**Week-1**

**Design principles & Patterns**

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

**Code Snippet:**

* **Logger.java:**

**package** com.core;

**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);

}

}

* **Test.java:**

**package** com.core;

**public** **class** Test {

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

Logger logger1 = Logger.*getInstance*();

logger1.log("First log message");

Logger logger2 = Logger.*getInstance*();

logger2.log("Second log message");

**if** (logger1 == logger2) {

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

} **else** {

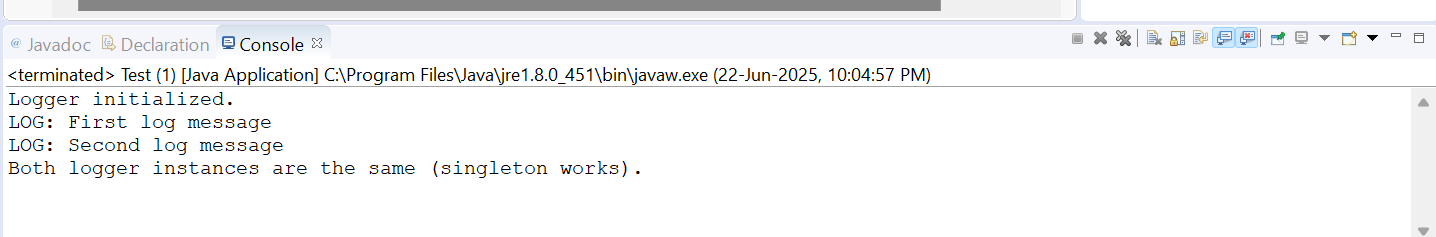
System.***out***.println("Logger instances are different (singleton failed).");

}

}

}

**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.

**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 Snippet:**

* **DocumentFactoryTest.java:**

**package** com.factory;

**interface** Document {

**void** open();

}

**class** WordDocument **implements** Document {

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

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

}

}

**class** PdfDocument **implements** Document {

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

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

}

}

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

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

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

}

}

**abstract** **class** DocumentFactory {

**public** **abstract** Document createDocument();

}

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

**public** Document createDocument() {

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

}

}

**class** PdfDocumentFactory **extends** DocumentFactory {

**public** Document createDocument() {

**return** **new** PdfDocument();

}

}

**class** ExcelDocumentFactory **extends** DocumentFactory {

**public** Document createDocument() {

**return** **new** ExcelDocument();

}

}

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

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

DocumentFactory wordFactory = **new** WordDocumentFactory();

wordFactory.createDocument().open();

DocumentFactory pdfFactory = **new** PdfDocumentFactory();

pdfFactory.createDocument().open();

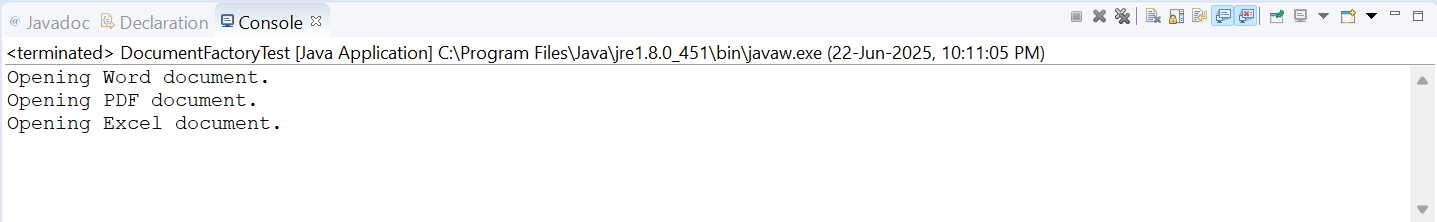
DocumentFactory excelFactory = **new** ExcelDocumentFactory();

excelFactory.createDocument().open();

}

}

**Output:**



**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.

**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.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Code Snippet:**

* **Product.java:**

**package** com.ecommerce;

**public** **class** Product **implements** Comparable<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** **int** compareTo(Product other) {

**return** Integer.*compare*(**this**.productId, other.productId);

}

@Override

**public** String toString() {

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

}

}

* **EcommerceSearch.java:**

**package** com.ecommerce;

**import** java.util.Arrays;

**public** **class** EcommerceSearch {

**public** **static** Product linearSearch(Product[] products, **int** id) {

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

**if** (product.productId == id) {

**return** product;

}

}

**return** **null**;

}

**public** **static** Product binarySearch(Product[] products, **int** id) {

**int** left = 0;

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

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

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

**if** (products[mid].productId == id) {

**return** products[mid];

} **else** **if** (products[mid].productId < id) {

left = mid + 1;

} **else** {

right = mid - 1;

}

}

**return** **null**;

}

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

Product[] products = {

**new** Product(104, "Laptop", "Electronics"),

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

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

**new** Product(101, "T-shirt", "Clothing"),

**new** Product(103, "Book", "Stationery")

};

System.***out***.println("Linear Search for product ID 103:");

Product result1 = *linearSearch*(products, 103);

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

Arrays.*sort*(products);

System.***out***.println("\nBinary Search for product ID 103:");

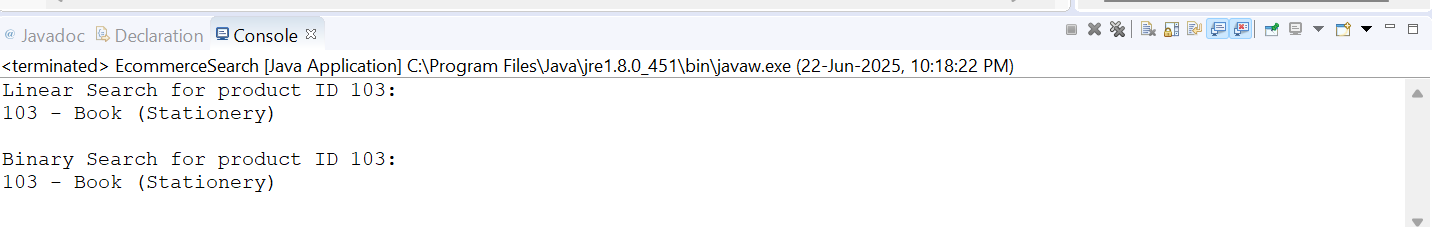
Product result2 = *binarySearch*(products, 103);

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

}

}

**Output:**



**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 Snippet:**

* **FinancialForecast.java:**

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

**public** **static** **double** forecastValueRecursive(**double** amount, **double** rate, **int** years) {

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

**return** amount;

}

**return** *forecastValueRecursive*(amount, rate, years - 1) \* (1 + rate);

}

**public** **static** **double** forecastValueMemo(**double** amount, **double** rate, **int** years, Double[] memo) {

**if** (years == 0) **return** amount;

**if** (memo[years] != **null**) **return** memo[years];

memo[years] = *forecastValueMemo*(amount, rate, years - 1, memo) \* (1 + rate);

**return** memo[years];

}

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

**double** initialAmount = 1000.0;

**double** annualGrowthRate = 0.10;

**int** years = 5;

System.***out***.println("Recursive Forecast (without memo):");

**double** forecast1 = *forecastValueRecursive*(initialAmount, annualGrowthRate, years);

System.***out***.printf("Value after %d years: Rs %.2f\n", years, forecast1);

System.***out***.println("\nRecursive Forecast (with memoization):");

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

**double** forecast2 = *forecastValueMemo*(initialAmount, annualGrowthRate, years, memo);

System.***out***.printf("Value after %d years: Rs %.2f\n", years, forecast2);

}

}

**Output:**

