# DESIGN 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.

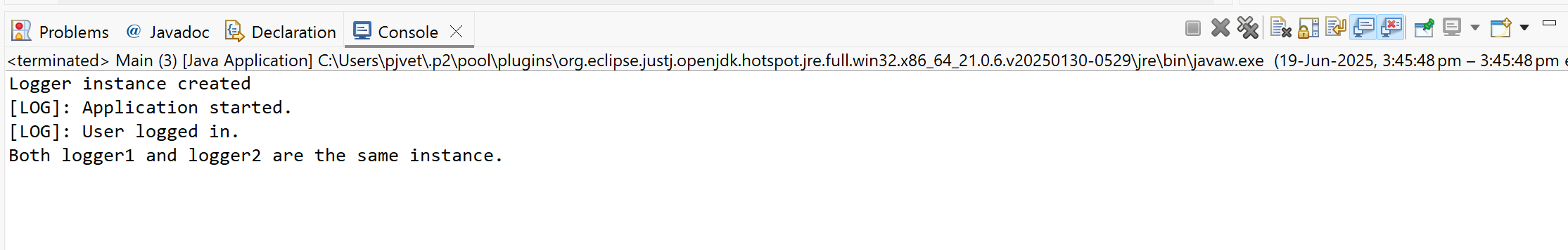
**Steps:**

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 :**



**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:**

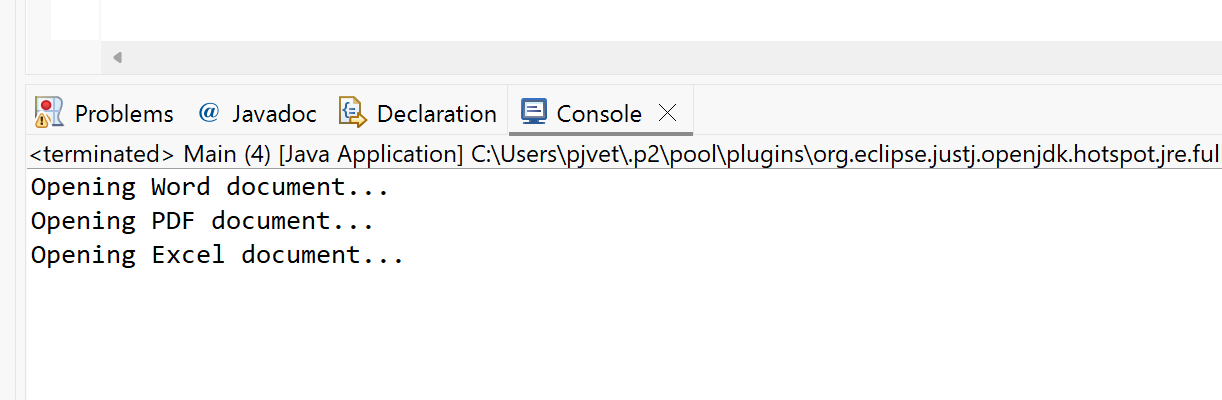
1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document 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:





OUTPUT:



# DATASTRUCTURES 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.

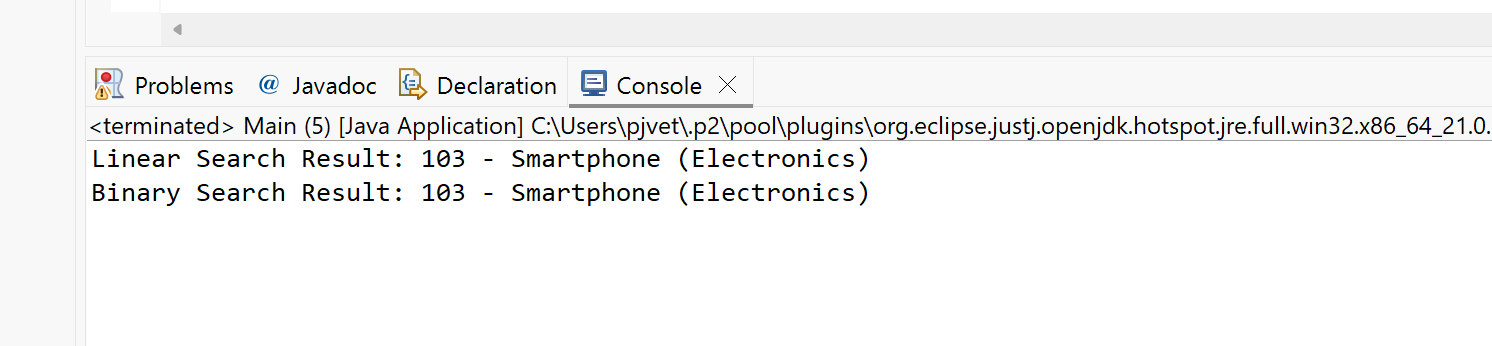
**Steps:**

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:**



OUTPUT:



**Analysis:**

**1. Comparison of Time Complexity:**

| **Search Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Sorted Required** |
| --- | --- | --- | --- | --- |
| **Linear Search** | O(1) | O(n) | O(n) | No |
| **Binary Search** | O(1) | O(log n) | O(log n) | Yes |

* **Linear Search:** Scans each element in the array sequentially until the target is found or the end is reached. It does not require the array to be sorted.
* **Binary Search:** Requires the array to be sorted. It repeatedly divides the search range in half, which significantly reduces the number of comparisons, especially for large datasets.

**2. Suitable Algorithm for the Platform:**

In the context of our Ecommerce platform, we are working with a list of products that users may search by name or ID. Here's the analysis:

* **Linear Search Suitability:**
  + Best for small or dynamically updated product lists.
  + No need to sort data before searching.
  + Easier to implement and maintain.
* **Binary Search Suitability:**
  + Best for large, static, or rarely changing product lists.
  + Requires sorting, which adds overhead if the list changes frequently.
  + Offers better performance for large datasets.

**Conclusion:**

For our current platform where product lists are relatively small and may not be sorted, **linear search is more suitable** due to its simplicity and flexibility. However, if the platform scales up with a large and sorted product database, **binary search** would be more efficient for search operations.

**Exercise\_7: Financial Forecasting**

**Scenario:**

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

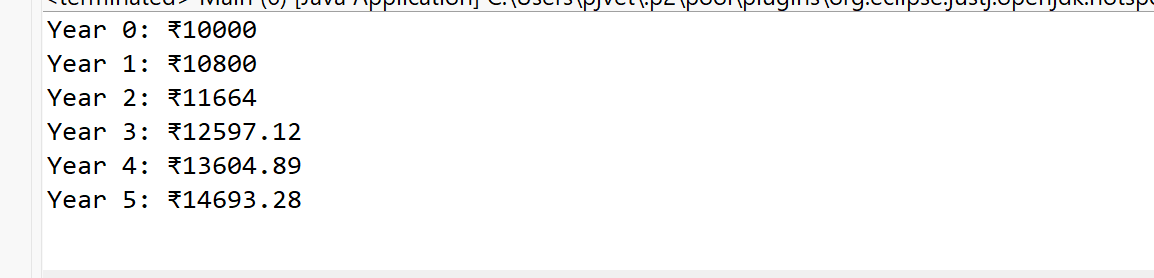
**Steps:**

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:**



OUTPUT:



**Analysis:**

**a) Time Complexity of Recursive Algorithm:**

The recursive function calculates the future value by making one call per year:

***predictValue(year - 1, initialValue, growthRate)***

* **Time Complexity: O(n)**  
  (where n is the number of years)

This is efficient for small n, but deep recursion can cause stack overflow for large values.

**b) Optimization of Recursive Solution:**

To avoid excessive computation:

* **Use Iteration or Formula**:  
  Final Value = Initial × (1 + rate)^years  
  This avoids recursion entirely.
* **Use Memoization** (if recursion is needed):  
  Store previously computed results to skip repeated calculations.