**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.

**Solution:**

**Logger.java**

public class Logger {

private static volatile Logger instance;

private Logger() {

System.out.println("Logger Initialized");

}

public static Logger getInstance() {

if (instance == null) {

synchronized (Logger.class) {

if (instance == null) {

instance = new Logger();

}

}

}

return instance;

}

public void log(String message) {

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

}

}

**LoggerTest.java**

public class LoggerTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Message from logger1");

Logger logger2 = Logger.getInstance();

logger2.log("Message from logger2");

if (logger1 == logger2) {

System.out.println("Both logger instances are the same (Thread-safe Singleton verified).");

} else {

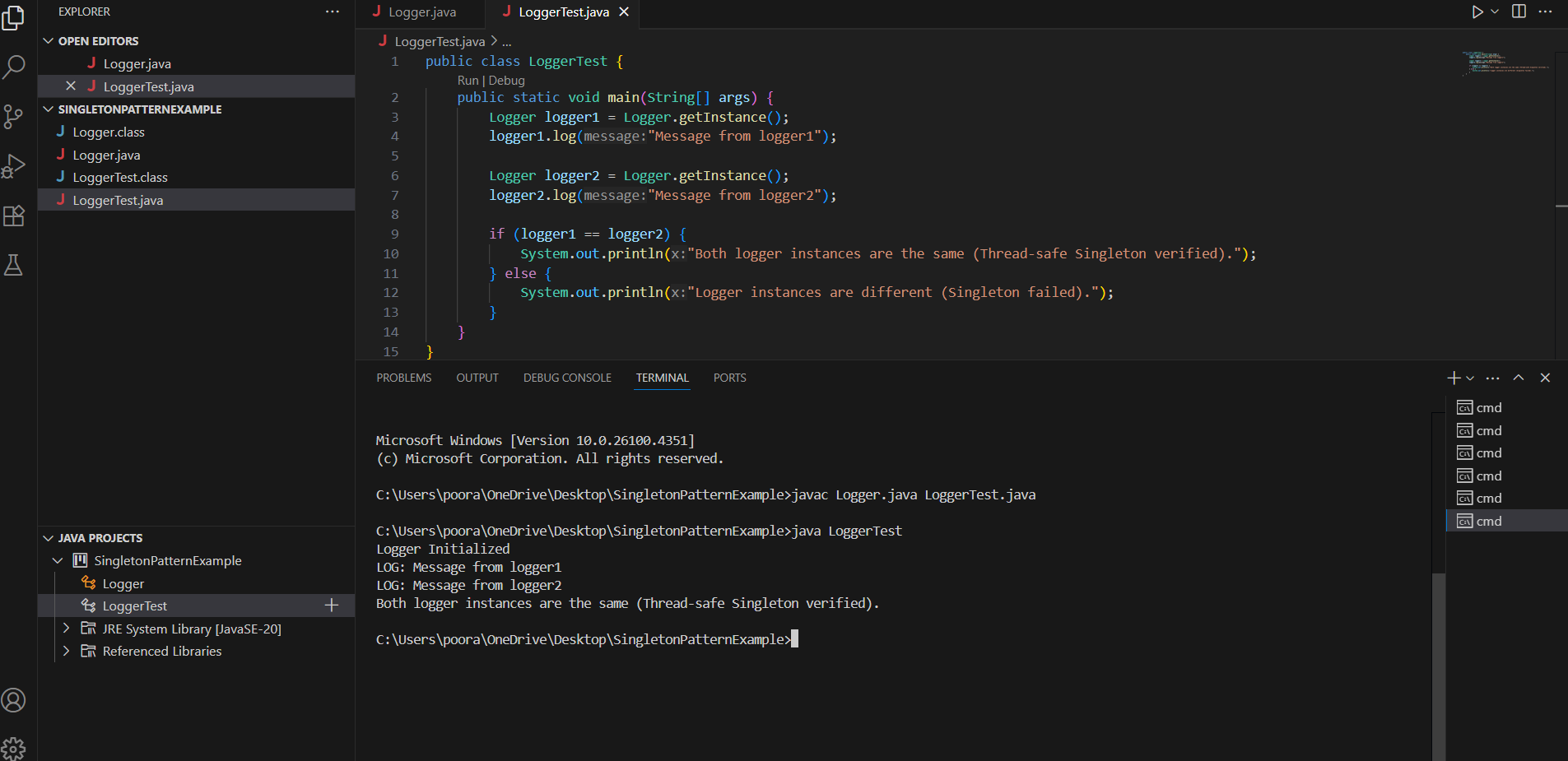
System.out.println("Logger instances are different (Singleton failed).");

}

}

}

**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.

**Solution:**

**Document.java**

public interface Document {

void open();

}

**WordDocument.java**

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

**PdfDocument.java**

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

**DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordFactory.java**

public class WordFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfFactory.java**

public class PdfFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelFactory.java**

public class ExcelFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**DocumentTest.java**

public class DocumentTest {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelFactory();

