Week1

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.

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.

```
package com.example.singleton;

public class Logger {
    private static Logger instance;

    private Logger() {
        System.out.println("Logger instance created!");
    }

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
```

```
    }

    public void log(String message) {
        System.out.println("LOG: " + message);
    }
}
```

3. Implement the Singleton Pattern:

- Write code to ensure that the Logger class follows the Singleton design pattern.
- Note: This is already implemented in the Logger class above with the private constructor and getInstance() method.

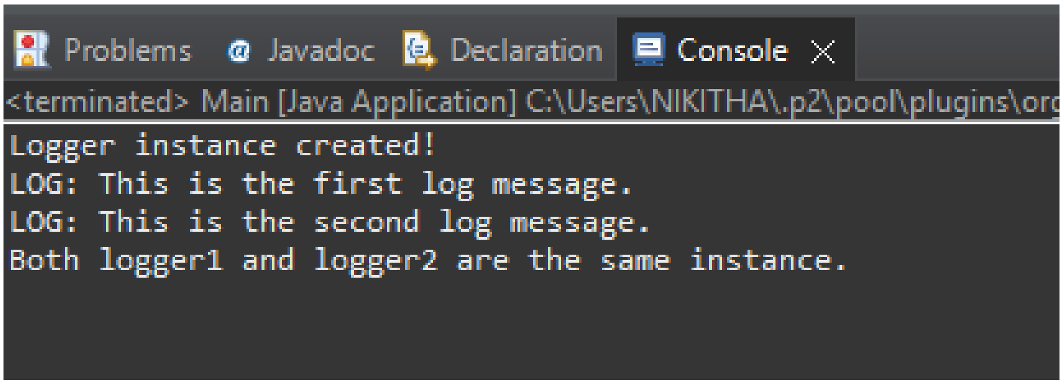
4. Test the Singleton Implementation:

- Create a test class to verify that only one instance of Logger is created and used across the application.

```
package com.example.singleton;

public class Main {
    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 logger1 and logger2 are the same instance.");
        } else {
            System.out.println("Different 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.

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**.

```
// IDocument.java
package com.example.factory;
```

```
public interface IDocument {
    void open();
}
```

3. **Create Concrete Document Classes:**
    - Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.

```
// WordDocument.java
package com.example.factory;

public class WordDocument implements IDocument {
    @Override
    public void open() {
        System.out.println("Opening Word Document...");
    }
}

// PdfDocument.java
package com.example.factory;

public class PdfDocument implements IDocument {
    @Override
    public void open() {
        System.out.println("Opening PDF Document...");
    }
}

// ExcelDocument.java
package com.example.factory;

public class ExcelDocument implements IDocument {
    @Override
    public void open() {
        System.out.println("Opening Excel Document...");
```

```
    }
}
```

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.

```
// DocumentFactory.java
package com.example.factory;

public abstract class DocumentFactory {
    public abstract IDocument createDocument();
}

// WordFactory.java
package com.example.factory;

public class WordFactory extends DocumentFactory {
    @Override
    public IDocument createDocument() {
        return new WordDocument();
    }
}

// PdfFactory.java
package com.example.factory;

public class PdfFactory extends DocumentFactory {
    @Override
    public IDocument createDocument() {
        return new PdfDocument();
    }
}
```

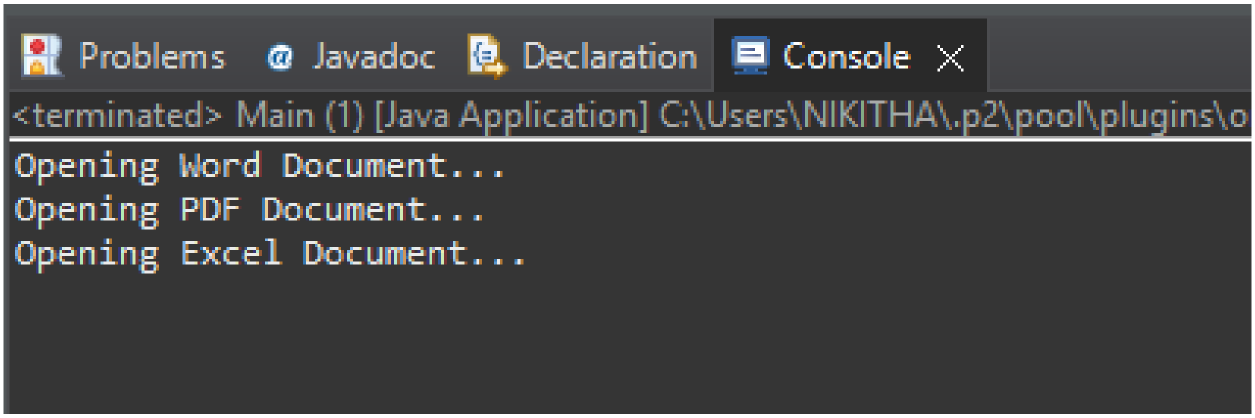
```java
// ExcelFactory.java
package com.example.factory;

