**WEEK-1**

**Design Patterns and Principles HandsOn**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

**Logger.java:**

package SingletonPatternExample;

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 showMsg(String msg){

        System.out.println(msg);

    }

}

**Test.java:**

package SingletonPatternExample;

public class Test {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.showMsg("First Logger Message");

        Logger logger2 = Logger.getInstance();

        logger2.showMsg("Second Logger Message");

        if(logger1==logger2){

            System.out.println("Singleton Created Successfully");

        }

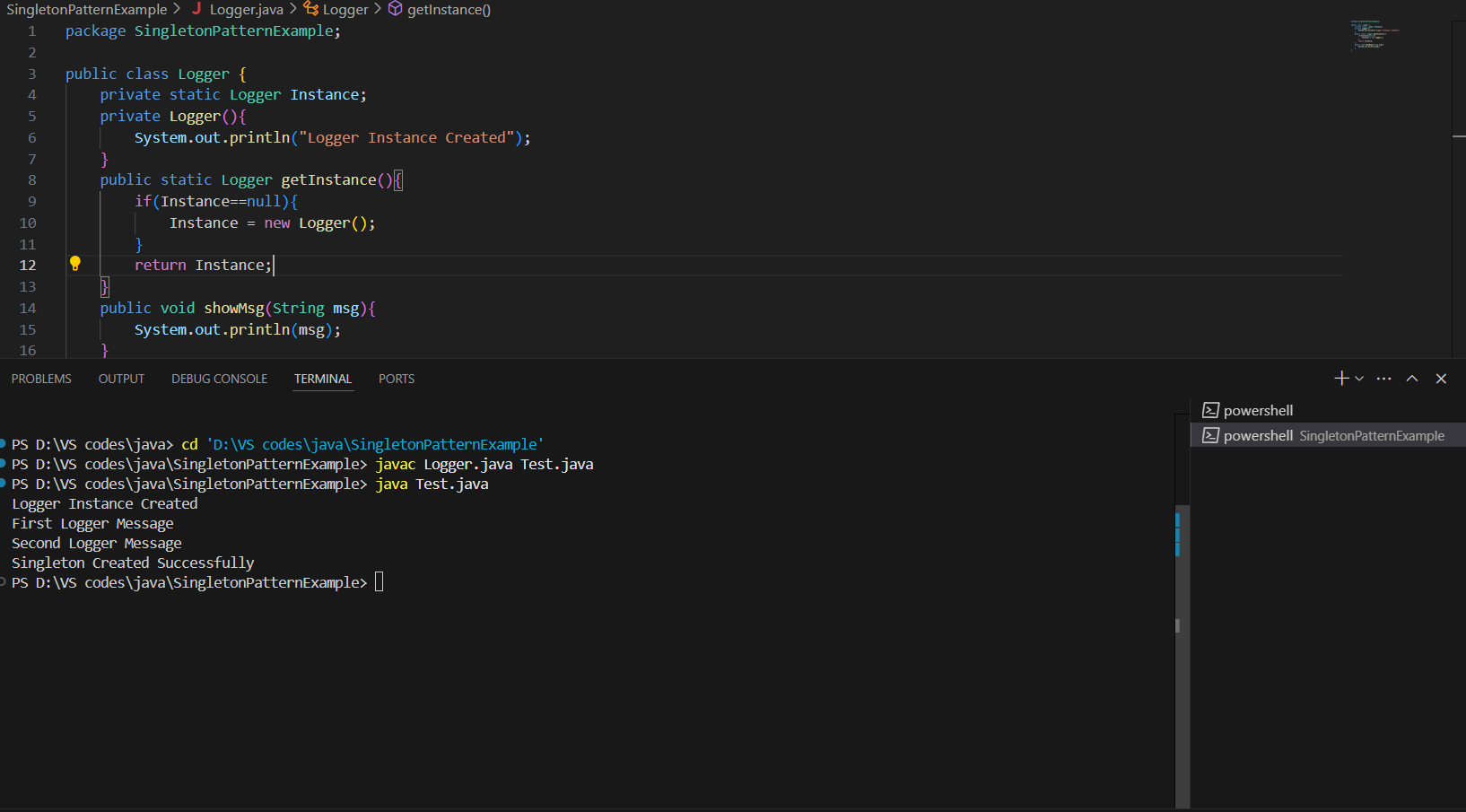
        else{

            System.out.println("Singleton creation failed");

        }

    }

}

**Output:  
**

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

**Code:**

**FactoryMethodPatternExample.java:**

package FactoryMethodPatternExample;

public class FactoryMethodPatternExample {

    interface Document{

        void open();

    }

    static class WordDocument implements Document{

        public void open(){

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

        }

    }

    static class PdfDocument implements Document{

        public void open(){

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

        }

    }

    static class ExcelDocument implements Document{

        public void open(){

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

        }

    }

    abstract static class DocumentFactory{

        public abstract Document createDocument();

    }

    static class WordDocumentFactory extends DocumentFactory{

        public Document createDocument(){

            return new WordDocument();

        }

    }

    static class PdfDocumentFactory extends DocumentFactory{

        public Document createDocument(){

            return new PdfDocument();

        }

    }

    static class ExcelDocumentFactory extends DocumentFactory{

        public Document createDocument(){

            return new ExcelDocument();

        }

    }

    public static void main(String[] args) {

        DocumentFactory word = new WordDocumentFactory();

        Document wordDocument = word.createDocument();

        wordDocument.open();

        DocumentFactory pdf = new PdfDocumentFactory();

        Document pdfDocument = pdf.createDocument();

        pdfDocument.open();

        DocumentFactory excel = new ExcelDocumentFactory();

        Document excelDocument = excel.createDocument();

        excelDocument.open();

    }

}

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 3: Implementing the Builder Pattern**

**Code:**

**BuilderPatternExample.java:**

package BuilderPatternExample;

public class BuilderPatternExample

{

    static class Computer{

        private String cpu;

        private String ram;

        private String rom;

        private String gpu;

        private String windowsVersion;

        private Computer(Builder builder){

            this.cpu=builder.cpu;

            this.ram=builder.ram;

            this.rom=builder.rom;

            this.gpu=builder.gpu;

            this.windowsVersion=builder.windowsVersion;

        }

        static class Builder{

            private String cpu;

            private String ram;

            private String rom;

            private String gpu;

            private String windowsVersion;

            public Builder setCpu(String cpu){

                this.cpu = cpu;

                return this;

            }

            public Builder setRam(String ram){

                this.ram = ram;

                return this;

            }

            public Builder setRom(String rom){

                this.rom = rom;

                return this;

            }

            public Builder setGpu(String gpu){

                this.gpu = gpu;

                return this;

            }

            public Builder setWindowsVersion(String windowsVersion){

                this.windowsVersion = windowsVersion;

                return this;

            }

            public Computer build(){

                return new Computer(this);

            }

        }

        public void display(){

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

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

            System.out.println("ROM - "+rom);

            System.out.println("GPU - "+gpu);

            System.out.println("Windows Version - "+windowsVersion);

            System.out.println();

        }

    }

    public static void main(String[] args){

        Computer gamingPC = new Computer.Builder()

        .setCpu("i7")

        .setRam("16GB")

        .setRom("1TB")

        .setGpu("RTX 4080")

        .setWindowsVersion("Windows 11")

        .build();

        Computer officePC = new Computer.Builder()

        .setCpu("i5")

        .setRam("8GB")

        .setRom("512GB")

        .setWindowsVersion("Windows 11")

        .build();

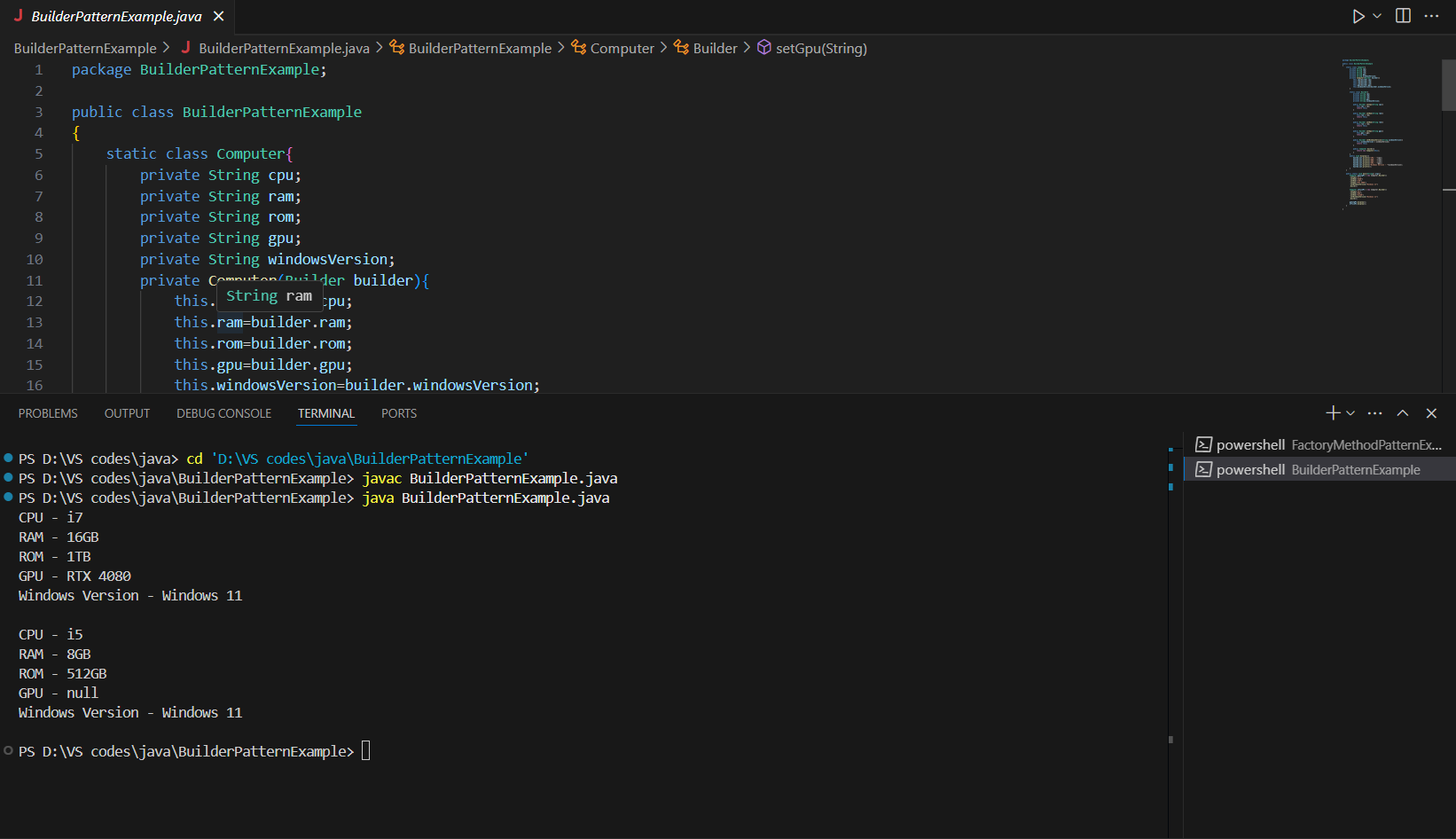
        gamingPC.display();

        officePC.display();

    }

}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

**AdapterPatternExample.java:**

package AdapterPatternExample;

interface Payment{

    void processPayment(double amount);

}

class RazorPay {

    public void RazorPayPayment(double amount){

        System.out.println("Processing Payment Amount of "+amount+" via RazorPay");

    }

}

class PayPal {

    public void PayPalPayment(double amount){

        System.out.println("Processing Payment Amount of "+amount+" via PayPal");

    }

}

class Paytm{

    public void PaytmPayment(double amount){

        System.out.println("Processing Payment Amount of "+amount+" via Paytm");

    }

}

class PaymentAdapter implements Payment{

    Object gateway;

    public PaymentAdapter(String gateway){

        if(gateway.equalsIgnoreCase("RazorPay")){

            this.gateway = new RazorPay();

        }

        else if(gateway.equalsIgnoreCase("PayPal")){

            this.gateway = new PayPal();

        }

        else if(gateway.equalsIgnoreCase("Paytm")){

            this.gateway = new Paytm();

        }

    }

    @Override

    public void processPayment(double amount){

        if(gateway instanceof RazorPay){

            ((RazorPay)gateway).RazorPayPayment(amount);

        }

        else if(gateway instanceof PayPal){

            ((PayPal)gateway).PayPalPayment(amount);

        }

        else if(gateway instanceof Paytm){

            ((Paytm)gateway).PaytmPayment(amount);

        }

        else{

            System.out.println("Invalid Gateway");

        }

    }

}

public class AdapterPatternExample implements Payment {

    PaymentAdapter paymentAdapter;

    @Override

    public void processPayment(double amount){

        System.out.println("Processing Amount "+amount);

    }

    public void pay(String gateWay,double amount){

        if(gateWay.equalsIgnoreCase("RazorPay") ||

        gateWay.equalsIgnoreCase("PayPal") ||

        gateWay.equalsIgnoreCase("Paytm")){

            paymentAdapter = new PaymentAdapter(gateWay);

            paymentAdapter.processPayment(amount);

        }

        else{

            System.out.println("Invalid Gateway");

        }

    }

    public static void main(String[] args) {

        AdapterPatternExample processor = new AdapterPatternExample();

        processor.pay("RazorPay",20000);

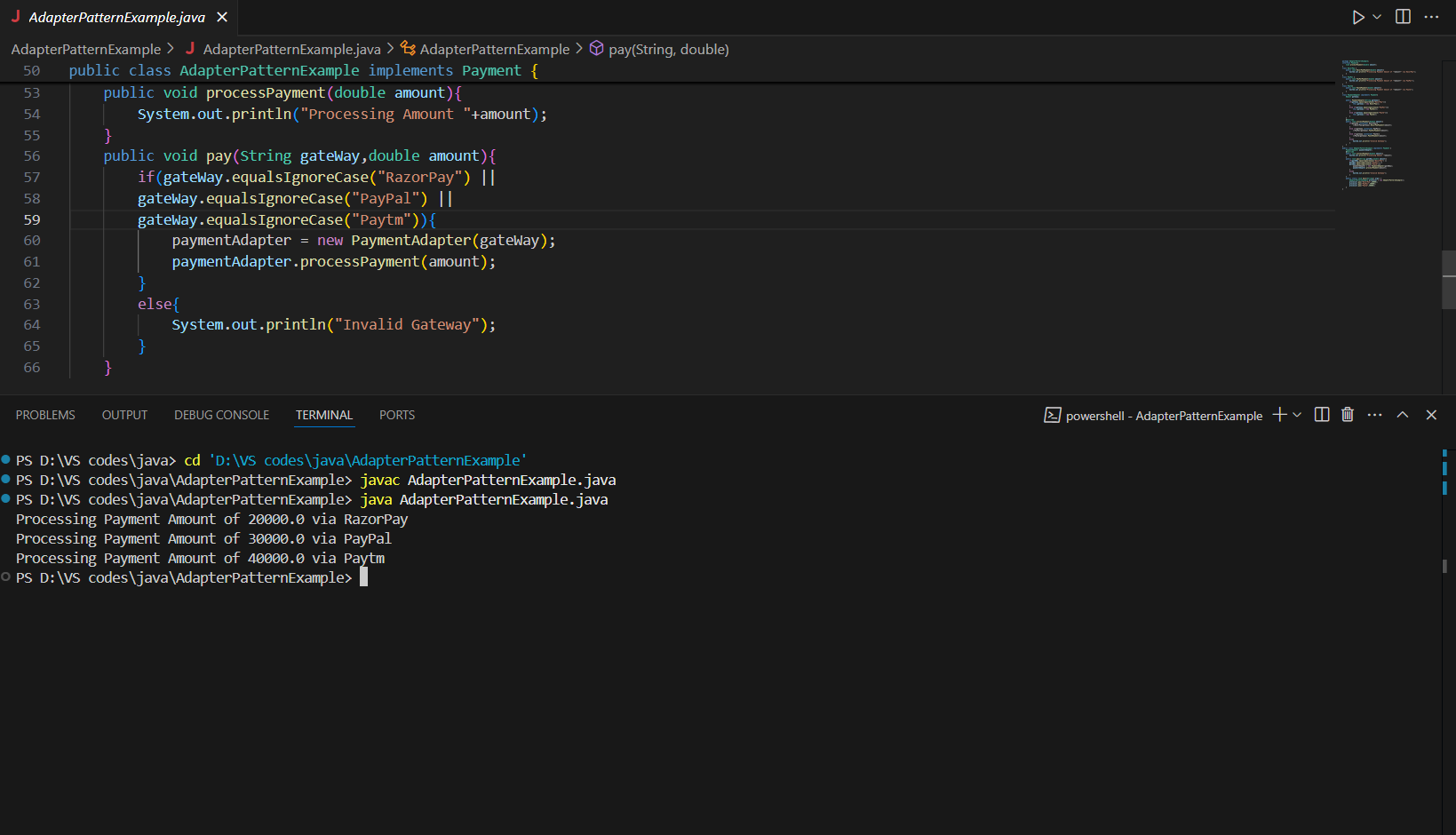
        processor.pay("PayPal",30000);

        processor.pay("Paytm",40000);

    }

}

**Output:**

****

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

**DecoratorPatternExample.java:**

package DecoratorPatternExample;

interface Notifier{

    void send(String message);

}

class EmailNotifier implements Notifier{

    @Override

    public void send(String message){

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

    }

}

abstract class NotifierDecorator implements Notifier{

    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier){

        this.notifier = notifier;

    }

    public void send(String message){

        notifier.send(message);

    }

}

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);

    }

}

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);

    }

}

public class DecoratorPatternExample {

    public static void main(String[] args) {

        Notifier notifier = new EmailNotifier();

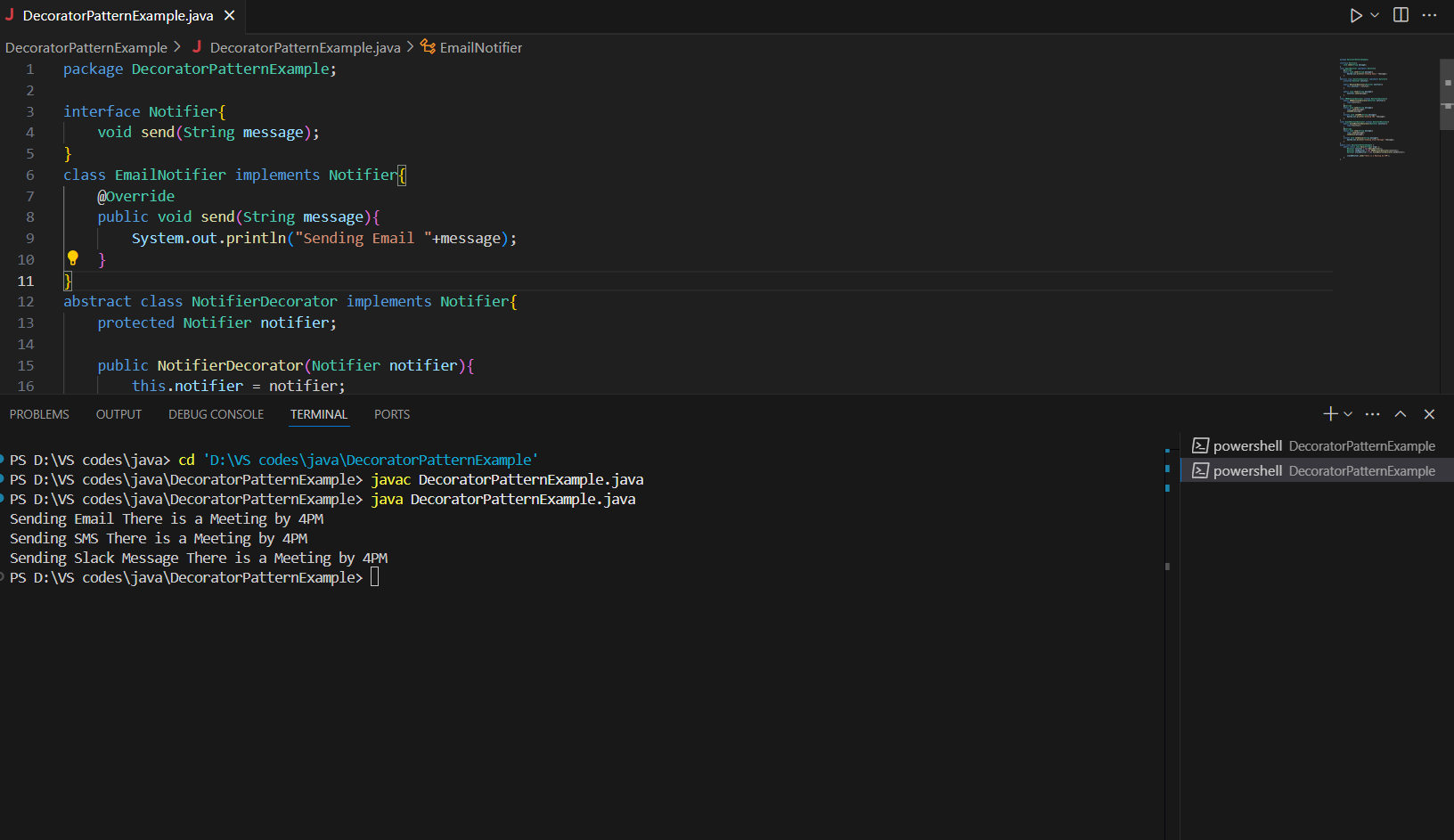
        Notifier smsNotifier = new SMSNotifierDecorator(notifier);

        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

        slackNotifier.send("There is a Meeting by 4PM");

    }

}

**Output:  
**

**Exercise 6: Implementing the Proxy Pattern**

**Code:**

**ProxyPatternExample.java:**

package ProxyPatternExample;

import java.util.HashMap;

import java.util.Map;

public class ProxyPatternExample {

    interface Image{

        void display();

    }

    static class RealImage implements Image{

        private String fileName;

        public RealImage(String fileName){

            this.fileName = fileName;

            remoteServer();

        }

        private void remoteServer(){

            System.out.println("Loading "+fileName+" from Remote Server");

        }

        public void display(){

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

        }

    }

    static class ProxyImage implements Image{

        private String fileName;

        Map<String,RealImage> cache = new HashMap<>();

        public ProxyImage(String fileName){

            this.fileName = fileName;

        }

        public void display(){

            RealImage realImage = cache.get(fileName);

            if(realImage == null){

                System.out.println("Creating a realImage object and storing in Cache"+fileName);

                realImage = new RealImage(fileName);

                cache.put(fileName, realImage);

            }

            else{

                System.out.println("Image is already there in cache "+fileName);

            }

            realImage.display();

        }

    }

    public static void main(String[] args) {

        Image img1 = new ProxyImage("img1.jpg");

        Image img2 = new ProxyImage("img2.jpg");

        Image img3 = new ProxyImage("img3.jpg");

        img1.display();

        System.out.println();

        img2.display();;

        System.out.println();

        img3.display();

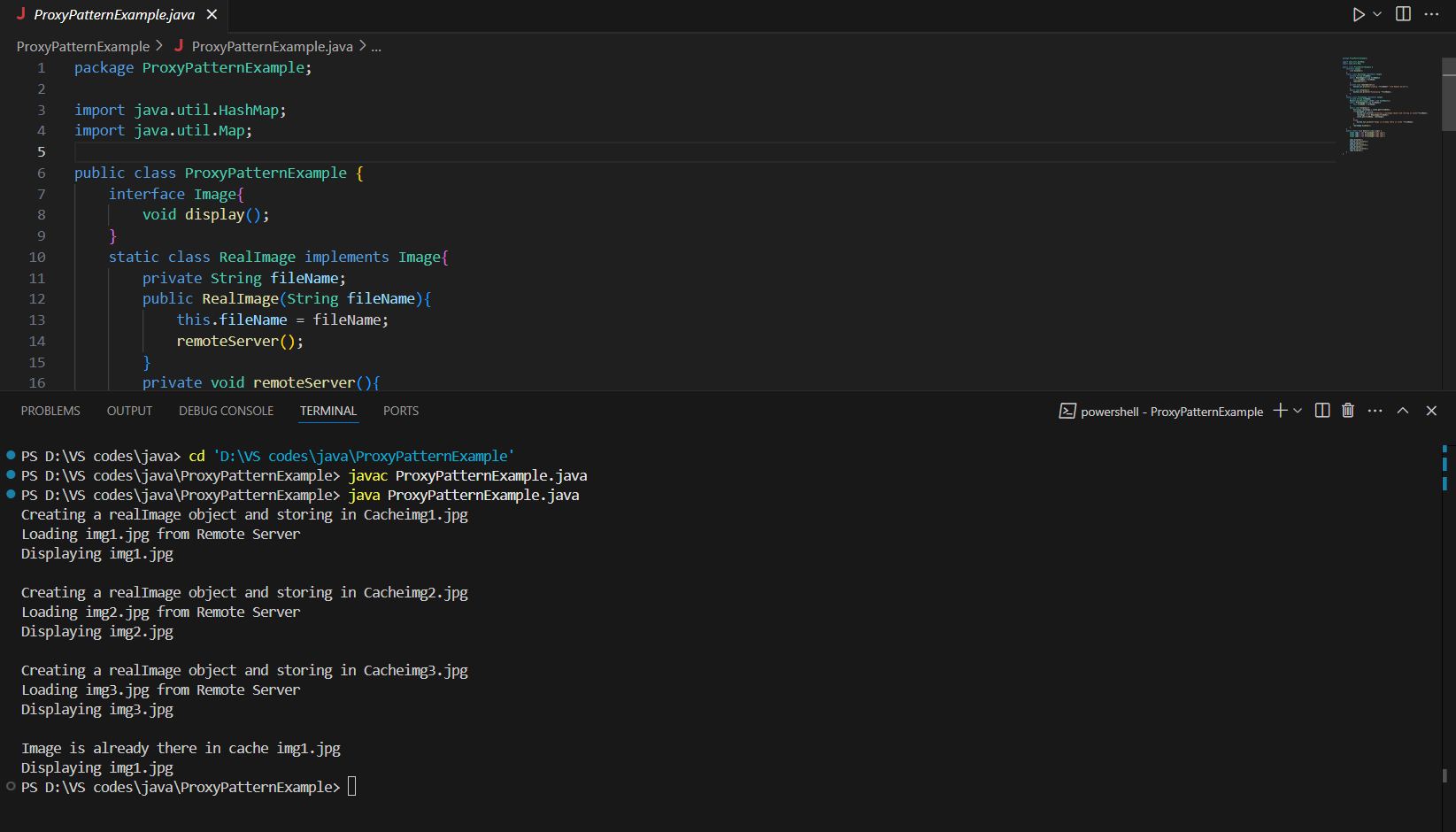
        System.out.println();

        img1.display();

    }

}

**Output:**

****

**Exercise 7: Implementing the Observer Pattern**

**Code:**

**ObserverPattern.java:**

package ObserverPatternExample;

import java.util.ArrayList;

import java.util.List;

public class ObserverPatternExample {

    interface Stock{

        void registerObserver(Observer o);

        void removeObserver(Observer o);

        void notifyObservers();

    }

    interface Observer{

        void update(String stockName,double newPrice);

    }

    static class StockMarket implements Stock{

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

        private String stockName;

        private double stockPrice;

        public StockMarket(String stockName,double price){

            this.stockName = stockName;

            this.stockPrice = 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, stockPrice);

            }

        }

        public void setStockPrice(double price){

            this.stockPrice = price;

            notifyObservers();

        }

    }

    static class MobileApp implements Observer{

        private String appName;

        public MobileApp(String appName){

            this.appName = appName;

        }

        public void update(String stockName,double stockPrice){

            System.out.println(appName+" : "+stockName+" = "+stockPrice);

        }

    }

    static class WebApp implements Observer{

        private String appName;

        public WebApp(String appName){

            this.appName = appName;

        }

        public void update(String stockName,double stockPrice){

            System.out.println(appName+" : "+stockName+" = "+stockPrice);

        }

    }

    public static void main(String[] args) {

        StockMarket samsung = new StockMarket("Samsung", 500);

        Observer mobileUser = new MobileApp("Mobile App");

        Observer webUser = new WebApp("Web App");

        samsung.registerObserver(mobileUser);

        samsung.registerObserver(webUser);

        samsung.setStockPrice(600);

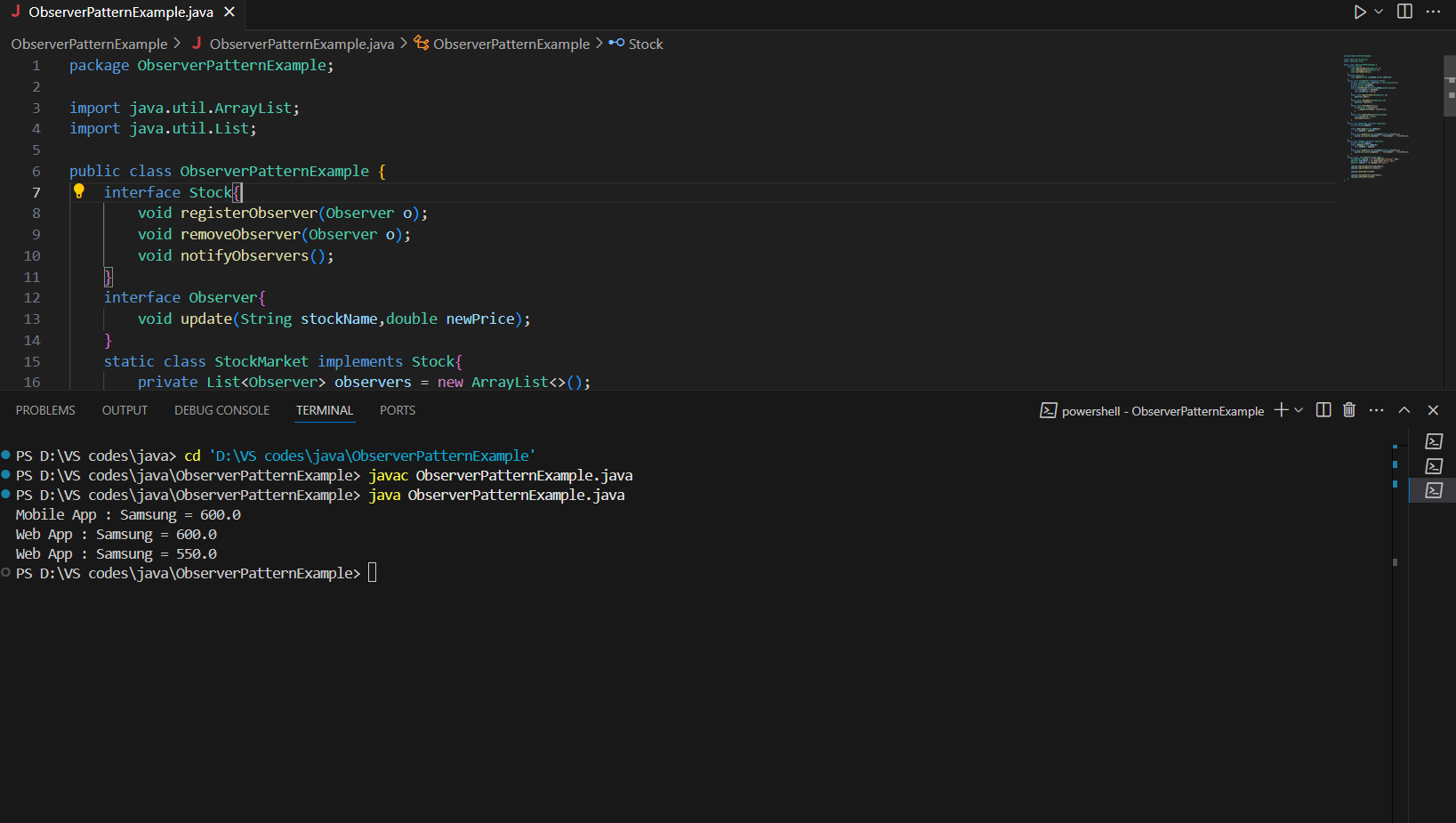
        samsung.removeObserver(mobileUser);

        samsung.setStockPrice(550);

    }

}

**Output:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Code:**

**StrategyPatternExample.java:**

package StrategyPatternExample;

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 email:"+email);

    }

}

class PaymentContext{

    private PaymentStrategy paymentStrategy;

    public void setPayment(PaymentStrategy paymentStrategy){

        this.paymentStrategy = paymentStrategy;

    }

    public void process(double amount){

        if(paymentStrategy==null){

            System.out.println("Choose a payment Method");

        }

        else{

            paymentStrategy.pay(amount);

        }

    }

}

public class StrategyPatternExample {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setPayment(new CreditCardPayment("1234-5678-9012-3456"));

        context.process(5000);

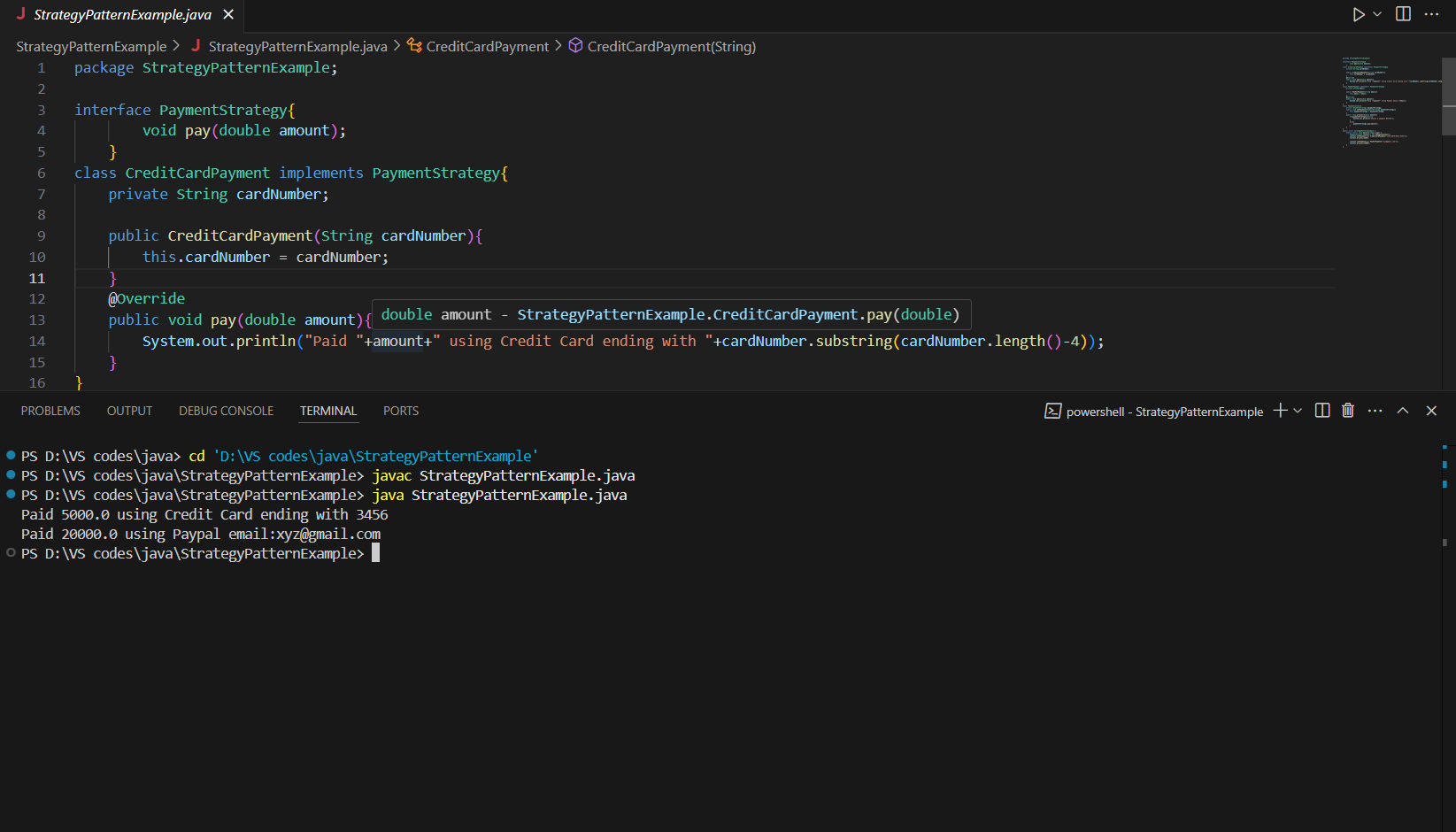
        context.setPayment(new PayPalPayment("xyz@gmail.com"));

        context.process(20000);

    }

}

**Output:**

****

**Exercise 9: Implementing the Command Pattern**

**Code:**

**CommandPatternExample.java:**

package CommandPatternExample;

interface Command{

    void execute();

}

class Light{

    public void turnOn(){

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

    }

    public void turnOff(){

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

    }

}

class LightOnCommand implements Command{

    private Light light;

    public LightOnCommand(Light light){

        this.light = light;

    }

    public void execute(){

        light.turnOn();

    }

}

class LightOffCommand implements Command{

    private Light light;

    public LightOffCommand(Light light){

        this.light = light;

    }

    public void execute(){

        light.turnOff();

    }

}

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 such Command");

        }

    }

}

public class CommandPatternExample {

    public static void main(String[] args) {

        Light roomLight = new Light();

        Command lightOn = new LightOnCommand(roomLight);

        Command lightOff = new LightOffCommand(roomLight);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);

        remote.pressButton();

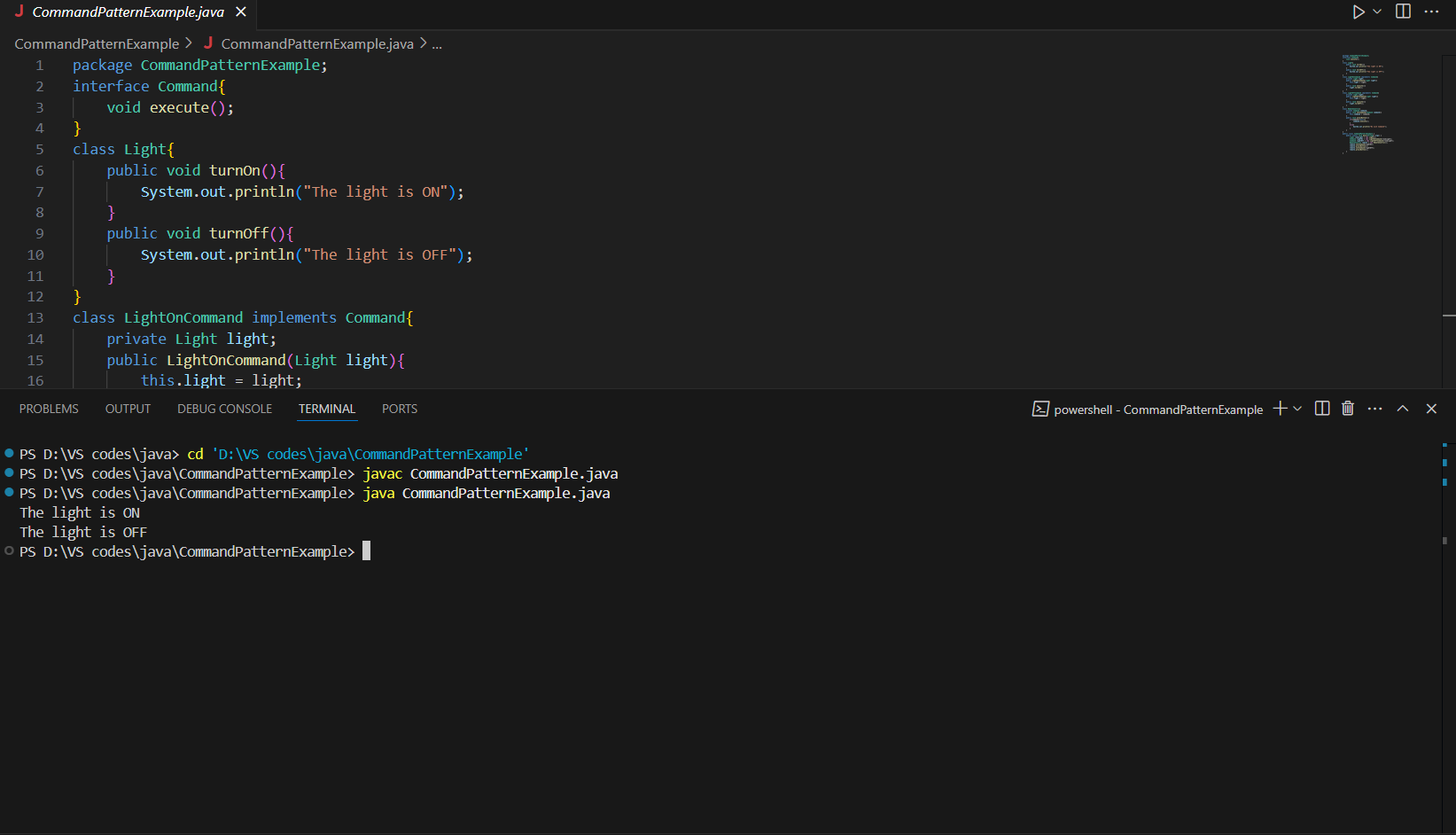
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**Code:**

**MVCPatternExample.java:**

package MVCPatternExample;

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;

    }

}

class StudentView{

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

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

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

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

        System.out.println();

    }

}

class StudentController{

    private Student model;

    private StudentView view;

    public StudentController(Student model,StudentView view){

        this.model = model;

        this.view = view;

    }

    public void setName(String name){

        model.setName(name);

    }

    public void setId(String id){

        model.setId(id);

    }

    public void setGrade(String grade){

        model.setGrade(grade);

    }

    public void update(){

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

    }

}

public class MVCPatternExample {

    public static void main(String[] args) {

        Student student = new Student("ABC", "123", "X");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.update();

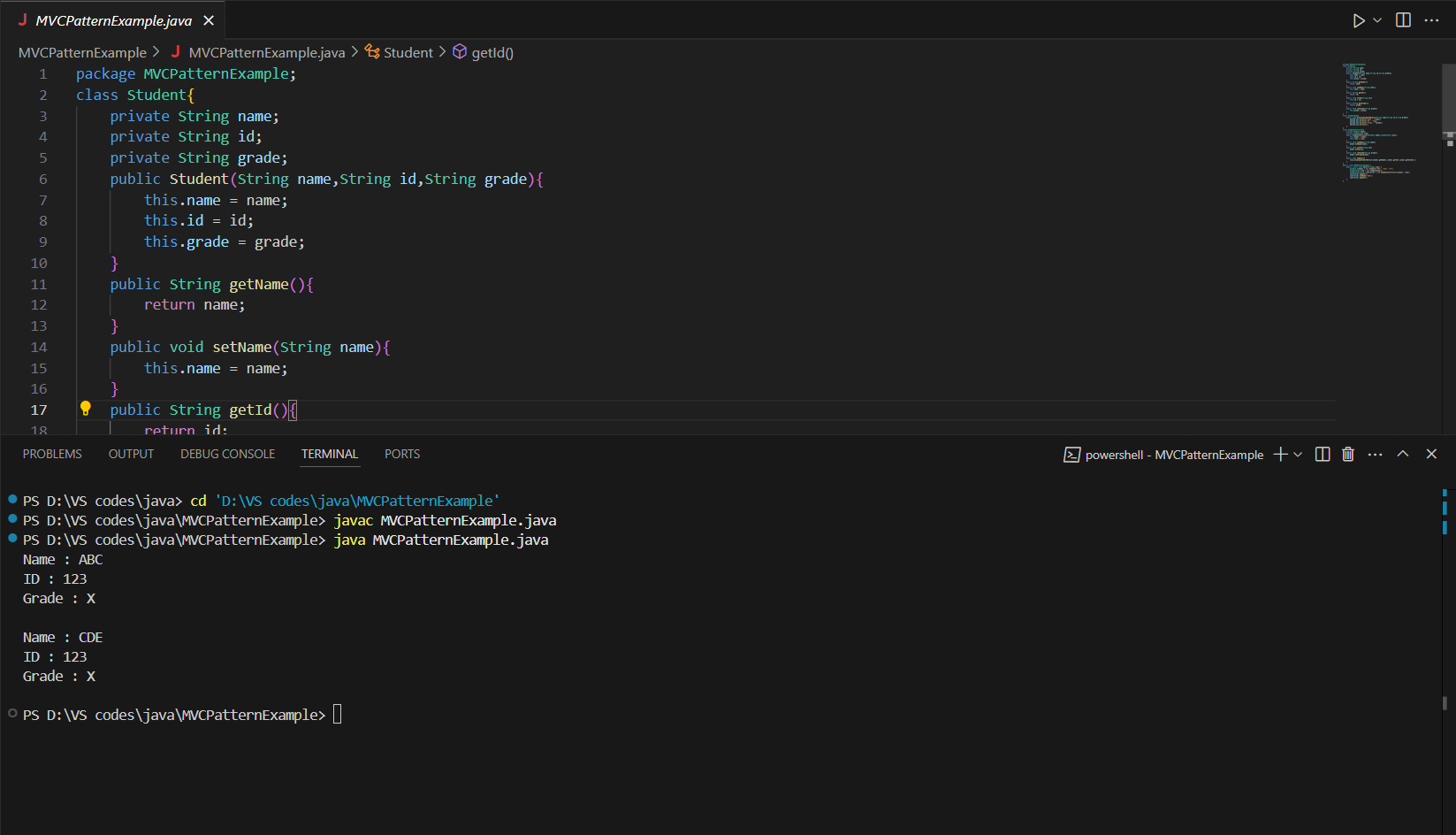
        controller.setName("CDE");

        controller.update();

    }

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Code:**

**DependencyInjectionExample.java:**

package DependencyInjectionExample;

interface CustomerRepository{

    String findCustomerById(String customerId);

}

class CustomerRepositoryImpl implements CustomerRepository{

    @Override

    public String findCustomerById(String customerId){

        return "ID - "+customerId+" Name:ABC ";

    }

}

class CustomerService{

    private CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository){

        this.customerRepository = customerRepository;

    }

    public void displayCustomer(String customerId){

        System.out.println("Customer Info "+customerRepository.findCustomerById(customerId));

    }

}

public class DependencyInjectionExample {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

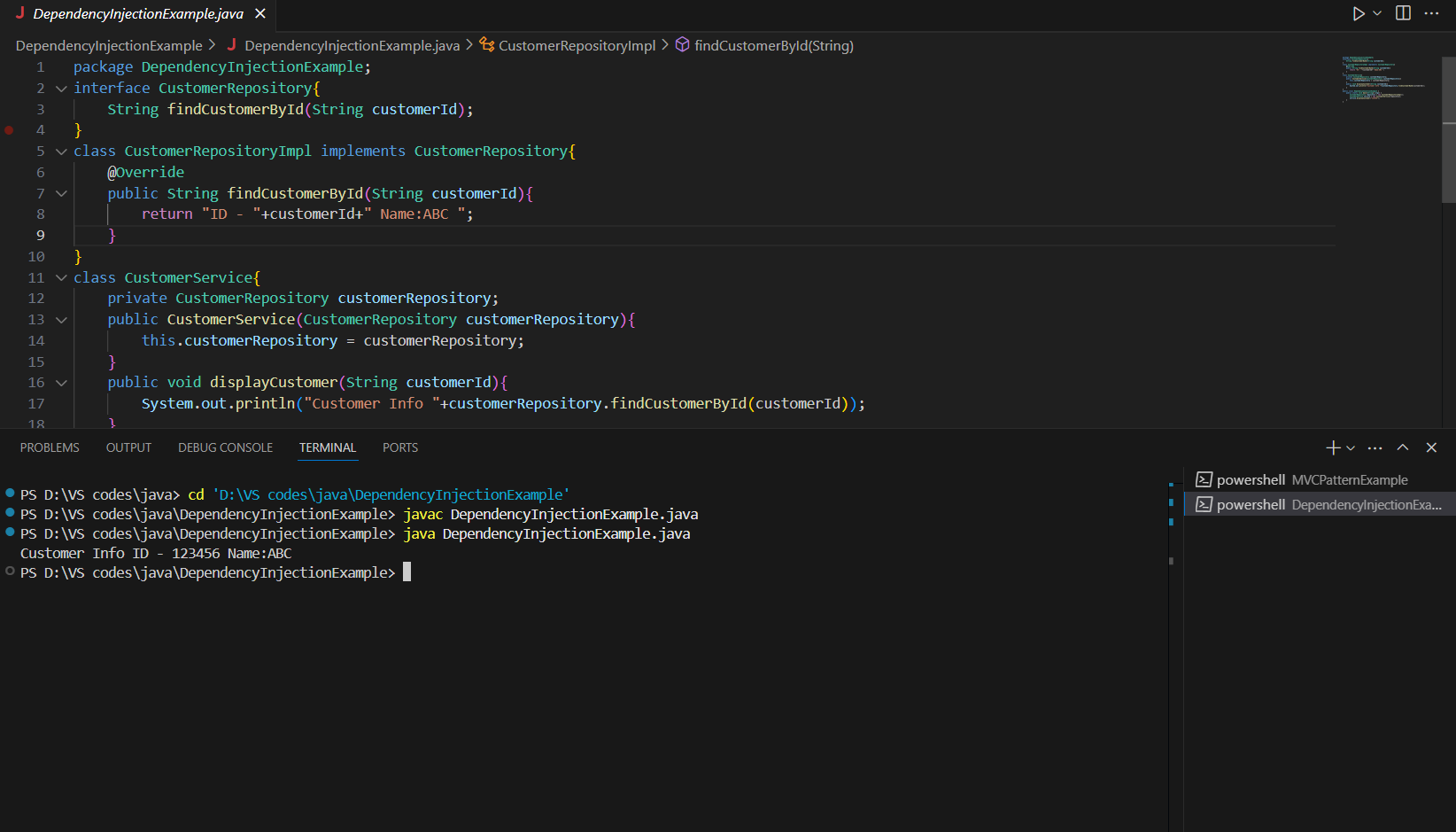
        CustomerService service = new CustomerService(repository);

        service.displayCustomer("123456");

    }

}

**Output:**

****