**WEEK 1:**

**Exercise 1: Implementing the Singleton Pattern**

* **Code:**

1. **Logger.java**

public class Logger {

private static Logger singleInstance;

private Logger() {

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

}

public static Logger getInstance() {

if (singleInstance == null) {

singleInstance = new Logger();

}

return singleInstance;

}

public void log(String message) {

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

}

}

1. **Main.java**

public class Logger {

private static Logger singleInstance;

private Logger() {

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

}

public static Logger getInstance() {

if (singleInstance == null) {

singleInstance = new Logger();

}

return singleInstance;

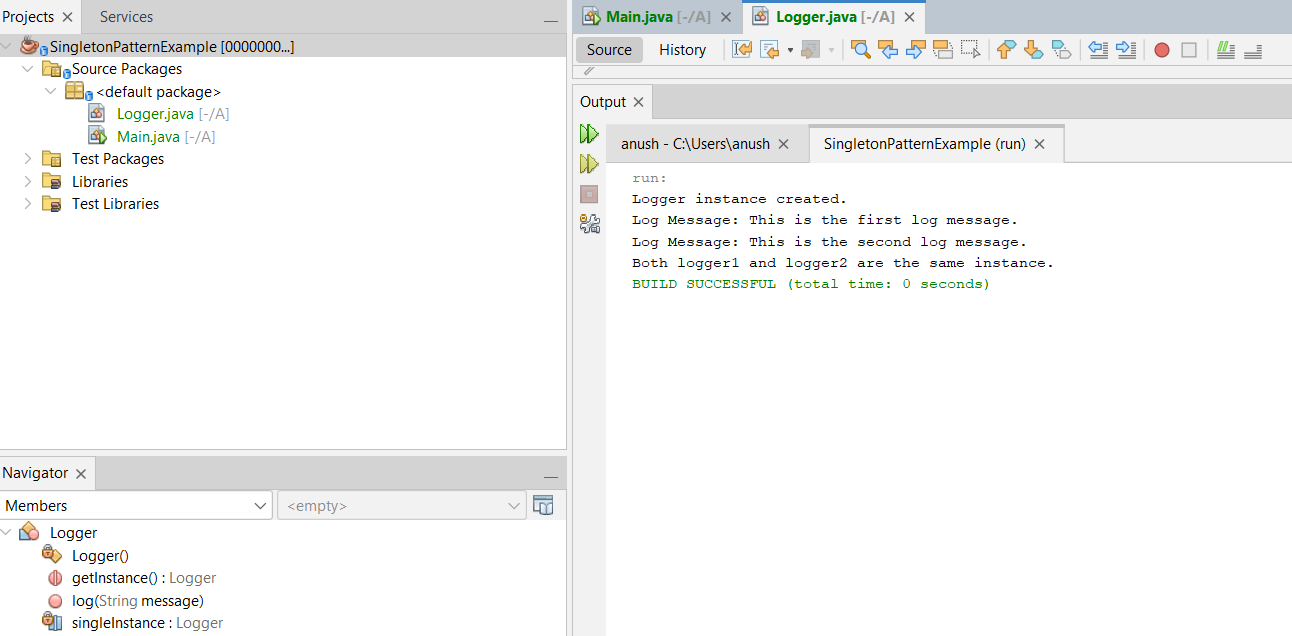
}

public void log(String message) {

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

}

}

* **Output:**  
    
  

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

* **Code:**

1. **Document.java interface**

public interface Document {

void open();

}

1. **WordDocument.java**

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

1. **PdfDocument.java**

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

1. **ExcelDocument.java**

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

1. **DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

}

1. **WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

1. **PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

1. **ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

1. **Main.java**

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

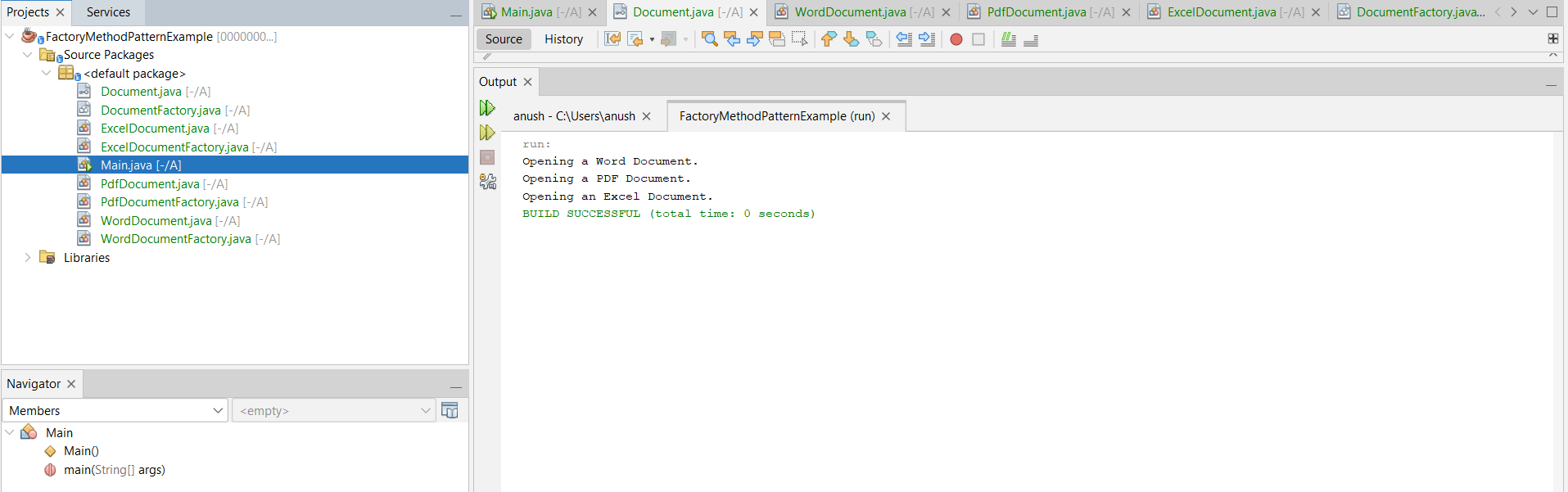
Document excel = excelFactory.createDocument();

excel.open();

}

}

* **Output:**

****

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

* **Big O** Notation is used to describe the time complexity or efficiency of an algorithm as the input size grows.

It tells us how fast an algorithm runs in terms of input size n.

| **Case** | **Meaning** |
| --- | --- |
| **Best Case** | Fastest execution time for ideal input |
| **Average Case** | Typical performance across various input types |
| **Worst Case** | Slowest performance, often used in Big O analysis |
|  |  |

* **Search Operation Complexities**

| **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) |

* **Code:**

1. **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;

}

public String toString() {

return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

1. **SearchUtility.java**

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtility {

// Linear Search by product name

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

for (Product product : products) {

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

return product;

}

}

return null;

}

// Binary Search by product name (array must be sorted)

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

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

}

1. **Main.java**

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Footwear"),

new Product(3, "Watch", "Accessories"),

new Product(4, "T-shirt", "Clothing"),

new Product(5, "Mobile", "Electronics")

};

String searchName = "Watch";

// Linear Search

long startTime = System.nanoTime();

Product resultLinear = SearchUtility.linearSearch(products, searchName);

long endTime = System.nanoTime();

System.out.println("Linear Search Result: " + resultLinear);

System.out.println("Linear Search Time: " + (endTime - startTime) + " ns");

// Binary Search

startTime = System.nanoTime();

Product resultBinary = SearchUtility.binarySearch(products, searchName);

endTime = System.nanoTime();

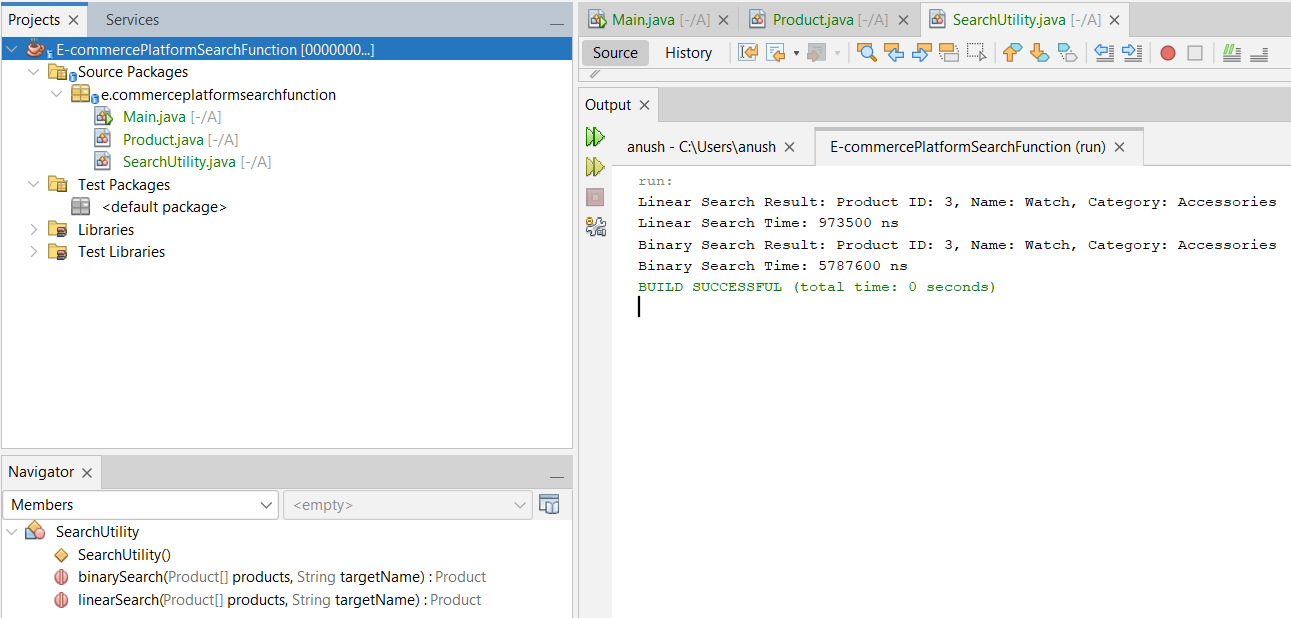
System.out.println("Binary Search Result: " + resultBinary);

System.out.println("Binary Search Time: " + (endTime - startTime) + " ns");

}

}

**Output:**



**Exercise 7: Financial Forecasting**

* Recursion is a technique where a method calls itself to solve smaller instances of a problem.
* Example:

To compute future value based on compound interest:

FutureValue(n)=FutureValue(n−1)×(1+growthRate)\text{FutureValue}(n) = \text{FutureValue}(n - 1) \times (1 + \text{growthRate})FutureValue(n)=FutureValue(n−1)×(1+growthRate)

Where:

* n = number of years
* growthRate = expected annual growth rate
* Base case: when n == 0, return initialValue
* **Code:**

**FinancialForecast.java**

public class FinancialForecast {

// Recursive method to calculate future value

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

// Base case

if (years == 0) {

return initialValue;

}

// Recursive step

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

}

public static void main(String[] args) {

double initialValue = 10000; // Initial investment

double growthRate = 0.08; // 8% growth per year

int years = 5;

double futureValue = calculateFutureValue(initialValue, growthRate, years);

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

}

}

* **Output:**