public class ExcelFactory extends DocumentFactory {
    @Override
    public IDocument createDocument() {
        return new ExcelDocument();
    }
}
```

5. **Test the Factory Method Implementation:**
    - Create a test class to demonstrate the creation of different document types using the factory method.

```java
// Main.java
package com.example.factory;

public class Main {
    public static void main(String[] args) {
        // Word document
        DocumentFactory wordFactory = new WordFactory();
        IDocument word = wordFactory.createDocument();
        word.open();

        // PDF document
        DocumentFactory pdfFactory = new PdfFactory();
        IDocument pdf = pdfFactory.createDocument();
        pdf.open();

        // Excel document
        DocumentFactory excelFactory = new ExcelFactory();
        IDocument excel = excelFactory.createDocument();
        excel.open();
```

```
    }
}
```

Output:



Exercise 3: 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.

1: Understand Asymptotic Notation

What is Big O Notation?

- **Big O Notation** describes how the runtime of an algorithm increases with the size of input.
- It is used to **measure the efficiency** of algorithms in terms of:
    - **Time Complexity** – how long it takes to run
    - **Space Complexity** – how much memory it uses

Common Complexities (Time):

Notation	Name	Example Operation

O(1)	Constant time	Accessing array by index
O(log n)	Logarithmic	Binary Search
O(n)	Linear	Linear Search
O(n log n)	Log-linear	Merge Sort
$O(n^2)$	Quadratic	Nested loops

Describe the best, average, and worst-case scenarios for search operations.

Algorithm	Best Case	Average Case	Worst Case
Linear	O(1)	O(n)	O(n)
Binary	O(1)	O(log n)	O(log n)

2: Setup

Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

Product.java

```
package com.ecommerce.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;
    }
}
```

3: Implementation

Implemen linear search and binary search algorithms.

SearchUtils.java

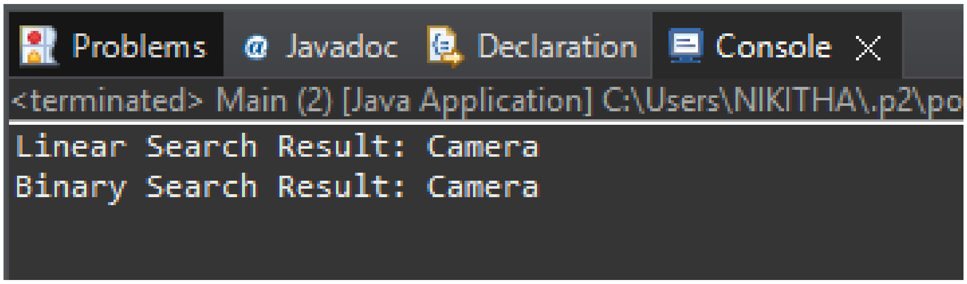
```
package com.ecommerce.search;

public class SearchUtils {

    // Linear Search (O(n))
    public static Product linearSearch(Product[] products, String targetName) {
        for (Product product : products) {
            if (product.productName.equalsIgnoreCase(targetName)) {
                return product;
            }
        }
        return null; // Not found
    }

    // Binary Search (O(log n)) – works only on sorted arrays
    public static Product binarySearch(Product[] products, String targetName) {
        int left = 0, right = products.length - 1;

        while (left <= right) {
            int mid = (left + right) / 2;
            int compare = products[mid].productName.compareToIgnoreCase(target

            if (compare == 0) return products[mid];
            else if (compare < 0) left = mid + 1;
            else right = mid - 1;
        }

        return null; // Not found
    }
}
```

4: Test the Implementation

Main.java

```
package com.ecommerce.search;

import java.util.Arrays;

public class Main {

    public static void main(String[] args) {

        Product[] products = {
            new Product(101, "Laptop", "Electronics"),
            new Product(102, "Shoes", "Fashion"),
            new Product(103, "Camera", "Electronics"),
            new Product(104, "Watch", "Accessories")
        };

        // Linear Search
        Product result1 = SearchUtils.linearSearch(products, "Camera");
        System.out.println("Linear Search Result: " +
            (result1 != null ? result1.productName : "Not Found"));

        // Binary Search – Requires sorted array
        Arrays.sort(products, (a, b) → a.productName.compareToIgnoreCase(b.proc
        Product result2 = SearchUtils.binarySearch(products, "Camera");
        System.out.println("Binary Search Result: " +
            (result2 != null ? result2.productName : "Not Found"));
    }
}
```

Output:



5: Analysis

Linear Search

- **Use Case:** When the dataset is small or unsorted.
- **Time Complexity:** O(n)
- **Drawback:** Slower with large data.

Binary Search

- **Use Case:** Best for large sorted datasets.
- **Time Complexity:** O(logn)
- **Requirement:** Data must be sorted before searching.

For our product search, binary search is more efficient if we can afford to sort the data first. Otherwise, linear search works for small/unsorted collections. I implemented both to compare performance in different scenarios.

Exercise 7: Financial Forecasting

Scenario

You are building a **financial forecasting tool** that predicts future investment values based on past data and annual growth rates.

1: Understand Recursive Algorithms

What is Recursion?

- **Recursion** is a technique in which a method **calls itself** to solve smaller instances of the same problem.
- It continues until it reaches a **base case**, which stops the recursion.

Why Use Recursion?

- Simplifies problems that are **naturally repetitive**, such as forecasting each year based on previous values.
- Helps break down complex logic into smaller, manageable tasks.

2: Setup

You need a method that calculates the **future value of an investment** based on:

- **Initial Value**
- **Annual Growth Rate**
- **Number of Years**

3: Implementation

Forecast.java

```
package com.finance.forecast;

import java.util.HashMap;
import java.util.Map;

public class Forecast {

    // Basic recursive method to calculate future value
    public static double forecastRecursive(double value, double growthRate, int years) {
        if (years == 0) return value; // Base case
        return forecastRecursive(value * (1 + growthRate), growthRate, years - 1);
    }
```

```
    // Optimized version using memoization
    private static final Map<Integer, Double> memo = new HashMap<>();

    public static double forecastMemoized(double value, double growthRate, in
t years) {
        if (years == 0) return value;
        if (memo.containsKey(years)) return memo.get(years);

        double result = forecastMemoized(value * (1 + growthRate), growthRate,
years - 1);
        memo.put(years, result);
        return result;
    }
}
```

Main.java

```
package com.finance.forecast;

public class Main {
    public static void main(String[] args) {
        double initialValue = 10000;     // ₹10,000 initial investment
        double growthRate = 0.1;         // 10% annual growth
        int years = 5;

        // Forecast using basic recursion
        double forecast = Forecast.forecastRecursive(initialValue, growthRate, ye
ars);
        System.out.println("Forecast after " + years + " years (Recursive): ₹" + fo
recast);

        // Forecast using memoized recursion
        double forecastMemo = Forecast.forecastMemoized(initialValue, growthR
ate, years);
        System.out.println("Forecast after " + years + " years (Memoized): ₹" + f
```

```
orecastMemo);
    }
}
```

4: Analysis

Time Complexity

Method	Time Complexity
forecastRecursive()	O(n)
forecastMemoized()	O(n), faster due to memoization

Disadvantages of Plain Recursion:

- May **recalculate values repeatedly**, leading to performance issues.
- Can cause **stack overflow** if the number of recursive calls is too high.

Optimization: Memoization

- **Memoization** stores previously computed results in a Map.
- Reduces redundant calculations.
- Improves performance especially when results are reused or the number of years is high.

Output: