**DESIGN PATTERNS AND 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.

**SOLUTION:**

**LOGGER CLASS:**

public class Logger {

private static Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); }

return instance;

}

public void log(String message) {

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

}

}

**MAIN CLASS:**

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

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

Logger logger2 = Logger.getInstance();

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

if (logger1 == logger2) {

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

} else {

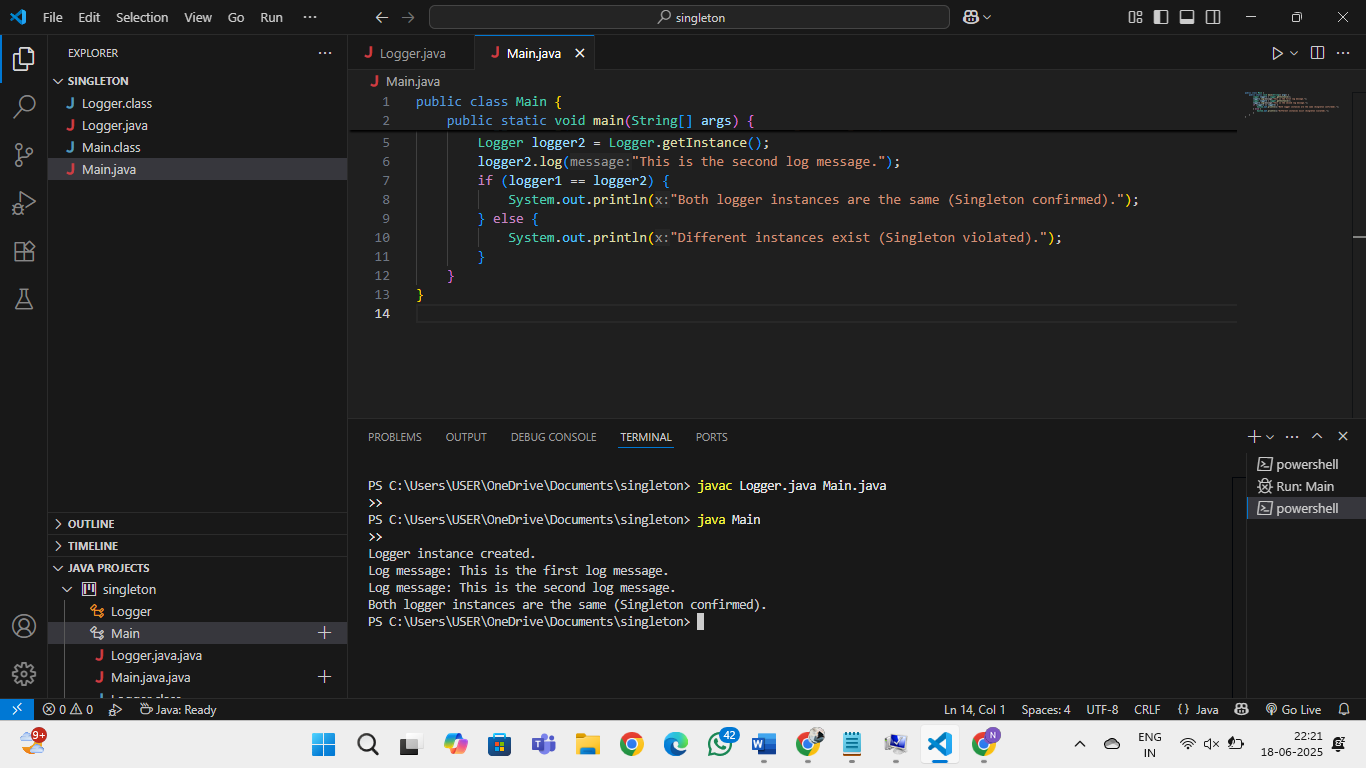
System.out.println("Different instances exist (Singleton violated).");

}

}

}

**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 class:**

public interface Document {

void open();

}

**WordDocument Class:**

public class WordDocument implements Document {

public void open() {

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

}

}

**PdfDocument Class:**

public class PdfDocument implements Document {

public void open() {

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

}

}

**ExcelDocument Class:**

public class ExcelDocument implements Document {

public void open() {

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

}

}

**DocumentFactory Class:**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory Class:**

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory Class:**

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory Class:**

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**Main Class:**

public class Main {

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();

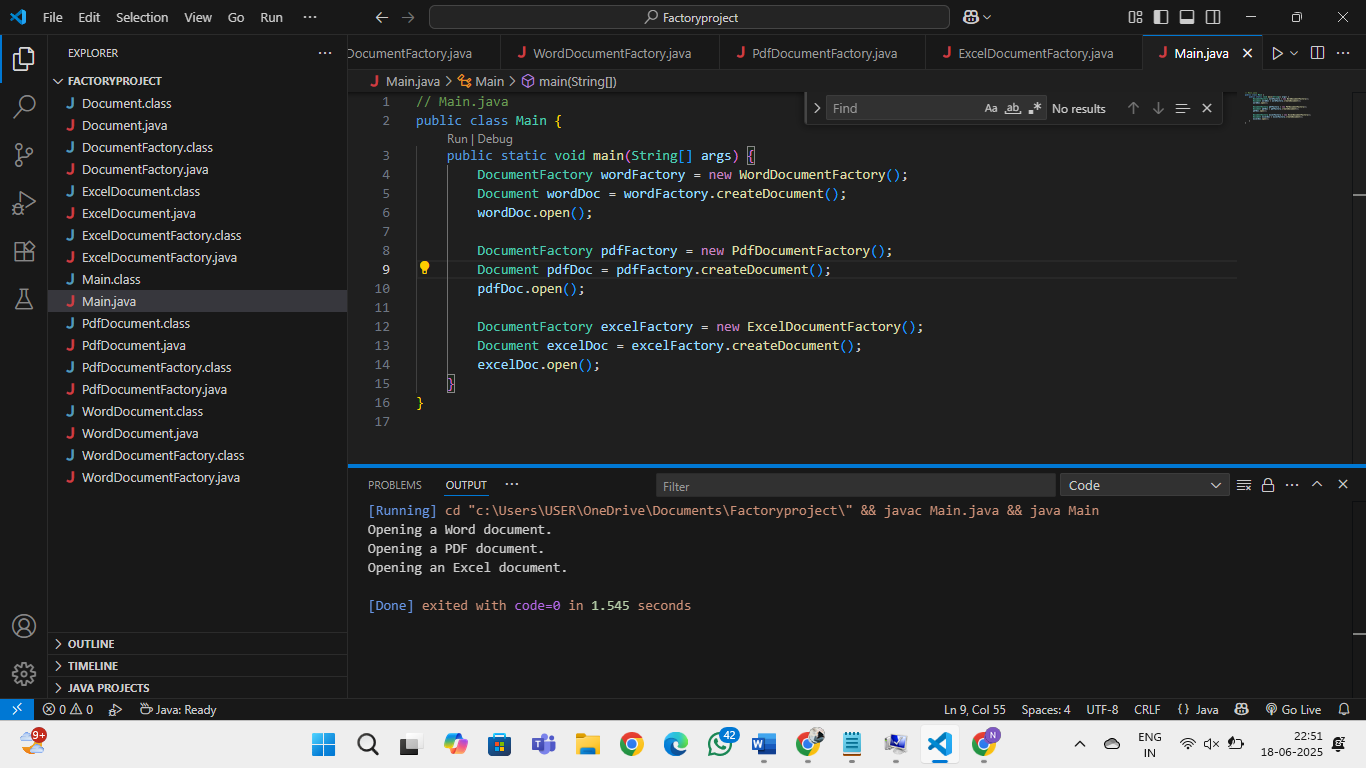
Document excelDoc = excelFactory.createDocument();

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.

**SOLUTION:**

public class BuilderPatternExample {

    public static class Computer {

        private String CPU;

        private String RAM;

        private String storage;

        private boolean isGraphicsCardEnabled;

        private boolean isBluetoothEnabled;

        private Computer(Builder builder) {

            this.CPU = builder.CPU;

            this.RAM = builder.RAM;

            this.storage = builder.storage;

            this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;

            this.isBluetoothEnabled = builder.isBluetoothEnabled;

        }

@Override

public String toString() {

    return "Computer [CPU=" + cpu +

           ", RAM=" + ram +

           ", Storage=" + storage +

           ", Graphics Card Enabled=" + hasGraphicsCard +

           ", Bluetooth Enabled=" + hasBluetooth + "]";

}

        @Override

        public String toString() {

            return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage +

                   ", Graphics Card Enabled=" + isGraphicsCardEnabled +

                   ", Bluetooth Enabled=" + isBluetoothEnabled + "]";

        }

        public static class Builder {

            private String CPU;

            private String RAM;

            private String storage;

            private boolean isGraphicsCardEnabled;

            private boolean isBluetoothEnabled;

            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 setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {

                this.isGraphicsCardEnabled = isGraphicsCardEnabled;

                return this;

            }

            public Builder setBluetoothEnabled(boolean isBluetoothEnabled) {

                this.isBluetoothEnabled = isBluetoothEnabled;

                return this;

            }

            public Computer build() {

                return new Computer(this);

            }

        }

    }

    public static void main(String[] args) {

        Computer gamingComputer = new Computer.Builder()

            .setCPU("Intel i9")

            .setRAM("32GB")

            .setStorage("1TB SSD")

            .setGraphicsCardEnabled(true)

            .setBluetoothEnabled(true)

            .build();

        Computer officeComputer = new Computer.Builder()

            .setCPU("Intel i5")

            .setRAM("16GB")

            .setStorage("512GB SSD")

            .setGraphicsCardEnabled(false)

            .setBluetoothEnabled(false)

            .build();

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

        System.out.println(gamingComputer);

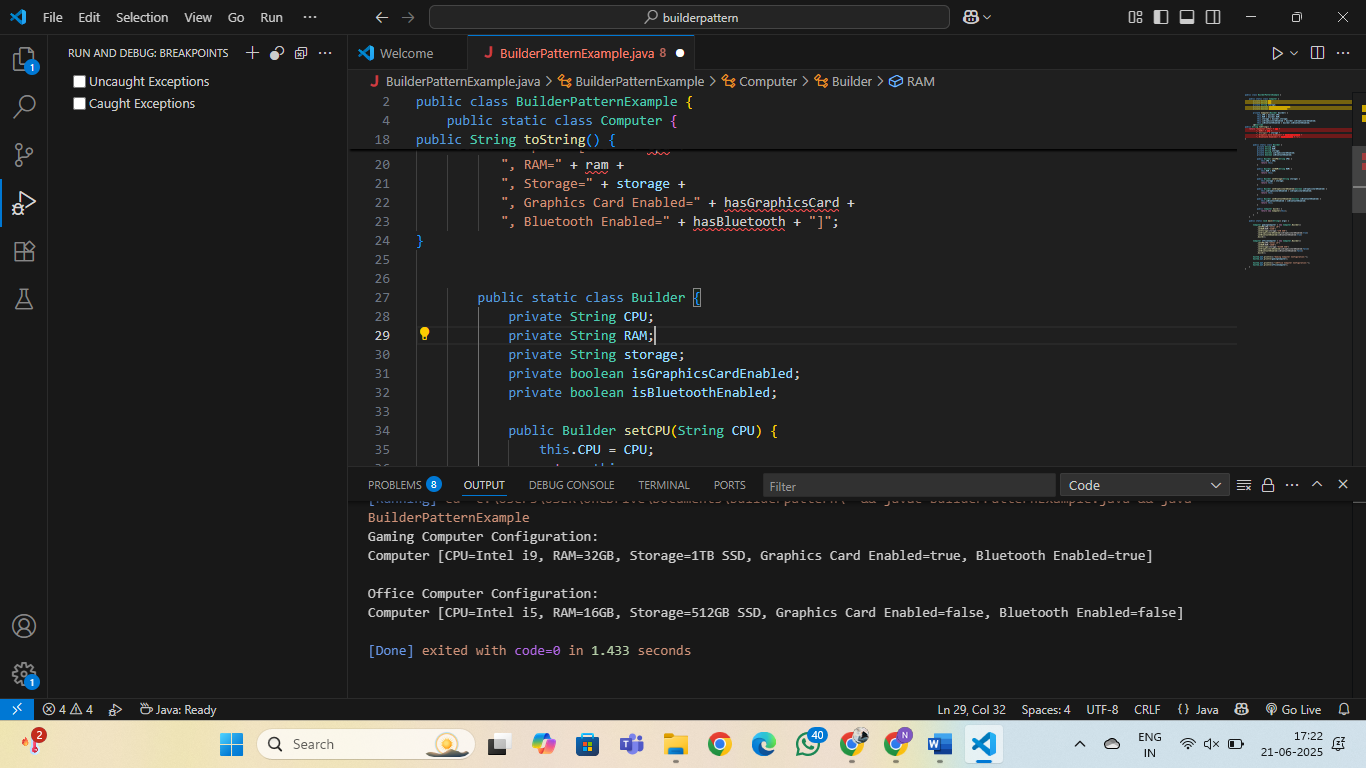
        System.out.println("\nOffice Computer Configuration:");

        System.out.println(officeComputer);

    }

}

**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 Class:**

public interface PaymentProcessor {

void processPayment(double amount);

}

**PayPalGateway:**

public class PayPalGateway {

public void makePayment(double amount) {

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

}

}

**StripeGateway:**

public class StripeGateway {

public void performPayment(double amount) {

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

}

}

**PayPalAdapter Class:**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

**StripeAdapter Class:**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.performPayment(amount);

}

}

**Main Class:**

public class Main {

public static void main(String[] args) {

PayPalGateway paypal = new PayPalGateway();

PaymentProcessor paypalAdapter = new PayPalAdapter(paypal);

paypalAdapter.processPayment(1000);

StripeGateway stripe = new StripeGateway();

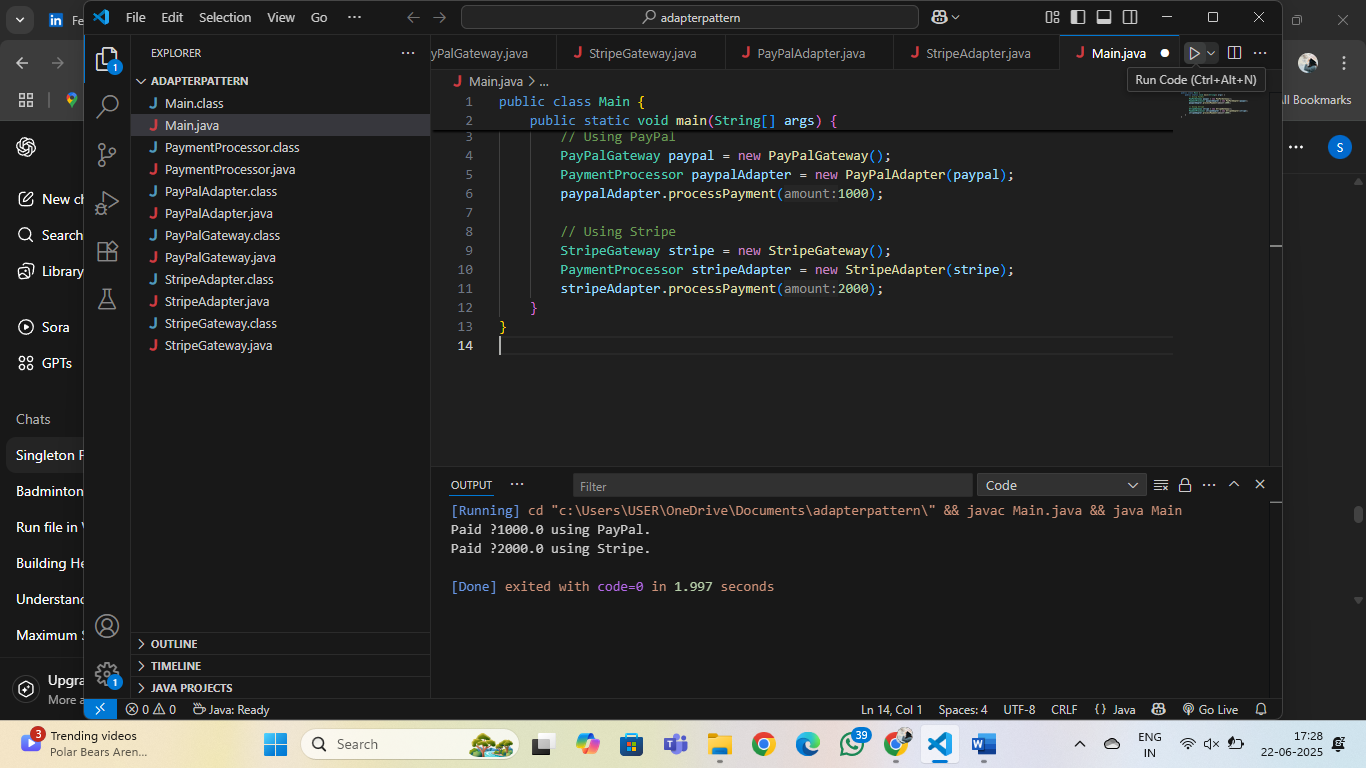
PaymentProcessor stripeAdapter = new StripeAdapter(stripe);

stripeAdapter.processPayment(2000);

}

}

**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.

**SOLUTION:**

**Notifier Class:**

public interface Notifier {

void send(String message);

}

**EmailNotifier Class:**

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

**NotifierDecorator Class:**

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

@Override

public void send(String message) {

wrappedNotifier.send(message); // Delegate call to wrapped object

}

}

**SMSNotifierDecorator Class:**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // Call previous notifier

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

}

}

**SlackNotifier Class:**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // Call previous notifier

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

}

}

**Main Class:**

public class Main {

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

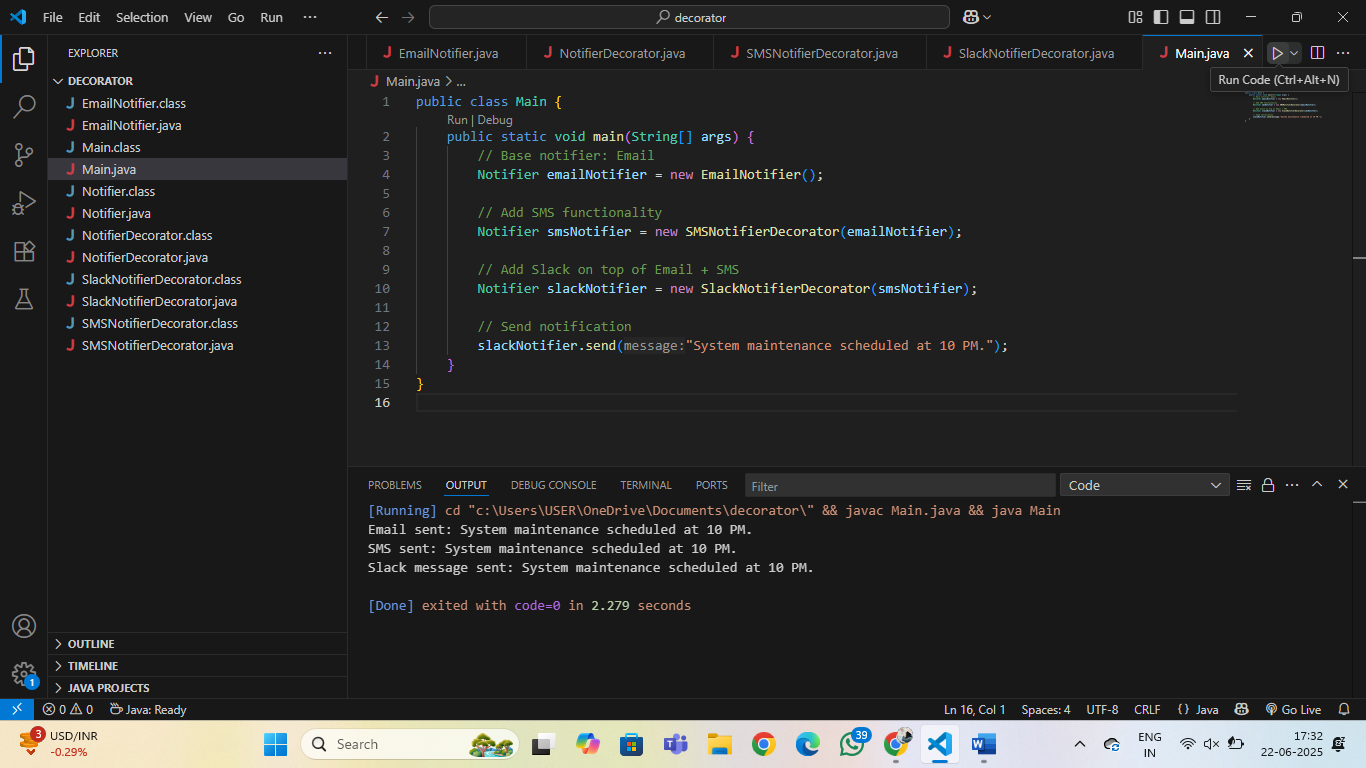
Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

slackNotifier.send("System maintenance scheduled at 10 PM.");

}

}

**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:**

interface Image {

    void display();

}

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);

        try {

            Thread.sleep(1000); // simulate delay

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    @Override

    public void display() {

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

    }

}

class ProxyImage implements Image {

    private RealImage realImage;

    private String filename;

    public ProxyImage(String filename) {

        this.filename = filename;

    }

    @Override

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(filename);

        } else {

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

        }

        realImage.display();

    }

}

public class ProxyPatternExample {

    public static void main(String[] args) {

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

        image1.display();

        System.out.println();

        image1.display();

        System.out.println();

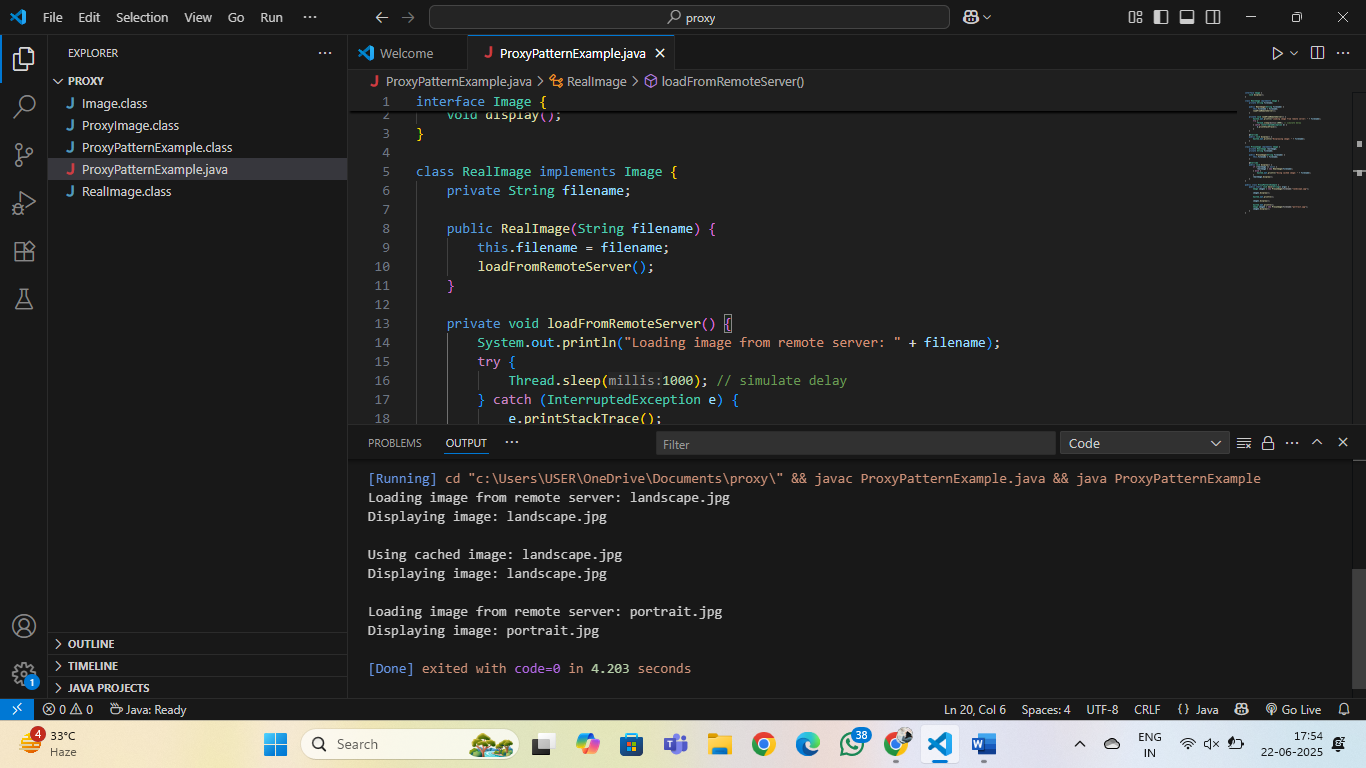
        Image image2 = new ProxyImage("portrait.jpg");

        image2.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.

**SOLUTION:**

import java.util.ArrayList;

import java.util.List;

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

interface Observer {

void update(String stockName, double price);

}

class StockMarket implements Stock {

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

private String stockName;

private double price;

@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(stockName, price);

}

}

public void setStockPrice(String stockName, double price) {

this.stockName = stockName;

this.price = price;

System.out.println("\nStock updated: " + stockName + " → ₹" + price);

notifyObservers();

}

}

class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(String stockName, double price) {

System.out.println(name + " [Mobile App] received update → " + stockName + ": ₹" + price);

}

}

class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(String stockName, double price) {

System.out.println(name + " [Web App] received update → " + stockName + ": ₹" + price);

}

}

public class ObserverPatternExample {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileClient = new MobileApp("Client A");

Observer webClient = new WebApp("Client B");

stockMarket.registerObserver(mobileClient);

stockMarket.registerObserver(webClient);

stockMarket.setStockPrice("TCS", 3650.75);

stockMarket.setStockPrice("INFY", 1523.40);

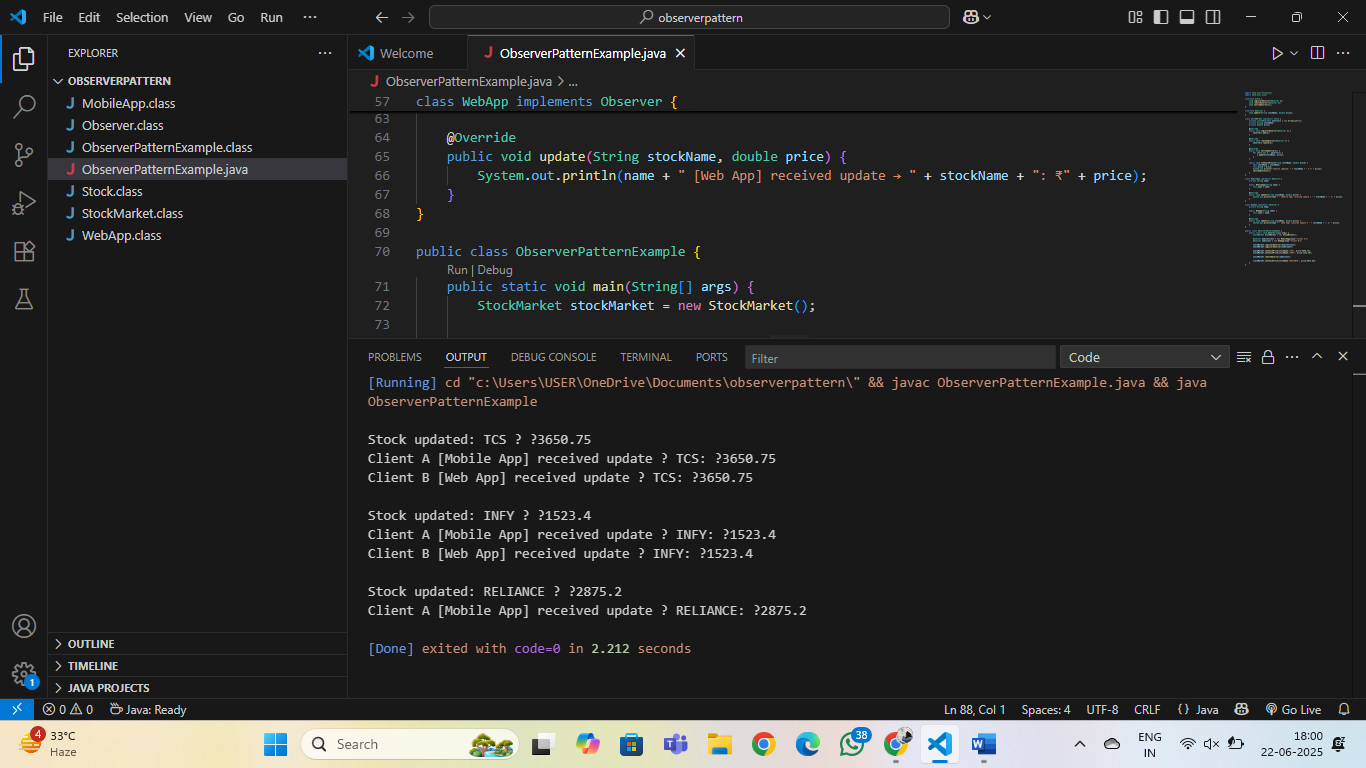
stockMarket.removeObserver(webClient);

stockMarket.setStockPrice("RELIANCE", 2875.20);

}

}

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:**

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

@Override

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;

}

@Override

public void pay(double amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void makePayment(double amount) {

strategy.pay(amount);

}

}

public class StrategyPatternExample {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234567812345678"));

context.makePayment(2500.00);

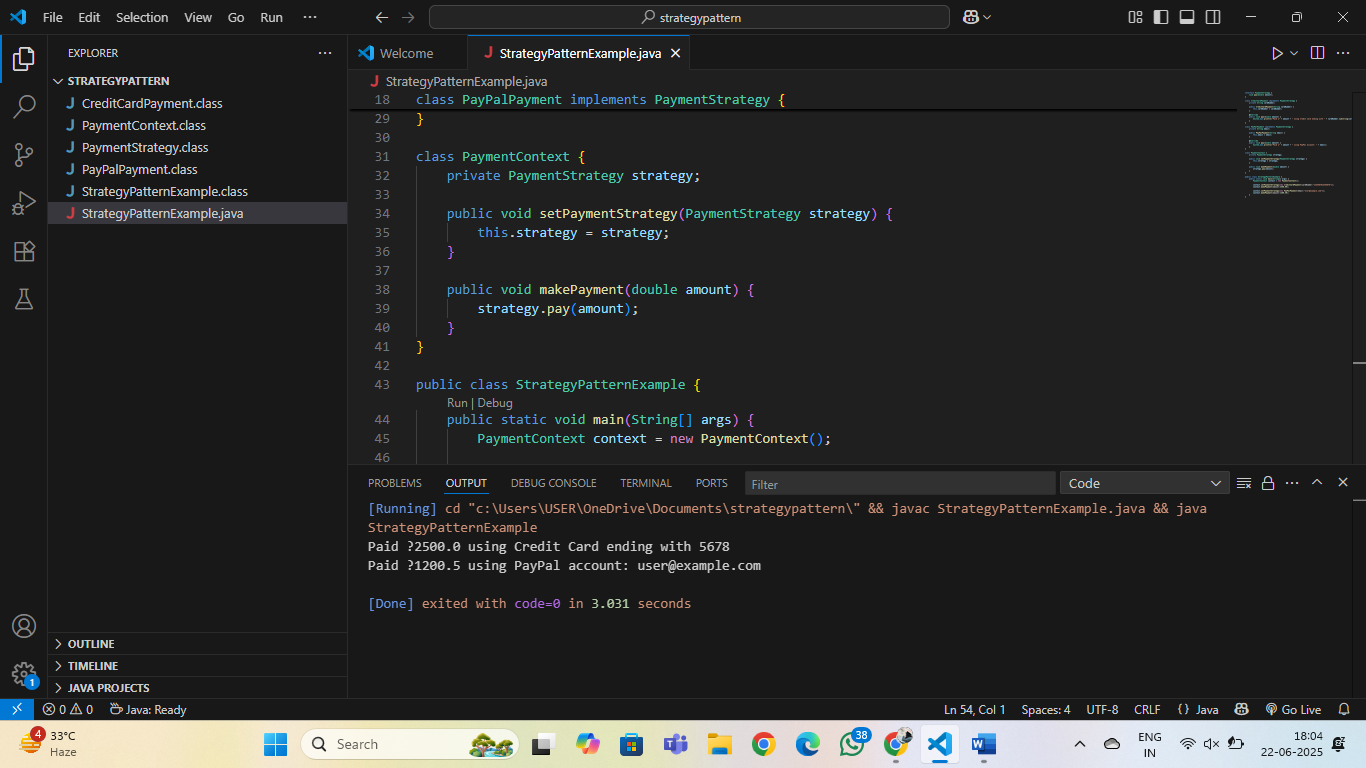
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.makePayment(1200.50);

}

}

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:**

interface Command {

void execute();

}

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

public class CommandPatternExample {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

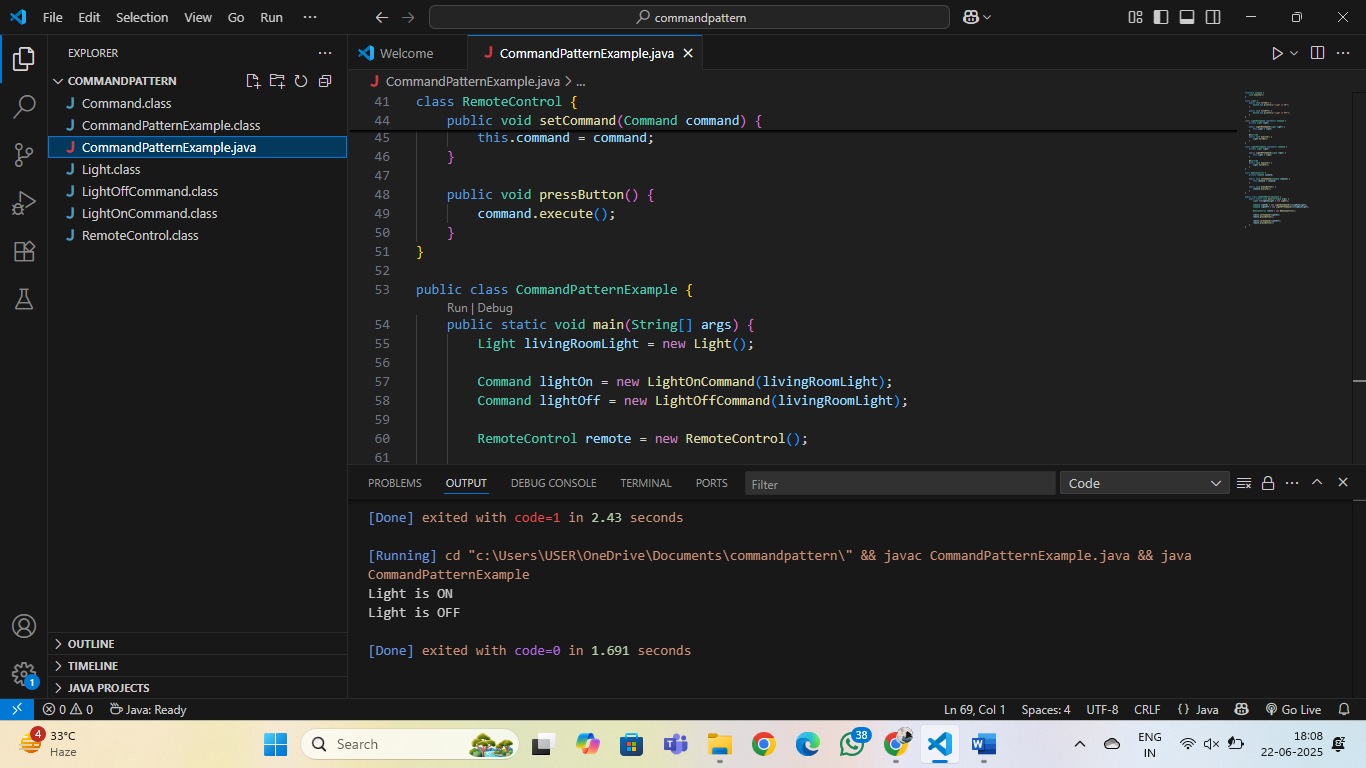
remote.setCommand(lightOff);

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**.

**SOLUTION:**

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;

    }

}

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);

    }

}

class StudentController {

    private Student model;

    private StudentView view;

    public StudentController(Student model, StudentView view) {

        this.model = model;

        this.view = view;

    }

    public void setStudentName(String name) {

        model.setName(name);

    }

    public void setStudentId(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());

    }

}

public class MVCPatternExample {

    public static void main(String[] args) {

        Student model = new Student();

        model.setName("Priya");

        model.setId("S102");

        model.setGrade("A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(model, view);

        controller.updateView();

        controller.setStudentName("Karthik");

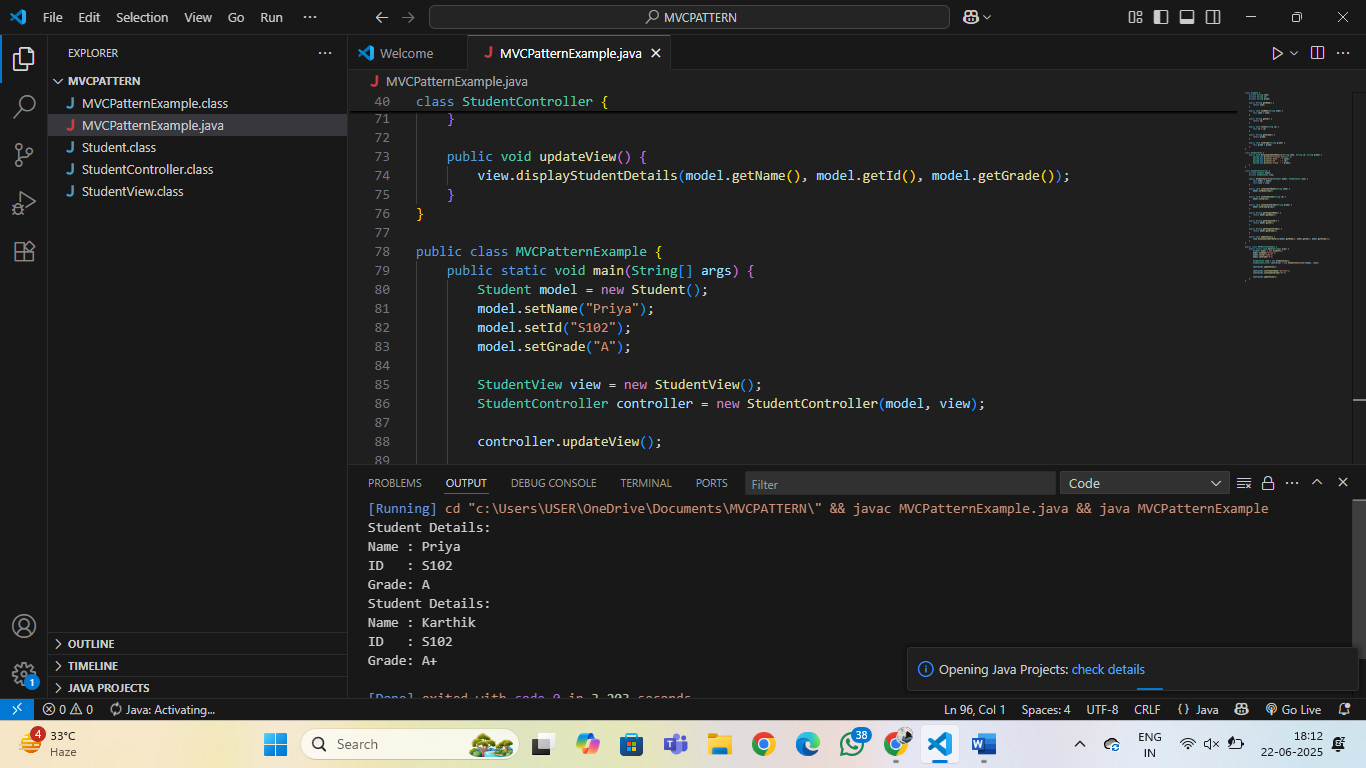
        controller.setStudentGrade("A+");

        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.

**SOLUTION:**

interface CustomerRepository {

    String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public String findCustomerById(String id) {

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

    }

}

class CustomerService {

    private CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void getCustomerDetails(String id) {

        String customer = customerRepository.findCustomerById(id);

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

    }

}

public class DependencyInjectionExample {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repository);

        service.getCustomerDetails("C101");

    }

}

**OUTPUT:**

