**Java FSE Week 1**

**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);

    }

}

App.java:

public class App {

    public static void main(String[] args) throws Exception {

        Logger logger1 = Logger.getInstance();

        logger1.log("this is the first instance");

        Logger logger2 = Logger.getInstance();

        logger2.log("this is the second instance");

        if(logger1==logger2)

        {

            System.out.println("Both instances are the same");

        }

        else

        {

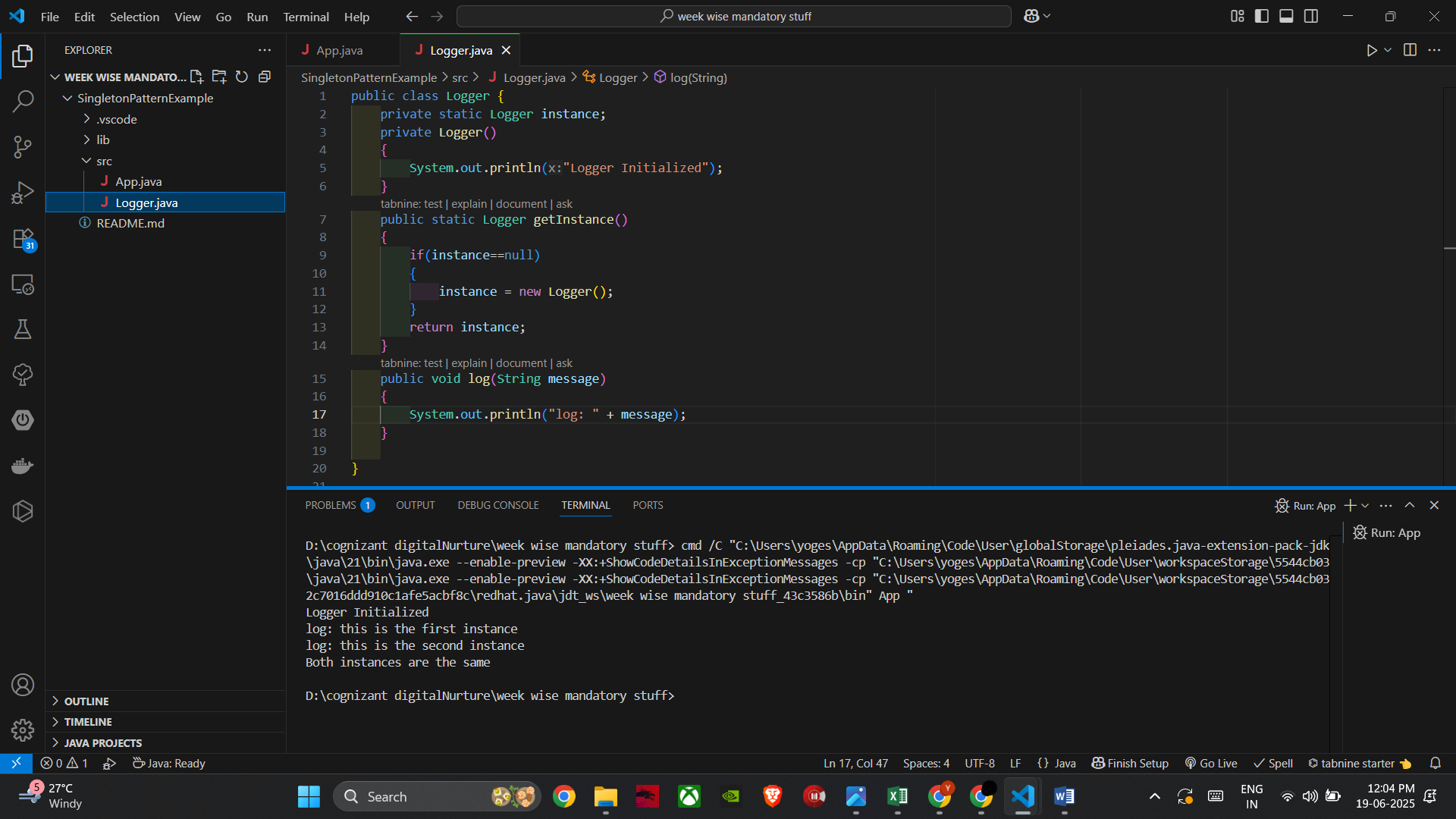
            System.out.println("Both are different");

        }

    }

}

Output:­

­­­

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

Document.java and DocumentFactory.java

public interface Document {

    void open();

}

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

WordDocument and wordFactory.java:

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

}

public class WordFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

ExcelDocument and ExcelFactory.java

public class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

}

public class ExcelFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

PdfDocument and PdfFactory.java

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

}

public class PdfFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

App.java

public class App {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelFactory();

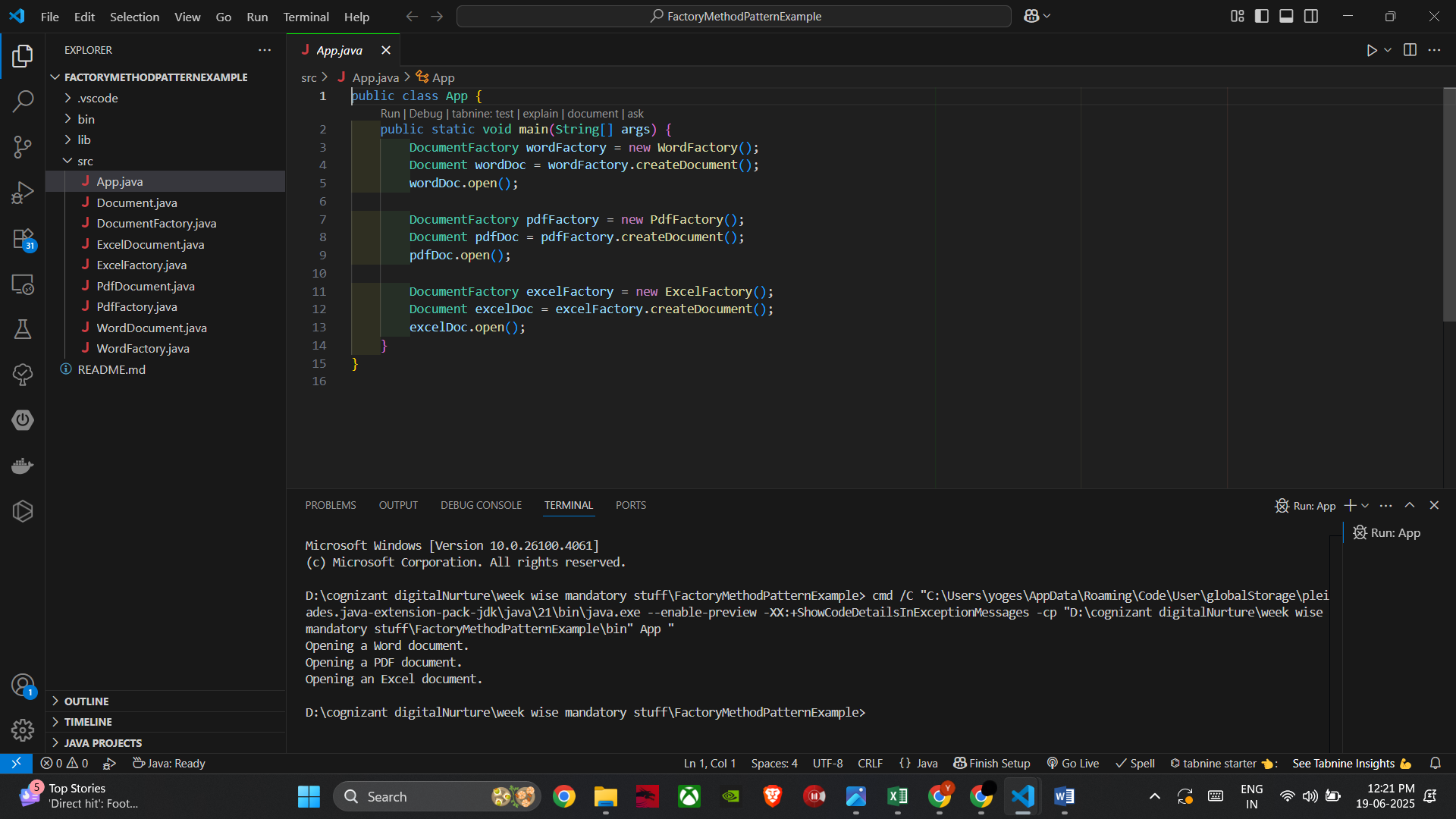
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

Output:



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

**Big O Notation**

* **Big O** notation describes the **time or space complexity** of an algorithm in terms of **input size** (n).
* It helps us understand how the **runtime grows** as input size increases.
* It’s **independent of hardware or language**—purely mathematical.
* Linear Search O(1)(Best case) or O(n)(worst case)
* Binary search O(1)(Best case) or O(n/2)(worst case)

Product.java

public class Product {

    int productId;

    String productName;

    String category;

    public Product(int productId, String productName, String category) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    @Override

    public String toString() {

        return productId + " - " + productName + " (" + category + ")";

    }

}

App.java

import java.util.Arrays;

import java.util.Comparator;

public class App {

    // Linear Search

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

        for (Product p : products) {

            if (p.productName.equalsIgnoreCase(target)) {

                return p;

            }

        }

        return null;

    }

    // Binary Search

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

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

        while (left <= right) {

            int mid = (left + right) / 2;

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

            if (cmp == 0)

                return products[mid];

            else 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", "Fashion"),

            new Product(103, "Phone", "Electronics"),

            new Product(104, "Watch", "Accessories")

        };

        System.out.println("Linear Search for 'Phone':");

        Product result1 = linearSearch(products, "Phone");

        System.out.println(result1 != null ? result1 : "Not found");

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

        System.out.println("\nBinary Search for 'Phone':");

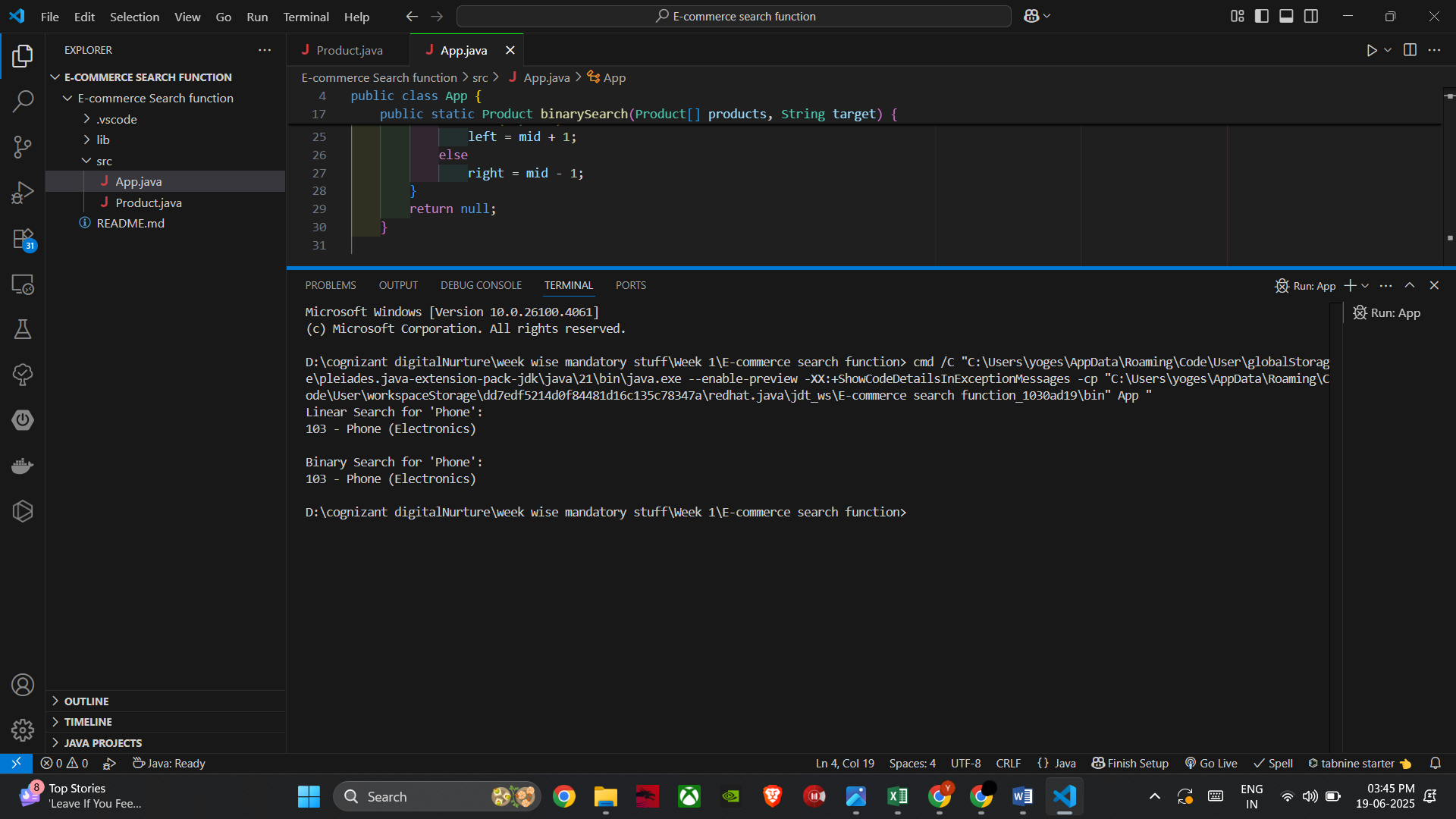
        Product result2 = binarySearch(products, "Phone");

        System.out.println(result2 != null ? result2 : "Not found");

    }

}

Output:



**Analysis:**

Linear search is suitable for small/test data and cannot be used in real time application whereas binary search can be used in real platform because it is faster and efficient.

So for an E-commerce platform binary search would be more suitable than the linear search.

**Exercise 7: Financial Forecasting**

Recursion:

Recursion is when a method **calls itself** to solve smaller versions of a problem until it reaches a base case.

App.java

public class App {

    public static double predictValue(int year, double currentValue, double growthRate) {

        if (year == 0) {

            return currentValue;

        }

        return predictValue(year - 1, currentValue, growthRate) \* (1 + growthRate);

    }

    public static void main(String[] args) {

        double currentValue = 10000;

        double growthRate = 0.08;

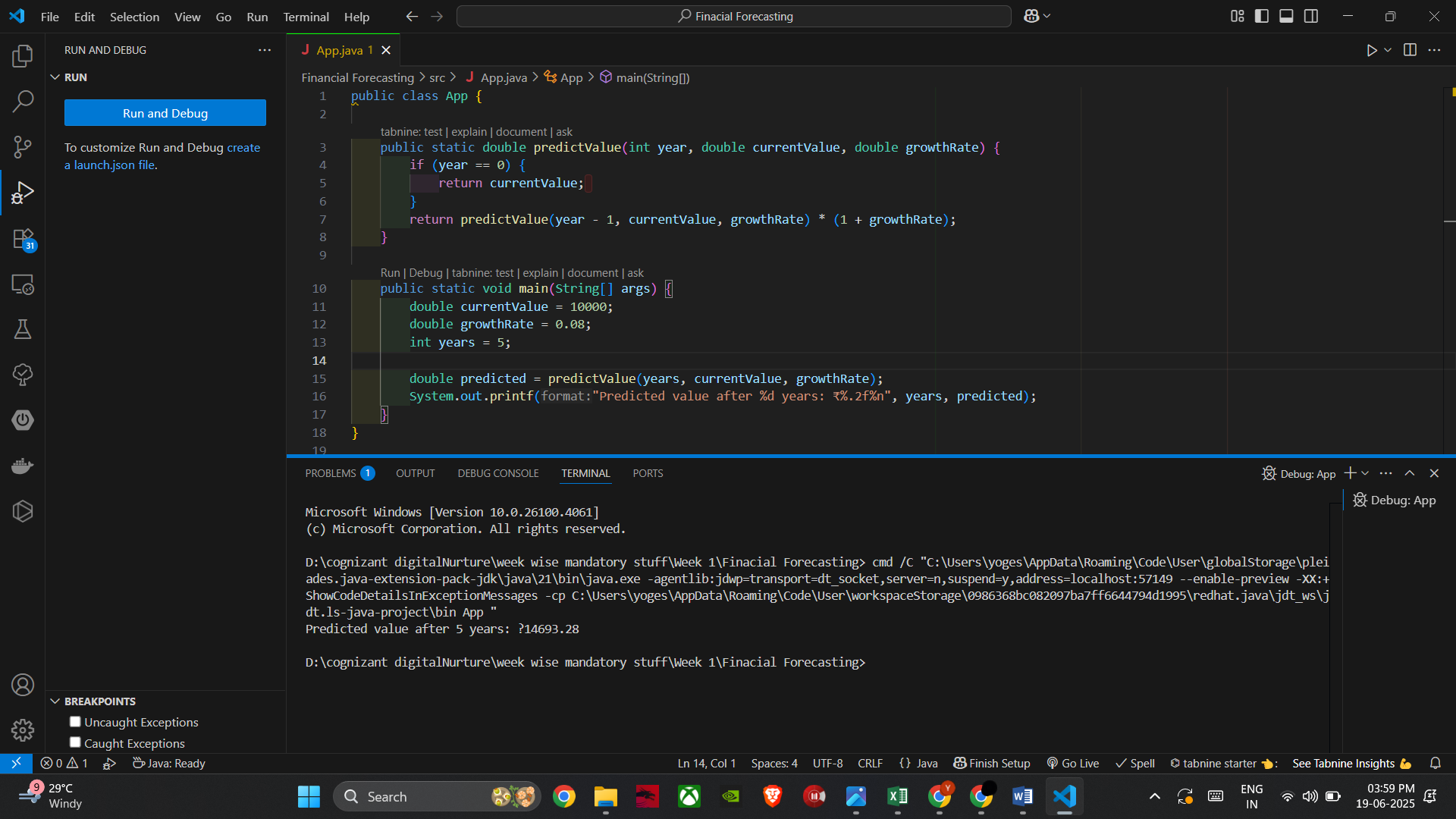
        int years = 5;

        double predicted = predictValue(years, currentValue, growthRate);

        System.out.printf("Predicted value after %d years: ₹%.2f%n", years, predicted);

    }

}

Output:

**Analysis:**

Time complexity: O(n)

Space complexity:O(n)

**Optimization Tips:**

### Problem with Deep Recursion

Too many recursive calls can **overflow the stack** or cause performance issues for large n.

### Tail Recursion / Iteration

Use **iteration instead** or optimize with **memoization** (for overlapping subproblems)