**Week-1 (Hands-on)**

**Date: 19-06-25**

**Design Patterns and Principles:**

**Exercise 1: Implementing the Singleton Pattern:**

**Logger.java**

**package** singleton;

**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);

}

}

**LoggerTest.java**

**package** singleton;

**public** **class** LoggerTest {

**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 logger instances are the same. Singleton works!");

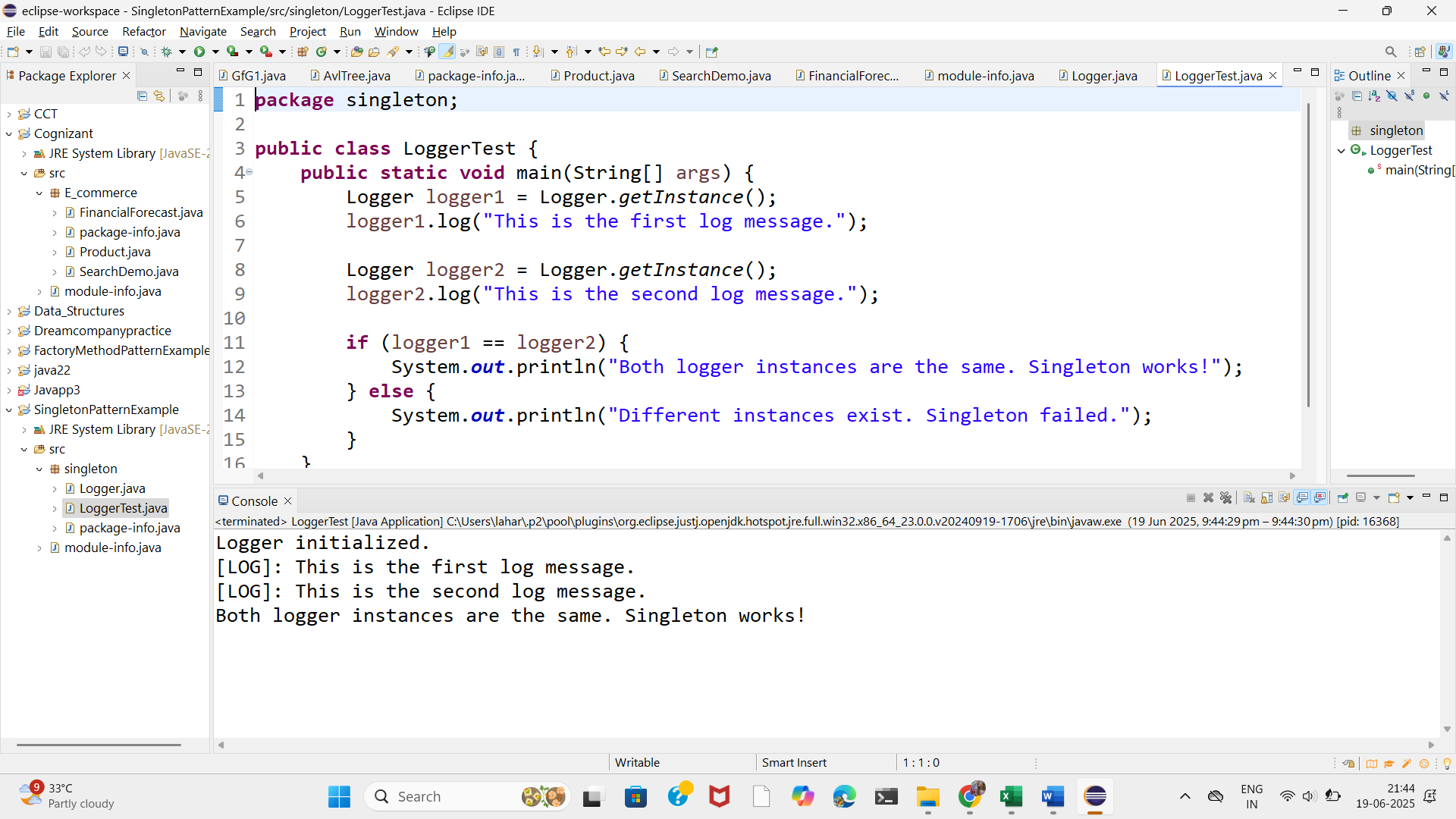
} **else** {

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

}

}}

**Output:**



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

**WordDocumentFactory.java**

**package** factory;

**public** **class** WordDocumentFactory **extends** DocumentFactory {

**public** Document createDocument() {

**return** **new** WordDocument();

}

}

**WordDocument:**

package factory;

public class WordDocument implements Document {

public void open() {

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

}

}

**PdfDocumentFactory.java**

package factory;

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**PdfDocument.java:**

package factory;

public class PdfDocument implements Document {

public void open() {

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

}

}

**ExcelDocumentFactory.java:**

package factory;

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**ExcelDocument.java:**

**package** factory;

**public** **class** ExcelDocument **implements** Document {

**public** **void** open() {

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

}

}

**DocumentFactory.java:**

package factory;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**Document.java:**

**package** factory;

**public** **interface** Document {

**void** open();

}

**DocumentFactoryTest.java:**

**package** factory;

**public** **class** DocumentFactoryTest {

**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();

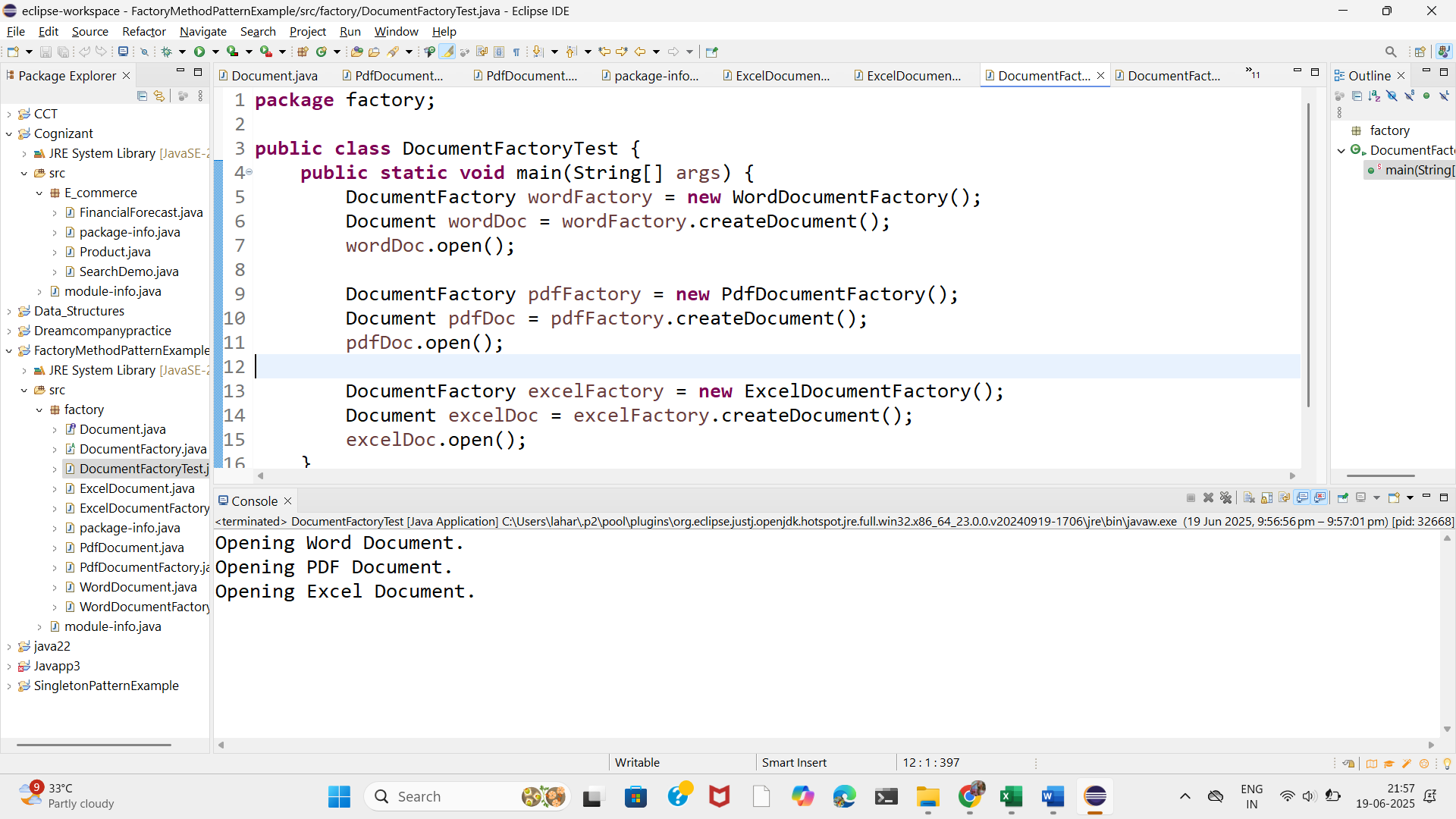
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**



**Algorithms\_Data Structures**

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

**Product.java:**

**1. Understand Asymptotic Notation**

**➤ Big O Notation:**

* Big O notation describes the **upper bound of time or space complexity** of an algorithm.
* It tells us how an algorithm performs as the **input size (n)** grows.
* **Example**:
  + O(1): Constant time — fastest
  + O(n): Linear time
  + O(log n): Logarithmic time — efficient for large data
  + O(n²): Quadratic time — inefficient for large n

**Best, Average, and Worst Case in Search:**

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

* **Best Case**: Found immediately
* **Average Case**: On average, in the middle
* **Worst Case**: Found at end (linear) or not found at all

**package** E\_commerce;

**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: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

**SearchDemo.java:**

**package** E\_commerce;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** SearchDemo {

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

**for** (Product product : products) {

**if** (product.productName.equalsIgnoreCase(targetName)) {

**return** product;

}

}

**return** **null**;

}

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

**int** left = 0;

**int** right = products.length - 1;

**while** (left <= right) {

**int** mid = (left + right) / 2;

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

**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, "Shampoo", "Personal Care"),

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

**new** Product(104, "Book", "Stationery"),

**new** Product(105, "Phone", "Electronics")

};

System.***out***.println("Linear Search:");

Product result1 = *linearSearch*(products, "Watch");

System.***out***.println(result1 != **null** ? result1 : "Product not found");

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

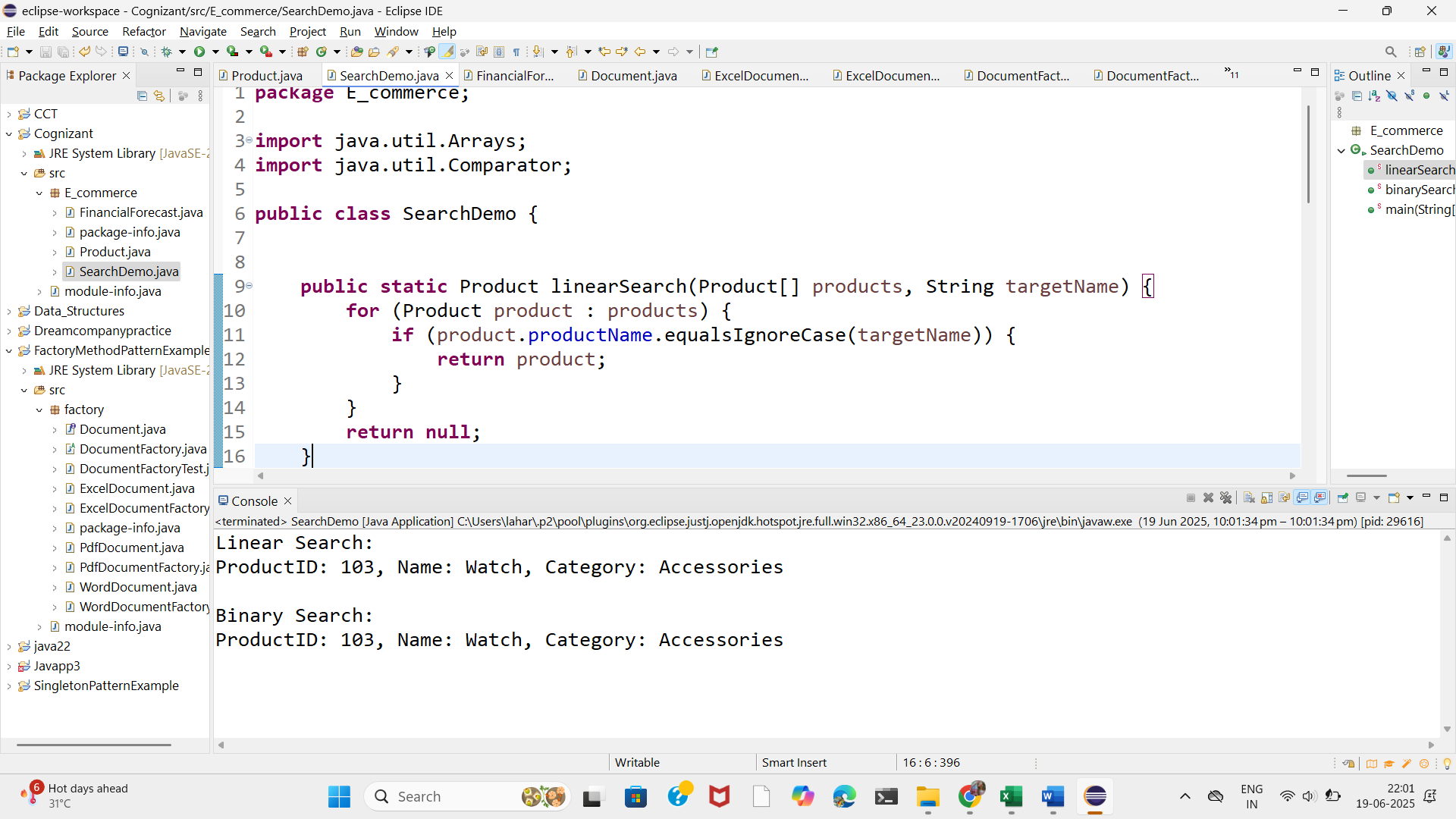
System.***out***.println("\nBinary Search:");

Product result2 = *binarySearch*(products, "Watch");

System.***out***.println(result2 != **null** ? result2 : "Product not found");

}

}



**Binary Search** is better for **large datasets** and frequent searches — it's much faster (logarithmic time).

But it requires the data to be **sorted first**, which can take **O(n log n)**.

**Linear Search** is better for **small datasets** or **one-time searches** with **unsorted data**.

**Real-world Recommendation:**

For an e-commerce platform with thousands of products:

* Use **binary search** (with preprocessing sort or maintain sorted data structures).
* Or better, use **hash maps or indexed databases** for real-time fast search (beyond the scope here but worth mentioning).

**Exercise 7: Financial Forecasting:**

Recursion is when a method calls itself to solve a problem by breaking it into smaller subproblems.

FV(n) = FV(n-1) \* (1 + r)

**Time Complexity:**

* The recursive function calls itself n times, where n = years.
* Each call does **constant work** (one multiplication and return).
* Therefore, the time complexity is:
* T(n) = T(n-1) + O(1)
* => O(n)

**Space Complexity:**

* Due to recursive call stack, the space used is also **O(n)**.

**Problem:**

* Although this specific recursion isn't very inefficient, **repeated calls** with the same parameters in **other problems** can cause exponential time.

**Solution:**

* Use a **memoization array** (top-down dynamic programming).
* Store the result of each year in a memo[] array.
* If the value already exists in the memo, reuse it instead of recalculating.

**FinancialForecast.java:**

**package** E\_commerce;

**public** **class** FinancialForecast {

**public** **static** **double** futureValueRecursive(**double** presentValue, **double** rate, **int** years) {

**if** (years == 0) {

**return** presentValue;

}

**return** *futureValueRecursive*(presentValue, rate, years - 1) \* (1 + rate);

}

**public** **static** **double** futureValueMemo(**double** presentValue, **double** rate, **int** years, **double**[] memo) {

**if** (years == 0) {

**return** presentValue;

}

**if** (memo[years] != 0) {

**return** memo[years];

}

memo[years] = *futureValueMemo*(presentValue, rate, years - 1, memo) \* (1 + rate);

**return** memo[years];

}

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

**double** presentValue = 10000;

**double** rate = 0.05;

**int** years = 10;

**double** recursiveResult = *futureValueRecursive*(presentValue, rate, years);

System.***out***.println("Recursive Future Value: " + recursiveResult);

**double**[] memo = **new** **double**[years + 1];

**double** memoResult = *futureValueMemo*(presentValue, rate, years, memo);

System.***out***.println("Memoized Future Value: " + memoResult);

}

}