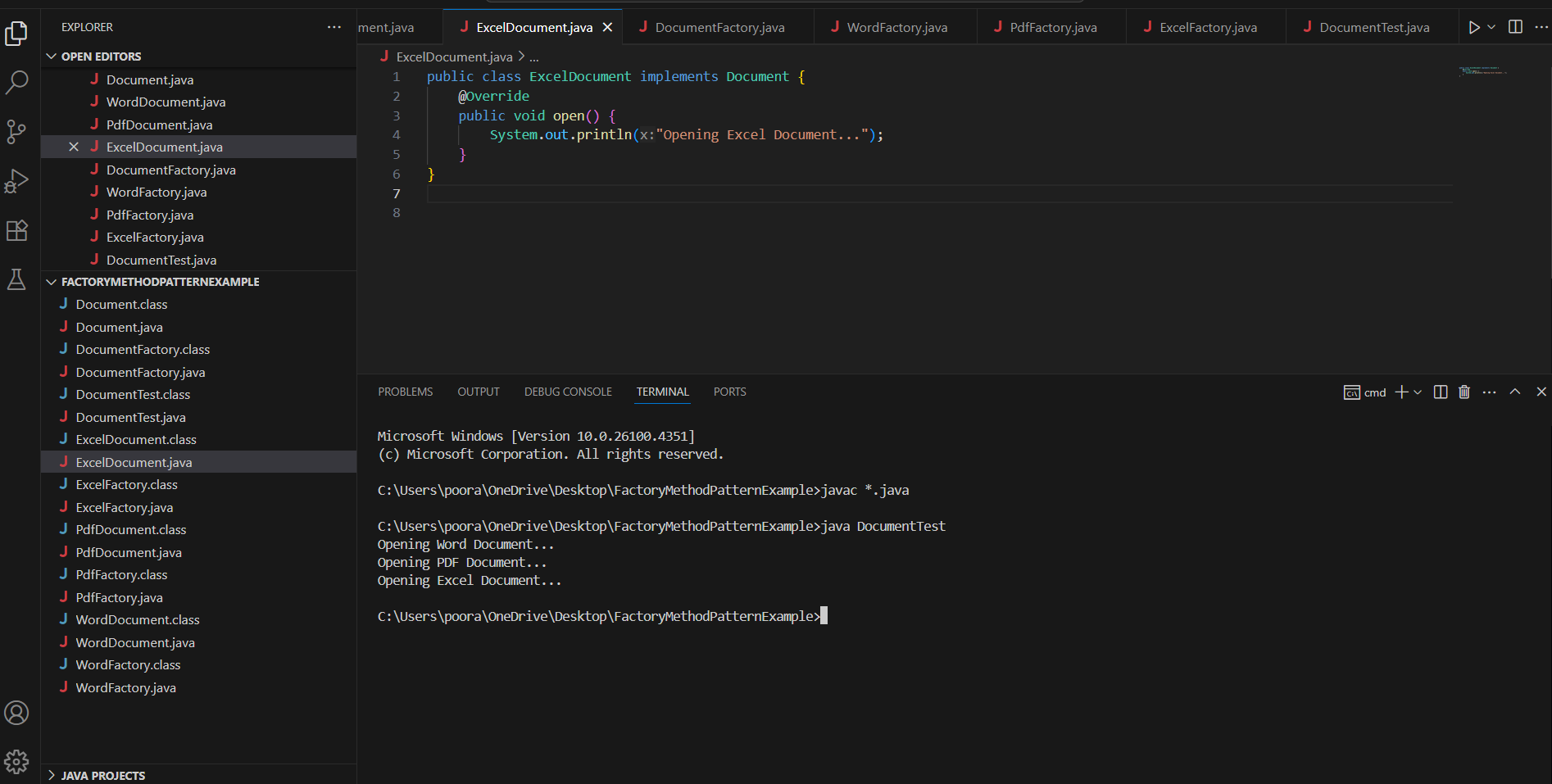
Document excel = excelFactory.createDocument();

excel.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.

**Solution:**

**Computer.java**

public class Computer {

// Required parameters

private String cpu;

private String ram;

// Optional parameters

private String storage;

private String graphicsCard;

private String os;

// Private constructor to enforce object creation via Builder

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.os = builder.os;

}

// Static nested Builder class

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private String os;

public Builder(String cpu, String ram) {

this.cpu = cpu;

this.ram = ram;

}

public Builder storage(String storage) {

this.storage = storage;

return this;

}

public Builder graphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder os(String os) {

this.os = os;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void showSpecs() {

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

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

System.out.println("Storage: " + (storage != null ? storage : "Not included"));

System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not included"));

System.out.println("OS: " + (os != null ? os : "Not included"));

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

}

}

**ComputerBuilderTest.java**

public class ComputerBuilderTest {

public static void main(String[] args) {

// Basic configuration

Computer basicComputer = new Computer.Builder("Intel i5", "8GB").build();

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

basicComputer.showSpecs();

// Gaming configuration

Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")

.storage("2TB SSD")

.graphicsCard("NVIDIA RTX 4090")

.os("Windows 11")

.build();

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

gamingComputer.showSpecs();

// Office configuration

Computer officeComputer = new Computer.Builder("AMD Ryzen 5", "16GB")

.storage("512GB SSD")

.os("Ubuntu Linux")

.build();

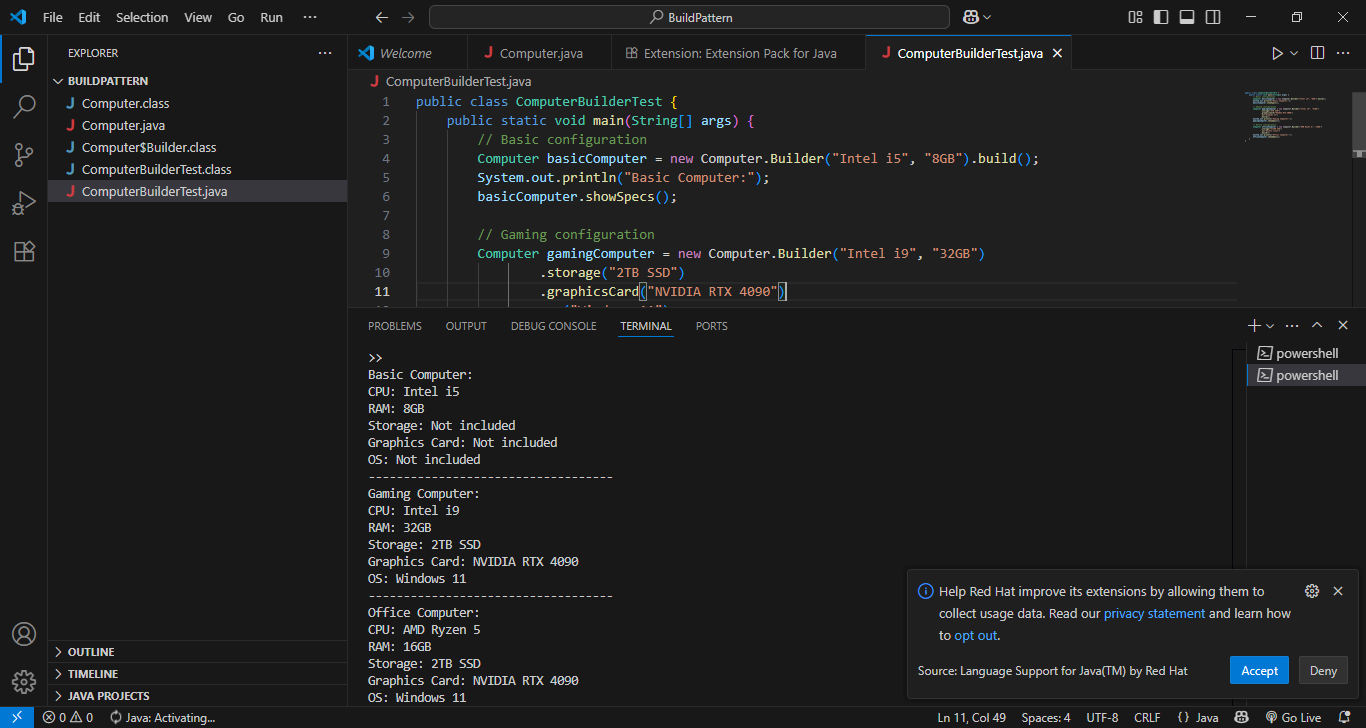
System.out.println("Office Computer:");

officeComputer.showSpecs();

}

}

**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.

**Solution:**

**PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment(double amount);

}

**PayPalGateway.java**

public class PayPalGateway {

public void makePayment(double amount) {

System.out.println("PayPal payment of $" + amount + " processed.");

}

}

**StripeGateway.java**

public class StripeGateway {

public void sendPayment(double amount) {

System.out.println("Stripe payment of $" + amount + " sent.");

}

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal = new PayPalGateway();

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe = new StripeGateway();

public void processPayment(double amount) {

stripe.sendPayment(amount);

}

}

**PaymentTest.java**

public class PaymentTest {

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter();

PaymentProcessor stripeProcessor = new StripeAdapter();

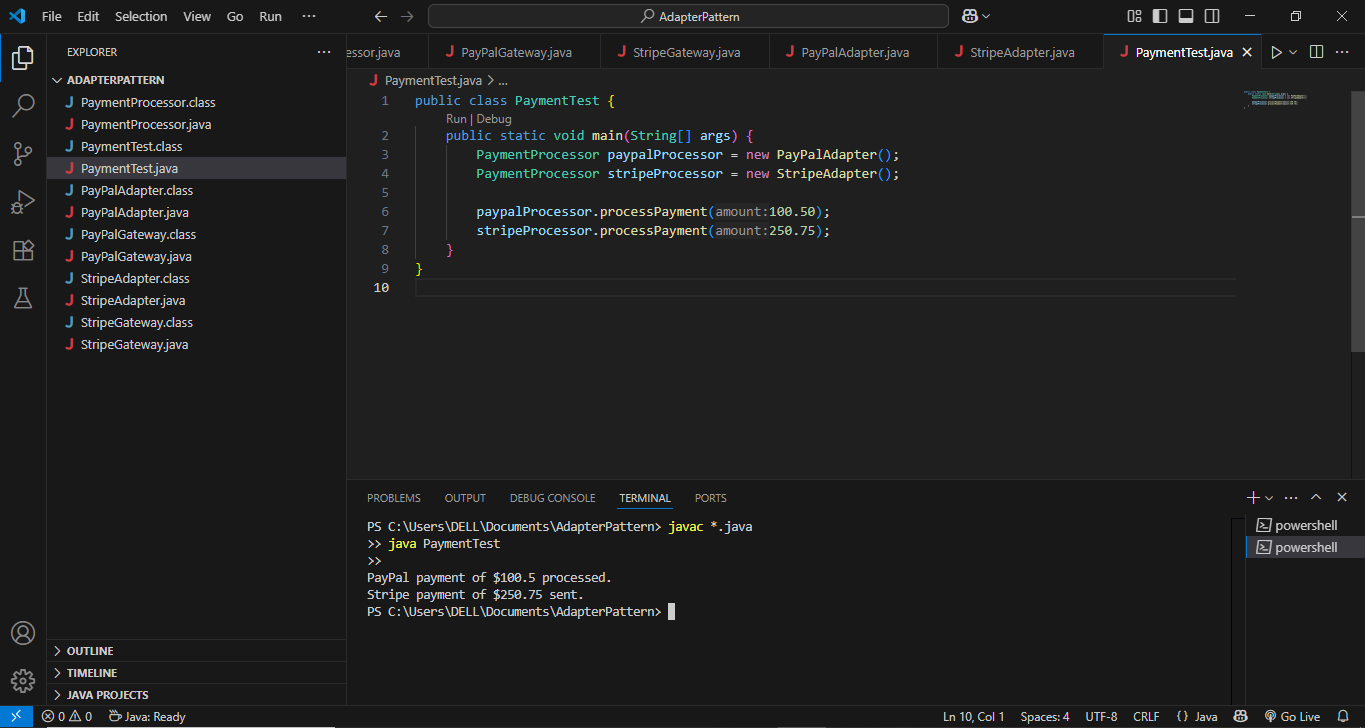
paypalProcessor.processPayment(100.50);

