**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

public class Logger {

    private static Logger instance;

    private Logger() {

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

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

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

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("Logging from logger1");

        Logger logger2 = Logger.getInstance();

        logger2.log("Logging from logger2");

        if (logger1 == logger2) {

            System.out.println("Both are the same instance (Singleton works).");

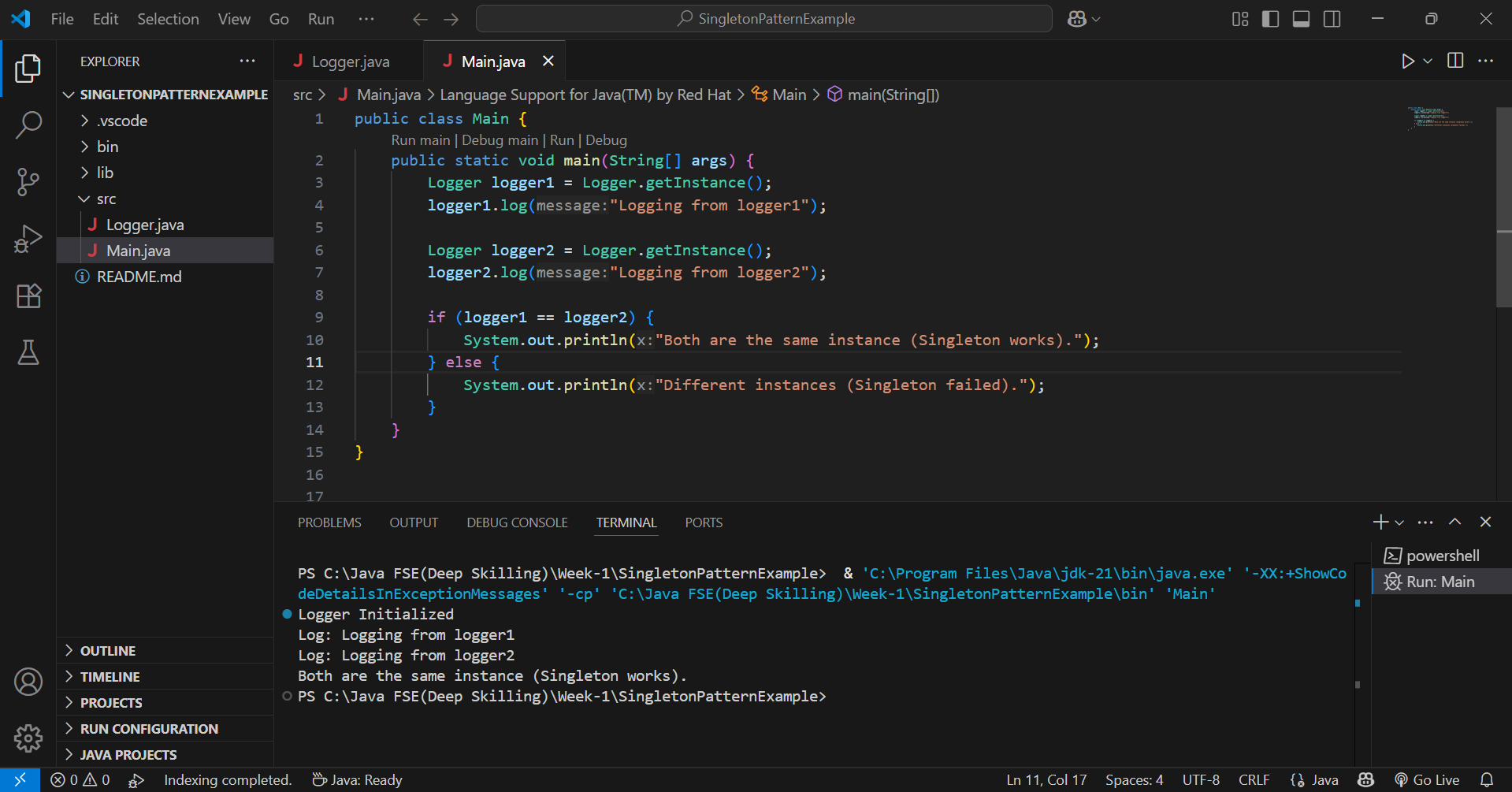
        } else {

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

        }

    }

}

****

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

**Document.java**

public interface Document {

void open();

}

**Creating Concrete Document Classes**

**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...");

}

}

**Implement the Factory Method**

**DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**TestDocumentFactory.java**

public class TestDocumentFactory {

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();

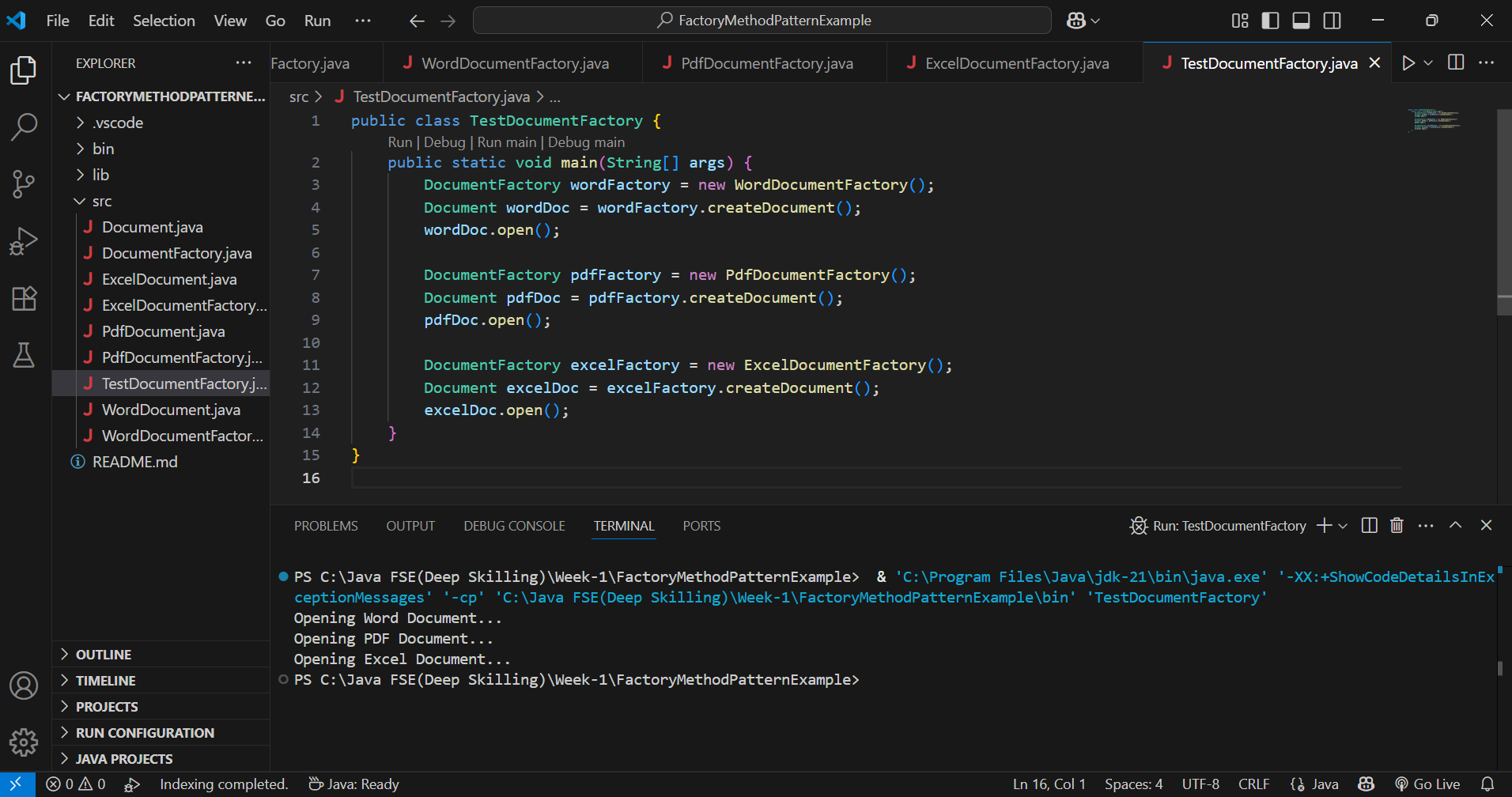
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

****

**Exercise 3: Implementing the Builder Pattern**

**Computer.java**

public class Computer {

    private String CPU;

    private String RAM;

    private String storage;

    private String graphicsCard;

    private String keyboard;

    private String monitor;

    private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

        this.keyboard = builder.keyboard;

        this.monitor = builder.monitor;

    }

    public static class Builder {

        private String CPU;

        private String RAM;

        private String storage;

        private String graphicsCard;

        private String keyboard;

        private String monitor;

        public Builder(String CPU, String RAM) {

            this.CPU = CPU;

            this.RAM = RAM;

        }

        public Builder setStorage(String storage) {

            this.storage = storage;

            return this;

        }

        public Builder setGraphicsCard(String graphicsCard) {

            this.graphicsCard = graphicsCard;

            return this;

        }

        public Builder setKeyboard(String keyboard) {

            this.keyboard = keyboard;

            return this;

        }

        public Builder setMonitor(String monitor) {

            this.monitor = monitor;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

    public void showSpecs() {

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

        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("Keyboard: " + (keyboard != null ? keyboard : "Not Included"));

        System.out.println("Monitor: " + (monitor != null ? monitor : "Not Included"));

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

    }

}

**TestBuilderPattern.java**

public class TestBuilderPattern {

    public static void main(String[] args) {

        Computer basicComputer = new Computer.Builder("Intel i3", "8GB") .build();

        Computer gamingComputer = new Computer.Builder("AMD Ryzen 7", "32GB")

                                    .setStorage("1TB SSD")

                                    .setGraphicsCard("NVIDIA RTX 4070")

                                    .setKeyboard("Mechanical RGB")

                                    .setMonitor("4K Monitor")

                                    .build();

        Computer businessComputer = new Computer.Builder("Intel i5", "16GB")

                                    .setStorage("512GB SSD")

                                    .setMonitor("Full HD Monitor")

                                    .build();

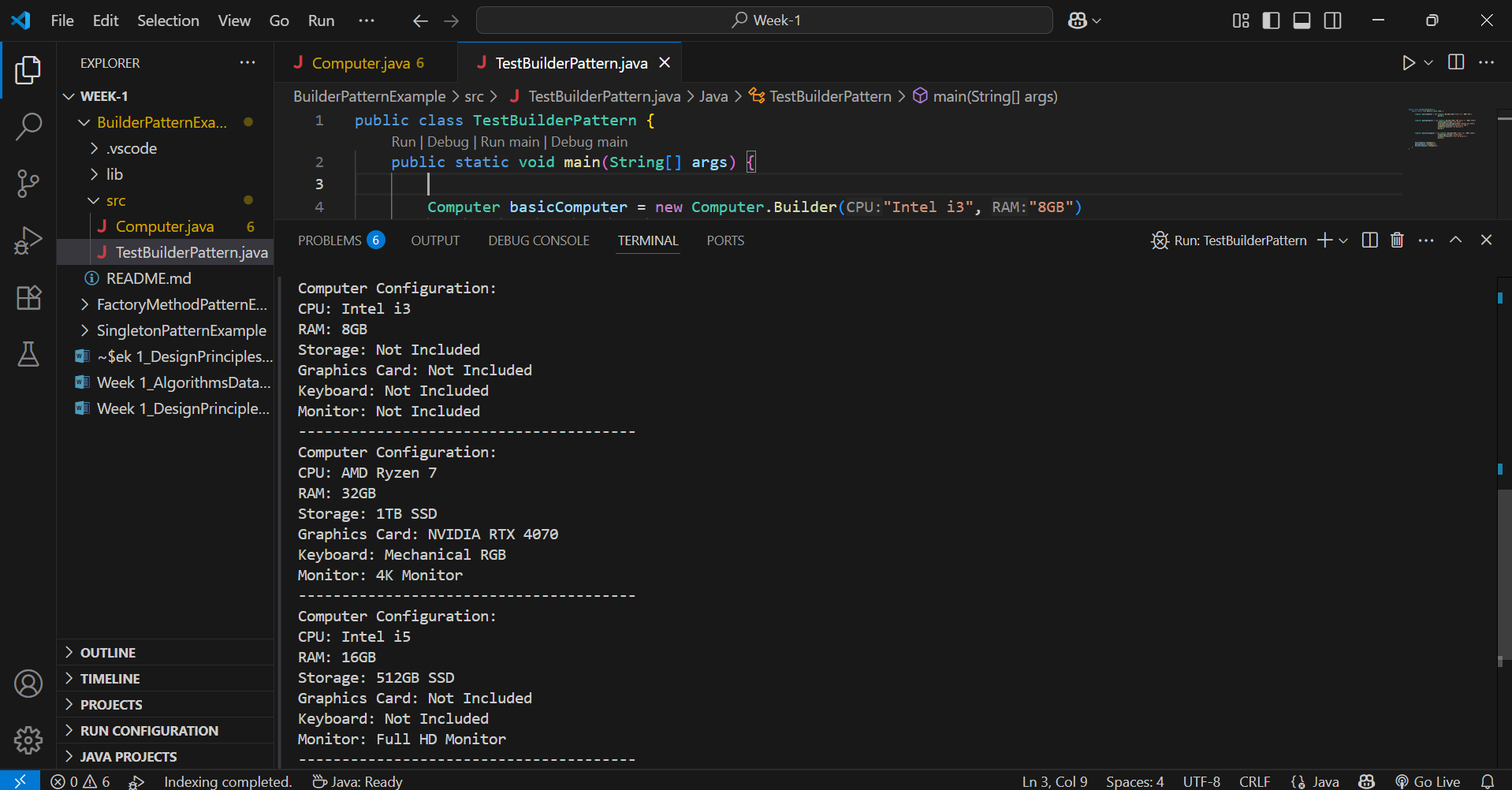
        basicComputer.showSpecs();

        gamingComputer.showSpecs();

        businessComputer.showSpecs();

    }

}



**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment(double amount);

}

**Create Adapter Classes (Third-party payment gateways)**

**PayPalGateway.java**

public class PayPalGateway {

public void makePayment(double amountInUSD) {

System.out.println("Paid $" + amountInUSD + " via PayPal.");

}

}

**StripeGateway.java**

public class StripeGateway {

public void pay(double amount) {

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

}

}

**Implement Adapter Classes**

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter() {

this.payPal = new PayPalGateway();

}

@Override

public void processPayment(double amount) {

payPal.makePayment(amount);

}

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter() {

this.stripe = new StripeGateway();

}

@Override

public void processPayment(double amount) {

stripe.pay(amount);

}

}

**TestPaymentAdapter.java**

public class TestPaymentAdapter {

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter();

PaymentProcessor stripeProcessor = new StripeAdapter();

System.out.println("Processing PayPal payment:");

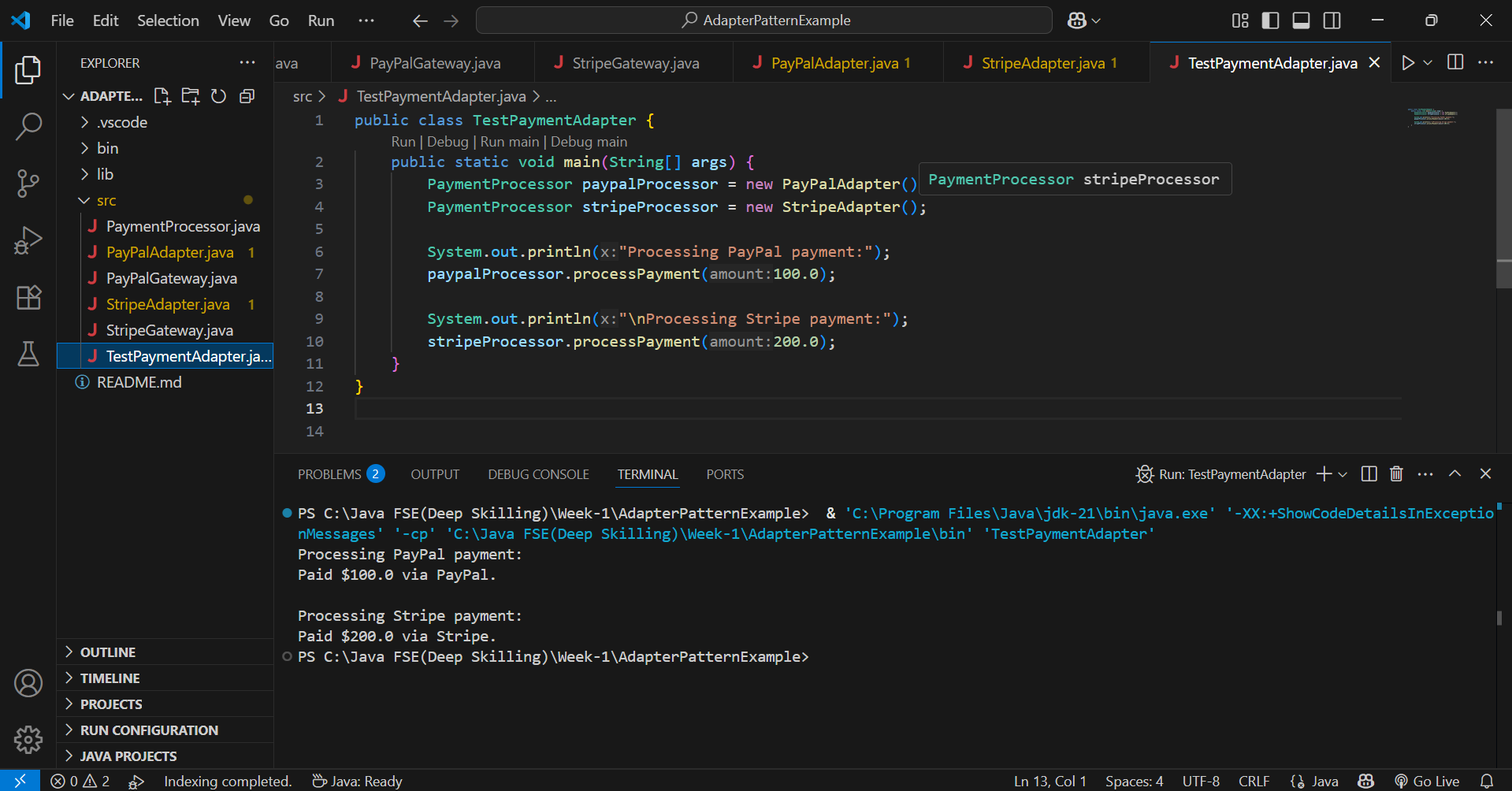
paypalProcessor.processPayment(100.0);

System.out.println("\nProcessing Stripe payment:");

stripeProcessor.processPayment(200.0);

}

}

****

**Exercise 5: Implementing the Decorator Pattern**

**Define Component Interface**

**Notifier.java**

public interface Notifier {

void send(String message);

}

**Implement Concrete Component**

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

**Implement Decorator Classes**

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message);

}

}

**SMSNotifierDecorator.java**

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: " + message);

}

}

**SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

**TestNotificationDecorator.java**

public class TestNotificationDecorator {

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

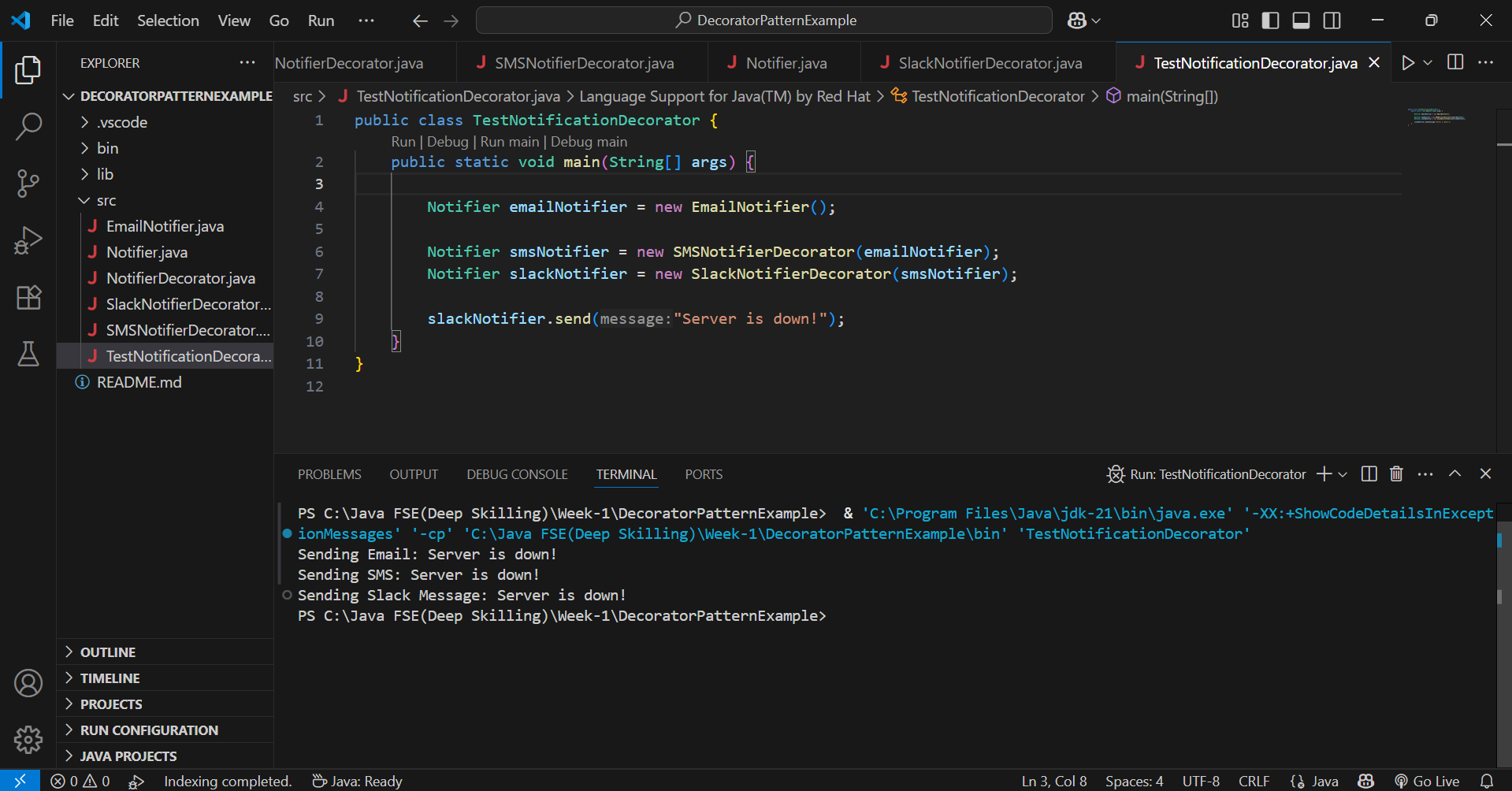
Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

slackNotifier.send("Server is down!");

}

}



**Exercise 6: Implementing the Proxy Pattern**

**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();

}

private void loadFromRemoteServer() {

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 String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

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

} else {

System.out.println("Image already loaded from cache: " + filename);

}

realImage.display();

}

}

**TestProxyPattern.java**

public class TestProxyPattern {

public static void main(String[] args) {

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

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

System.out.println("First time display (cat.png):");

image1.display();

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

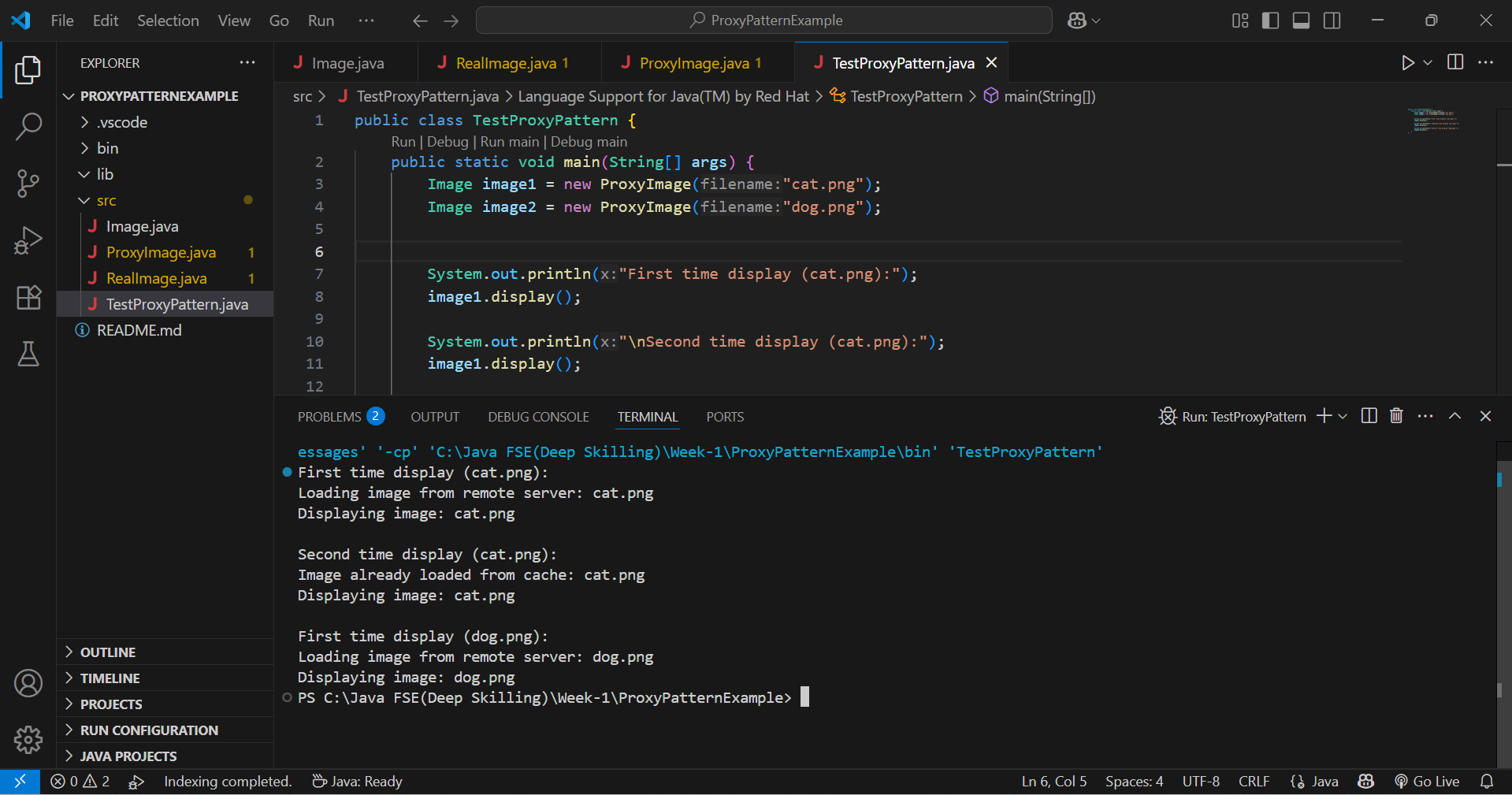
image1.display();

System.out.println("\nFirst time display (dog.png):");

image2.display();

}

}

****

**Exercise 7: Implementing the Observer Pattern**

**Define Subject Interface**

**Stock.java**

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**Implement Concrete Subject**

**StockMarket.java**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers;

private double stockPrice;

public StockMarket() {

observers = new ArrayList<>();

}

@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) {

System.out.println("\nStock Price Updated to: ₹" + newPrice);

this.stockPrice = newPrice;

notifyObservers();

}

}

**Define Observer Interface**

**Observer.java**

public interface Observer {

void update(double stockPrice);

}

**Implement Concrete Observers**

**MobileApp.java**

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("MobileApp " + name + ": New stock price: ₹" + stockPrice);

}

}

**WebApp.java**

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("WebApp " + name + ": New stock price: ₹" + stockPrice);

}

}

**TestObserverPattern.java**

public class TestObserverPattern {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp1 = new MobileApp("Client-A");

Observer webApp1 = new WebApp("Client-B");

stockMarket.registerObserver(mobileApp1);

stockMarket.registerObserver(webApp1);

stockMarket.setStockPrice(500.50);

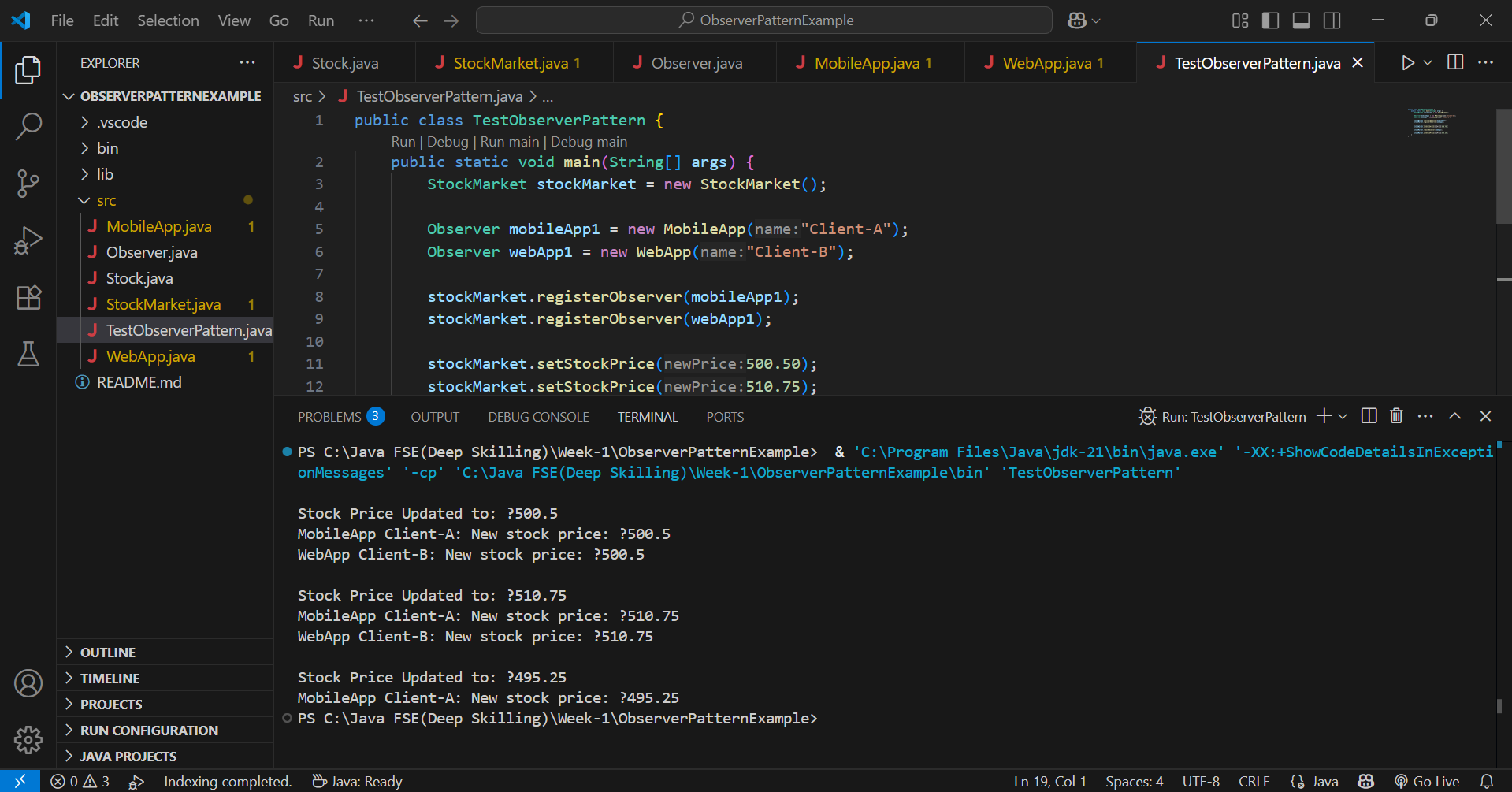
stockMarket.setStockPrice(510.75);

stockMarket.removeObserver(webApp1);

stockMarket.setStockPrice(495.25);

}

}



**Exercise 8: Implementing the Strategy Pattern**

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**Implement Concrete Strategies**

**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 [" + cardNumber + "] - " + 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 account: " + email);

}

}

**Implement Context Class**

**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("No payment strategy selected!");

} else {

strategy.pay(amount);

}

}

}

**Test the Strategy Implementation**

**TestStrategyPattern.java**

public class TestStrategyPattern {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

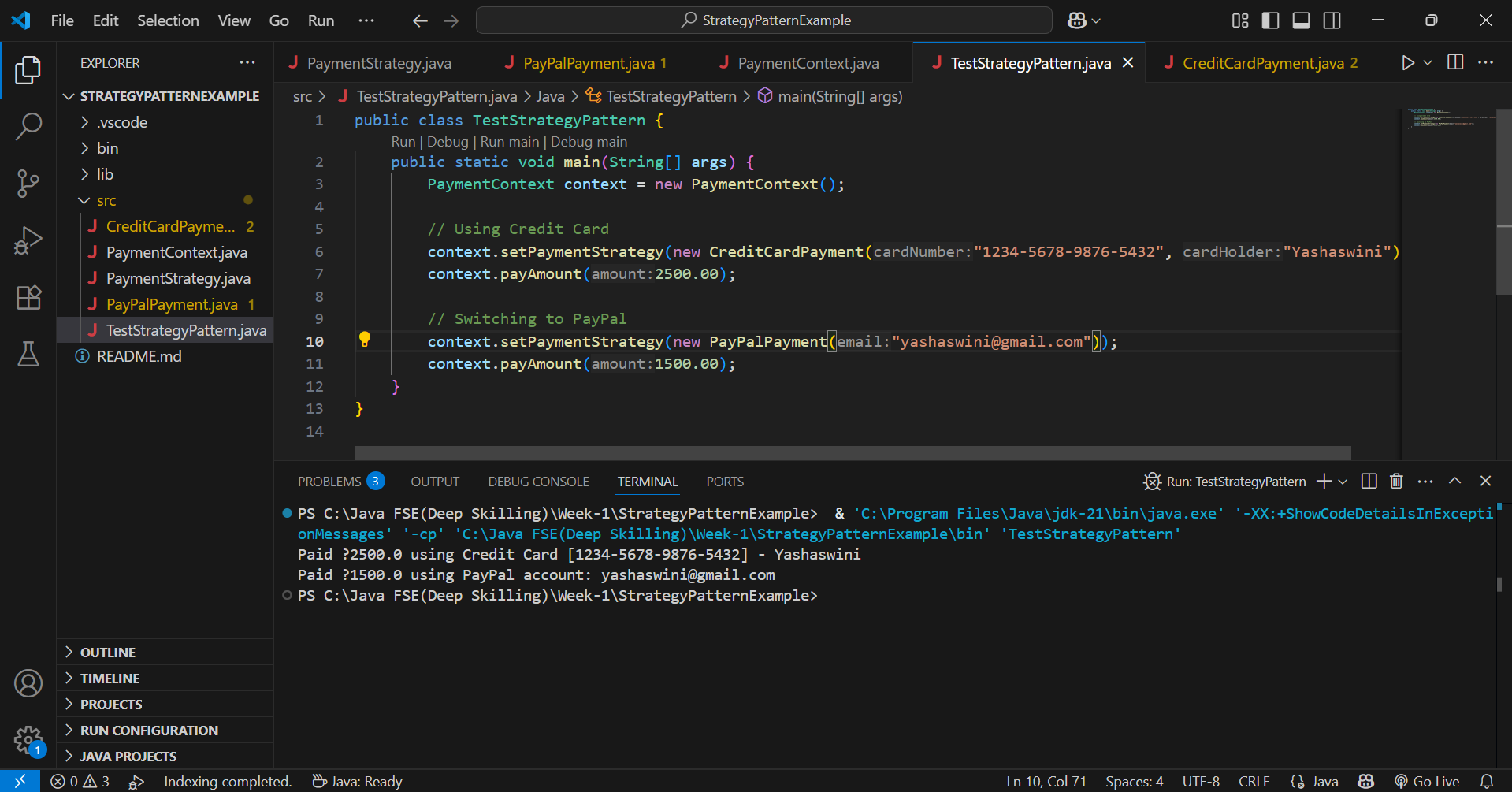
context.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432", "Yashaswini"));

context.payAmount(2500.00);

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

context.payAmount(1500.00);

}

****

**Exercise 9: Implementing the Command Pattern**

**Define Command Interface**

**Command.java**

public interface Command {

void execute();

}

Implement Concrete Commands

**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();

}

}

**Implement Invoker Class**

**RemoteControl.java**

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

**Implement Receiver Class**

**Light.java**

public class Light {

public void turnOn() {

System.out.println("The light is ON");

}

public void turnOff() {

System.out.println("The light is OFF");

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Light light = new Light();

Command lightOn = new LightOnCommand(light);

Command lightOff = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

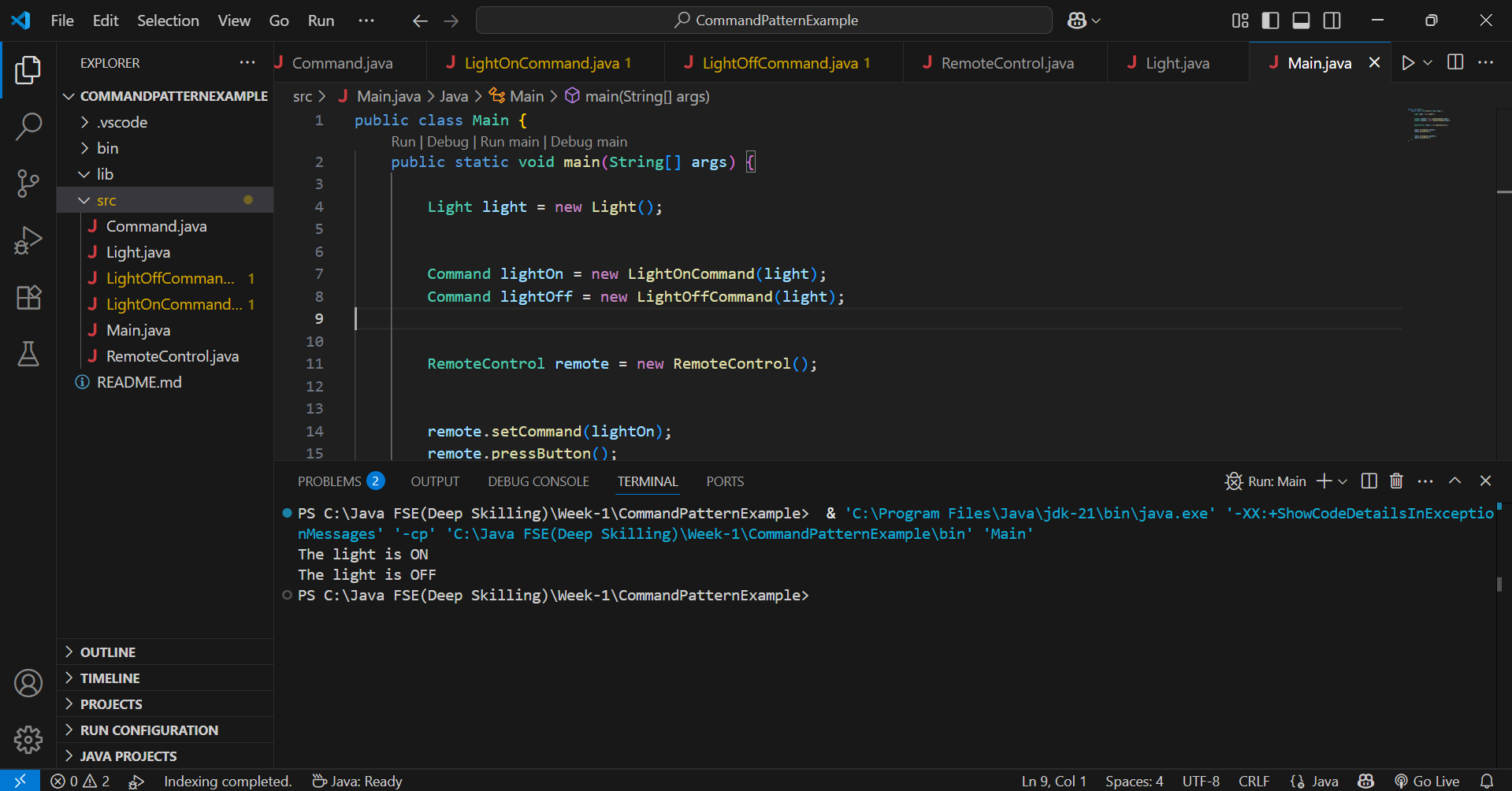
remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

**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;

}

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;

}

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());

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Student student = new Student("Yashaswini", "S101", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Yashaswini Kurapati");

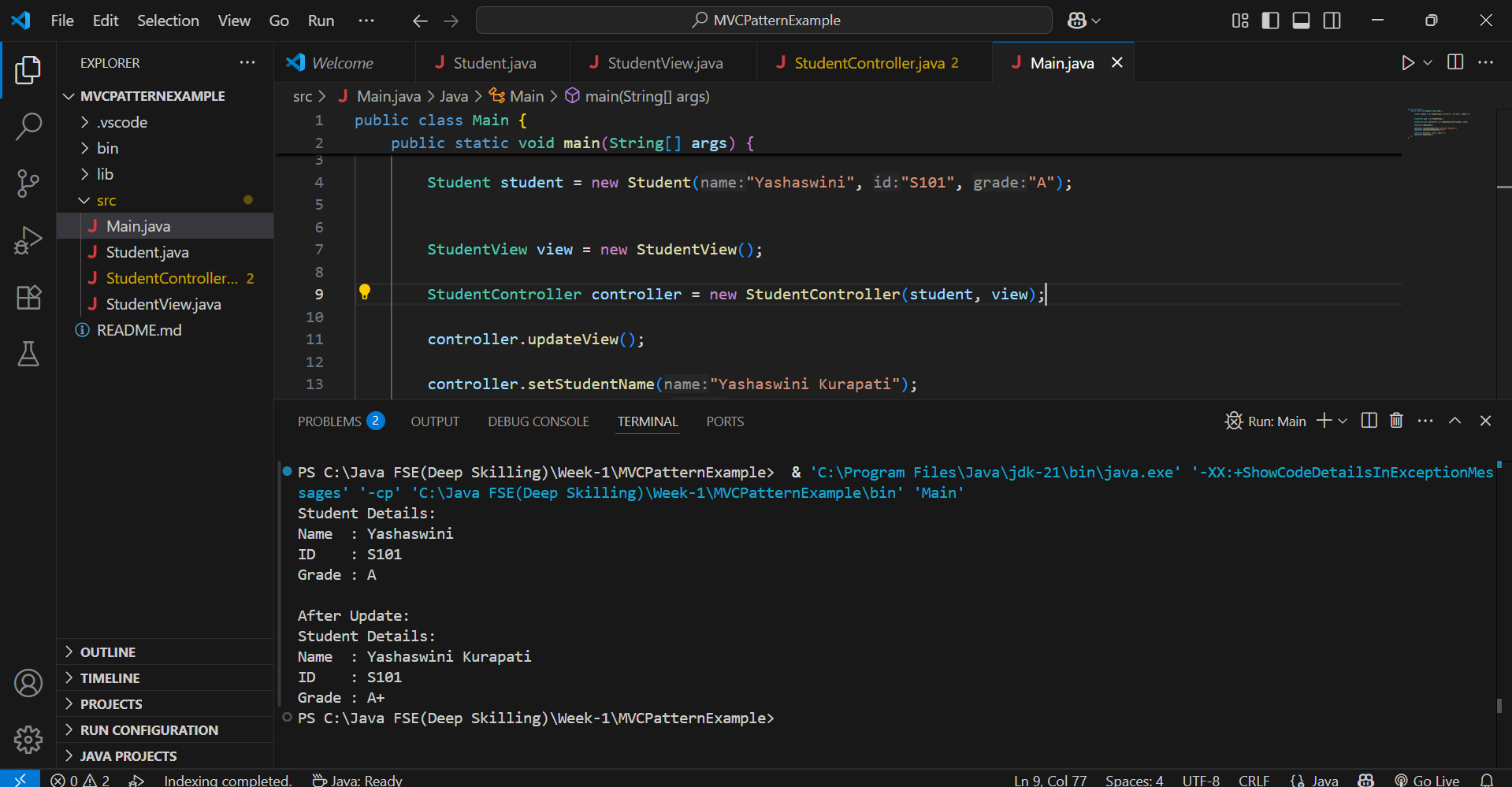
controller.setStudentGrade("A+");

System.out.println("\nAfter Update:");

controller.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

**CustomerRepository.java**

public interface CustomerRepository {

String findCustomerById(int id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(int id) {

if (id == 1) {

return "Customer: Yashaswini Kurapati";

} else {

return "Customer not found";

}

}

}

**CustomerService.java**

public class CustomerService {

private CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(int id) {

String customer = customerRepository.findCustomerById(id);

System.out.println(customer);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.getCustomerDetails(1);

service.getCustomerDetails(2);

}

}

