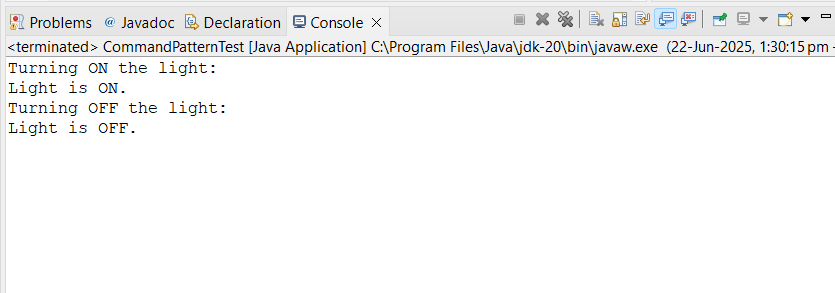
remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Steps:**

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

**Source Code:**

// Student.java

public class Student {

private String name;

private String id;

private String grade;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

// StudentView.java

public class StudentView {

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

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

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

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

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

}

}

// StudentController.java

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Controller logic: Setters & Getters

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

// MVCPatternTest.java

public class MVCPatternTest {

public static void main(String[] args) {

// Create model

Student student = new Student();

student.setName("Sravanthi");

student.setId("STU123");

student.setGrade("A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Display initial data

controller.updateView();

System.out.println("\nUpdating student grade...\n");

// Update model data via controller

controller.setStudentGrade("A+");

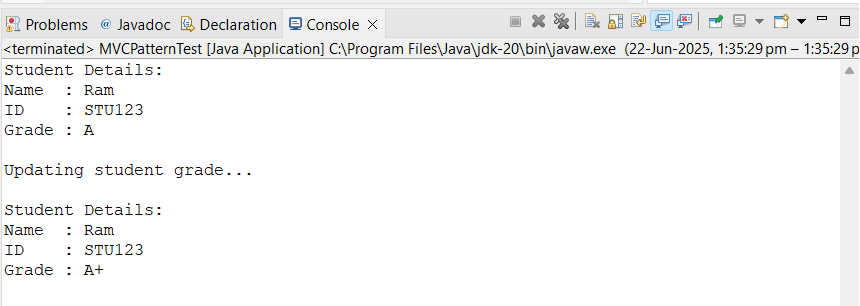
// Display updated data

controller.updateView();

}

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Steps:**

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

**Source Code:**

// CustomerRepository.java

public interface CustomerRepository {

String findCustomerById(String id);

}

// CustomerRepositoryImpl.java

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

// Simulate fetching customer data

return "Customer{id='" + id + "', name='Sravanthi', email='sravanthi@example.com'}";

}

}

// CustomerService.java

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor-based dependency injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(String id) {

String customer = customerRepository.findCustomerById(id);

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

}

}

// CustomerService.java

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor-based dependency injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(String id) {

String customer = customerRepository.findCustomerById(id);

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

}

}

// DependencyInjectionTest.java

public class DependencyInjectionTest {

public static void main(String[] args) {

// Create the repository

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject the repository into the service

CustomerService service = new CustomerService(repository);

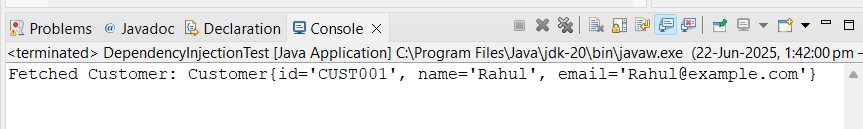
// Use the service to get customer info

service.getCustomerDetails("CUST001");

}

}

**Output:**

****