COGNIZANT

Digital Nurture 4.0

Deep Skilling - Java FSE

WEEK-1 HANDS ON

By Kaviya P

**TASK 1: IMPLEMENTATION OF SINGLETON PATTERN**

**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.**

**//CODE**

// Thread-safe Singleton Logger class using Double-Checked Locking

class Logger {

    // Static and volatile instance variable

    private static volatile Logger instance;

    // Private constructor to prevent external instantiation

    private Logger() {

        System.out.println("Logger instance created");

    }

    // Public method to provide global access point

    public static Logger getInstance() {

        // First check (no locking)

        Logger localInstance = instance;

        if (localInstance == null) {

            // Synchronized block to handle multithreading

            synchronized (Logger.class) {

                // Second check (with locking)

                if (localInstance == null) {

                    localInstance = new Logger();

                    instance = localInstance;

                }

            }

        }

        return localInstance;

    }

    // Sample log method

    public void log(String message) {

        System.out.println(Thread.currentThread().getName() + " LOG: " + message);

    }

}

// Test class to verify Singleton behaviour with multithreading

public class SingletonPatternExample {

    public static void main(String[] args) {

        // Runnable task to access Logger instance

        Runnable task = () -> {

            Logger logger = Logger.getInstance();

            logger.log("Logging from thread.");

            System.out.println(Thread.currentThread().getName() + " -> HashCode: " + logger.hashCode());

        };

        // Create multiple threads

        Thread t1 = new Thread(task, "Thread-1");

        Thread t2 = new Thread(task, "Thread-2");

        // Start threads

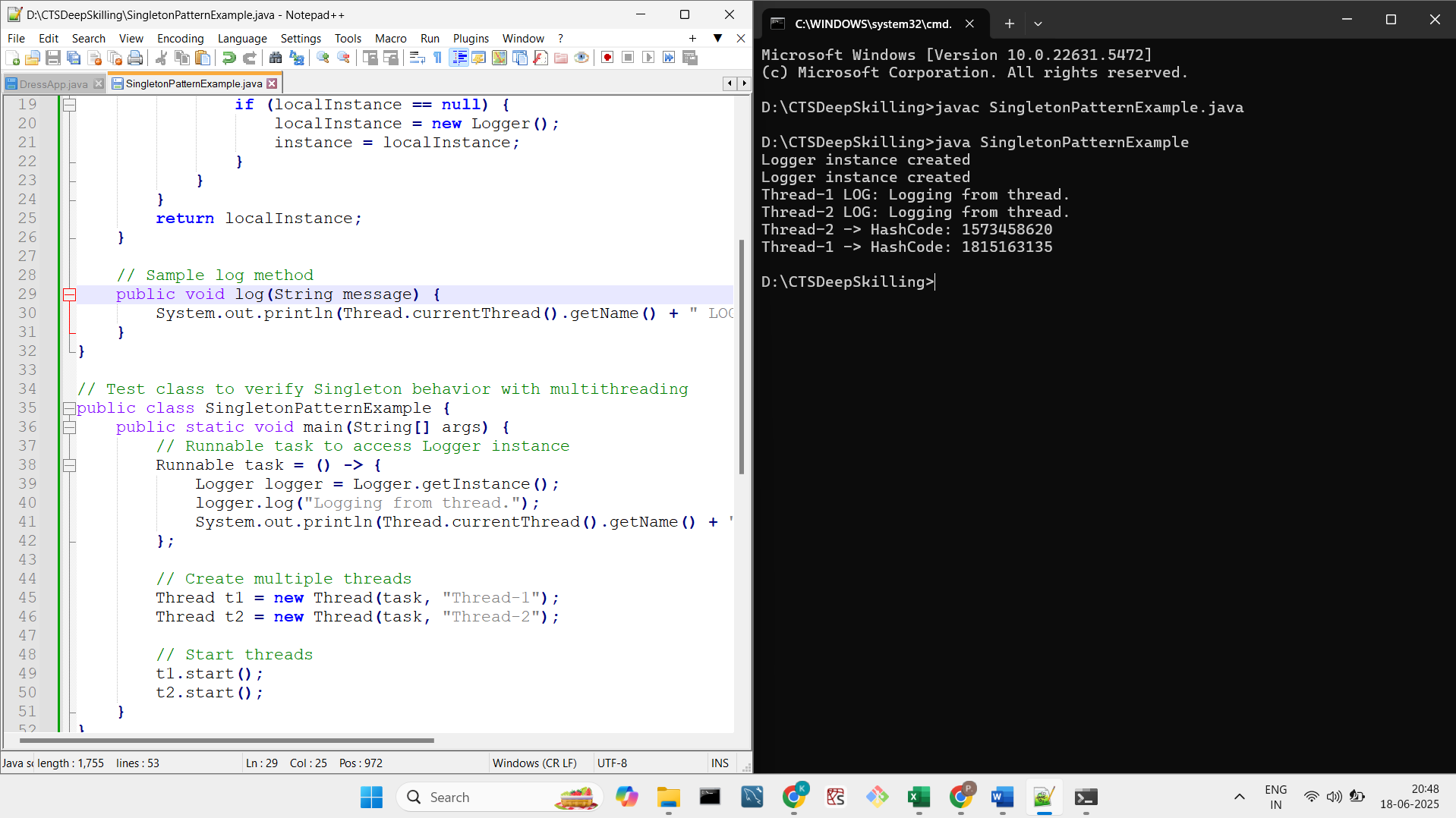
        t1.start();

        t2.start();

    }

}

**OUTPUT OF METHOD :**



**TASK 2: IMPLEMETATION OF FACTORY METHOD PATTERN**

**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.**

**//CODE**

**// Step 1: Define Document Interface**

**interface Document {**

**void open();**

**}**

**// Step 2: Concrete Document Classes**

**class WordDocument implements Document {**

**public void open() {**

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

**}**

**}**

**class PdfDocument implements Document {**

**public void open() {**

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

**}**

**}**

**class ExcelDocument implements Document {**

**public void open() {**

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

**}**

**}**

**// Step 3: Abstract Factory-IF ANY DOCUMENT EXTENDS IT YOU CAN ACCOMODATE IN FUTURE**

**abstract class DocumentFactory {**

**public abstract Document createDocument();**

**}**

**// Step 4: Concrete Factories-PREPARES INTERMEIDATE**

**class WordDocumentFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new WordDocument();**

**}**

**}**

**class PdfDocumentFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new PdfDocument();**

**}**

**}**

**class ExcelDocumentFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new ExcelDocument();**

**}**

**}**

**// Step 5: Test Class-CLIENT**

**public class FactoryMethodPatternExample {**

**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();**

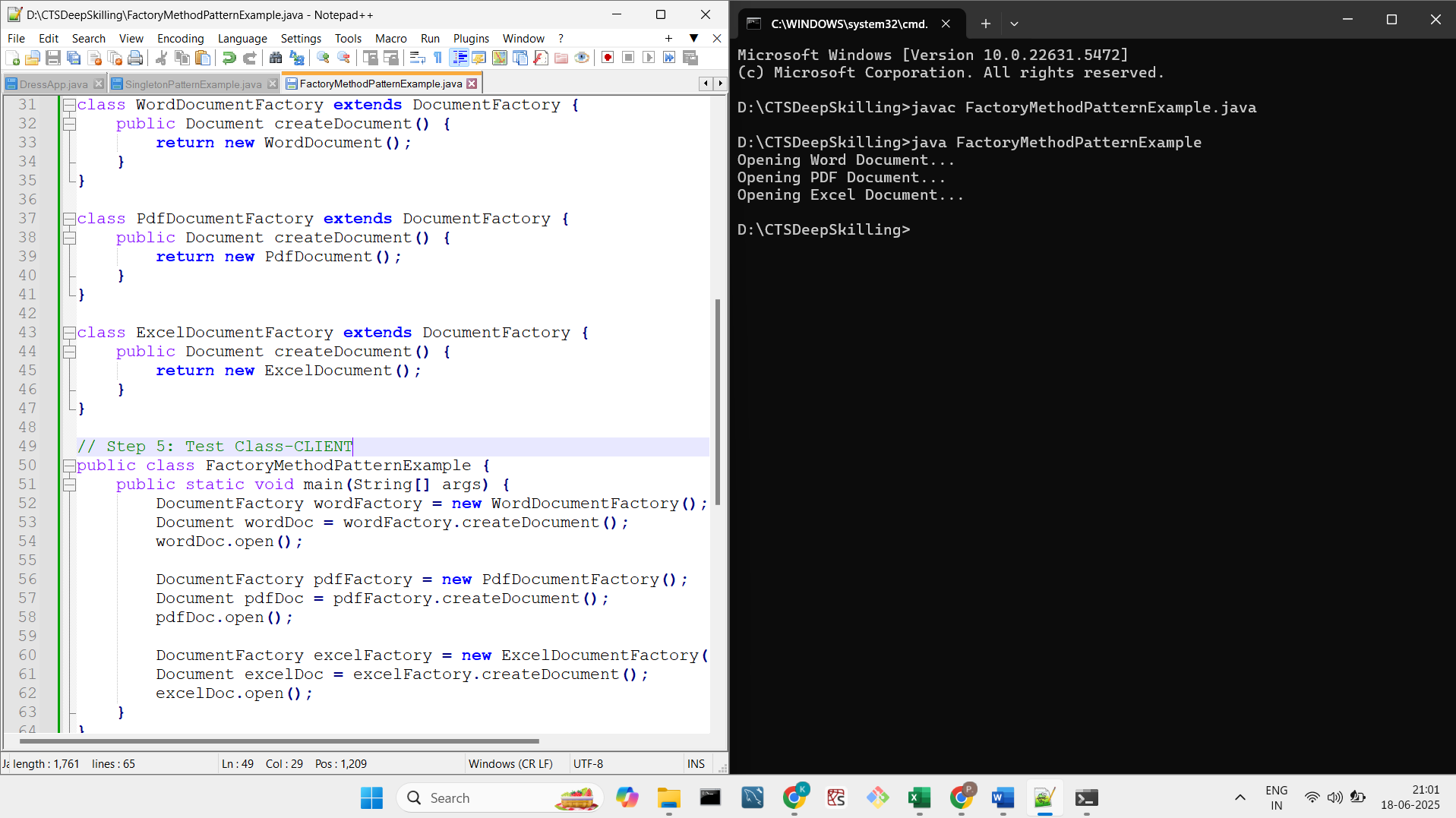
**Document excelDoc = excelFactory.createDocument();**

**excelDoc.open();**

    }

}

**OUTPUT:**

****

**TASK 3: E-COMMERCE PLATFORM SEARCH APPLICATION**

**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.**

**//CODE**

**import java.util.\*;**

**import java.lang.\*;**

**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;**

**}**

**}**

**public class Main {**

**public static void main(String[] args) {**

**List<Product> p=new ArrayList<>();**

**p.add(new Product(1,"HpLaptop","Laptop"));**

**p.add(new Product(2, "DellLaptop", "Laptop"));**

**p.add(new Product(3, "iPhone 14", "Mobile"));**

**p.add(new Product(4, "Samsung Galaxy", "Mobile"));**

**p.add(new Product(5, "Sony Bravia TV", "Electronics"));**

**p.add(new Product(6, "LG Refrigerator", "Appliances"));**

**p.add(new Product(7, "Whirlpool Washing Machine", "Appliances"));**

**p.add(new Product(8, "Redmi Note 12", "Mobile"));**

**p.add(new Product(9, "Apple Watch", "Wearables"));**

**p.add(new Product(10, "Boat Earbuds", "Audio"));**

**p.add(new Product(11, "Asus Monitor", "Electronics"));**

**p.add(new Product(12, "Lenovo Keyboard", "Accessories"));**

**p.add(new Product(13, "Logitech Mouse", "Accessories"));**

**p.add(new Product(14, "Amazon Echo", "Smart Devices"));**

**p.add(new Product(15, "Philips Trimmer", "Personal Care"));**

**p.add(new Product(16, "Canon Printer", "Office"));**

**p.add(new Product(17, "Realme Tablet", "Tablet"));**

**p.add(new Product(18, "MI Air Purifier", "Home Appliances"));**

**p.add(new Product(19, "Nike Shoes", "Footwear"));**

**p.add(new Product(20, "Puma Backpack", "Bags"));**

**System.out.println("Welcome USER");**

**System.out.println("Enter the product ID You want to search between 1 and 20");**

**Scanner in=new Scanner(System.in);**

**int userProduct=in.nextInt();**

**//LINEAR SEARCH**

**System.out.println("LINEAR SEARCH");**

**for(int i=0;i<p.size();i++)**

**{**

**if(userProduct==p.get(i).productId)**

**{**

**System.out.println("Your product found");**

**System.out.println("PRODUCT DETAILS");**

**System.out.println(p.get(i).productId);**

**System.out.println(p.get(i).productName);**

**System.out.println(p.get(i).category);**

**}**

**}**

**//BINARY SEARCH**

**System.out.println("BINARY SEARCH");**

**int n=p.size();**

**int s=0;**

**int e=n-1;**

**boolean f=false;**

**while(s<=e)**

**{**

**int m=(s+e)/2;**

**if(userProduct==p.get(m).productId)**

**{**

**System.out.println("Your product found");**

**System.out.println("PRODUCT DETAILS");**

**System.out.println(p.get(m).productId);**

**System.out.println(p.get(m).productName);**

**System.out.println(p.get(m).category);**

**f=true;**

**break;**

**}**

**else if(userProduct>p.get(m).productId)**

**{**

**s=m+1;**

**}**

**else**

**{**

**e=m-1;**

**}**

**}**

**if(!f)**

**{**

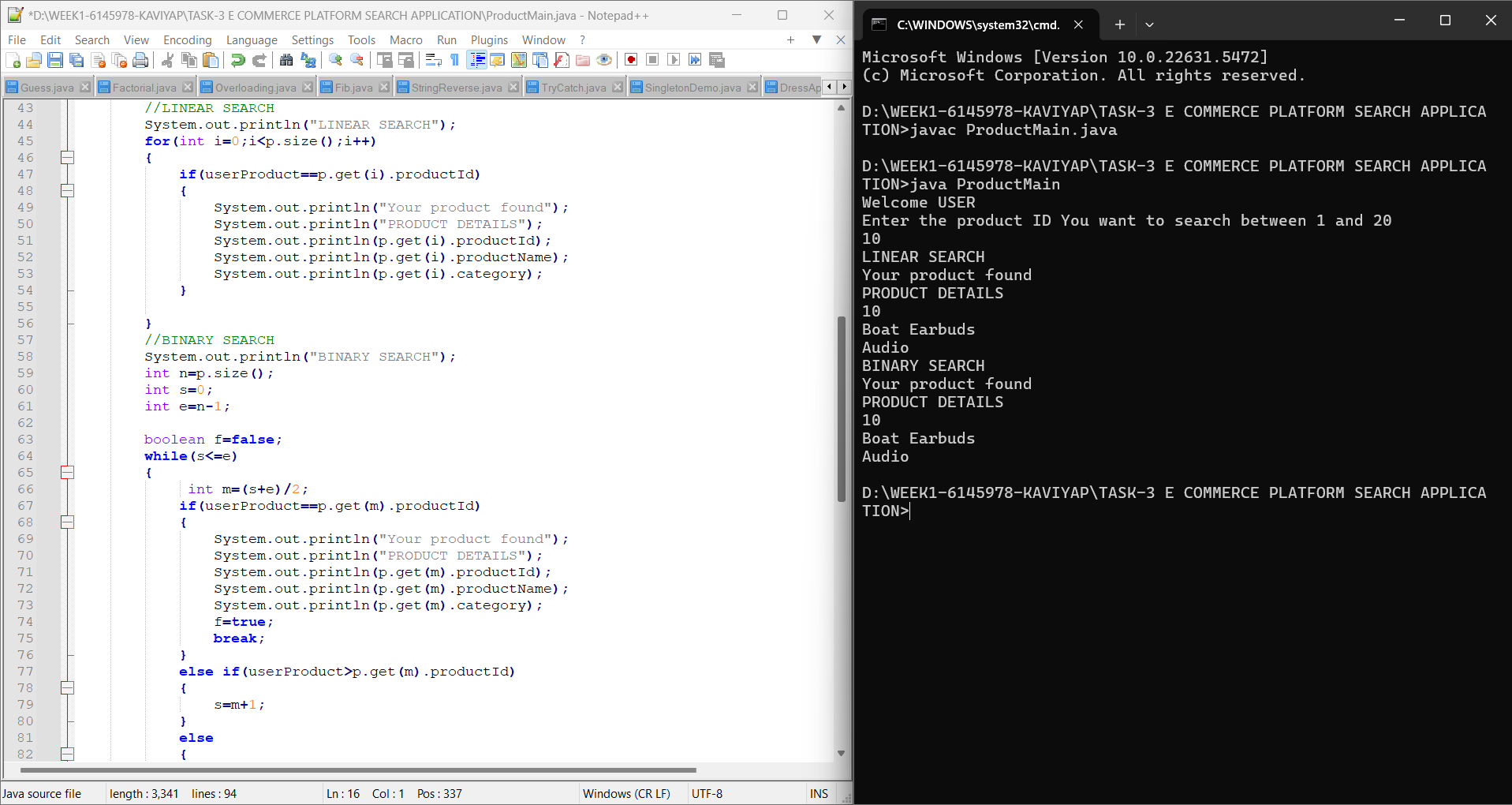
**System.out.println("No product");**

**}**

**}**

**}**

**OUTPUT:**

****

**1. Big O Notation**

* Describes the performance or complexity of an algorithm.
* Example: O(n) means time grows linearly with input size.

**2.TIME COMPLEXITY**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | **O(1)** | **O(n)** | **O(n)** |
| **Binary Search** | **O(1)** | **O(log n)** | **O(log n)** |

**3. WHICH IS EFFICIENT**

**Binary Search is more efficient but requires a sorted array/list. Because it takes less time.**

**TASK 4: FINANCIAL FORECASTING**

**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.**

**//CODE**

**import java.util.\*;**

**import java.lang.\*;**

**class FinancialForeCast {**

**public static double forecase(double current,double growthRate,int years)**

**{**

**if(years==0)**

**{**

**return current;**

**}**

**return forecase(current,growthRate,years-1)\*(1+growthRate);**

**}**

**public static void main(String[] args) {**

**System.out.println("Hi! LET'S PREDICT FURTURE FORECASE");**

**Scanner in=new Scanner(System.in);**

**System.out.println("Enter the starting investment");**

**double current=in.nextDouble();**

**System.out.println("Enter the growth rate by converting the percentage to number if 8% enter as 0.08");**

**double growthRate=in.nextDouble();**

**System.out.println("Enter the number of years from now");**

**int years=in.nextInt();**

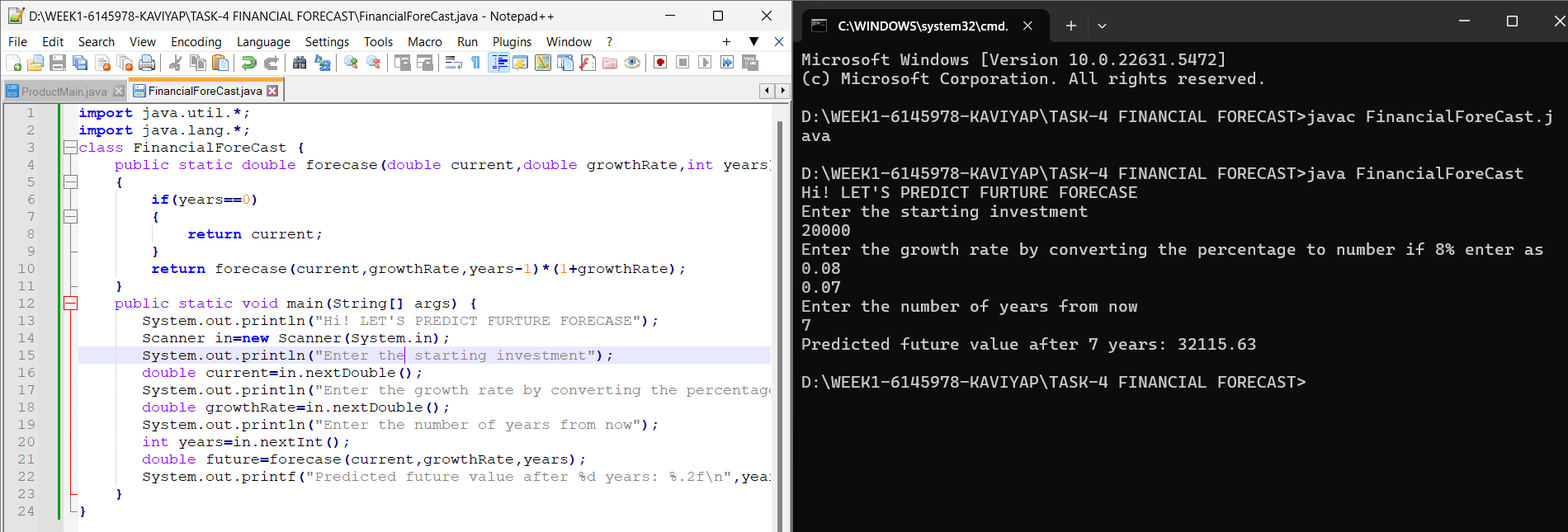
**double future=forecase(current,growthRate,years);**

**System.out.printf("Predicted future value after %d years: %.2f\n",years,future);**

**}**

**}**

**OUTPUT**

****

**1.TIME COMPLEXITY :** O(n)

**2.SPACE COMPLEXITY :** O(n)

**3.OPTMIZIED APPROACH :**

To use FOR LOOP

public static double forecase(int years, double current double growthRate)

{

    for (int i = 0; i < years; i++) {

        current \*= (1 + growthRate);

    }

    return current;

}

**TIME COMPLEXITY :** O(n)

**SPACE COMPLEXITY :** O(1)