JAVA FSE ASSIGNMENT:

WEEK 01: Design principles & Patterns

Exercise 1: Implementing the Singleton Pattern

Project Name: SingletonPatternExample

Inside my project, I created two files:

* Logger.java
* Main.java (or TestLogger.java)

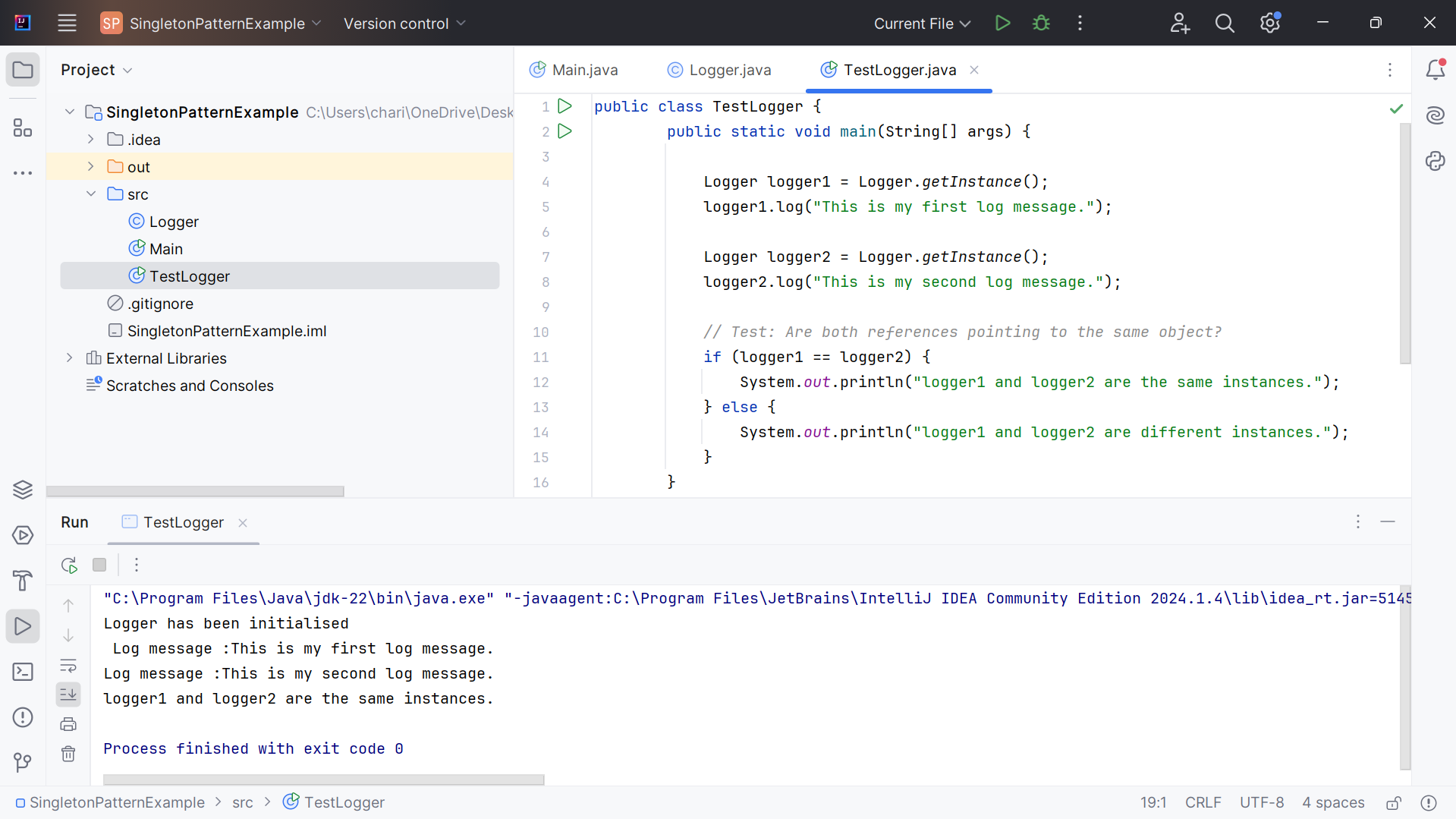
Logger.java

public class Logger {  
 private static Logger *instance*;  
 private Logger(){  
 System.*out*.print("Logger has been initialised\n ");  
 }  
  
 public static Logger getInstance(){  
 if(*instance*==null){  
 *instance*=new Logger();  
 }  
 return *instance*;  
 }  
  
 public void log(String message){  
 System.*out*.println("Log message :"+message);  
 }  
}

TestLogger.java

public class TestLogger {  
 public static void main(String[] args) {  
  
 Logger logger1 = Logger.*getInstance*();  
 logger1.log("This is my first log message.");  
  
 Logger logger2 = Logger.*getInstance*();  
 logger2.log("This is my second log message.");  
if (logger1 == logger2) {  
 System.*out*.println("logger1 and logger2 are the same instances.");  
 } else {  
 System.*out*.println("logger1 and logger2 are different instances.");  
 }  
 }  
 }

Output :



Exercise 2: Implementing the Factory Method Pattern

* **Project Name:** FactoryMethodPatternExample
* Created the following files:
  + Document.java (interface)
  + WordDocument.java, PdfDocument.java, ExcelDocument.java (concrete implementations)
  + DocumentFactory.java (abstract factory)
  + WordDocumentFactory.java, PdfDocumentFactory.java, ExcelDocumentFactory.java (concrete factories)
  + Main.java (test class)

Document.java

public interface Document {

void open();

}

WordDocument**.**java

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

PdfDocument.java

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

ExcelDocument.java

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

DocumentFactory.java

public abstract class DocumentFactory {

public abstract Document createDocument();

}

WordDocumentFactory.java

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

PdfDocumentFactory.java

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

ExcelDocumentFactory.java

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

Main.java

public class Main {

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();

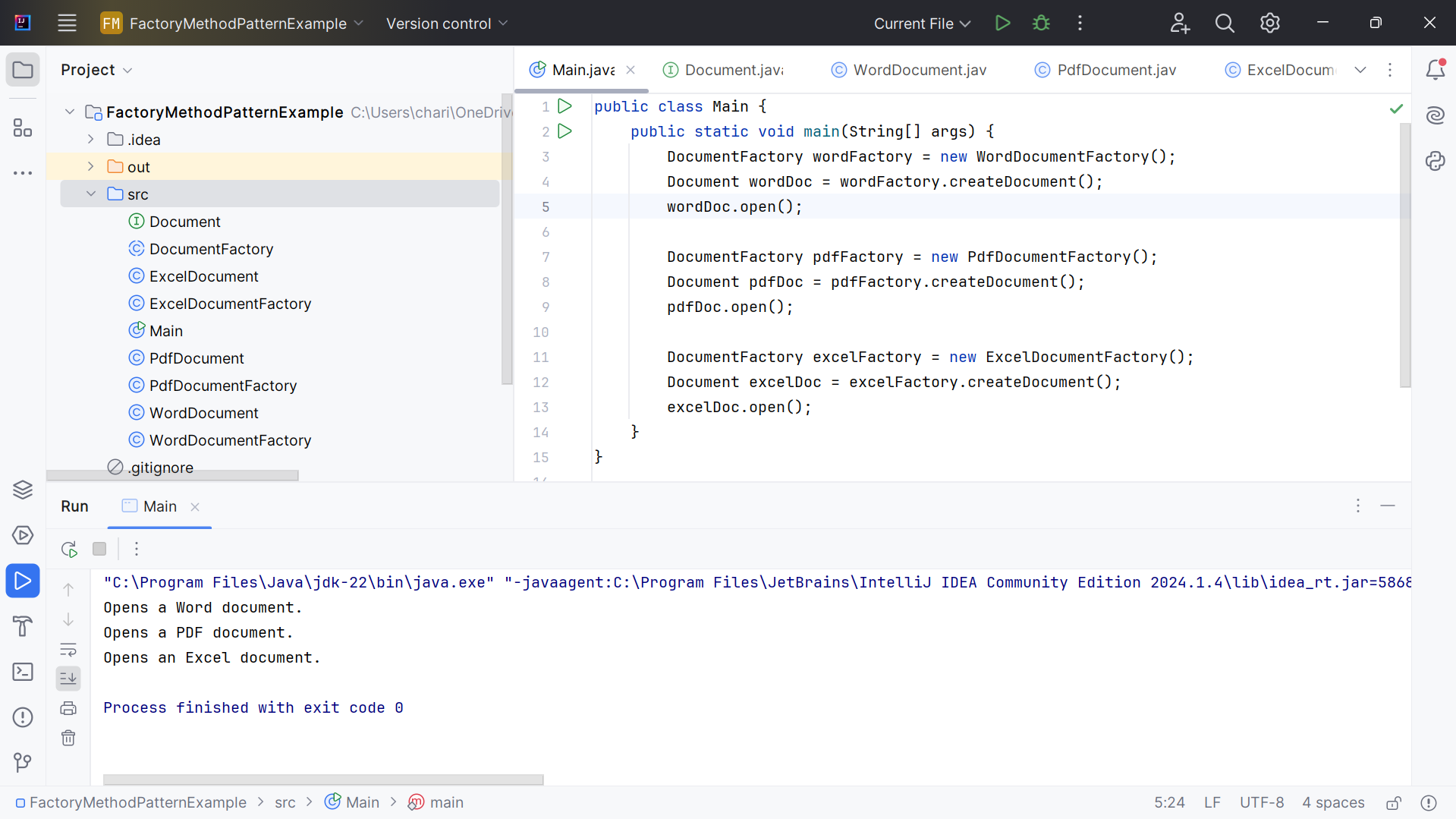
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

Output:



WEEK 01: Data structures and Algorithms

Exercise 2: E-commerce Platform Search Function

What is Big O Notation?

* Big O notation describes the upper bound of an algorithm's running time as the input size (n) grows.
* It abstractsawayconstantsand low-order terms, focusing only on how performance scales.
* Helps evaluate efficiency and scalability.
* Guides in choosing optimal algorithms for large data.

**🔹** Best, Average, and Worst Cases :

| Scenario | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Best Case | O(1) – first element match | O(1) – middle element match |
| Average Case | O(n/2) ≈ O(n) | O(log n) |
| Worst Case | O(n) – last or no match | O(log n) – item not found |

Product.java

public class Product {

private String productId;

private String productName;

private String category;

public Product(String productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

@Override

public String toString() {

return productId + " | " + productName + " | " + category;

}

}

SearchFunctions.java

import java.util.Arrays;

import java.util.Comparator;

public class SearchFunctions {

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

for (Product p : products) {

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

return p;

}

}

return null;

}

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

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0)

return products[mid];

else if (cmp < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

public static void sortByProductName(Product[] products) {

Arrays.sort(products, Comparator.comparing(Product::getProductName));

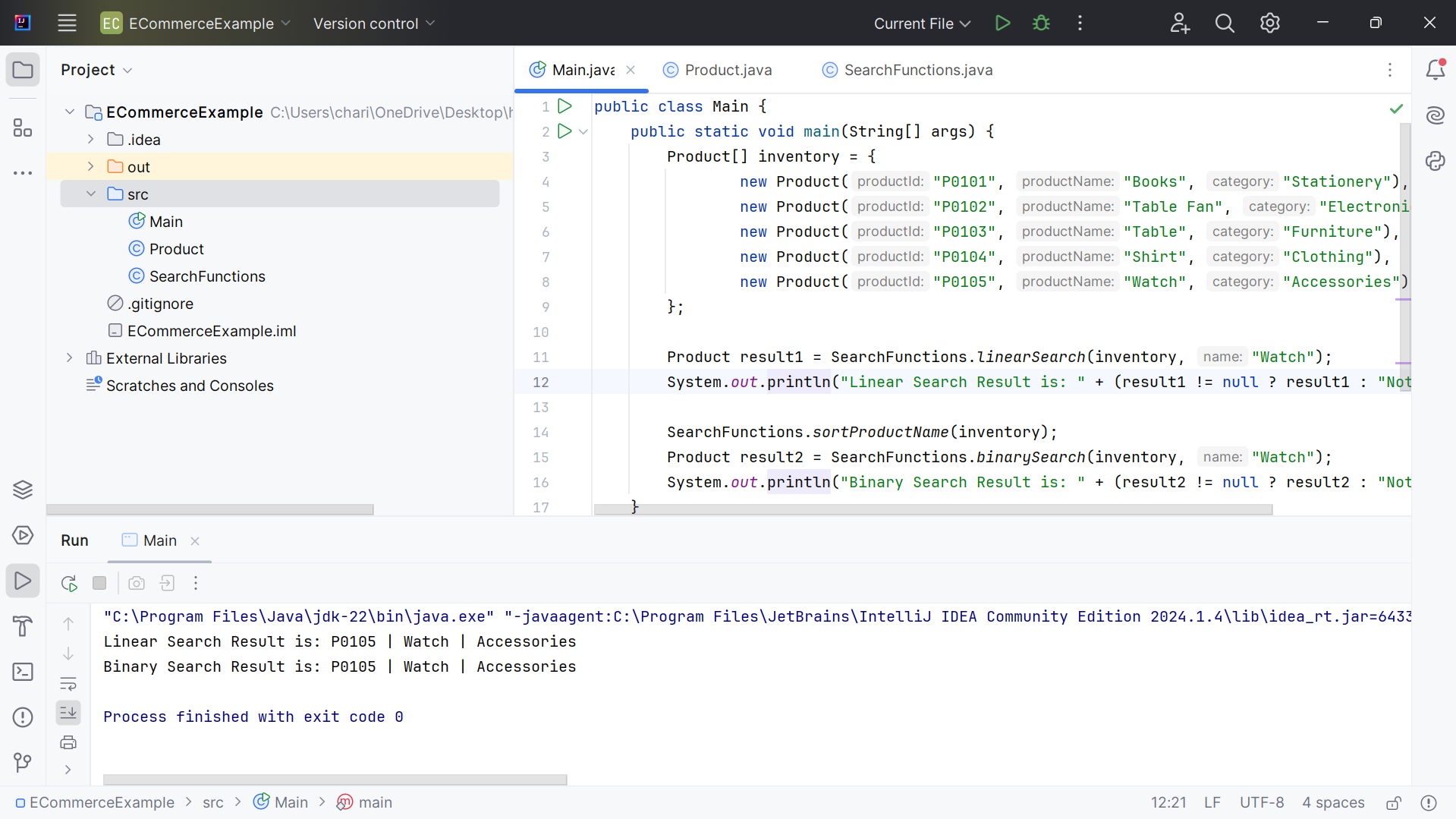
}

}

Main.java

public class Main {  
 public static void main(String[] args) {  
 Product[] inventory = {  
 new Product("P0101", "Books", "Stationery"),  
 new Product("P0102", "Table Fan", "Electronics"),  
 new Product("P0103", "Table", "Furniture"),  
 new Product("P0104", "Shirt", "Clothing"),  
 new Product("P0105", "Watch", "Accessories")  
 };  
  
 Product result1 = SearchFunctions.*linearSearch*(inventory, "Watch");  
 System.*out*.println("Linear Search Result is: " + (result1 != null ? result1 : "Not Found"));  
  
 SearchFunctions.*sortProductName*(inventory);  
 Product result2 = SearchFunctions.*binarySearch*(inventory, "Watch");  
 System.*out*.println("Binary Search Result is: " + (result2 != null ? result2 : "Not Found"));  
 }  
}

Output:



Exercise 7: Financial Forecasting

Recursion :

Recursionis technique used in computer science to solve big problems by breaking them into smaller, similar problems. The process in which a function calls itself directly or indirectly is called recursionand the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily.

For Example:

Simplifies problems like factorial, Fibonacci, tree traversal, etc.

Breaks problems into smaller sub-problems for elegance and clarity.

Recursive algorithm

A recursive algorithm is an algorithm that uses recursion to solve a problem. Recursive algorithms typically have two parts:

1. Base case**:**Which is a condition that stops the recursion.
2. Recursive case**:** Which is a call to the function itself with a smaller version of the problem.

Recursive Formula for Forecasting:

We can model future value using compound growth:

FV = PV \* (1 + r)^n

Where:

* FV = Future Value
* PV = Present Value
* r = growth rate
* n = number of periods

This can be expressed recursively as:

futureValue(PV, r, n) = futureValue(PV, r, n-1) \* (1 + r)

Base case: if n == 0, return PV

FinacialForcast:

public class FinacialForcast {  
 public static double futureValue(double presentValue, double rate, int years) {  
 if (years == 0) {  
 return presentValue;  
 }  
 return *futureValue*(presentValue, rate, years - 1) \* (1 + rate);  
 }  
 public static void main(String[] args) {  
 double pv = 10000;  
 double rate = 0.08;  
 int years = 5;  
  
 double forecast = *futureValue*(pv, rate, years);  
 System.*out*.println("Future Value: ₹" + forecast);  
  
 }  
}

Time Complexity:

| Method | Time Complexity | Space Complexity |
| --- | --- | --- |
| futureValue (Recursive) | O(n) | O(n) (due to call stack) |
| futureValue(Iterative) | O(n) | O(1) |

Optimization:

* Use iteration instead of recursion for better space efficiency.
* For performance-critical or large n, avoid recursion due to stack overhead.
* Memoization isn’t necessary here since no overlapping subproblems exist.

Output:

