**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

public class SingletonPatternExample {

static 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);

}

}

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

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

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

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 are the same instance");

} else {

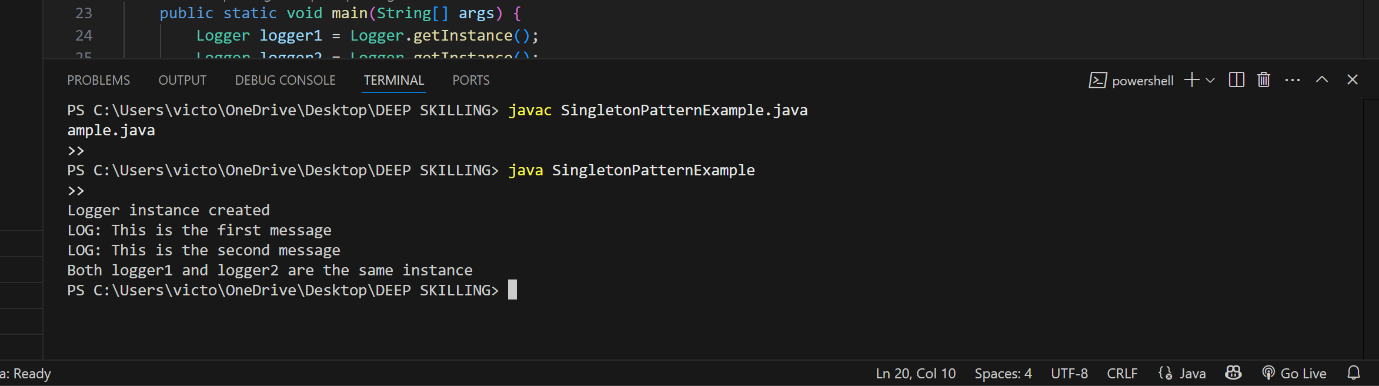
System.out.println("Different instances exist — Singleton failed");

}

}

}

**OUTPUT:**



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

**Code:**

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 wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

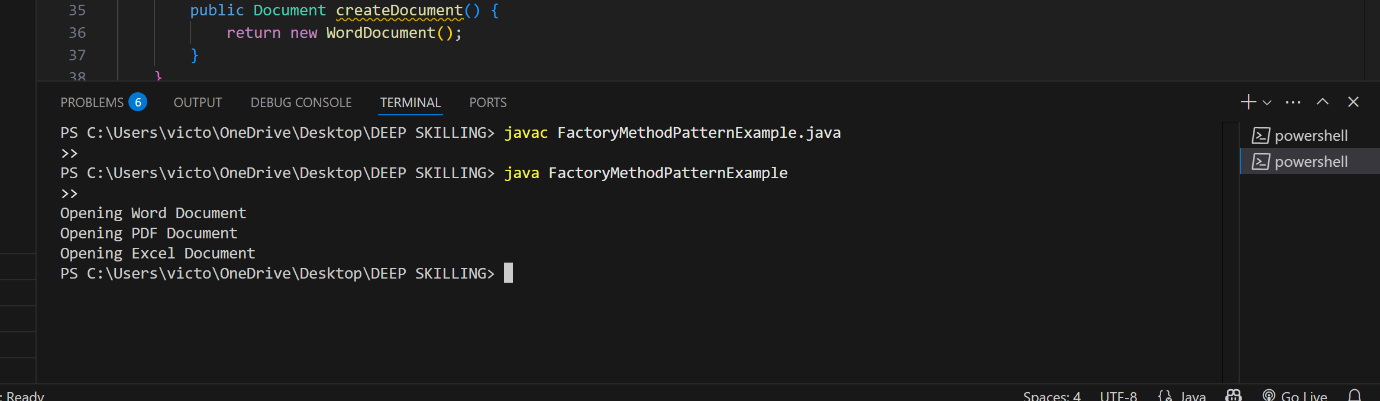
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

OUTPUT:



**Exercise 3: Implementing the Builder Pattern**

Code:

public class BuilderPatternExample {

static class Computer {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

public void displaySpecs() {

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

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

System.out.println("Storage: " + storage);

System.out.println("Graphics Card: " + graphicsCard);

}

static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

public Builder setCPU(String CPU) {

this.CPU = CPU;

return this;

}

public Builder setRAM(String RAM) {

this.RAM = RAM;

return this;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("512GB SSD")

.build();

Computer gamingComputer = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.build();

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

basicComputer.displaySpecs();

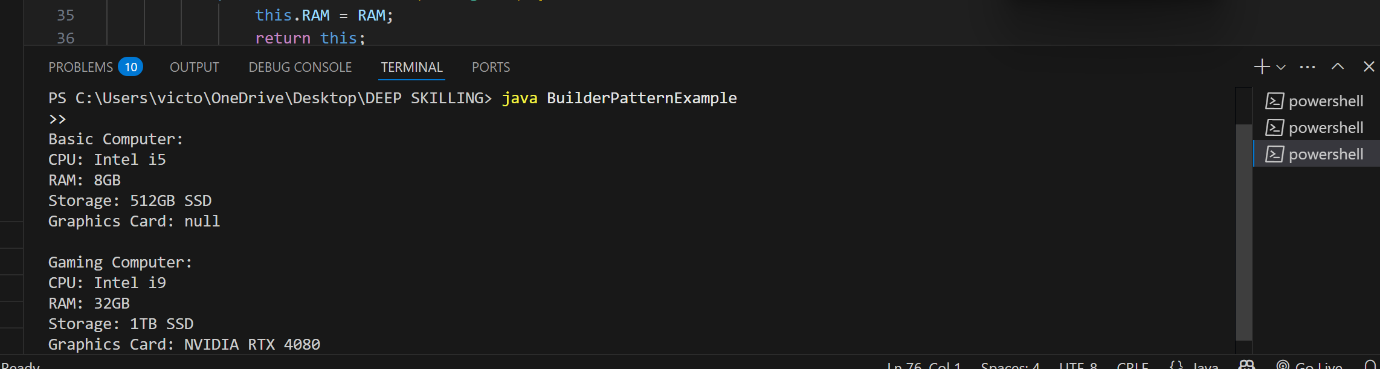
System.out.println("\nGaming Computer:");

gamingComputer.displaySpecs();

}

}

OUTPUT:



**Exercise 4: Implementing the Adapter Pattern**

Code:

public class AdapterPatternExample {

interface PaymentProcessor {

void processPayment(double amount);

}

static class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Payment of ₹" + amount + " processed using PayPal.");

}

}

static class StripeGateway {

public void executeTransaction(double amount) {

System.out.println("Payment of ₹" + amount + " processed using Stripe.");

}

}

static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter(PayPalGateway payPal) {

this.payPal = payPal;

}

public void processPayment(double amount) {

payPal.makePayment(amount);

}

}

static class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.executeTransaction(amount);

}

}

public static void main(String[] args) {

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

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

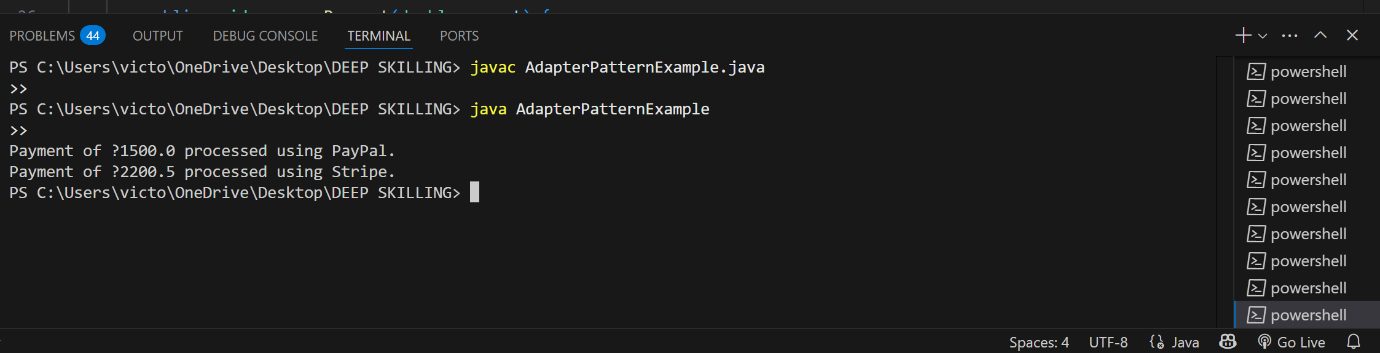
paypalProcessor.processPayment(1500.00);

stripeProcessor.processPayment(2200.50);

}

}

OUTPUT:



**Exercise 5: Implementing the Decorator Pattern**

Code:

public class DecoratorPatternExample {

interface Notifier {

void send(String message);

}

static class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

static abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message);

}

}

static class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

static class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

public static void main(String[] args) {

Notifier baseNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

Notifier slackAndSmsNotifier = new SlackNotifierDecorator(smsNotifier);

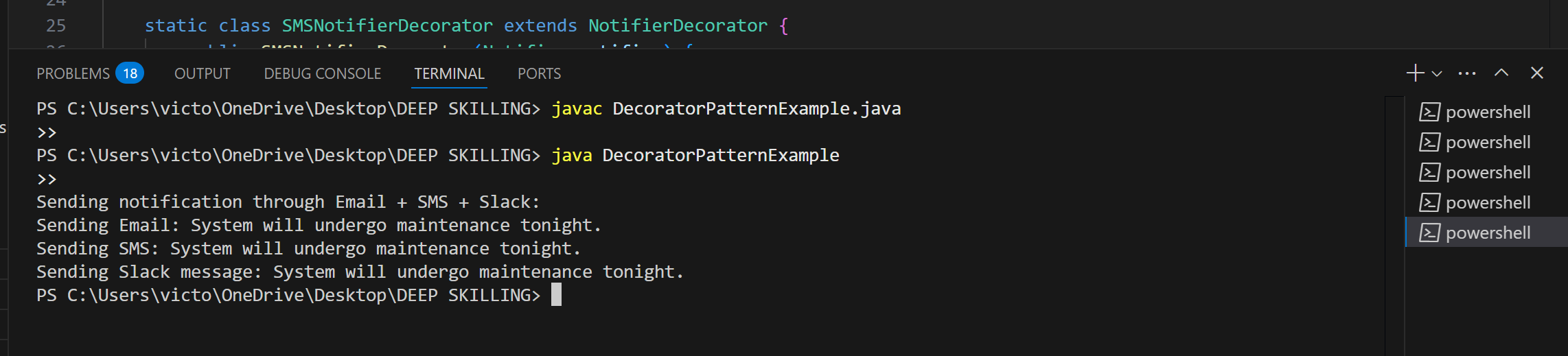
System.out.println("Sending notification through Email + SMS + Slack:");

slackAndSmsNotifier.send("System will undergo maintenance tonight.");

}

}

OUTPUT:



**Exercise 6: Implementing the Proxy Pattern**

Code:

public class ProxyPatternExample {

interface Image {

void display();

}

static class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

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

}

public void display() {

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

}

}

static class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

public static void main(String[] args) {

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

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

image1.display();

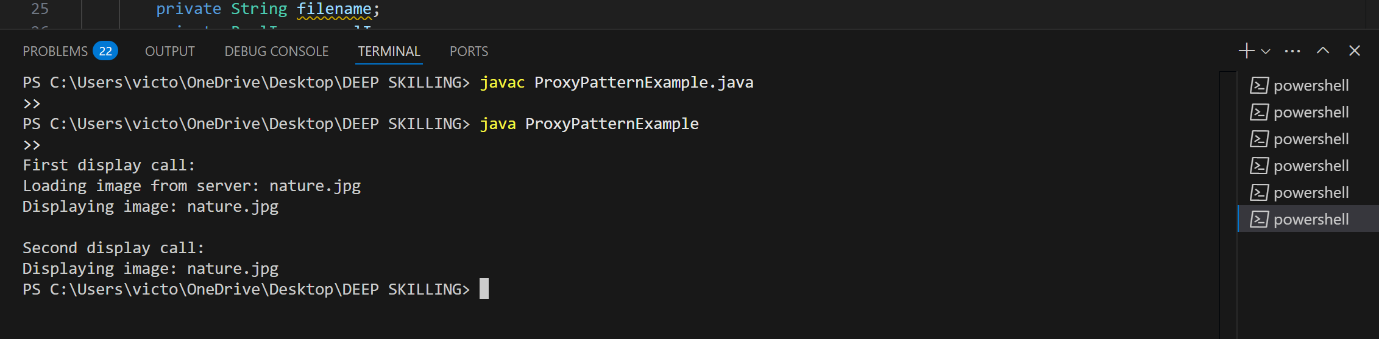
System.out.println("\nSecond display call:");

image1.display();

}

}

OUTPUT:



**Exercise 7: Implementing the Observer Pattern**

Code:

import java.util.ArrayList;

import java.util.List;

public class ObserverPatternExample {

interface Observer {

void update(String stockName, double newPrice);

}

interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

static class StockMarket implements Stock {

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

private String stockName;

private double price;

public StockMarket(String stockName, double price) {

this.stockName = stockName;

this.price = price;

}

public void registerObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockName, price);

}

}

public void setPrice(double newPrice) {

this.price = newPrice;

notifyObservers();

}

}

static class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

public void update(String stockName, double newPrice) {

System.out.println("MobileApp " + name + ": " + stockName + " price changed to ₹" + newPrice);

}

}

static class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

public void update(String stockName, double newPrice) {

System.out.println("WebApp " + name + ": " + stockName + " price changed to ₹" + newPrice);

}

}

public static void main(String[] args) {

StockMarket infosysStock = new StockMarket("INFY", 1500.0);

Observer mobileApp1 = new MobileApp("Android");

Observer webApp1 = new WebApp("Chrome");

infosysStock.registerObserver(mobileApp1);

infosysStock.registerObserver(webApp1);

System.out.println("Price update #1:");

infosysStock.setPrice(1525.0);

System.out.println("\nPrice update #2:");

infosysStock.setPrice(1498.0);

}

}

OUTPUT:



**Exercise 8: Implementing the Strategy Pattern**

Code:

public class StrategyPatternExample {

interface PaymentStrategy {

void pay(double amount);

}

static class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

public void pay(double amount) {

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

}

}

static class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

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

}

}

static class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy != null) {

strategy.pay(amount);

} else {

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

}

}

}

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

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

context.executePayment(1200.00);

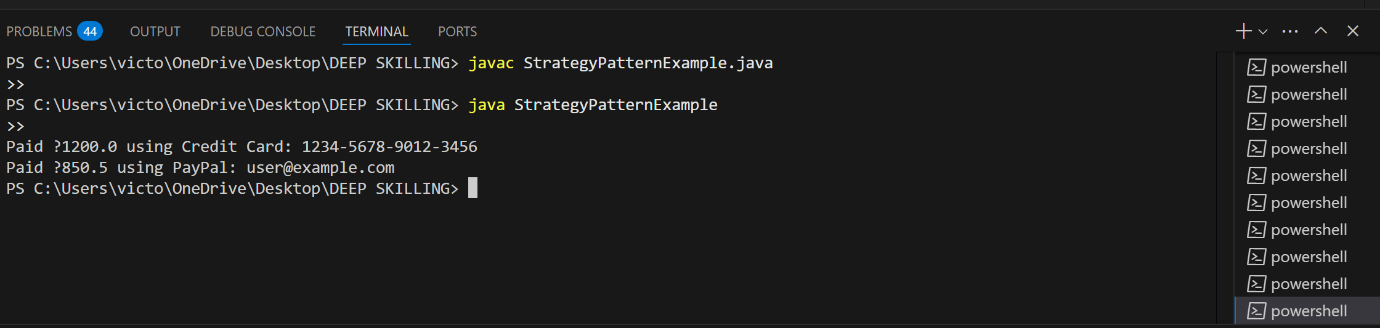
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.executePayment(850.50);

}

}

OUTPUT:



**Exercise 9: Implementing the Command Pattern**

Code:

public class CommandPatternExample {

interface Command {

void execute();

}

static class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

static class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

static 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.");

}

}

}

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();

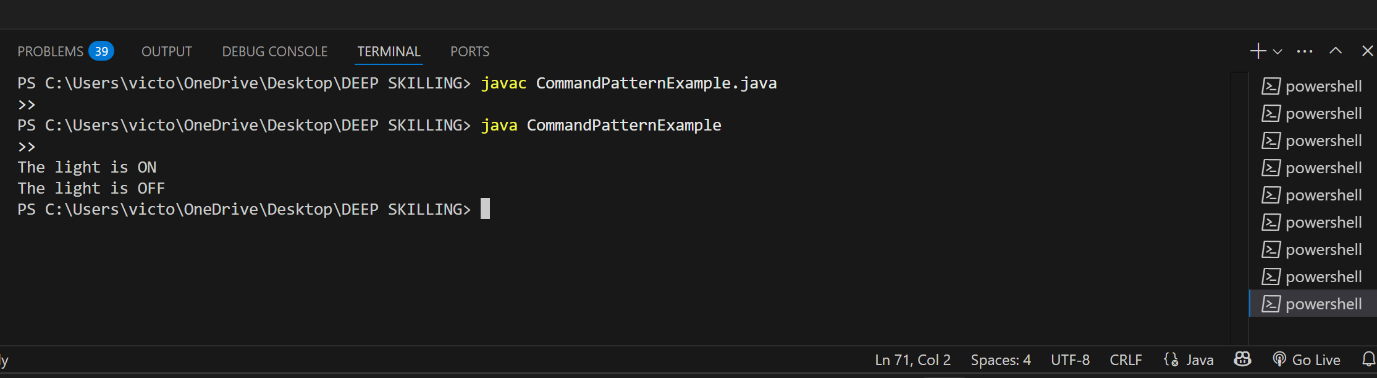
remote.setCommand(lightOff);

remote.pressButton();

}

}

OUTPUT:



**Exercise 10: Implementing the MVC Pattern**

Code:

public class MVCPatternExample {

static 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 String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

static class StudentView {

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

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

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

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

}

}

static 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 setStudentGrade(String grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

public static void main(String[] args) {

Student student = new Student("John", "S001", "B+");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Alice");

controller.setStudentGrade("A");

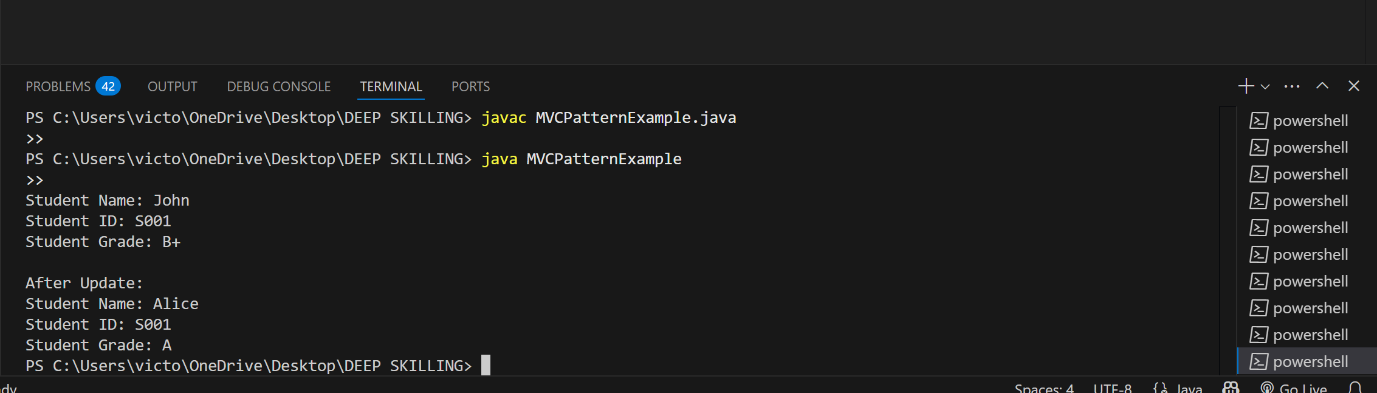
System.out.println("\nAfter Update:");

controller.updateView();

}

}

OUTPUT:



**Exercise 11: Implementing Dependency Injection**

Code:

public class DependencyInjectionExample {

interface CustomerRepository {

String findCustomerById(String id);

}

static class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer[ID=" + id + "]: Alice";

}

}

static class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

String customer = repository.findCustomerById(id);

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

}

}

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomerDetails("C102");

}

}

OUTPUT:

