

Name : KAKUMANU TARAKA LAKSHMI PRASANNA

Superset ID : 6314755

Email : 22BQ1A4268@vvit.net

Exercise 1 : Implementing a Singleton Pattern

Scenario:

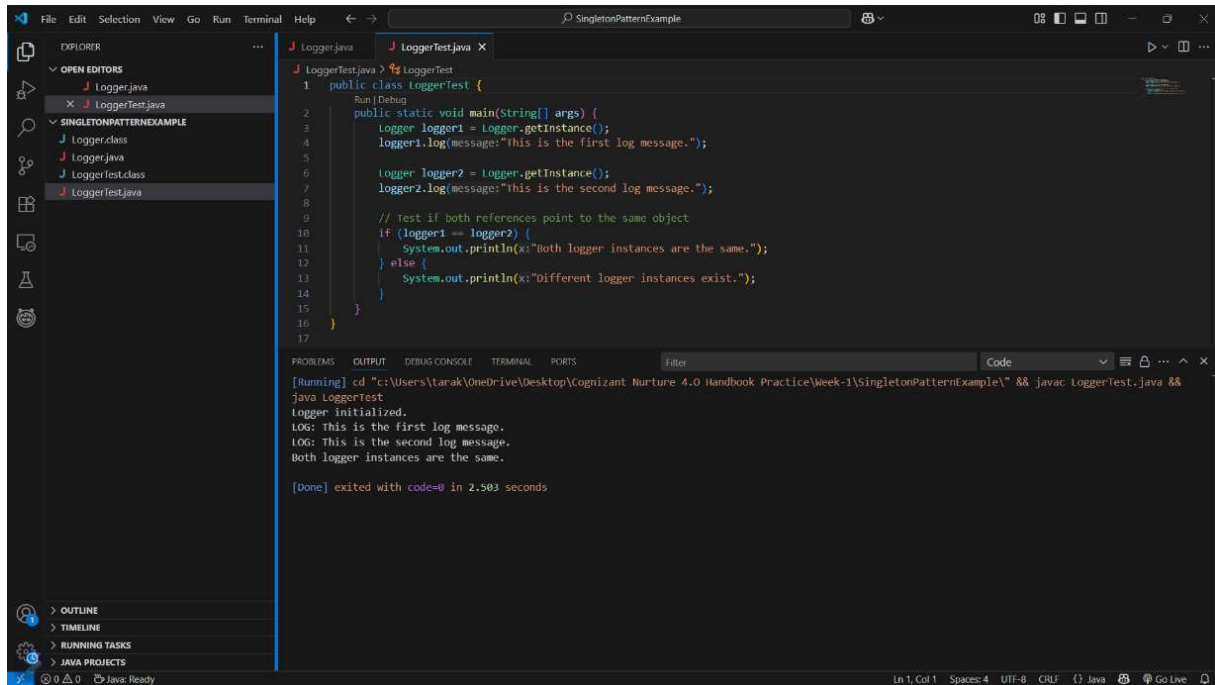
You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

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.

Singleton Pattern:

- Ensures that only one instance of a class is created and provides a global point of access to it.
- Useful when you want to have a single, shared instance of a class throughout the application.
- Ensuring thread safety is crucial when implementing a singleton class to prevent issues that may arise from concurrent access by multiple threads.



```
1 public class LoggerTest {
2     public static void main(String[] args) {
3         Logger logger1 = Logger.getInstance();
4         logger1.log(message: "this is the first log message.");
5
6         Logger logger2 = Logger.getInstance();
7         logger2.log(message: "this is the second log message.");
8
9         // Test if both references point to the same object
10        if (logger1 == logger2) {
11            System.out.println(x: "Both logger instances are the same.");
12        } else {
13            System.out.println(x: "Different logger instances exist.");
14        }
15    }
16 }
17 }
```

[Running] cd "c:\Users\tarak\OneDrive\Desktop\Cognizant Nurture 4.0 Handbook Practice\Week-1\SingletonPatternExample\" && javac LoggerTest.java && java LoggerTest

Logger initialized.

LOG: This is the first log message.

LOG: This is the second log message.

Both logger instances are the same.

[Done] exited with code=0 in 2.503 seconds

Exercise 2: Implementing the Factory Method Pattern

Scenario:

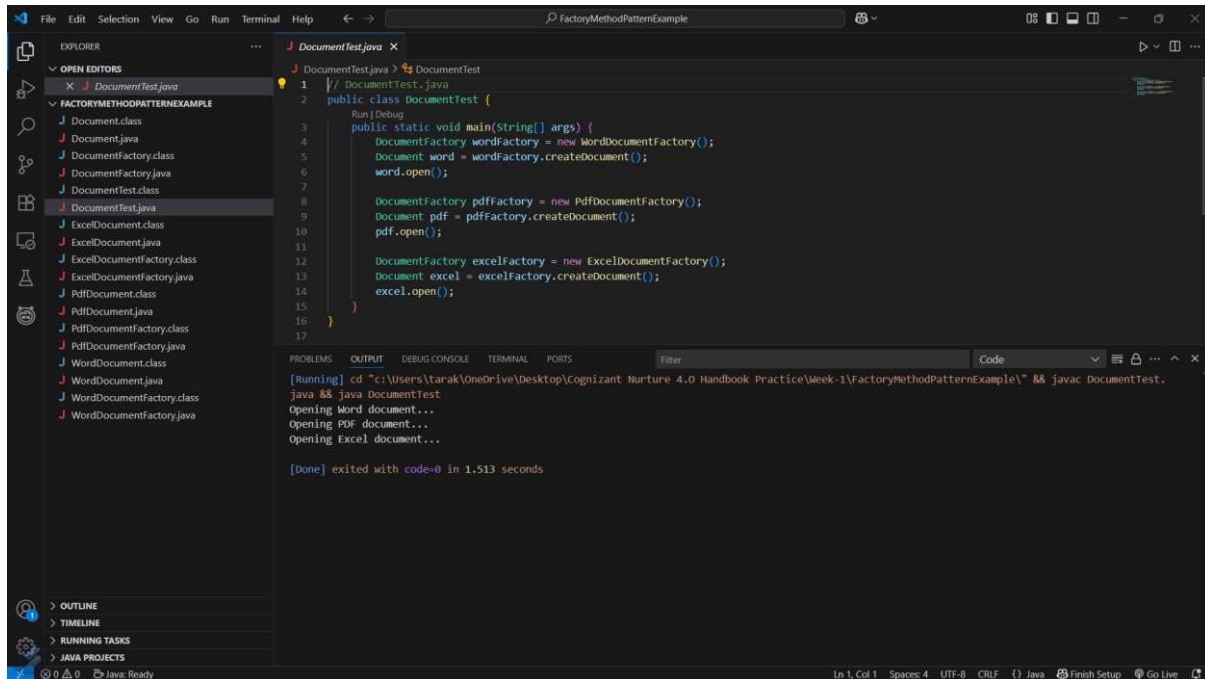
You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

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.

Factory Pattern :

The Factory Pattern is a creational design pattern that provides an interface for creating objects but allows subclasses or implementing classes to decide which class to instantiate. It encapsulates the object creation logic within a separate factory class, providing a common interface for creating objects without exposing the instantiation logic to the client.



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.

Steps:

1. Understand Asymptotic Notation:

- Explain Big O notation and how it helps in analysing algorithms
 - Big O describes **how the performance of an algorithm changes with input size**.
 - It helps analyse **time and space complexity** of algorithms.
 - Focuses on the **worst-case scenario** to guarantee performance limits.
 - Ignores constants and machine-specific factors, focusing on **growth rates**.
 - Helps in choosing the **most efficient algorithm** for scalability.

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

Search Type	Best Case	Average Case	Worst Case
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$

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.

In my opinion, Binary search is most suitable search function for e-commerce platform since data will be searched mostly by product name which will be in lexicographical order. Data will be in sorted order which helps binary search to perform well.

The screenshot shows an IDE window titled "E-Commerce platform Search Function". The Explorer panel on the left shows a project structure with "E-COMMERCE PLATFORM SEARCH FUNCTION" containing "Product.class", "Product.java", "Search.class", and "Search.java". The main editor displays the code for "Search.java":

```

1 import java.util.Arrays;
2 import java.util.Comparator;
3
4 public class Search {
5     public static Product linearSearch(Product[] products, String key){
6         for(Product p : products){
7             if(p.productName.equalsIgnoreCase(key)){
8                 return p;
9             }
10        }
11        return null;
12    }
13
14    public static Product binarySearch(Product[] products, String key){
15        Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));
16        int low = 0;
17        int high = products.length - 1;
18        while(low <= high){
19            int mid = (low + high) / 2;
20        }
21    }
22 }

```

The Output panel at the bottom shows the execution results:

```

[Running] cd "c:\Users\tarak\OneDrive\Desktop\Cognizant Nurture 4.0 Handbook Practice\Week-1\E-commerce platform Search Function" && javac Search.
java && java Search
Linear Search Result: 101 - aaa - a
Binary Search Result: 101 - aaa - a
[Done] exited with code=0 in 2.398 seconds

```

Exercise 4: 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.

Recursion is a programming technique where a function calls itself to solve smaller instances of the problem.

Recursion Simplify the problem as it has the nature of “ Divide and Conquer ” structure. This method is mostly preferred when we want to solve problems with repeated patterns.

Time Complexity:

- Each recursive call reduces n by 1, so the time complexity is $O(n)$.
- Each function call does a single multiplication and call.

Space Complexity:

- Due to the call stack, space complexity is also $O(n)$.

The screenshot shows an IDE with the following components:

- Explorer Panel:** Displays the project structure. The file `FinancialForecast.java` is selected under the `FINANCIAL FORECASTING` folder.
- Editor Panel:** Shows the source code of `FinancialForecast.java`. The code defines a `FinancialForecast` class with a recursive method `forecastRecursive` and a `main` method.


```

1  class FinancialForecast {
2      public static double forecastRecursive(double currentValue, double growthRate, int years) {
3          if (years == 0) {
4              return currentValue;
5          }
6          return forecastRecursive(currentValue * (1 + growthRate), growthRate, years - 1);
7      }
8
9      public static void main(String[] args) {
10         double initialValue = 10000;
11         double annualGrowthRate = 0.08;
12         int forecastYears = 5;
13
14         double futureValue = forecastRecursive(initialValue, annualGrowthRate, forecastYears);
15         System.out.printf("Forecasted Value after %d years: %.2f\n", forecastYears, futureValue);
16     }
17 }
      
```
- Run and Debug Panel:** Shows the execution output. The command executed is `java -cp . FinancialForecast`. The output is:


```

[Running] cd "C:\Users\tarak\OneDrive\Desktop\Cognizant Nuture 4.0 Handbook Practice\Week-1\Financial Forecasting\" && javac FinancialForecast.java && java FinancialForecast
Forecasted Value after 5 years: 714693.28

[Done] exited with code=0 in 1.876 seconds
      
```