**WEEK 1**

**MODULE 1 - DESIGN PATTERNS 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.

**APPROACH**

* Create a class Logger with a private static variable to hold single instance of the class.
* Make the constructor private so that no other class can instantiate Logger directly using new.
* Provide a public static method getInstance() that returns the single instance; if it doesn’t exist yet, create it.
* Add a method like log(String message) in Logger class to simulate logging behaviour.
* In a separate test class, call Logger.getInstance() multiple times and verify that all returned instances are the same (using ==).

**CODE**

**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);

}}

**Main.java**

package singleton;

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First log message.");

logger2.log("Second log message.");

if (logger1 == logger2) {

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

}

else {

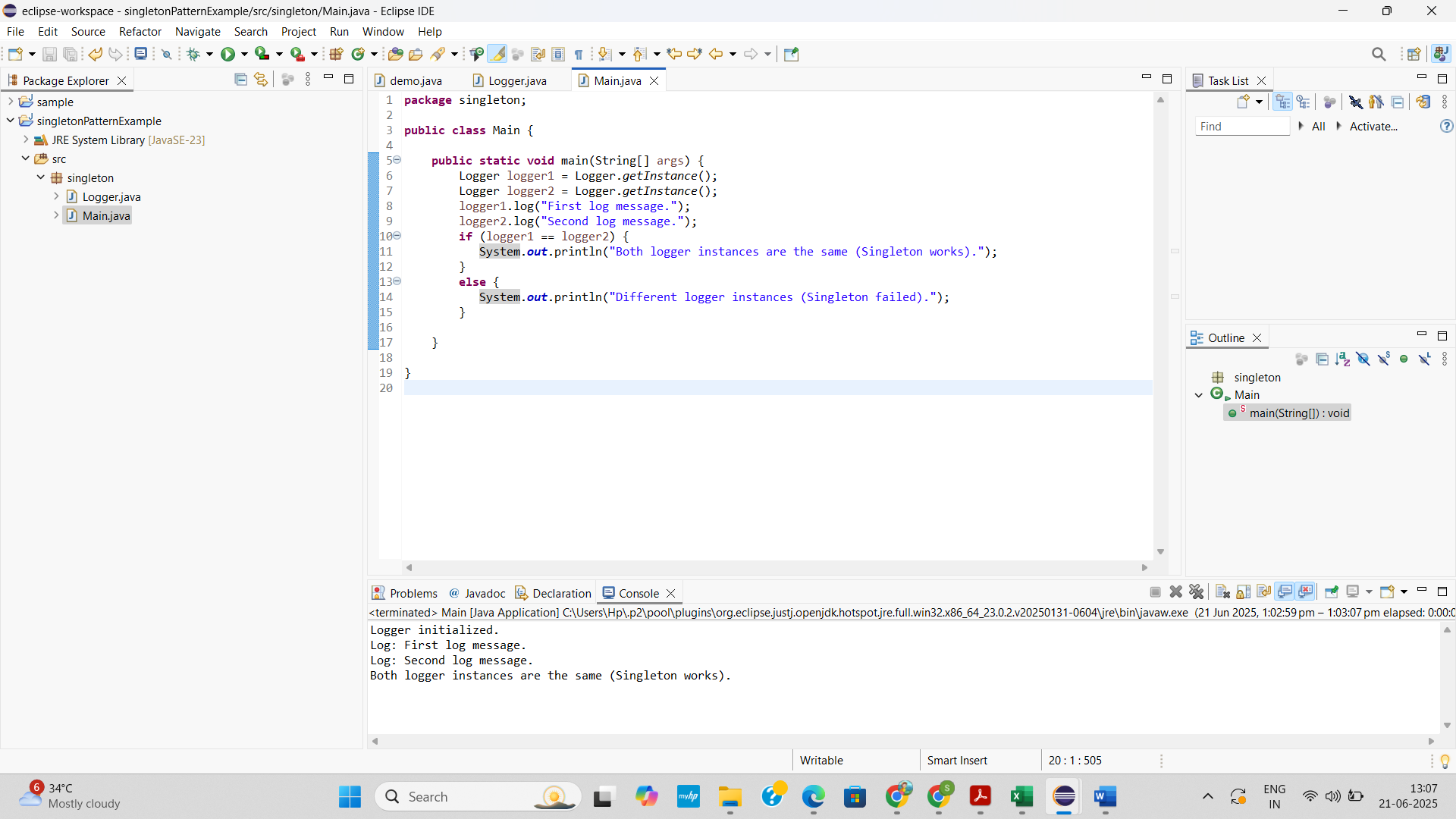
System.out.println("Different logger instances (Singleton failed).");

}

}

}

**OUTPUT**

****

**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.

**APPROACH**

* Create a Document interface or abstract class that defines a common method like open() for all document types.
* Implement concrete classes like WordDocument, PdfDocument, and ExcelDocument that implement or extend the Document interface/class.
* Create an abstract class DocumentFactory with an abstract method createDocument() to define the factory method.
* Implement concrete factory classes such as WordFactory, PdfFactory, and ExcelFactory that extend DocumentFactory and override createDocument() to return their respective document types.
* Create a test class (Main) that uses the factory classes to create and open different documents, demonstrating the Factory Method in action.

**CODE**

**Document.java**

package factory;

public interface Document {

void open();

}

**WordDocument.java**

package factory;

public class WordDocument implements Document{

public void open() {

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

}

}

**PdfDocument.java**

package factory;

public class PdfDocument implements Document {

public void open() {

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

}

}

**ExcelDocument.java**

package factory;

public class ExcelDocument implements Document{

public void open() {

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

}

}

**DocumentFactory.java**

package factory;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordFactory.java**

package factory;

public class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfFactory.java**

package factory;

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelFactory.java**

package factory;

public class ExcelFactory extends DocumentFactory{

public Document createDocument() {

return new ExcelDocument();

}

}

**Main.java**

package factory;

public class Main {

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();

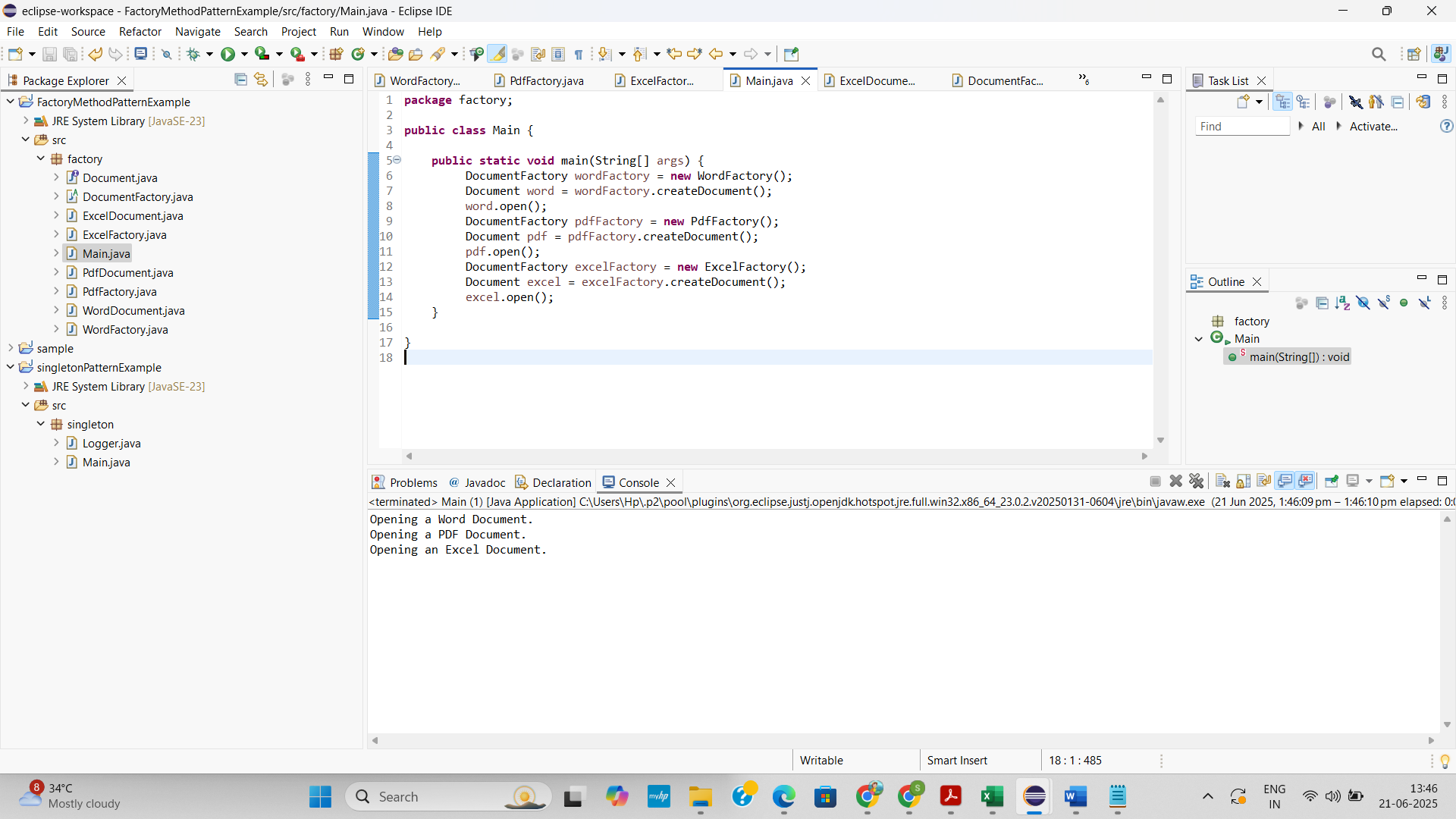
Document excel = excelFactory.createDocument();

excel.open();

}

}

**OUTPUT**



**MODULE 2-DATA STRUCTURES AND ALGORITHMS**

**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.

**APPROACH**

* Create a Product class with productId, productName, and category.
* Implement linearSearch and binarySearch methods in a SearchService class.
* Store products in an array, and sort it before applying binary search.
* Use a Main class to test both search methods with sample product data.
* Compare performance: use linear search for unsorted data, binary search for sorted data (faster: O(log n)).

**UNDERSTAND ASYMPTOTIC NOTATION**

* Big O notation describes the performance (time or space) of an algorithm as input size grows.
* For search operations:
  + Best Case: Few or one comparison (e.g., first item match).
  + Average Case: Match in the middle.
  + Worst Case: Match at the end or not found at all.
* Helps predict scalability and performance.

**SETUP PRODUCT CLASS**

**CODE**

**Product.java**

package search;

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;

}

public String toString() {

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

}

}

**SearchService.java**

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

for (Product p : products) {

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

return p;

}

}

return null;

}

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

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

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

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

else if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

**Main.java**

package search;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Shampoo", "Personal Care"),

new Product(103, "T-Shirt", "Clothing"),

new Product(104, "Mobile", "Electronics"),

new Product(105, "Notebook", "Stationery")

};

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

Product result1 = SearchService.linearSearch(products, "Mobile");

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

System.out.println("\nBinary Search for 'T-Shirt':");

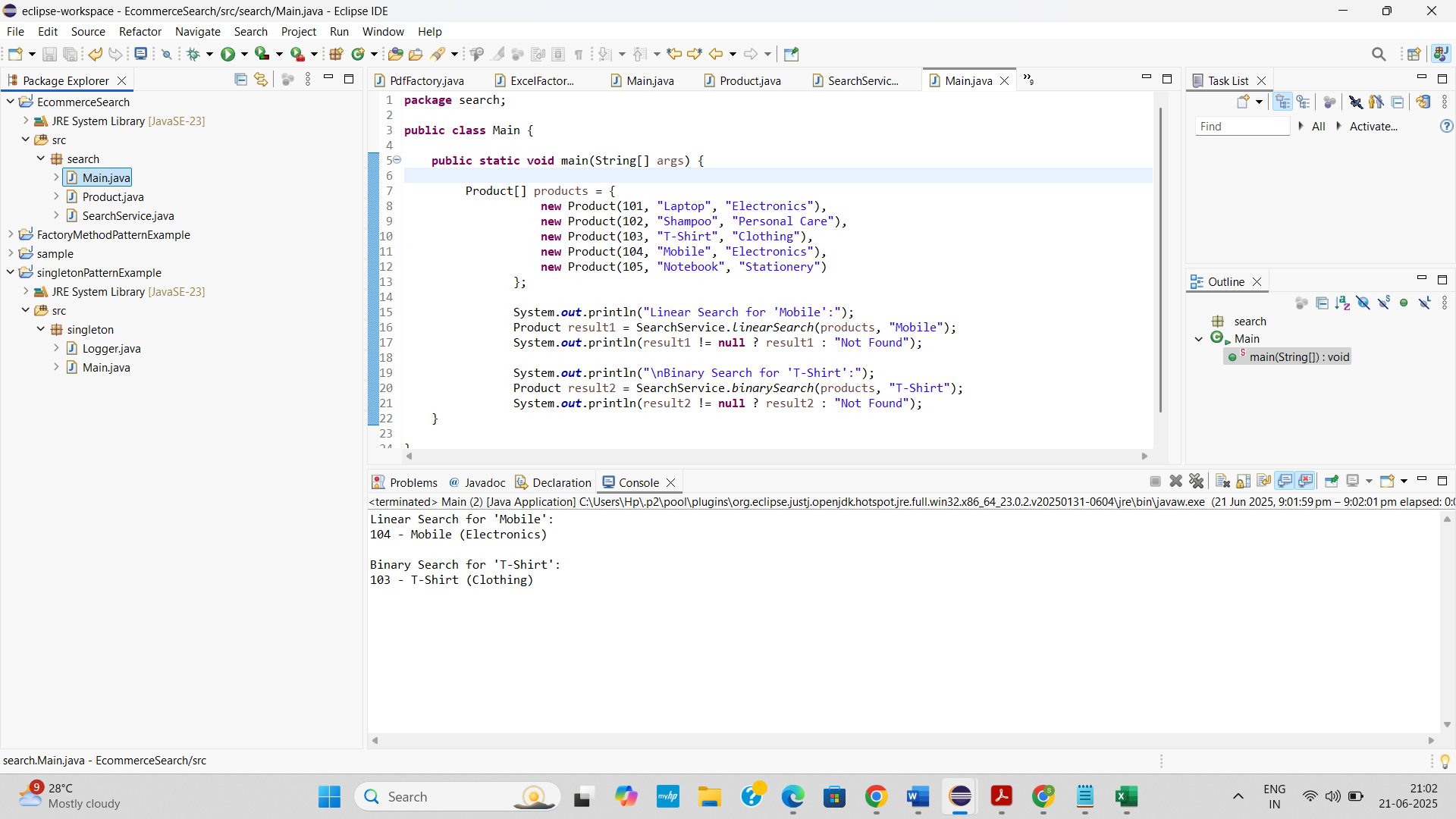
Product result2 = SearchService.binarySearch(products, "T-Shirt");

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

}

}

**OUTPUT**



**ANALYSIS OF TIME COMPLEXITIES**

|  |  |  |  |
| --- | --- | --- | --- |
| **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) |

* Linear search works on unsorted data but is slower for large datasets.
* Binary search is faster but needs sorted data.
* The best algorithm for this scenario is Binary Search because it offers faster performance (O(log n)) on large, sorted product lists, which is ideal for e-commerce platforms that require quick and frequent searches. It's efficient when the data is static or rarely changes, ensuring minimal overhead compared to linear search.

**EXERCISE 7: FINANCIAL FORECASTING**

**SCENARIO**

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

**APPROACH**

* Model the forecasting problem so that each future value depends on the previous year's value using a fixed growth rate.
* Implement a recursive method that calculates the future value by calling itself for the previous year.
* Define a base case where the recursion stops, typically when the number of years is zero, returning the initial value.
* In the recursive step, reduce the year by one and multiply the result by (1 + growthRate) to compute the current year's value.
* Analyze the time complexity (O(n)) and consider optimization techniques like memoization or iteration for better performance.

**UNDERSTANDING RECURSION**

Recursion is a programming technique where a method calls itself to solve a smaller instance of the same problem. It continues until it reaches a base case—a condition where the recursion stops.

In financial forecasting, recursion is particularly useful when each year’s value depends on the previous year’s value. For example, predicting future value based on compound interest or annual growth involves repeatedly applying a growth rate, which naturally fits a recursive pattern.

**CODE**

**ForecastCalculator.java**

package forecast;

public class ForecastCalculator {

public static double predictFutureValue(int years, double initialValue, double growthRate) {

if (years == 0) {

return initialValue;

}

return predictFutureValue(years - 1, initialValue, growthRate) \* (1 + growthRate);

}

}

**Main.java**

package forecast;

public class Main {

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 0.10;

int years = 5;

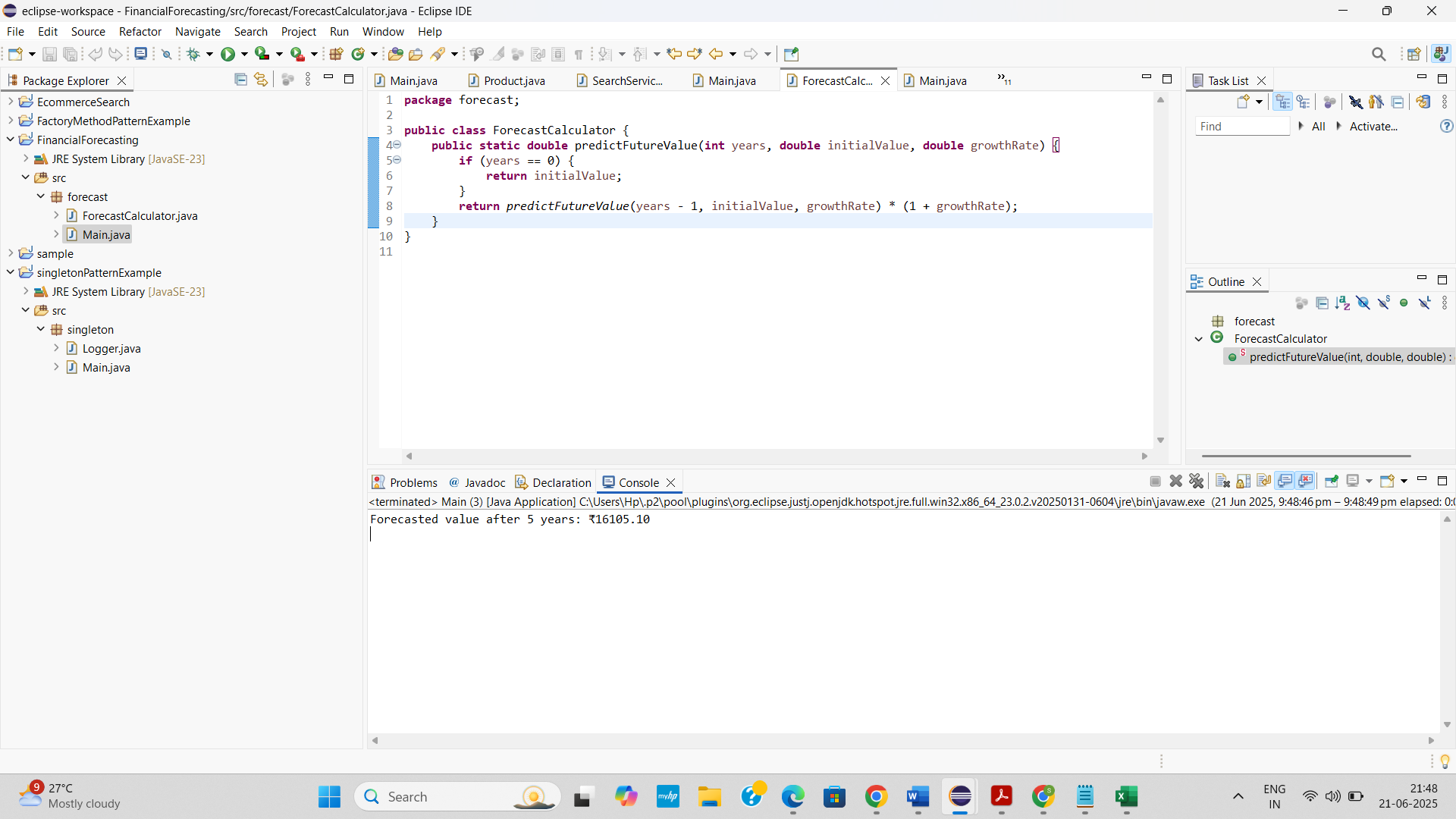
double futureValue = ForecastCalculator.predictFutureValue(years, initialValue, growthRate);

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

}

}

**OUTPUT**

****

**ANALYSIS OF TIME COMPLEXITY**

* Time Complexity: O(n) — one recursive call per year.
* Recursive depth increases with years, which may lead to performance issues or stack overflow for large inputs.
* To optimize recursion, use **memoization** by storing computed results in a map (e.g., Map<Integer, Double>), allowing the function to return precomputed values for repeated inputs instead of recalculating them, thus reducing redundant calls and improving performance.