**Candidate Name**:-  Uday R

**Superset ID**:-6417149

**WEEK – 1 HANDS ON EXERCISE (JAVA FSE DEEPSKILLING)**

**(DESIGN PATTERN AND PRINCIPLES)**

**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 LogManager that has a private static instance of itself.**
   * **Ensure the constructor of LogManager is private.**
   * **Provide a public static method to get the instance of the LogManager class.**
3. **Implement the Singleton Pattern:**
   * **Write code to ensure that the LogManager class follows the Singleton design pattern.**
4. **Test the Singleton Implementation:**
   * **Create a test class to verify that only one instance of LogManager is created and used across the application.**

**Code for the above question:-**

 public class LogManager {

private static LogManager instance;

private LogManager() {

// Private constructor to prevent instantiation

}

public static LogManager getInstance() {

if (instance == null) {

instance = new LogManager();

}

return instance;

}

public void logMessage(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

LogManager logManagerOne = LogManager.getInstance();

logManagerOne.logMessage("System initialization started.");

LogManager logManagerTwo = LogManager.getInstance();

logManagerTwo.logMessage("User login event recorded.");

if (logManagerOne == logManagerTwo) {

System.out.println("Both logManager references point to the same instance.");

} else {

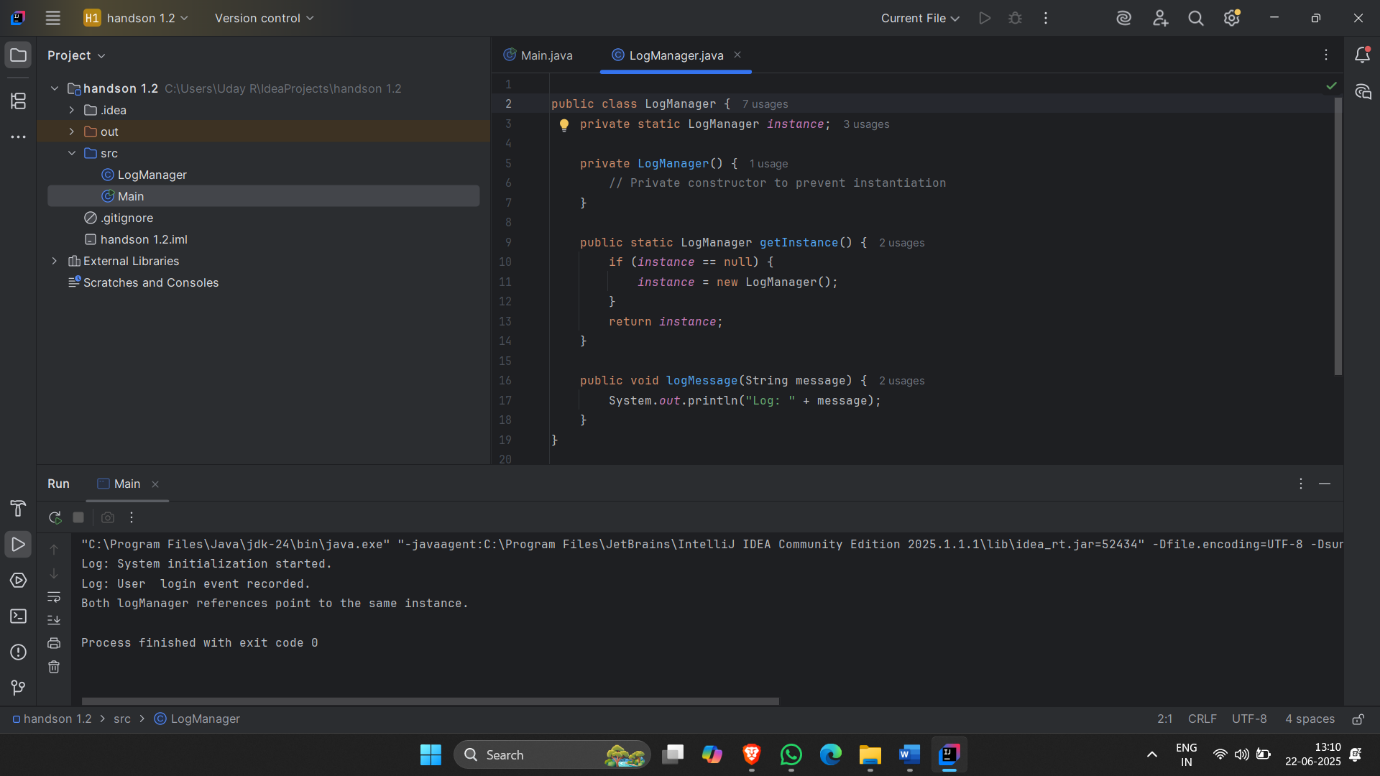
System.out.println("Different instances exist — Singleton failed.");