stripeProcessor.processPayment(250.75);

}

}

**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.

**Solution**

**Notifier.java**

public interface Notifier {

void send(String message);

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.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("SMS sent: " + 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("Slack message sent: " + message);

}

}

**NotificationTest.java**

public class NotificationTest {

public static void main(String[] args) {

// Send notification via Email only

Notifier emailOnly = new EmailNotifier();

System.out.println("Single Channel Notification:");

emailOnly.send("System update at 5 PM");

// Send notification via Email + SMS

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

System.out.println("\nEmail + SMS Notification:");

emailAndSMS.send("Server down alert");

// Send notification via Email + SMS + Slack

Notifier fullNotification = new SlackNotifierDecorator(

new SMSNotifierDecorator(

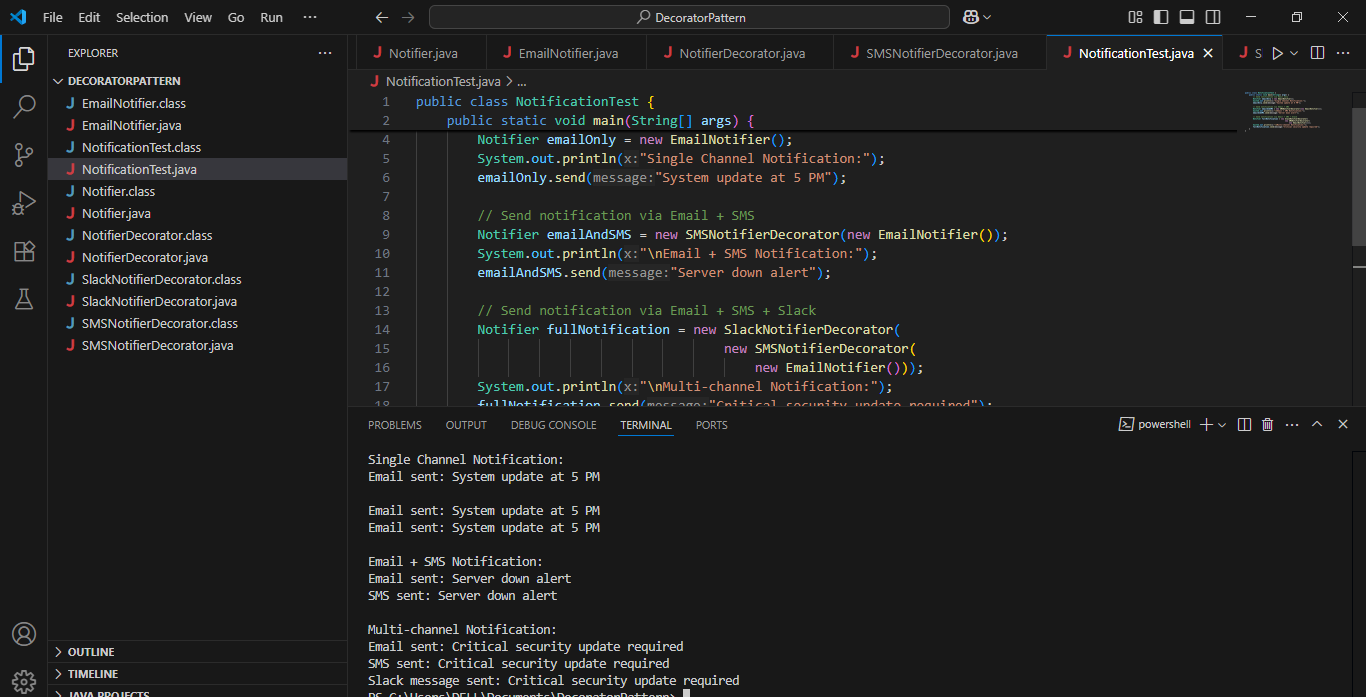
new EmailNotifier()));

System.out.println("\nMulti-channel Notification:");

fullNotification.send("Critical security update required");

}

}

**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.

**Solution**

**Image.java**

public interface Image {

void display();

}

**RealImage.java**

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromServer();

}

private void loadFromServer() {

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

}

@Override

public void display() {

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

}

}

**ProxyImage.java**

public class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

@Override

public void display() {

if (realImage == null) {

// Lazy initialization

realImage = new RealImage(fileName);

} else {

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

}

realImage.display();

}

}

**ProxyPatternTest.java**

public class ProxyPatternTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

// Image is loaded only when display() is called

System.out.println("First display:");

image1.display();

System.out.println("\nSecond display:");

image1.display();

// Another image

System.out.println("\nAnother image:");

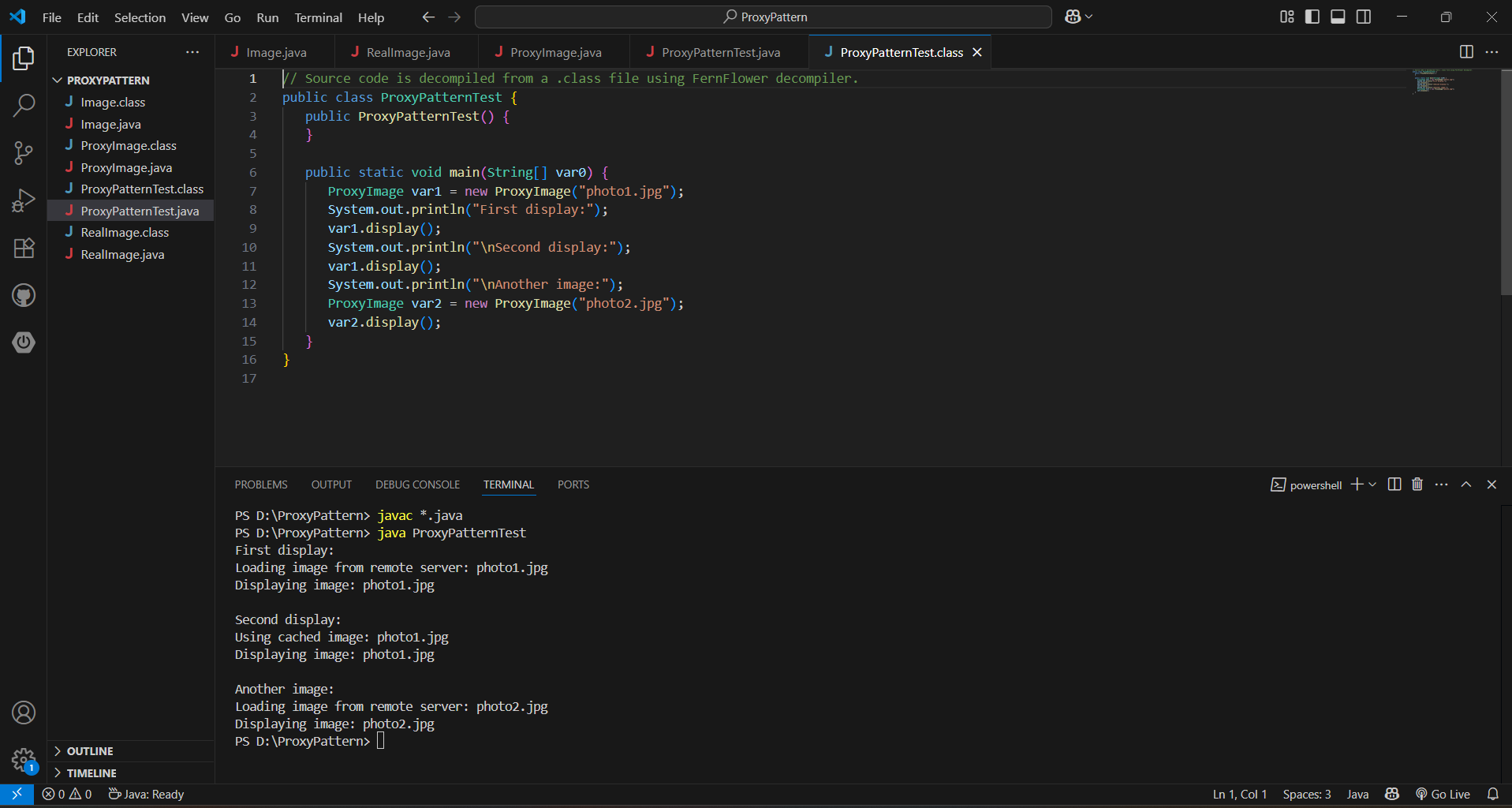
Image image2 = new ProxyImage("photo2.jpg");

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.

**Solution**

**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 double stockPrice;

public void setStockPrice(double price) {

this.stockPrice = price;

notifyObservers();

}

public double getStockPrice() {

return 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);

}

}

}

**Observer.java**

public interface Observer {

void update(double stockPrice);

}

**MobileApp.java**

public class MobileApp implements Observer {

private String appName;

public MobileApp(String name) {

this.appName = name;

}

@Override

public void update(double stockPrice) {

System.out.println(appName + " received stock price update: $" + stockPrice);

}

}

**WebApp.java**

public class WebApp implements Observer {

private String siteName;

public WebApp(String name) {

this.siteName = name;

}

@Override

public void update(double stockPrice) {

System.out.println(siteName + " received stock price update: $" + stockPrice);

}

}

**ObserverPatternTest.java**

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobileApp = new MobileApp("StockMobile");

Observer webApp = new WebApp("StockWeb");

market.registerObserver(mobileApp);

market.registerObserver(webApp);

System.out.println("Setting stock price to 100.0");

market.setStockPrice(100.0);

System.out.println("\nRemoving MobileApp observer...");

market.removeObserver(mobileApp);

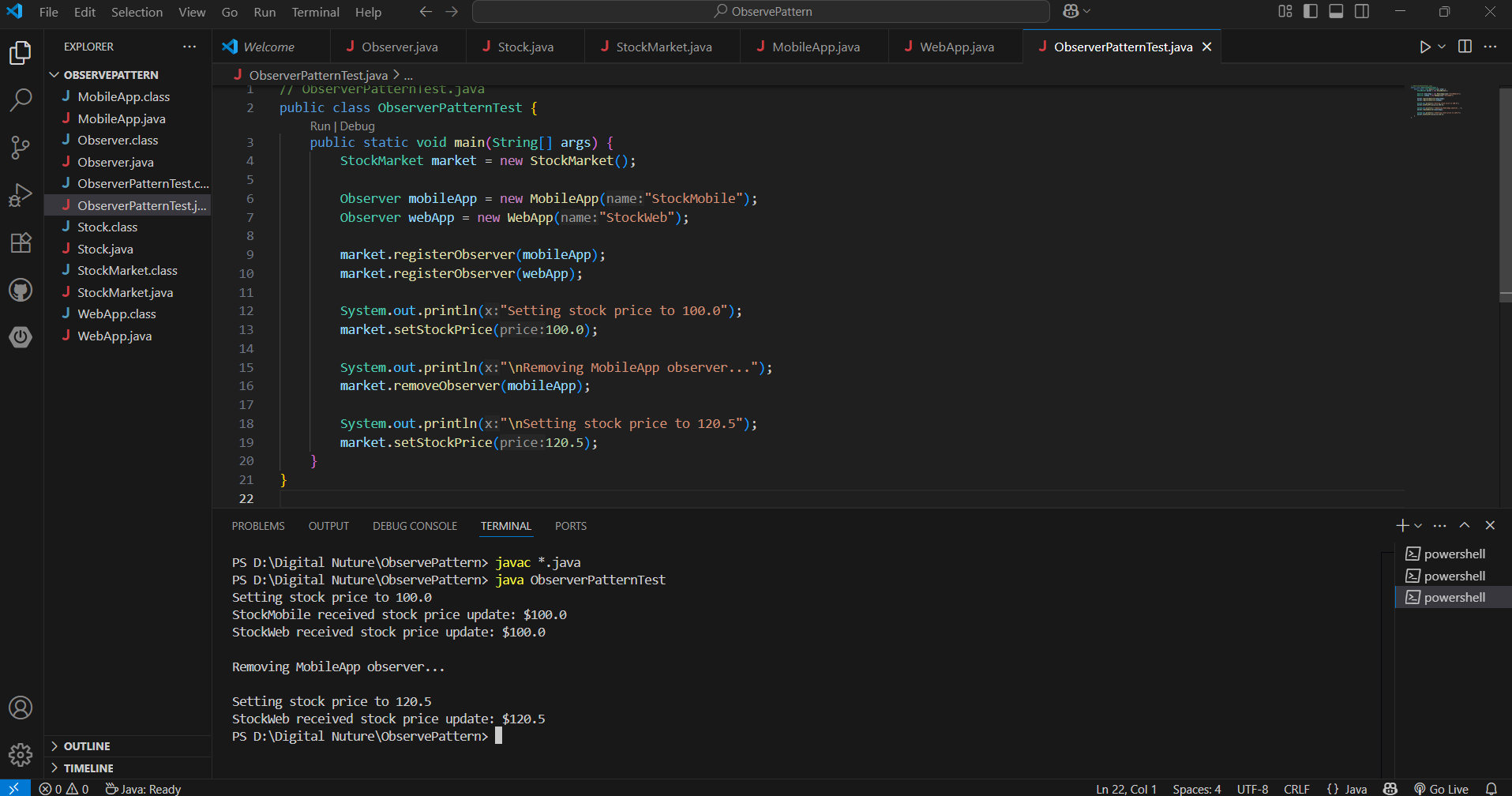
System.out.println("\nSetting stock price to 120.5");

market.setStockPrice(120.5);

}

}

**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.

**Solution**

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolder;

public CreditCardPayment(String cardNumber, String cardHolder) {

this.cardNumber = cardNumber;

this.cardHolder = cardHolder;

}

@Override

public void pay(double amount) {

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

}

}

**PayPalPayment.java**

public 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 (" + email + ").");

}

}

**PaymentContext.java**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void payAmount(double amount) {

if (strategy == null) {

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

} 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", "User ");

context.setPaymentStrategy(creditCard);

context.payAmount(1500.0);

System.out.println();

// Pay using PayPal

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

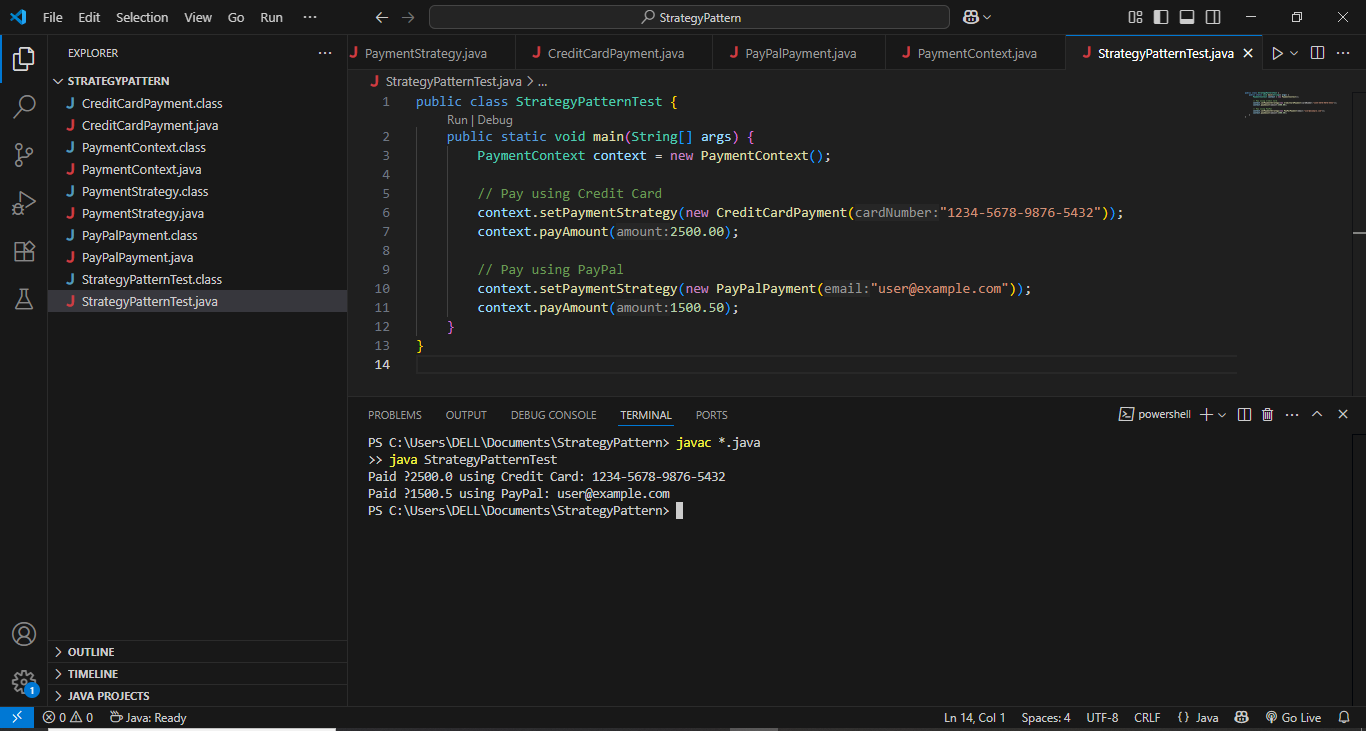
context.setPaymentStrategy(paypal);

context.payAmount(800.0);

}

}

**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.

**Solution**

**Command.java**

public interface Command {

void execute();

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**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.");

}

}

}

**Light.java**

public class Light {

private String location;

public Light(String location) {

this.location = location;

}

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**CommandPatternTest.java**

public class CommandPatternTest {

public static void main(String[] args) {

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

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

System.out.println("Pressing ON button:");

remote.setCommand(lightOn);

remote.pressButton();

System.out.println("\nPressing OFF button:");

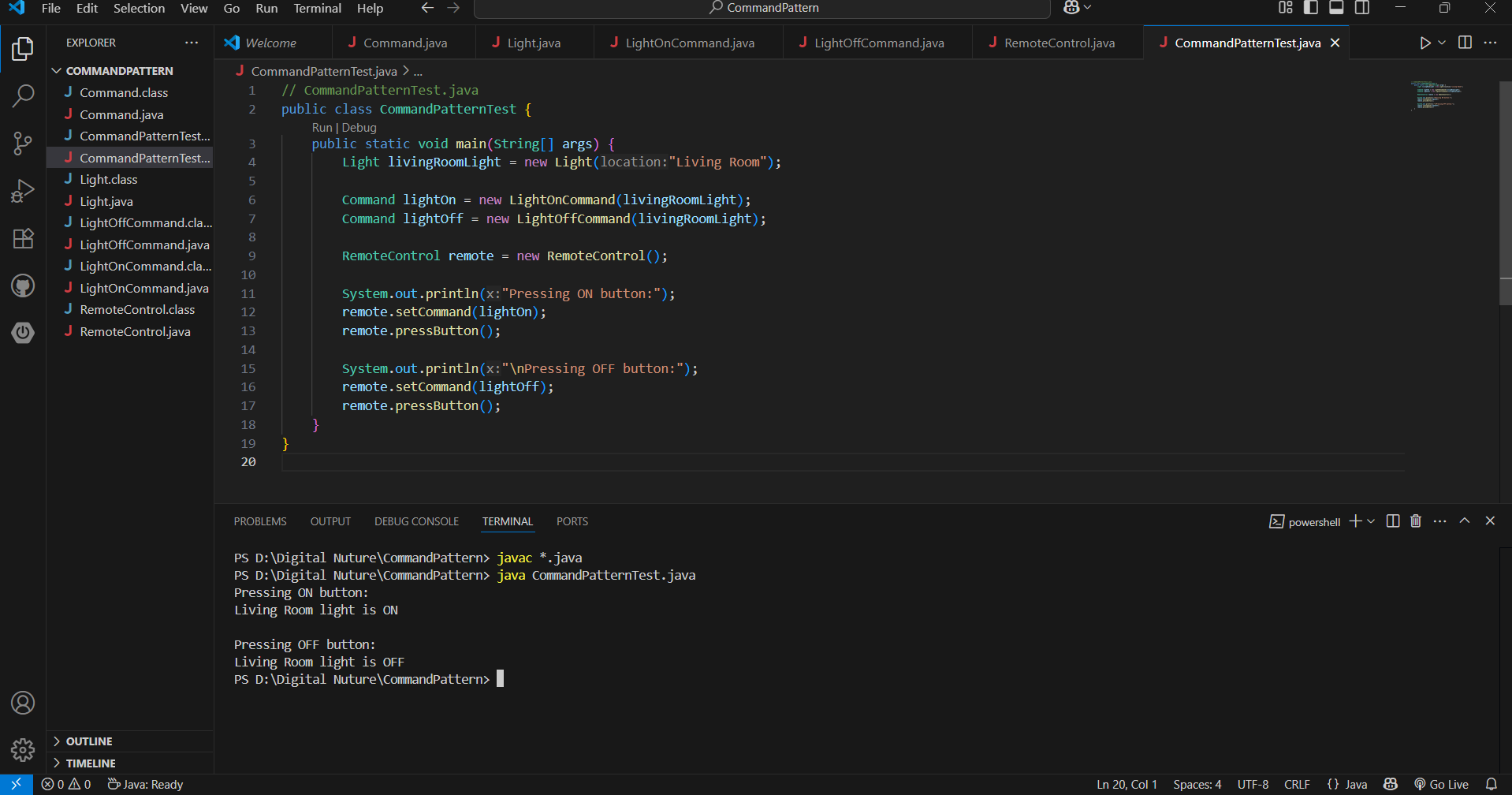
remote.setCommand(lightOff);

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.

**Solution**

**Student.java**

public class Student {

private String name;

private String id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

// Getters and setters

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getId() { return 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;

}

// Setters to update the model

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

// Getters to access model data

public String getStudentName() { return model.getName(); }

public String getStudentId() { return model.getId(); }

public String getStudentGrade() { return model.getGrade(); }

// Update the view with model data

public void updateView() {

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

}

}

**MVCPatternTest.java**

public class Main {

public static void main(String[] args) {

// Create model

Student student = new Student("Alice", "S001", "A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Display initial data

controller.updateView();

// Update model data

controller.setStudentName("Bob");

controller.setStudentGrade("B+");

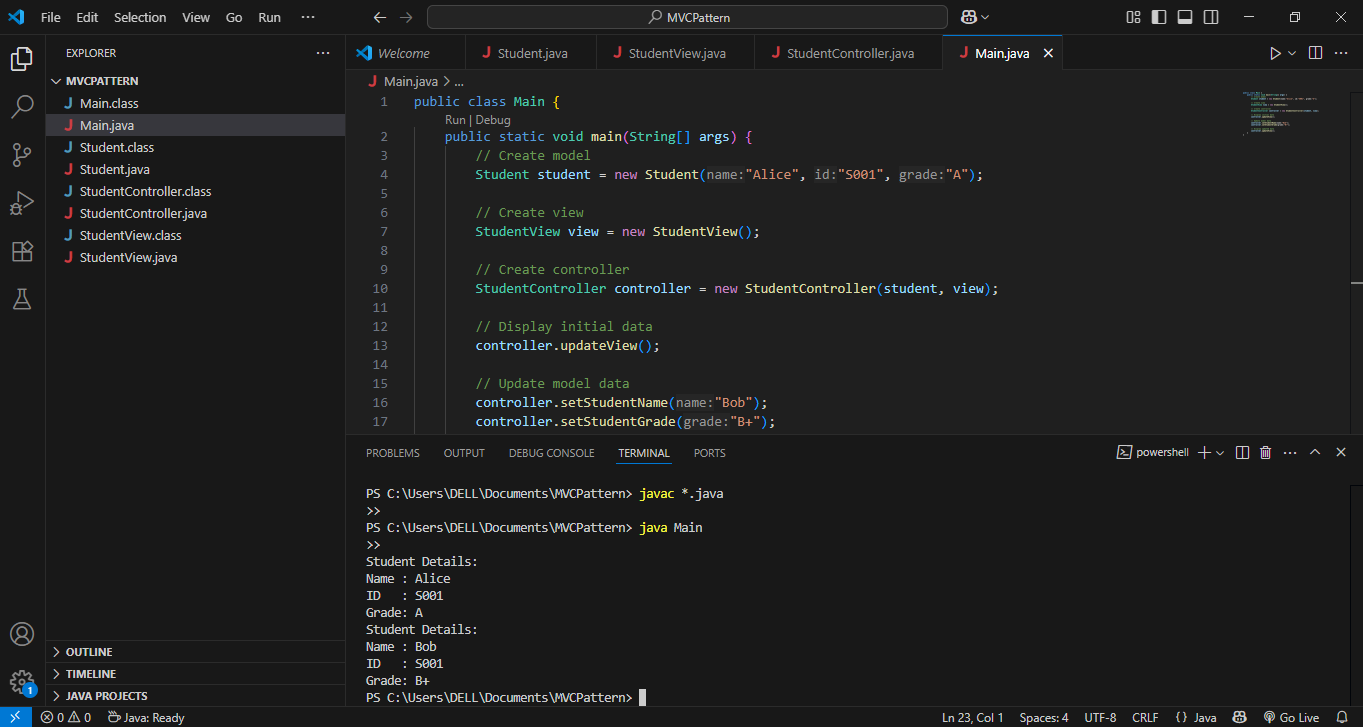
// Display updated data

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.

**Solution**

**CustomerRepository.java**

public interface CustomerRepository {

Customer findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

import java.util.HashMap;

import java.util.Map;

public class CustomerRepositoryImpl implements CustomerRepository {

private Map<Integer, Customer> database = new HashMap<>();

public CustomerRepositoryImpl() {

database.put(1, new Customer(1, "Alice"));

database.put(2, new Customer(2, "Bob"));

}

@Override

public Customer findCustomerById(int id) {

return database.get(id);

}

}

**CustomerService.java**

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void printCustomerName(int id) {

Customer customer = customerRepository.findCustomerById(id);

if (customer != null) {

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

} else {

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

}

}

}

**Customer.java**

public class Customer {

private int id;

private String name;

public Customer(int id, String name) {

this.id = id;

this.name = name;

}

public String getName() {

return name;

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

// Inject dependency

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

// Use service

service.printCustomerName(1);

service.printCustomerName(3);

}

}

**Output:**

