**Design principles & Patterns :**

**QUESTION 1:**

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

**SOLUTION:**

**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);

    }

}

**TestLogger.java:**

public class TestLogger {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("First log message");

        Logger logger2 = Logger.getInstance();

        logger2.log("Second log message");

        if (logger1 == logger2) {

            System.out.println("Only one instance of Logger is used.");

        } else {

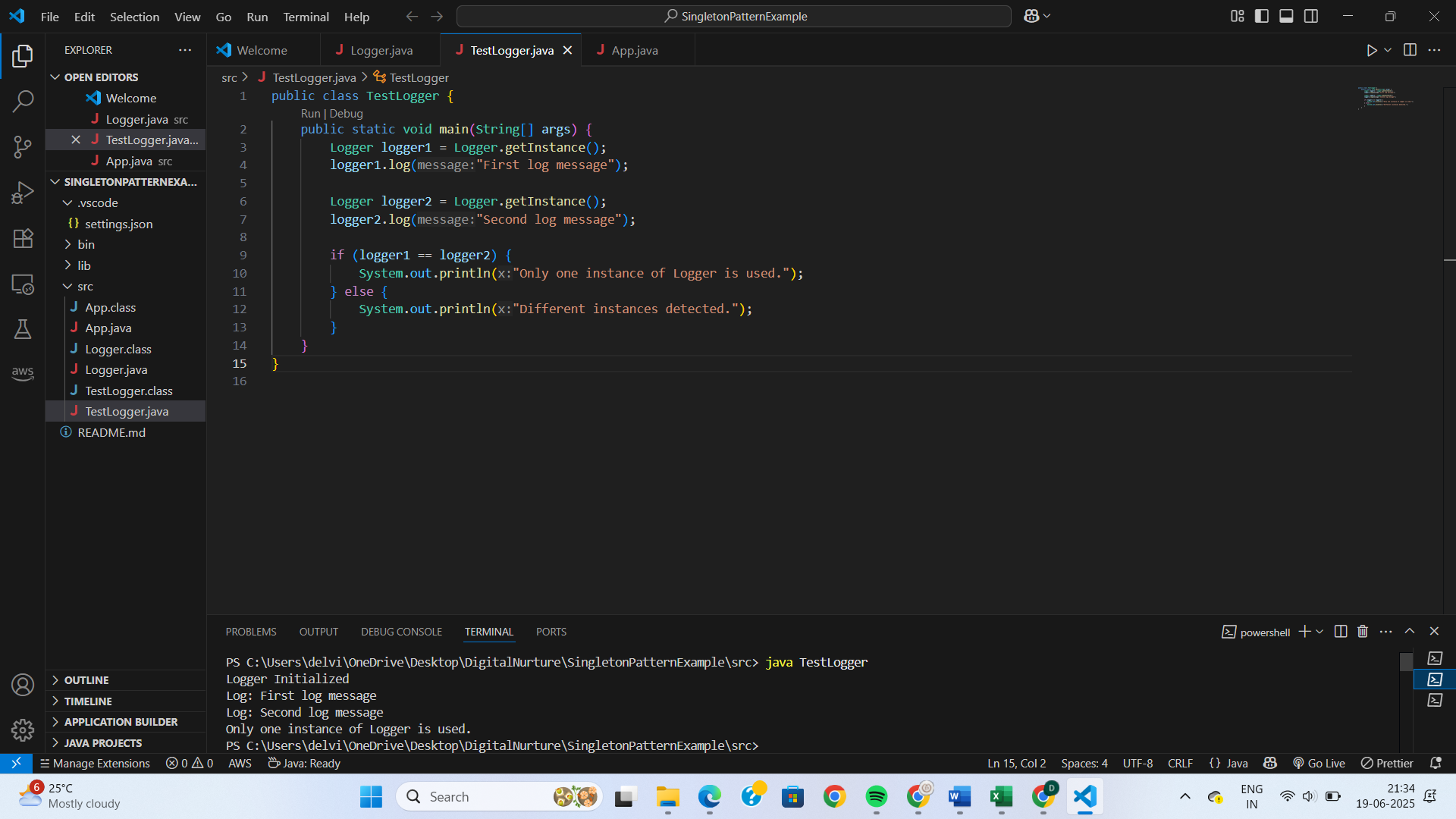
            System.out.println("Different instances detected.");

        }

    }

}

**OUTPUT:**



**QUESTION 2:**

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

**SOLUTION:**

**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 Excel document...");

    }

}

**ExcelFactory.java:**

public class ExcelFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**PdfDocument.java:**

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

}

**PdfFactory.java:**

public class PdfFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

**TestFactory.java:**

public class TestFactory {

    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();

    }

}

**WordDocument.java:**

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

}

**WordFactory.java:**

public class WordFactory extends DocumentFactory {

    @Override

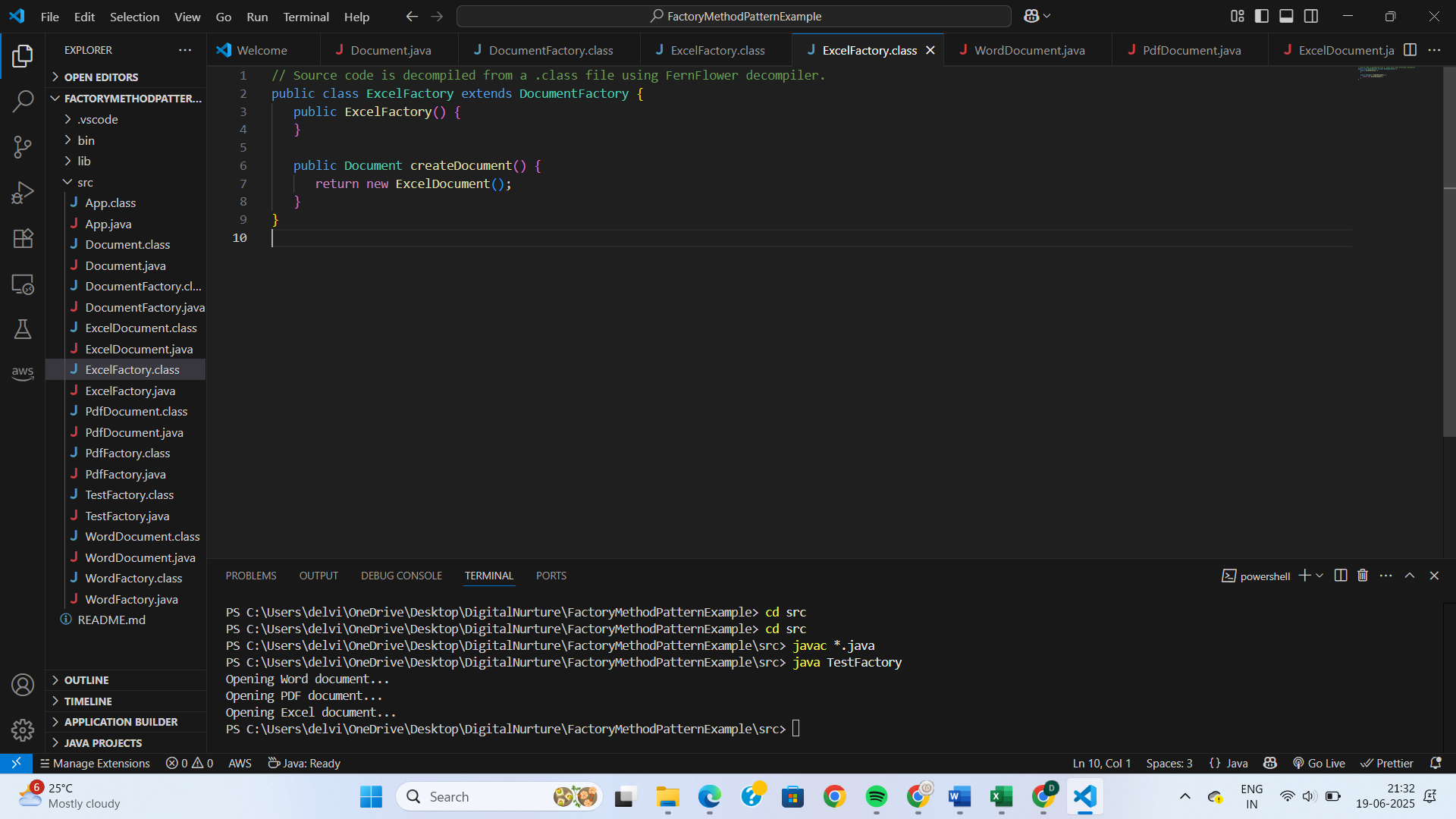
    public Document createDocument() {

        return new WordDocument();

    }

}

**OUTPUT:**



**Data structures and Algorithms:**

**QUESTION 1:**

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**SOLUTION:**

**Product.java:**

public class Product {

int productId;

String productName;

String category;

public Product(int id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

public String toString() {

return "[" + productId + ", " + productName + ", " + category + "]";

}

}

**SearchDemo.java:**

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

public static Product linearSearch(Product[] products, String name) {

for (Product product : products) {

if (product.productName.equalsIgnoreCase(name)) {

return product;

}

}

return null;

}

public static Product binarySearch(Product[] products, String name) {

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = products[mid].productName.compareToIgnoreCase(name);

if (cmp == 0) return products[mid];

if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(102, "Shoes", "Footwear"),

new Product(103, "Watch", "Accessories"),

new Product(104, "Book", "Education")

};

Product foundLinear = linearSearch(products, "Shoes");

System.out.println("Linear Search: " + foundLinear);

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

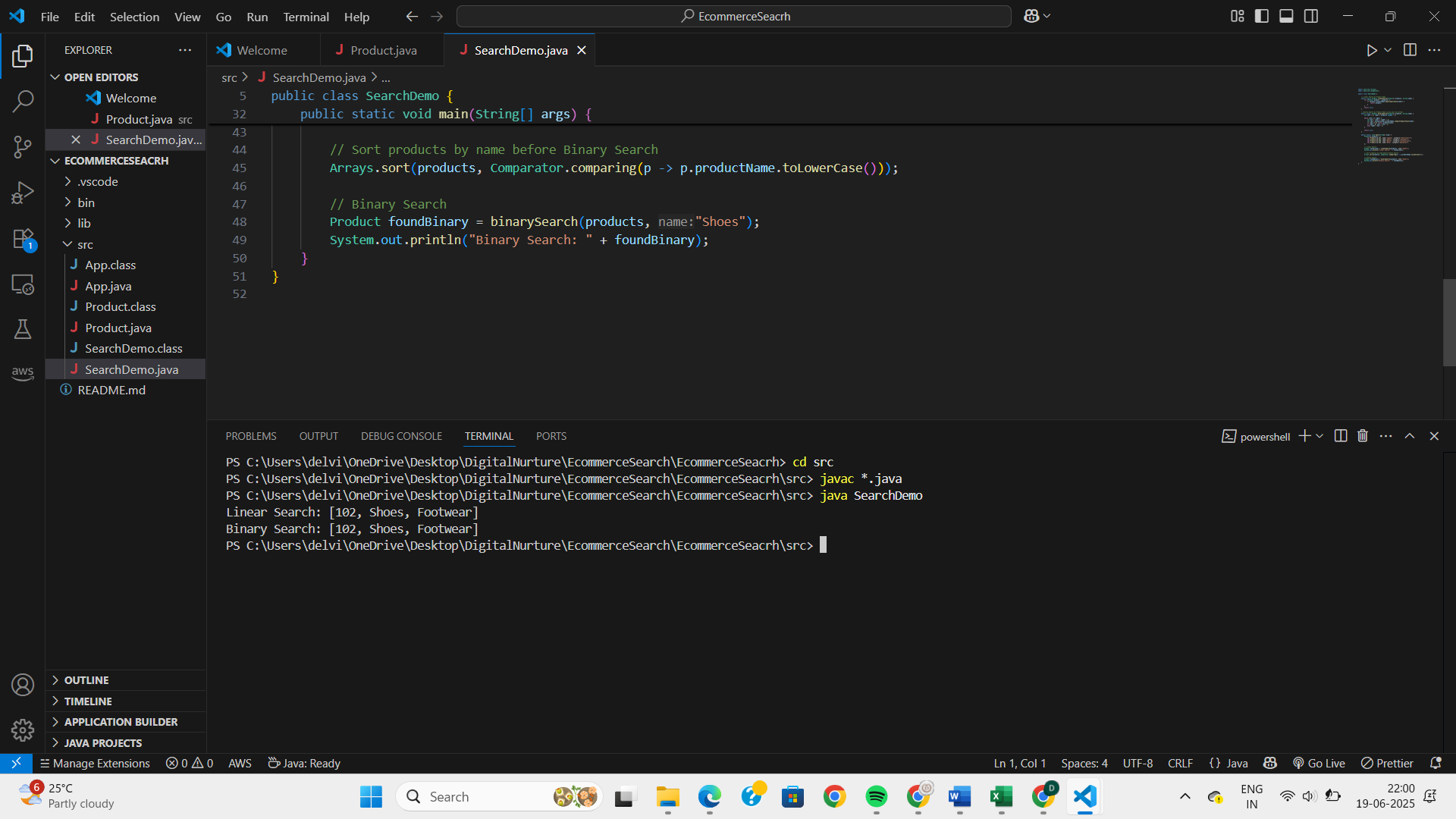
Product foundBinary = binarySearch(products, "Shoes");

System.out.println("Binary Search: " + foundBinary);

}

}

**OUTPUT:**



**QUESTION 2:**

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**SOLUTION:**

**Forecasting.java:**

public class Forecasting {

public static double futureValue(double principal, double rate, int periods) {

if (periods == 0) return principal;

return (1 + rate) \* futureValue(principal, rate, periods - 1);

}

public static void main(String[] args) {

double P = 10000;

double r = 0.05;

int n = 5;

double result = futureValue(P, r, n);

System.out.printf("Future Value after %d years: %.2f\n", n, result);

}

}

**OUTPUT:**

