Week 1: Design Patterns and Principles

Exercise 1: Implementing the Singleton Pattern

-THEJASHRI NARAYANAN

Scenario:

You need to ensure that a logging utility class in your application has only one instance thtroughout the application lifecycle to ensure consistent logging.

Steps:

1. Created a New Java Project:

- Project name: SingletonPatternExample

2. Defined the Singleton Class:

- Created a class named Logger
- Made the constructor private
- Used a private static instance variable
- Provided a public static method to return the single instance

Logger.java

```java
public class Logger {
    private static Logger loggerInstance;

    private Logger() {
        System.out.println(x:"Logger initialized...");
    }

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

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

```

3. Implemented Singleton Logic:

- Used lazy initialization in getInstance() to create only one instance
- Verified using object references

4. Tested Singleton Implementation:

- Wrote a Main class to test and verify that both variables point to the same Logger instance

Main.java

```java
public class Main {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Logger log1 = Logger.getInstance();
        log1.log(message:"First log message");

        Logger log2 = Logger.getInstance();
        log2.log(message:"Second log message");

        if (log1 == log2) {
            System.out.println(x:"Both loggers are the same instance (Singleton Verified )");
        } else {
            System.out.println(x:"Different instances (Singleton Failed )");
        }
    }
}
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\design nd patterns\pgm> javac Logger.java Main.java
>>
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\design nd patterns\pgm> java Main
Logger initialized...
[LOG]: First log message
[LOG]: Second log message
Both loggers are the same instance (Singleton Verified )
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\design nd patterns\pgm>
```

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 :

CustomFileManager.java > Language Support for Java(TM) by Red Hat > CustomFileManager

```java
import java.util.Scanner;

// STEP 1: Define the interface

interface FileHandler {
    void open();
    void save();
}

// STEP 2: Create concrete implementations

class TextFile implements FileHandler {
    @Override
    public void open() {
        System.out.println(x:"Opening a plain text file...");
    }

    @Override
    public void save() {
        System.out.println(x:"Saving changes to the text file.");
    }
}

class CsvFile implements FileHandler {
    @Override
    public void open() {
        System.out.println(x:"Opening a CSV spreadsheet...");
    }

    @Override
    public void save() {
        System.out.println(x:"Saving CSV data...");
```

```
    }
}
class MarkdownFile implements FileHandler {
    @Override
    public void open() {
        System.out.println(x:"Opening a Markdown document...");
    }

    @Override
    public void save() {
        System.out.println(x:"Saving Markdown content...");
    }
}

// STEP 3: Create the abstract factory

abstract class FileFactory {
    public abstract FileHandler generateFile();
}

// STEP 4: Implement concrete factories

class TextFileFactory extends FileFactory {
    @Override
    public FileHandler generateFile() {
        return new TextFile();
    }
}

class CsvFileFactory extends FileFactory {
    @Override
```

```
class CsvFileFactory extends FileFactory {
    @Override
    public FileHandler generateFile() {
        return new CsvFile();
    }
}

class MarkdownFileFactory extends FileFactory {
    @Override
    public FileHandler generateFile() {
        return new MarkdownFile();
    }
}


// STEP 5: Main class to test everything

public class CustomFileManager {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {

        try (Scanner scanner = new Scanner(System.in)) {
            System.out.println(x:"Choose file type to create: text / csv / markdown");
            String choice = scanner.nextLine().toLowerCase();

            FileFactory factory;

            switch (choice) {
                case "text":
                    factory = new TextFileFactory();
                    break;
```

```
                    break;
                case "csv":
                    factory = new CsvFileFactory();
                    break;
                case "markdown":
                    factory = new MarkdownFileFactory();
                    break;
                default:
                    System.out.println(x:"Invalid choice.");
                    return;
            }
            FileHandler file = factory.generateFile();
            file.open();
            file.save();
        }
    }
}
```

Output

```
PS C:\Usejavac CustomFileManager.javas\Engineering concepts\design nd p
>> erns>
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\design nd patterns> java CustomFileManager
>>
Choose file type to create: text / csv / markdown
text
Opening a plain text file...
Saving changes to the text file.
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\design nd patterns> java CustomFileManager
>>
Choose file type to create: text / csv / markdown
csv
Opening a CSV spreadsheet...
csv
Opening a CSV spreadsheet...
Saving CSV data...
csv
Opening a CSV spreadsheet...
Opening a CSV spreadsheet...
Saving CSV data...
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\design nd patterns> java CustomFileManager
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\design nd patterns> java CustomFileManager
>>
Choose file type to create: text / csv / markdown
markdown
Opening a Markdown document...
Opening a Markdown document...
Saving Markdown content...
```

Algorithms and Data Structures

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. Understanding Big O Notation

Big O notation is just a way to measure how fast or slow an algorithm is when the input size gets really big. It helps us know which algorithm is better when we have lots of data.

- Best case: The search finds the item right away (super fast).
- Average case: On average, how long it takes to find the item.
- Worst case: The longest it could take, like if the item isn't there or at the very end.

2.Program:

ProductSearchApp.java > ...

```java
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;

public class ProductSearchApp {

    // STEP 2: Define Product class
    static class Product {
        int id;
        String name;
        String type;

        Product(int id, String name, String type) {
            this.id = id;
            this.name = name;
            this.type = type;
        }

        @Override
        public String toString() {
            return "Product ID: " + id + ", Name: " + name + ", Type: " + type;
        }
    }

    // STEP 3a: Implement Linear Search
    public static Product findUsingLinearSearch(Product[] items, String keyword) {
        for (Product item : items) {
            if (item.name.equalsIgnoreCase(keyword)) {
                return item;
            }
        }
        return null;
    }

    // STEP 3b: Implement Binary Search
    public static Product findUsingBinarySearch(Product[] items, String keyword) {
        int low = 0, high = items.length - 1;
```

```
38
39          while (low <= high) {
40              int mid = (low + high) / 2;
41              int compare = items[mid].name.compareToIgnoreCase(keyword);
42
43              if (compare == 0) return items[mid];
44              else if (compare < 0) low = mid + 1;
45              else high = mid - 1;
46          }
47          return null;
48      }
49
50      // STEP 5: Main method to test both searches
        Run main | Debug main | Run | Debug
51      public static void main(String[] args) {
52          Scanner sc = new Scanner(System.in);
53
54          // Creating sample product list
55          Product[] catalog = {
56              new Product(id:1, name:"Laptop", type:"Electronics"),
57              new Product(id:2, name:"Keyboard", type:"Accessories"),
58              new Product(id:3, name:"Shoes", type:"Fashion"),
59              new Product(id:4, name:"Book", type:"Education"),
60              new Product(id:5, name:"Mobile", type:"Electronics")
61          };
62
63
64          System.out.print(s:"Enter product name to search: ");
65          String inputName = sc.nextLine();
66
67          // Linear search (no need to sort)
68          System.out.println(x:"\n📦 Linear Search Result:");
69          Product result1 = findUsingLinearSearch(catalog, inputName);
70          System.out.println(result1 != null ? result1 : "Product not found");
```

```
71
72          // Binary search (sort first)
73          Arrays.sort(catalog, Comparator.comparing(p -> p.name.toLowerCase()));
74          System.out.println(x:"\n📦 Binary Search Result:");
75          Product result2 = findUsingBinarySearch(catalog, inputName);
76          System.out.println(result2 != null ? result2 : "Product not found");
77
78          sc.close();
79      }
80  }
```

Output:

```
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\ADS> javac ProductSearchApp.java
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\ADS> java ProductSearchApp
Enter product name to search: Book

Enter product name to search: Book

? Linear Search Result:
Product ID: 4, Name: Book, Type: Education

? Binary Search Result:
Product ID: 4, Name: Book, Type: Education
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\ADS>
```

4. Analysis :

In my project, I compared **linear search** and **binary search** to find products faster.

What I Observed:

- **Linear Search** checks each product one by one. So if I have 1000 products, it may need to check all 1000 in the worst case. That's why it's **slow for big data** – time complexity is **O(n)**.

- **Binary Search** first sorts the product list, then keeps dividing the list to find the target quickly.
  So it's way **faster**, especially when the list is large – time complexity is **O(log n)**.

Which Is Better?

- For **small lists**, linear search is fine and easy to use.
- But for **big platforms like Amazon**, **binary search is much better** because it saves time.

Exercise 7: Financial Forecasting

Scenario:

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

Steps:

1. Understand Recursive Algorithm

Recursion is when a method calls itself to solve a smaller version of the same problem.

It's ideal for problems with repetitive structure, like forecasting over months/years.

2&3. Recursive Forecasting Method:

```java
import java.util.Scanner;
public class FinancePredictor {
    // STEP 1 & 2: Recursive method to estimate future investment value
    public static double estimateGrowth(double baseAmount, double growthRate, int periods) {

        if (periods == 0) {
            return baseAmount;
        }

        double updatedAmount = baseAmount * (1 + growthRate);
        return estimateGrowth(updatedAmount, growthRate, periods - 1);
    }
    // STEP 3: Main method to test the forecast
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println(x:"=== Financial Growth Forecast Tool ===");
        System.out.print(s:"Enter initial amount (₹): ");
        double startingValue = input.nextDouble();

        System.out.print(s:"Enter annual growth rate (e.g., 0.07 for 7%): ");
        double rate = input.nextDouble();

        System.out.print(s:"Enter number of years: ");
        int years = input.nextInt();

        // STEP 4: Forecast using recursion
        double predictedAmount = estimateGrowth(startingValue, rate, years);
        System.out.printf(format:"Estimated value after %d years: ₹%.2f\n", years, predictedAmount);
        input.close();
    }
}
```

Output:

```
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\ADS> javac FinancePredictor.java
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\ADS> java FinancePredictor
=== Financial Growth Forecast Tool ===
Enter initial amount (?): 160000
Enter annual growth rate (e.g., 0.07 for 7%): 5
Enter number of years: 9
Estimated value after 9 years: ?1612431360000.00
PS C:\Users\Hp\Desktop\cts assessments\Engineering concepts\ADS>
```

4.Analysis :

Time Complexity:

Recursive depth = years (n) → So it's O(n) time

No repeated work = No exponential growth = Still efficient

Optimization Techniques (if needed for complex models):

Memoization (store already computed results)

Tail Recursion (optimize for deep stacks)

Iterative version (preferred for large-scale datasets)