**Week 1- Design Patterns- Hands-On Exercises**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

Logger.java

**public class Logger {**

    private static Logger instance;

    private Logger() {

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

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

    }

}

SingleTon.java

public class SingletonTest {

    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 logger1 and logger2 are the same instance.");

        } else {

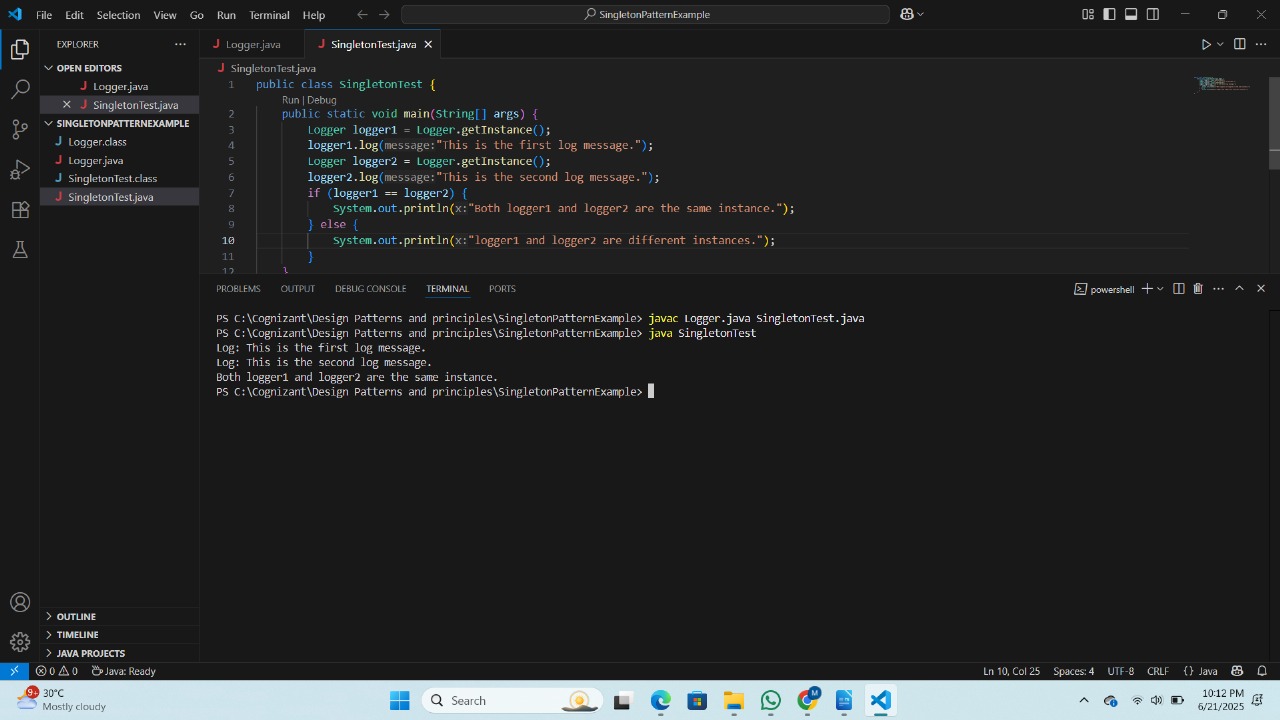
            System.out.println("logger1 and logger2 are different instances.");

        }

    }

}

**Output:**

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

**Code:**

Document.java

**public interface Document {**

    void open();

}

DocumentFactory.java

**public abstract class DocumentFactory {**

    public abstract Document createDocument();

}

ExcelDocument.java

**public class ExcelDocument implements Document {**

    @Override

    public void open() {

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

    }

}

ExcelDocumentFactory.java

**public class ExcelDocumentFactory extends DocumentFactory {**

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

FactoryMethodTest.java

**public class FactoryMethodTest {**

    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();

    }

}

PdfDocument.java

**public class PdfDocument implements Document {**

    @Override

    public void open() {

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

    }

}

PdfDocumentFactory.java

**public class PdfDocumentFactory extends DocumentFactory {**

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

WordDocument.java

**public class WordDocument implements Document {**

    @Override

    public void open() {

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

    }

}

WordDocumentFactory.java

**public class WordDocumentFactory extends DocumentFactory {**

    @Override

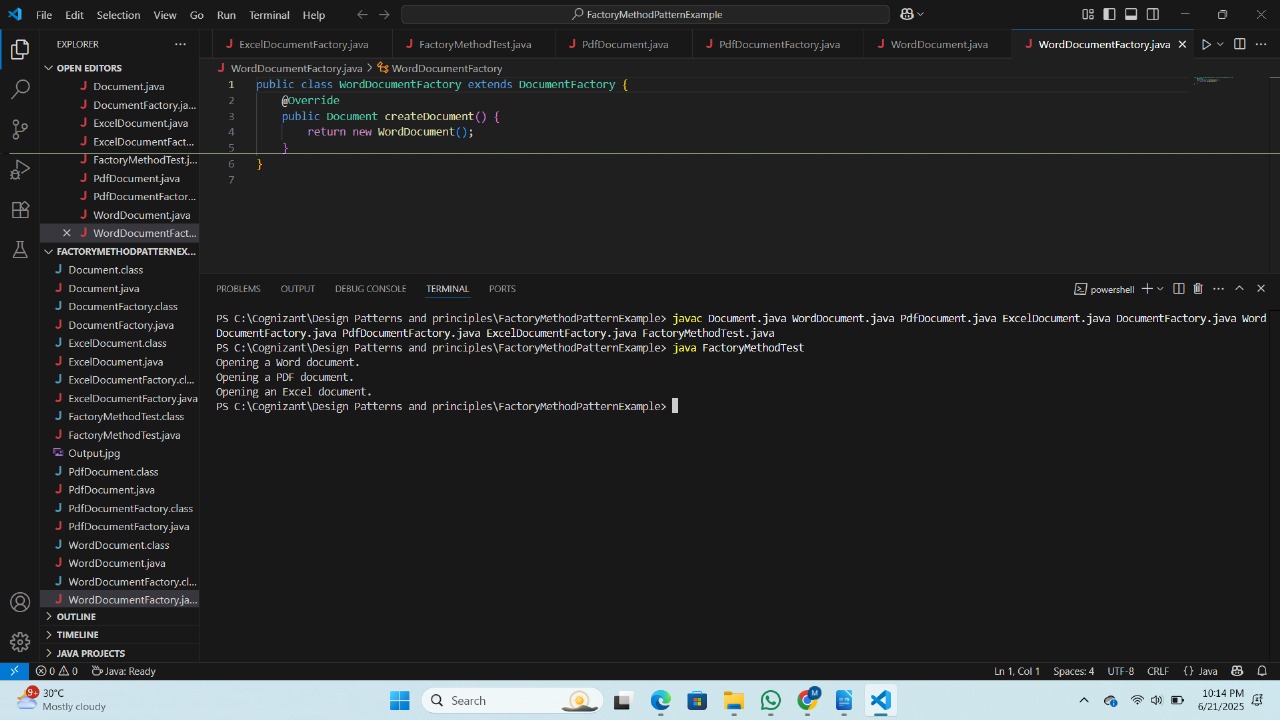
    public Document createDocument() {

        return new WordDocument();

    }

}

**Output:**

  
  
**Exercise 3: Implementing the Builder Pattern**

**Code:**

BuilderTest.java

**public class BuilderTest {**

    public static void main(String[] args) {

        Computer gamingPC = new Computer.Builder()

                .setCPU("Intel i9")

                .setRAM("32GB")

                .setStorage("1TB SSD")

                .setGPU("NVIDIA RTX 3080")

                .setOS("Windows 10")

                .build();

        Computer officePC = new Computer.Builder()

                .setCPU("Intel i5")

                .setRAM("16GB")

                .setStorage("512GB SSD")

                .setOS("Windows 10")

                .build();

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

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

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

        System.out.println("Storage: " + gamingPC.getStorage());

        System.out.println("GPU: " + gamingPC.getGPU());

        System.out.println("OS: " + gamingPC.getOS());

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

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

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

        System.out.println("Storage: " + officePC.getStorage());

        System.out.println("GPU: " + officePC.getGPU());

        System.out.println("OS: " + officePC.getOS());

    }

}

Computer.java

**public class Computer {**

    private String CPU;

    private String RAM;

    private String storage;

    private String GPU;

    private String OS;

    private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.storage = builder.storage;

        this.GPU = builder.GPU;

        this.OS = builder.OS;

    }

    public String getCPU() {

        return CPU;

    }

    public String getRAM() {

        return RAM;

    }

    public String getStorage() {

        return storage;

    }

    public String getGPU() {

        return GPU;

    }

    public String getOS() {

        return OS;

    }

    public static class Builder {

        private String CPU;

        private String RAM;

        private String storage;

        private String GPU;

        private String OS;

        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 setGPU(String GPU) {

            this.GPU = GPU;

            return this;

        }

        public Builder setOS(String OS) {

            this.OS = OS;

            return this;

        }

        public Computer build() {

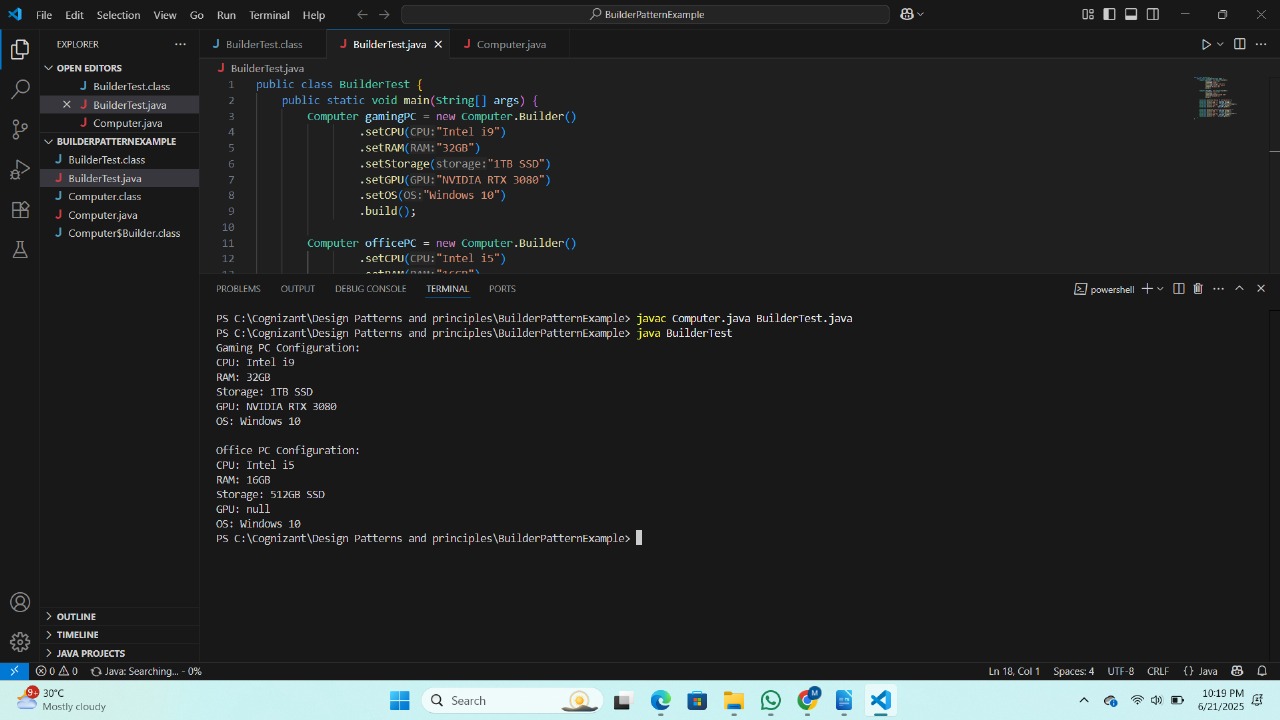
            return new Computer(this);

        }

    }

}

**Output:**

 **Exercise 4: Implementing the Adapter Pattern**

**Code:**

AdapterTest.java

**public class AdapterTest {**

    public static void main(String[] args) {

        PaymentProcessor payPalProcessor = new PayPalAdapter();

        payPalProcessor.processPayment(100.0);

        PaymentProcessor stripeProcessor = new StripeAdapter();

        stripeProcessor.processPayment(200.0);

    }

}

PayementProcessor.java

**public interface PaymentProcessor {**

    void processPayment(double amount);

}

PayPalAdapter.java

**public class PayPalAdapter implements PaymentProcessor {**

    private PayPalGateway payPalGateway;

    public PayPalAdapter() {

        this.payPalGateway = new PayPalGateway();

    }

    @Override

    public void processPayment(double amount) {

        payPalGateway.makePayment(amount);

    }

}

PayPalGateway.java

**public class PayPalGateway {**

    public void makePayment(double amount) {

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

    }

}

StripeAdapter.java

**public class StripeAdapter implements PaymentProcessor {**

    private StripeGateway stripeGateway;

    public StripeAdapter() {

        this.stripeGateway = new StripeGateway();

    }

    @Override

    public void processPayment(double amount) {

        stripeGateway.charge(amount);

    }

}

StripeGateway.java

**public class StripeGateway {**

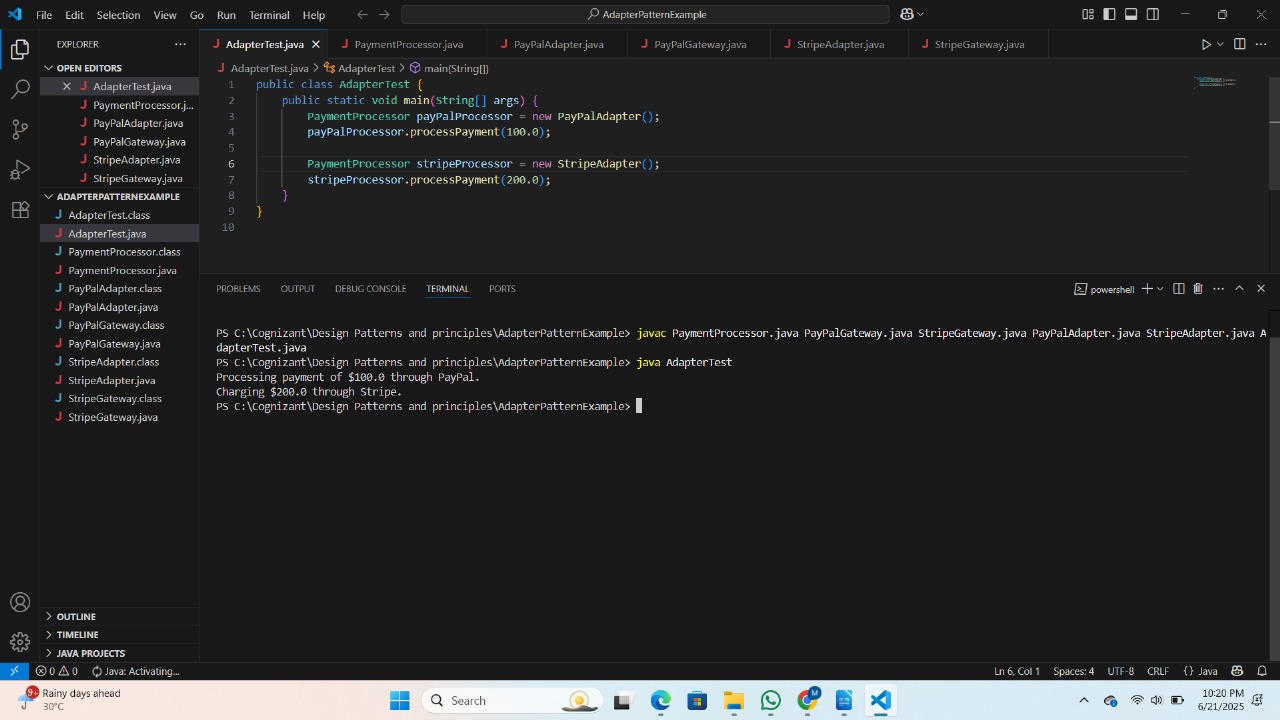
    public void charge(double amount) {

        System.out.println("Charging $" + amount + " through Stripe.");

    }

}

**Output:**

 **Exercise 5: Implementing the Decorator Pattern**

**Code:**

DecoratorTest.java

**public class DecoratorTest {**

    public static void main(String[] args) {

        Notifier emailNotifier = new EmailNotifier();

        emailNotifier.send("Hello, this is a test notification!");

        Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

        smsNotifier.send("Hello, this is a test notification!");

        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

        slackNotifier.send("Hello, this is a test notification!");

    }

}

EmailNotifier.java

**public class EmailNotifier implements Notifier {**

    @Override

    public void send(String message) {

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

    }

}

Notifier.java

**public interface Notifier {**

    void send(String message);

}

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);

    }

}

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 notification: " + 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);

    }

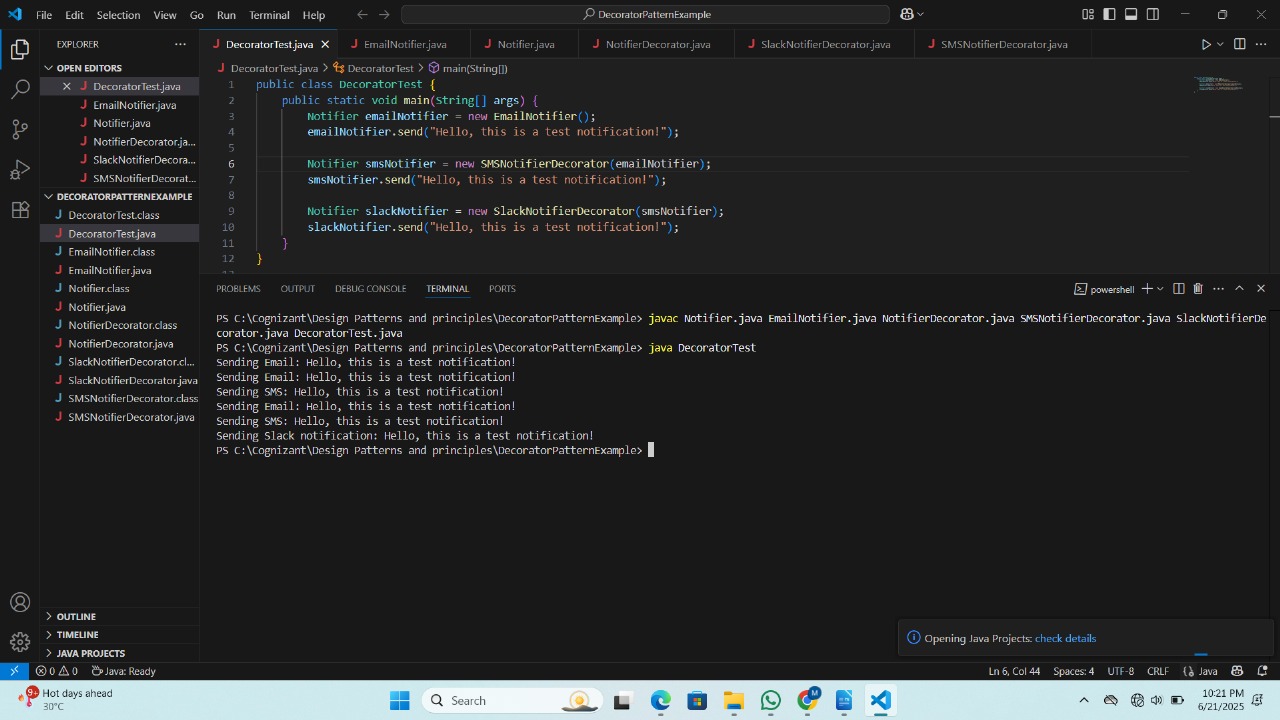
    private void sendSMS(String message) {

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

    }

}

**Output:**

 **Exercise 6: Implementing the Proxy Pattern**

**Code:**

Image.java

**public interface Image {**

    void display();

}

ProxyImage.java

public interface Image {

   void display();

}

ProxyTest.java

**public class ProxyTest {**

    public static void main(String[] args) {

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

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

        image1.display();

        image1.display();

        image2.display();

    }

}

RealImage.java

**public class RealImage implements Image {**

    private String filename;

    public RealImage(String filename) {

        loadImageFromServer();

        this.filename = filename;

    }

    private void loadImageFromServer() {

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

    }

    @Override

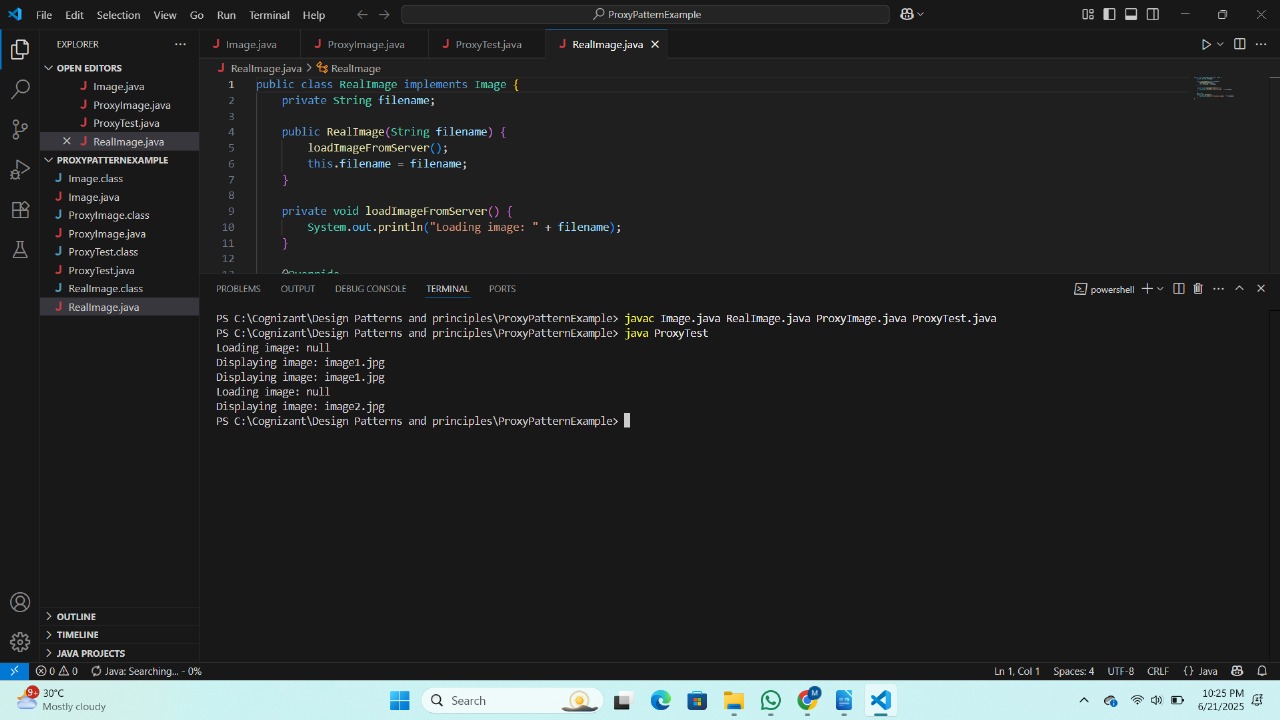
    public void display() {

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

    }

}

**Output:**

 **Exercise 7: Implementing the Observer Pattern**

**Code:**

MobileApp.java

**public class MobileApp implements Observer {**

    @Override

    public void update(double stockPrice) {

        System.out.println("Mobile App: Stock price updated to $" + stockPrice);

    }

}

Observer.java

**public interface Observer {**

    void update(double stockPrice);

}

ObserverTest.java

**public class ObserverTest {**

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        MobileApp mobileApp = new MobileApp();

        WebApp webApp = new WebApp();

        stockMarket.registerObserver(mobileApp);

        stockMarket.registerObserver(webApp);

        stockMarket.setStockPrice(100.50);

        stockMarket.setStockPrice(102.75);

        stockMarket.deregisterObserver(mobileApp);

        stockMarket.setStockPrice(99.99);

    }

}

Stock.java

**import java.util.ArrayList;**

import java.util.List;

public interface Stock {

    void registerObserver(Observer observer);

    void deregisterObserver(Observer observer);

    void notifyObservers();

}

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<>();

    }

    public void setStockPrice(double stockPrice) {

        this.stockPrice = stockPrice;

        notifyObservers();

    }

    @Override

    public void registerObserver(Observer observer) {

        observers.add(observer);

    }

    @Override

    public void deregisterObserver(Observer observer) {

        observers.remove(observer);

    }

    @Override

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(stockPrice);

        }

    }

}

WebApp.java

**public class WebApp implements Observer {**

    @Override

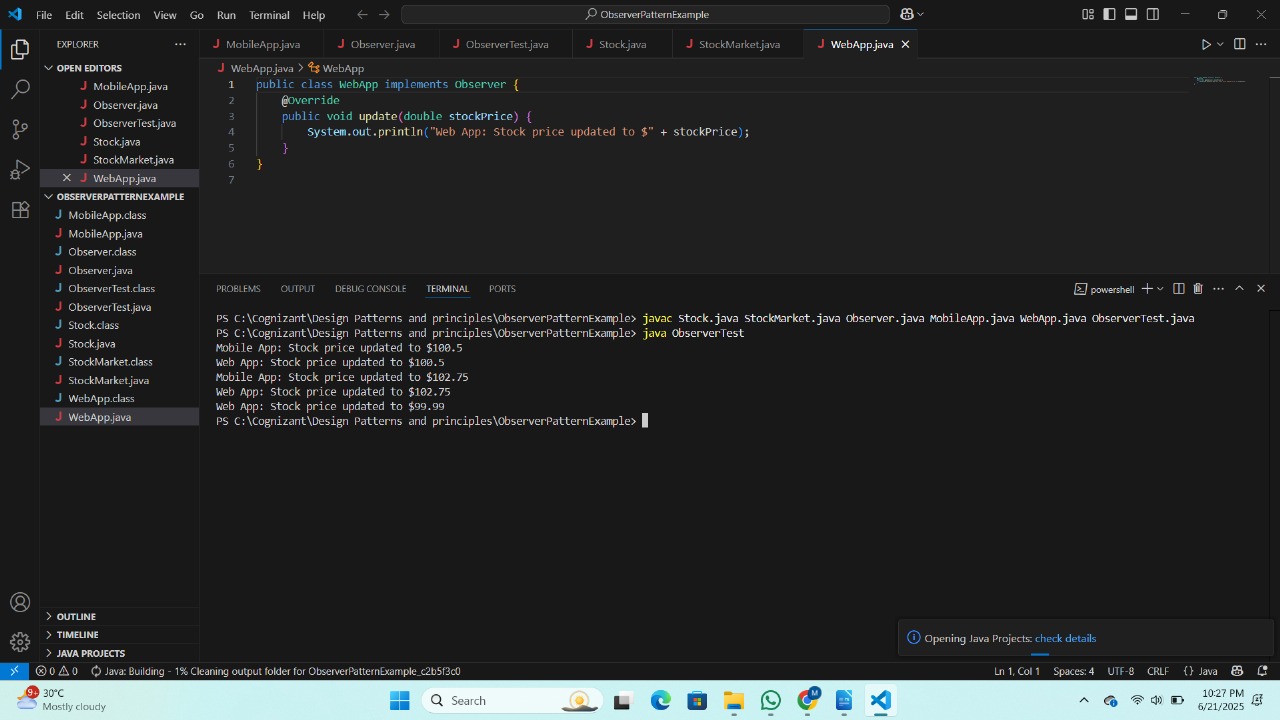
    public void update(double stockPrice) {

        System.out.println("Web App: Stock price updated to $" + stockPrice);

    }

}

**Output:**

 **Exercise 8: Implementing the Strategy Pattern**

**Code:**

CreditCardPayment.java

**public 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: " + cardNumber);

    }

}

PaymentContext.java

**public class PaymentContext {**

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentStrategy = paymentStrategy;

    }

    public void executePayment(double amount) {

        if (paymentStrategy != null) {

            paymentStrategy.pay(amount);

        } else {

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

        }

    }

}

PaymentStrategy.java

**public interface PaymentStrategy {**

    void pay(double amount);

}

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);

    }

}

StrategyTest.java

**public class StrategyTest {**

    public static void main(String[] args) {

        PaymentContext paymentContext = new PaymentContext();

        PaymentStrategy creditCardPayment = new CreditCardPayment("1234-5678-9876-5432");

        paymentContext.setPaymentStrategy(creditCardPayment);

        paymentContext.executePayment(150.00);

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

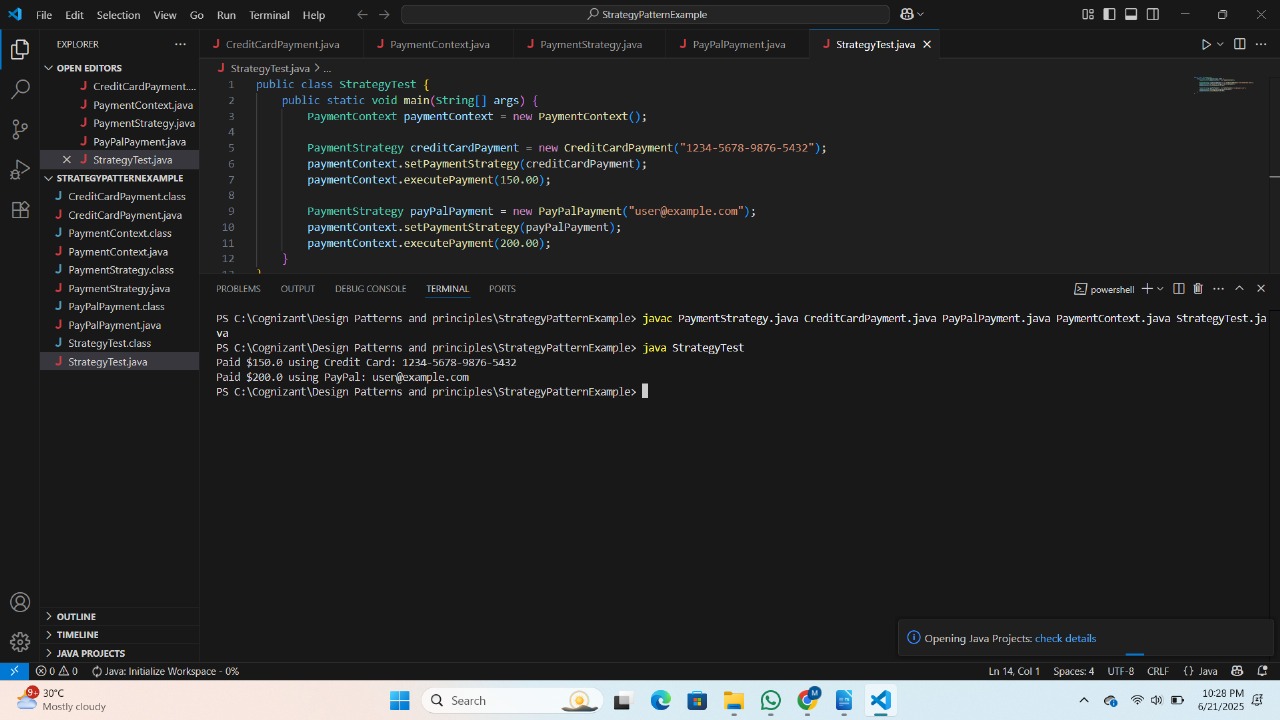
        paymentContext.setPaymentStrategy(payPalPayment);

        paymentContext.executePayment(200.00);

    }

}

**Output:**

 **Exercise 9: Implementing the Command Pattern**

**Code:**

Command.java

**public interface Command {**

    void execute();

}

CommandTest.java

**public class CommandTest {**

    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();

    }

}

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");

    }

}

LightOffCommand.java

**public class LightOffCommand implements Command {**

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

LightOnCommand.java

**public class LightOnCommand implements Command {**

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

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 {

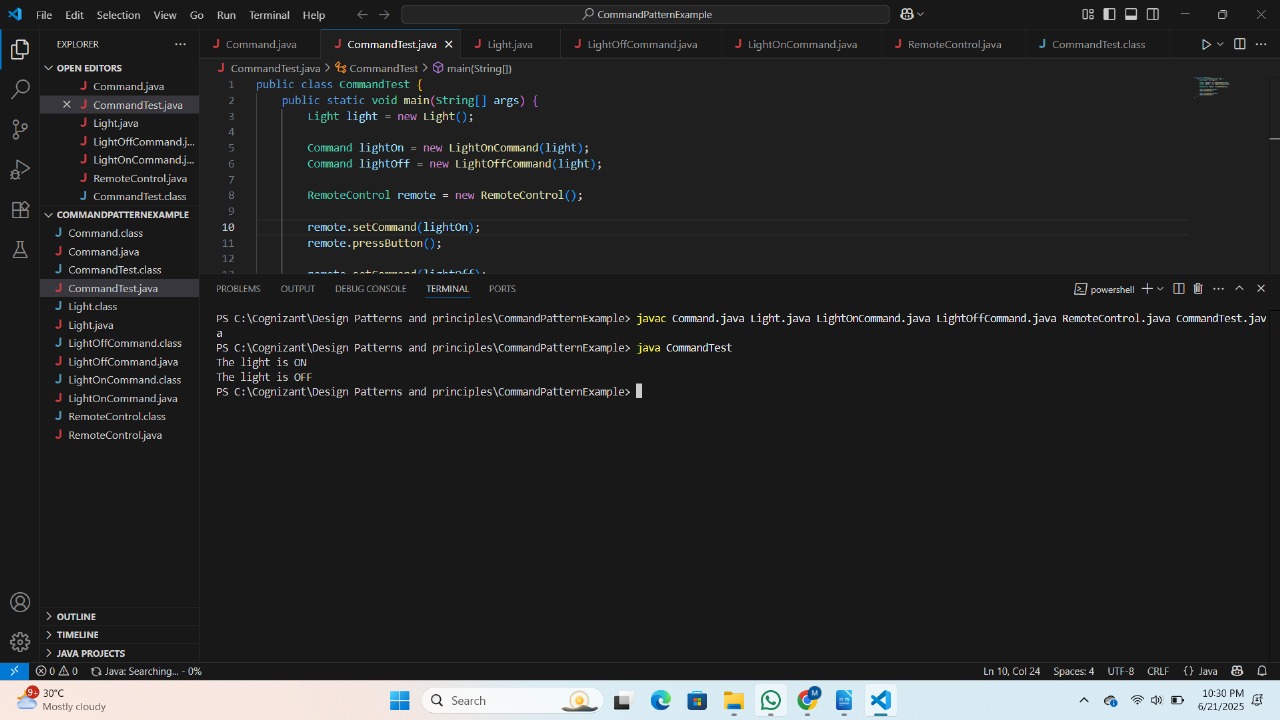
            System.out.println("No command set.");

        }

    }

}

**Output:**

 **Exercise 10: Implementing the MVC Pattern**

**Code:**

MVCTest.java

**public class MVCTest {**

    public static void main(String[] args) {

        Student student = new Student("John Doe", 1, "A");

        StudentView studentView = new StudentView();

        StudentController studentController = new StudentController(student, studentView);

        studentController.updateView();

        studentController.setStudentName("Jane Doe");

        studentController.setStudentId(2);

        studentController.setStudentGrade("B");

        studentController.updateView();

    }

}

Student.java

**public class Student {**

    private String name;

    private int id;

    private String grade;

    public Student(String name, int 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 int getId() {

        return id;

    }

    public void setId(int id) {

        this.id = id;

    }

    public String getGrade() {

        return grade;

    }

    public void setGrade(String grade) {

        this.grade = grade;

    }

}

StudentController.java

**public class StudentController {**

    private Student student;

    private StudentView studentView;

    public StudentController(Student student, StudentView studentView) {

        this.student = student;

        this.studentView = studentView;

    }

    public void setStudentName(String name) {

        student.setName(name);

    }

    public String getStudentName() {

        return student.getName();

    }

    public void setStudentId(int id) {

        student.setId(id);

    }

    public int getStudentId() {

        return student.getId();

    }

    public void setStudentGrade(String grade) {

        student.setGrade(grade);

    }

    public String getStudentGrade() {

        return student.getGrade();

    }

    public void updateView() {

        studentView.displayStudentDetails(student.getName(), student.getId(), student.getGrade());

    }

}

StudentView.java

**public class StudentView {**

    public void displayStudentDetails(String studentName, int studentId, String studentGrade) {

        System.out.println("Student Details:");

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

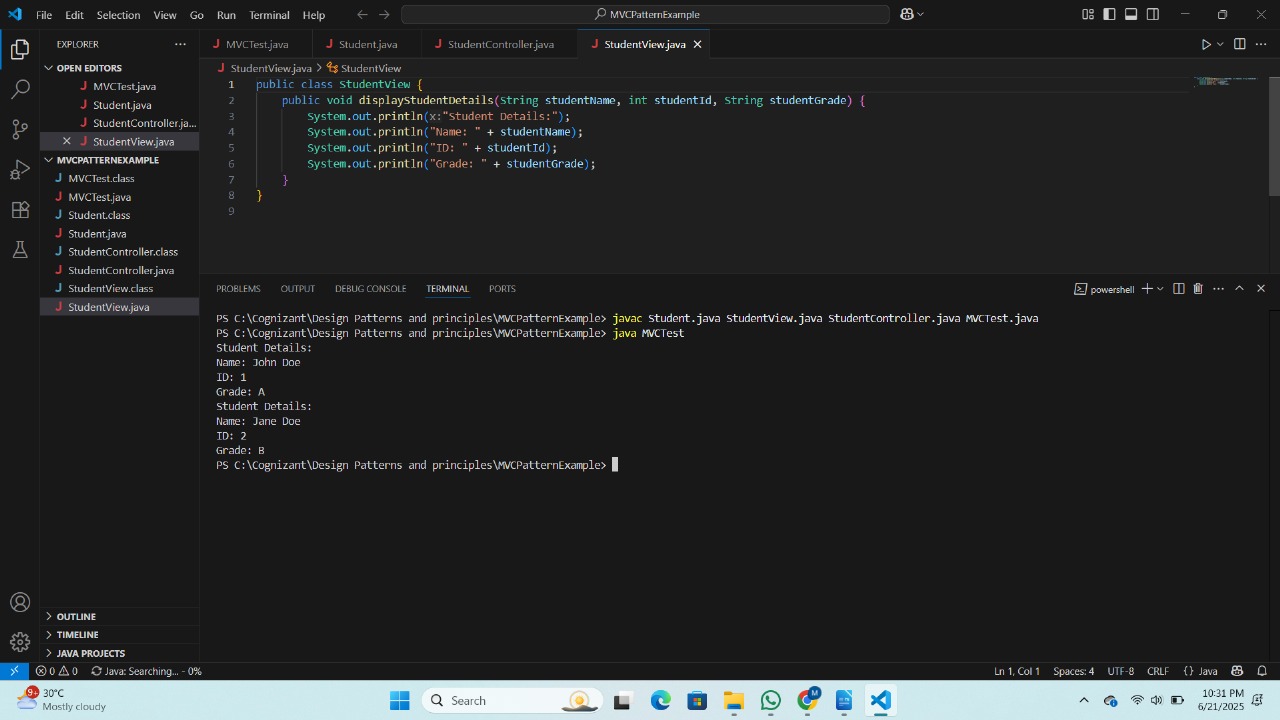
        System.out.println("ID: " + studentId);

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

    }

}

**Output:**

 **Exercise 11: Implementing Dependency Injection**

**Code:**

CustomerRepository.java

**public interface CustomerRepository {**

    String findCustomerById(int id);

}

CustomerRepositoryImpl.java

**public class CustomerRepositoryImpl implements CustomerRepository {**

    @Override

    public String findCustomerById(int id) {

        return "Customer with ID: " + id;

    }

}

CustomerService.java

**public class CustomerService {**

    private CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public String getCustomer(int id) {

        return customerRepository.findCustomerById(id);

    }

}

DependencyInjectionTest.java

**public class DependencyInjectionTest {**

    public static void main(String[] args) {

        CustomerRepository customerRepository = new CustomerRepositoryImpl();

        CustomerService customerService = new CustomerService(customerRepository);

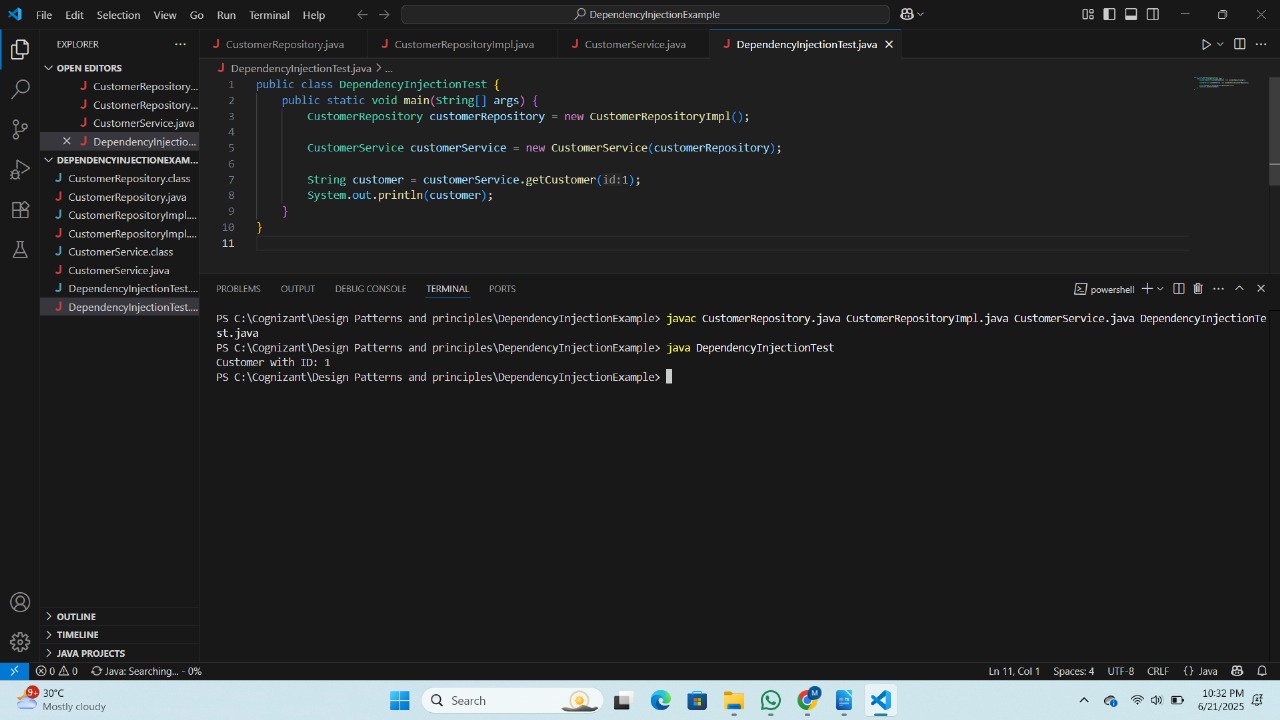
        String customer = customerService.getCustomer(1);

        System.out.println(customer);

    }

}

**Output:**

****