}

}

}

**Output Image:-**



**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 for the above question:-**

public class DocumentManager {

public static void main(String[] args) {

DocumentCreator textFactory = new TextDocumentFactory();

DocumentCreator portableFactory = new PortableDocumentFactory();

DocumentCreator spreadsheetFactory = new SpreadsheetDocumentFactory();

Document textDoc = textFactory.createDocument();

Document portableDoc = portableFactory.createDocument();

Document spreadsheetDoc = spreadsheetFactory.createDocument();

textDoc.open();

portableDoc.open();

spreadsheetDoc.open();

}

}

interface Document {

void open();

}

class TextFile implements Document {

public void open() {

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

}

}

class PortableFile implements Document {

public void open() {

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

}

}

class SpreadsheetFile implements Document {

public void open() {

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

}

}

abstract class DocumentCreator {

public abstract Document createDocument();

}

class TextDocumentFactory extends DocumentCreator {

public Document createDocument() {

return new TextFile();

}

}

class PortableDocumentFactory extends DocumentCreator {

public Document createDocument() {

return new PortableFile();

}

}

class SpreadsheetDocumentFactory extends DocumentCreator {

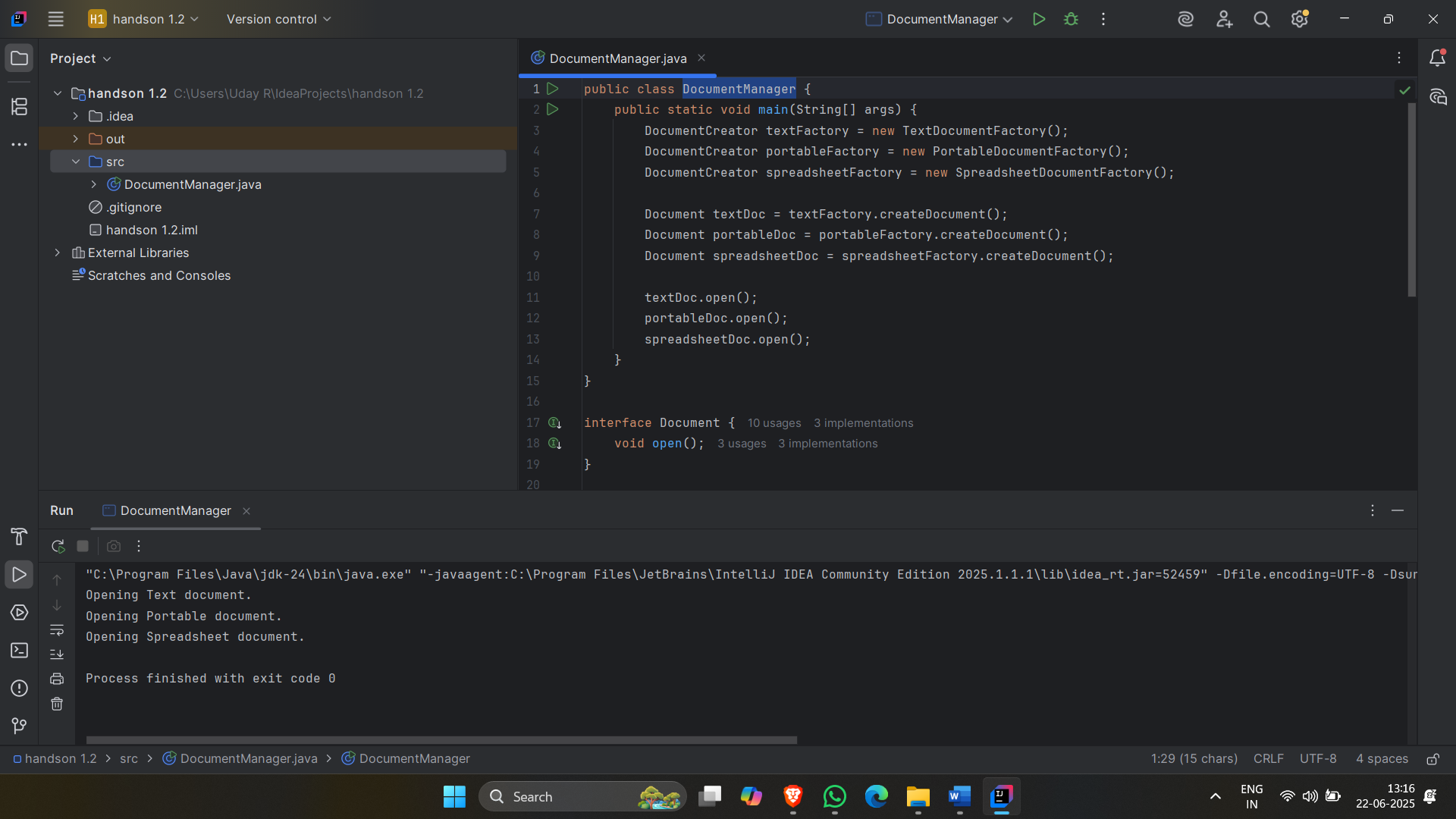
public Document createDocument() {

return new SpreadsheetFile();

}

}

**Output Image:-**



**(ALGORITHM DATA STRUCTURES)**

**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.**

**Code for above question:-**

import java.util.Scanner;

public class FinancialForecast {

public static double computeFutureValue(double initialAmount, double growthRate, int duration) {

if (duration == 0) return initialAmount;

return computeFutureValue(initialAmount, growthRate, duration - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

Scanner inputScanner = new Scanner(System.in);

System.out.print("Enter initial investment amount: ");

double investment = inputScanner.nextDouble();

System.out.print("Enter annual growth rate (e.g., 0.06 for 6%): ");

double growthRate = inputScanner.nextDouble();

System.out.print("Enter number of years for prediction: ");

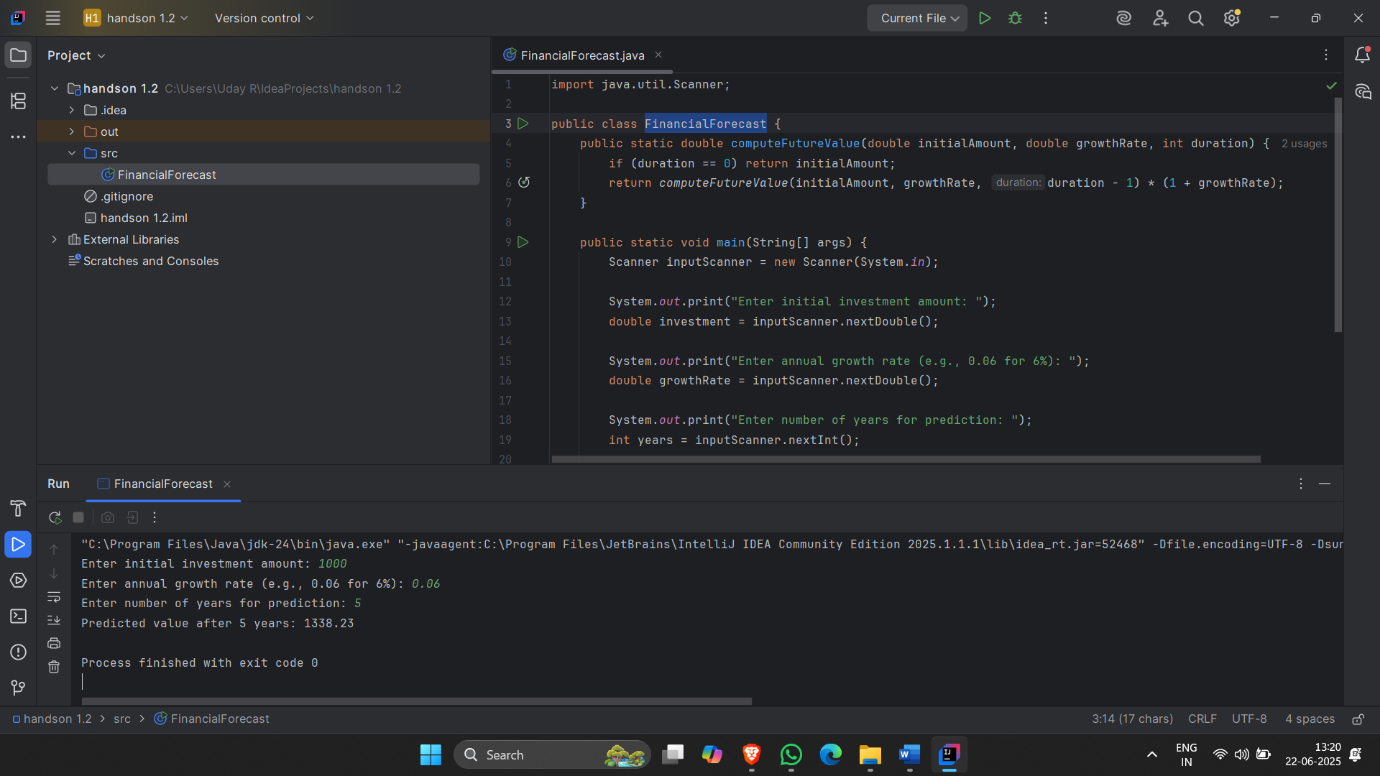
int years = inputScanner.nextInt();

double futureValue = computeFutureValue(investment, growthRate, years);

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

}

}**Output Image:-**



**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.

1. **Setup:**

* Create a class Product with attributes for searching, such as productId, productName, and category.

1. **Implementation:**

* Implement linear search and binary search algorithms.
* Store products in an array for linear search and a sorted array for binary search.

1. **Analysis:**

* Compare the time complexity of linear and binary search algorithms.
* Discuss which algorithm is more suitable for your platform and why.

**Code for above question:-**

    class Item {

int itemId;

String itemName;

public Item(int itemId, String itemName) {

this.itemId = itemId;

this.itemName = itemName;

}

@Override

public String toString() {

return "ID: " + itemId + ", Name: " + itemName;

}

}

public class BinarySearchExample {

public static void main(String[] args) {

Item[] items = {

new Item(1, "Apple"),

new Item(2, "Banana"),

new Item(3, "Cherry")

};

// Perform binary search for item with ID 2

Item foundItem = binarySearch(items, 2);

if (foundItem != null) {

System.out.println("Found item: " + foundItem);

} else {

System.out.println("Item not found.");

}

}

// Binary search method

public static Item binarySearch(Item[] items, int targetId) {

int start = 0;

int end = items.length - 1;

while (start <= end) {

int middle = (start + end) / 2;

if (items[middle].itemId == targetId) {

return items[middle];

} else if (items[middle].itemId < targetId) {

start = middle + 1;

} else {

end = middle - 1;

}

}

return null; // Not found

}

}

**Output Image:-**

