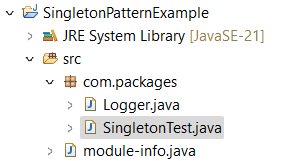
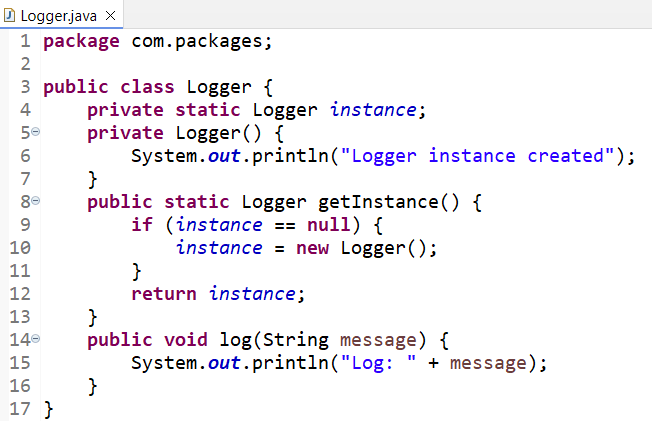
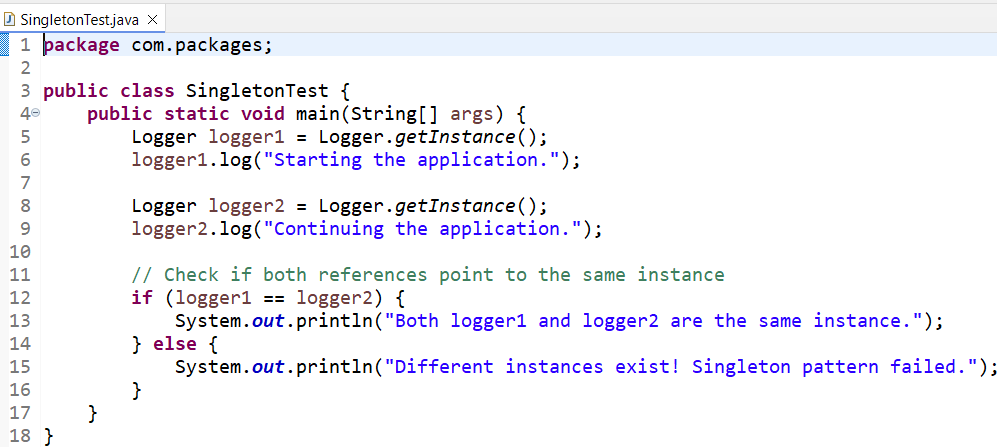
**WEEK1 HANDS-ON**

**DESIGN PRINCIPLES AND PATTERNS**

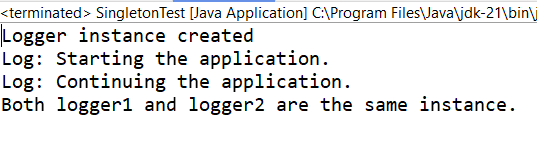
1. **Exercise1: Implementing the Singleton Pattern**



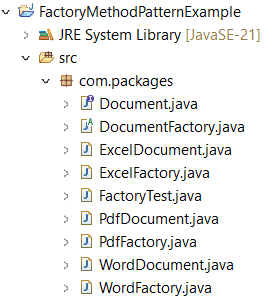


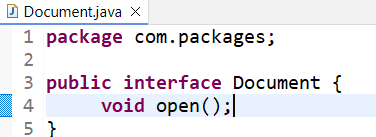


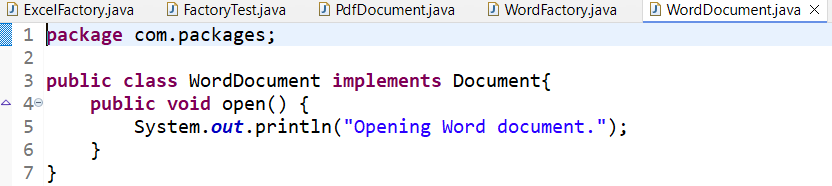
**OUTPUT:**

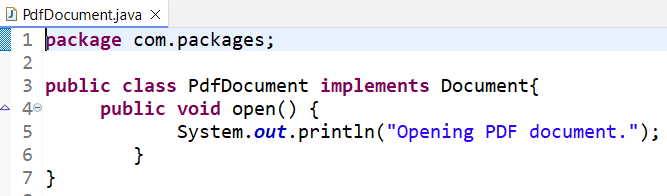
****

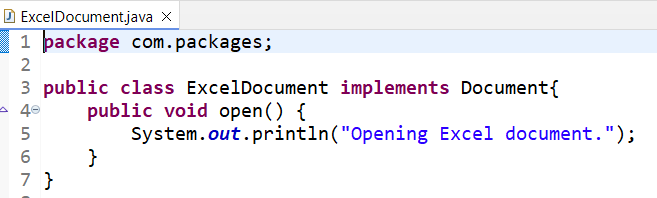
1. **Exercise2: Implementing the Factory Method Pattern**

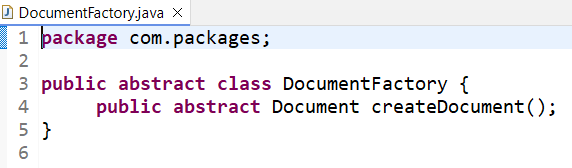
****

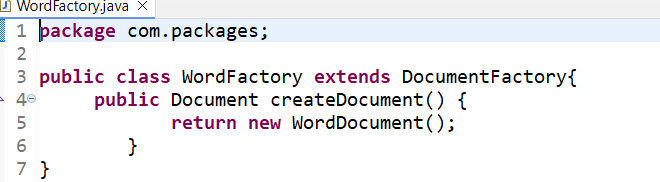
****

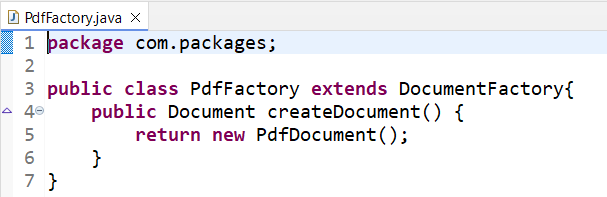
****

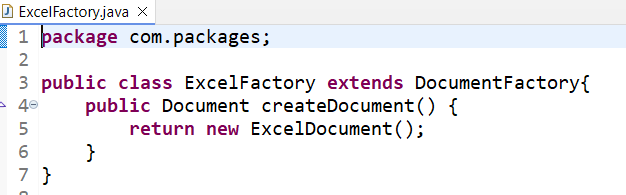
****

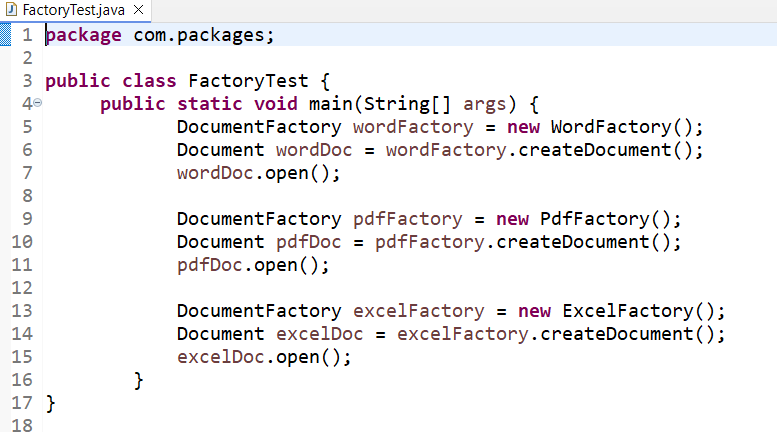
****

****

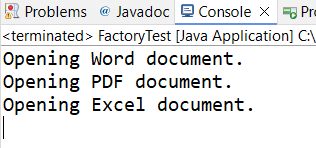
****

****

****

****

**OUTPUT:**

****

**DATA STRUCTURES AND ALGORTIHMS**

1. **EXERCISE 2: E-commerce Platform Search Function**
2. **Understand Asymptotic Notation:**

* Explain Big O notation and how it helps in analysing algorithms.

**Big O notation** is a mathematical way to describe how the performance of an algorithm scales with the size of input data. It provides a high-level understanding of the time or space complexity of algorithms — especially when the input grows very large.

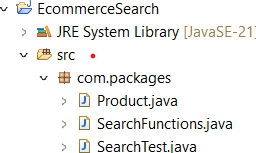
* Helps compare algorithms independently of hardware.
* Focuses on growth rate, not actual time.
* Useful for understanding worst-case performance - critical for building efficient systems.
* Describe the best, average, and worst-case scenarios for search operations.

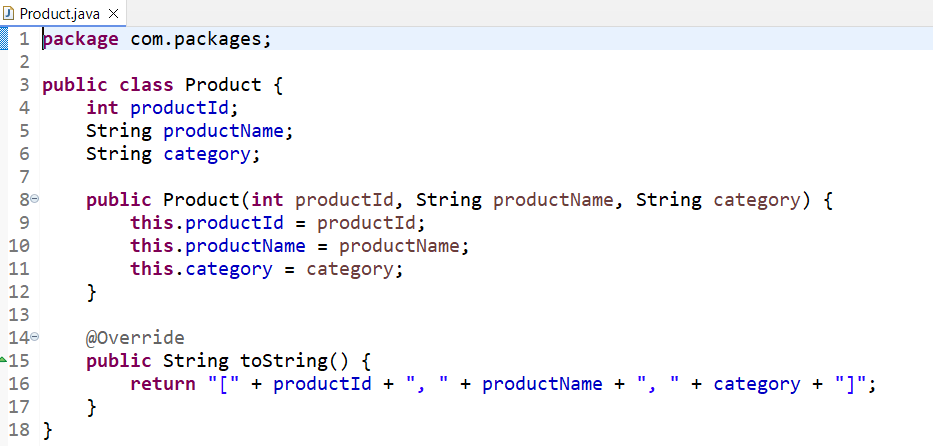
**Best Case:** The best-case occurs when the target element is found immediately.  
For example, in a linear search, the item may be the first element in the array.  
In a binary search, the element might be found at the middle in the first comparison.  
This results in the fastest execution time and is typically represented as O(1).

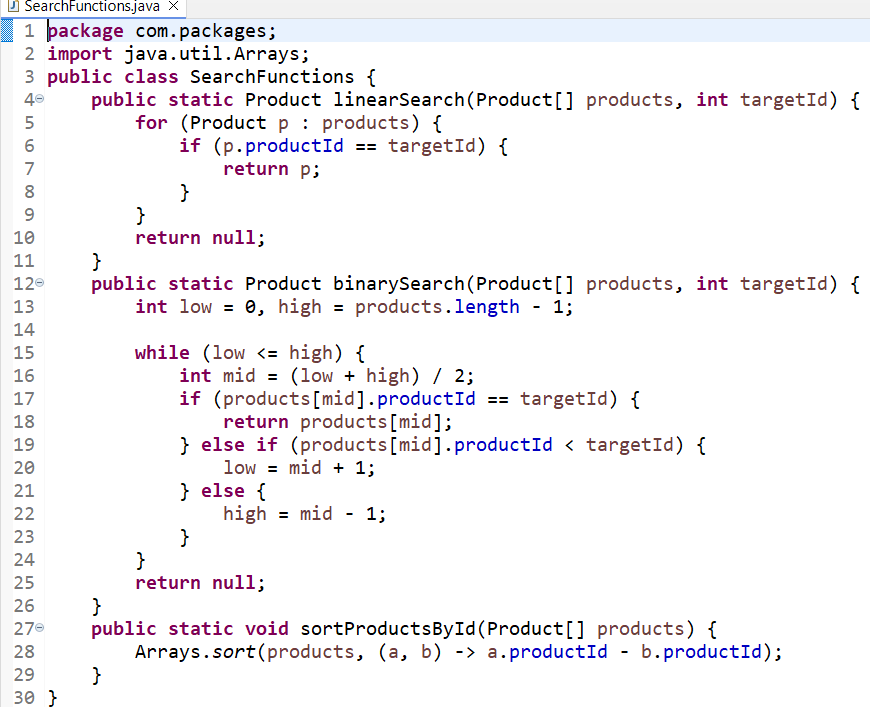
**Average Case:** The average-case assumes that the element could be anywhere in the data and is found randomly. In linear search, this means it may be found halfway through, giving a time complexity of approximately O(n/2), simplified to O(n).In binary search, it averages around O(log n) because the list is repeatedly divided in half.

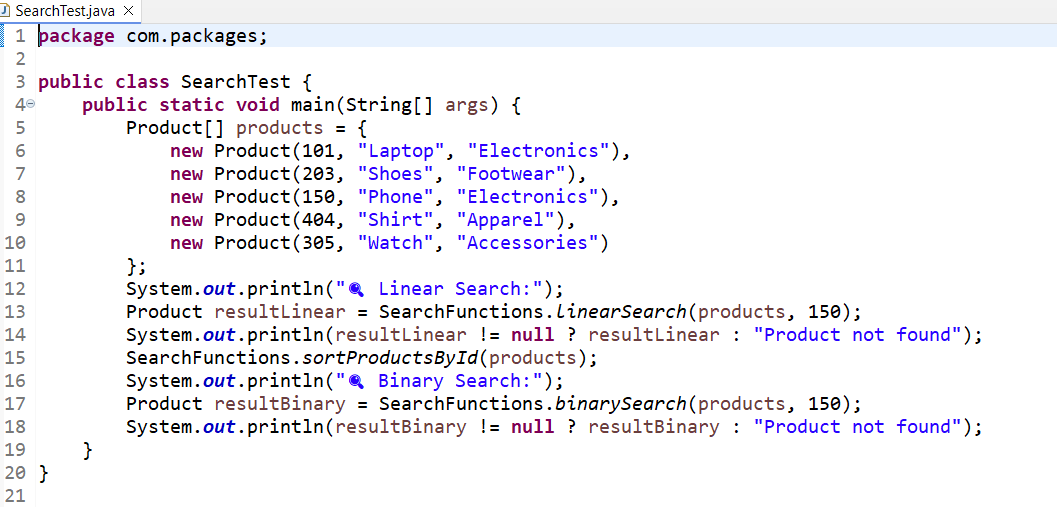
**Worst Case:** The worst-case happens when the element is at the end of the list or not present at all. In linear search, every single element must be checked, giving a time complexity of O(n). In binary search, even in the worst case, the algorithm only needs to divide the list log n times, so the time complexity remains O(log n).

2 and 3:

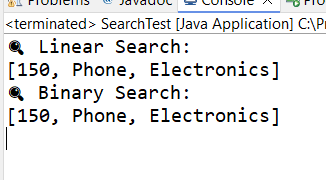








**OUTPUT:**

****

4**. Analysis:**

* Compare the time complexity of linear and binary search algorithms.

The time complexity of linear search and binary search differs significantly and impacts performance based on the size and nature of the dataset. **Linear search** has a time complexity of **O(n)** in the worst and average cases, as it may need to check every element in the array to find the target. In the best case, it is **O(1)** when the element is found at the beginning. On the other hand, **binary search** offers a much faster time complexity of **O(log n)** in the average and worst cases because it repeatedly divides the search space in half. However, binary search requires that the array is **sorted** in advance, which might add overhead if the data frequently changes. Overall, binary search is significantly more efficient than linear search for large, sorted datasets.

* Discuss which algorithm is more suitable for your platform and why.

For an e-commerce platform where search speed is critical and the number of products is large, **binary search is generally more suitable** due to its logarithmic time complexity. It allows users to find products quickly, enhancing the overall user experience. Since product data such as productId can be maintained in a sorted structure or indexed database, binary search becomes an ideal choice for optimizing performance. However, if the product list is unsorted or frequently updated in real time without re-sorting, linear search might be more flexible despite its slower speed. In most real-world e-commerce systems, data is pre-sorted or indexed, making **binary search the preferred and more scalable option** for implementing efficient search functionality.

1. **EXERCISE 7: Financial Forecasting**

1.**Understand Recursive Algorithms:**

* Explain the concept of recursion and how it can simplify certain problems.

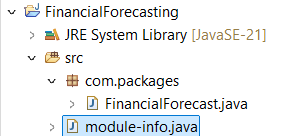
**Recursion** is a programming concept where a method calls **itself** to solve a problem. The idea is to break a large or complex problem into **smaller, more manageable sub-problems** that are easier to solve. The recursive method keeps calling itself with a reduced input until it reaches a **base case**, which stops further calls and starts returning values back up the chain.

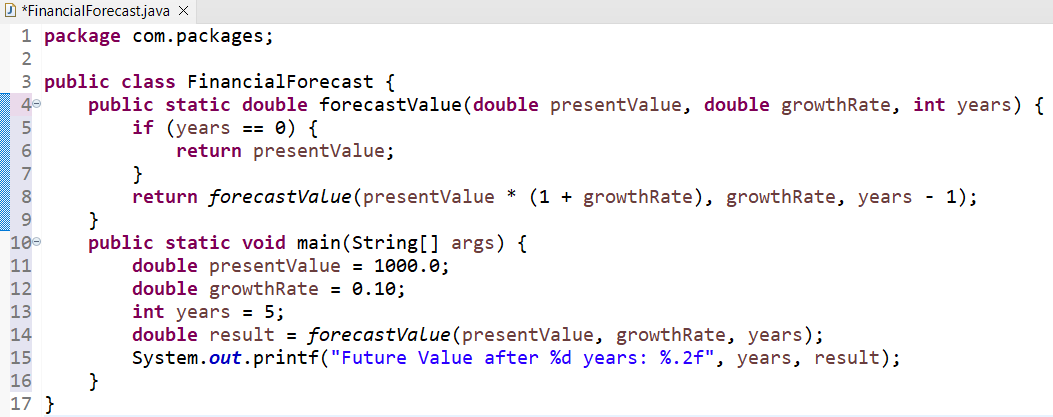
A recursive method generally has two main parts:

* **Base Case** – A condition that stops the recursion (prevents infinite calls).
* **Recursive Case** – The function calls itself with smaller input to approach the base case.

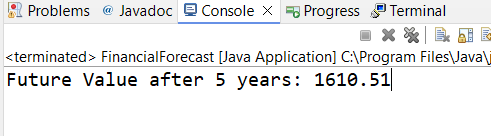
Recursion simplifies certain problems by breaking them down into smaller, similar sub-problems that are easier to solve. Instead of using complex loops and manual tracking of steps, recursion allows the same function to be reused with a smaller portion of the input, making the code more readable and elegant. This is especially helpful in problems with repetitive or hierarchical structures, such as mathematical computations (like factorials or Fibonacci numbers), tree traversals, and divide-and-conquer algorithms like merge sort. By solving the simplest case first (the base case) and building up the solution step-by-step through recursive calls, complex problems can be expressed in a concise and logical way.

2 and 3:





**OUTPUT :**



4. **Analysis:**

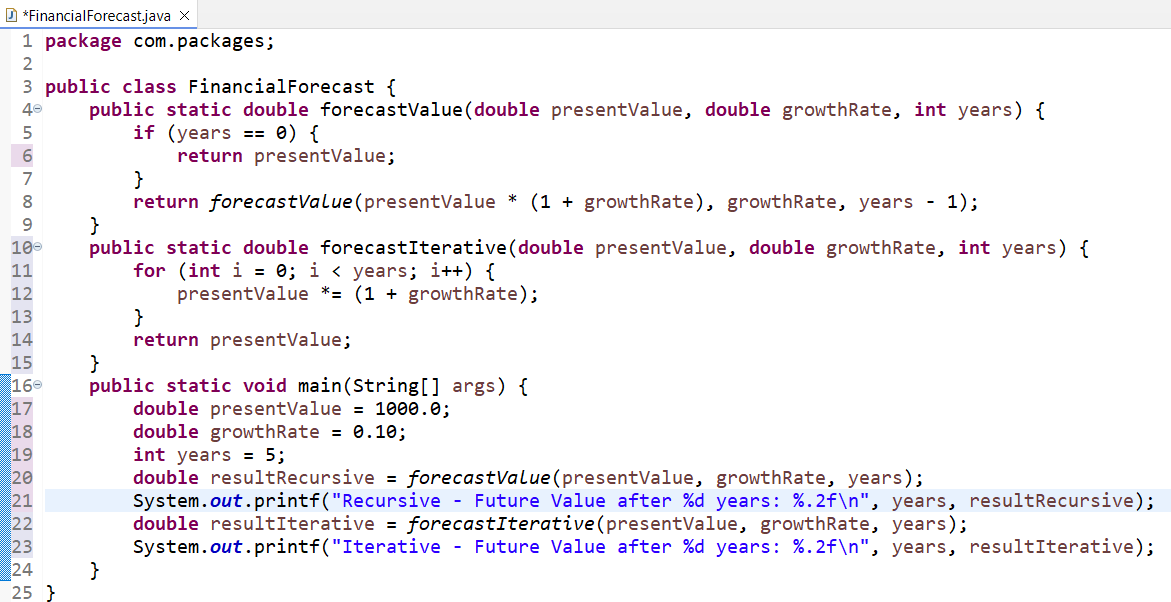
* + Discuss the time complexity of your recursive algorithm.

The time complexity of the recursive algorithm used for financial forecasting is **O(n)**, where n is the number of years being forecasted. This is because the recursive method calls itself once for each year, and each call performs a constant amount of work (a multiplication and a subtraction). As the number of years increases, the number of recursive calls increases linearly, which directly affects the execution time. Although the logic is simple and elegant, the recursive depth grows with the input, and for very large values of n, this can lead to performance concerns or even a stack overflow.

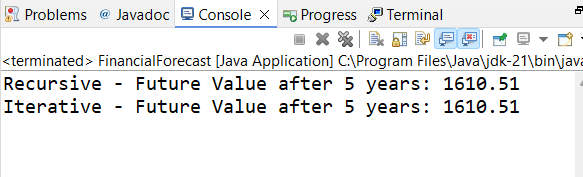
* + Explain how to optimize the recursive solution to avoid excessive computation.

To avoid excessive computation and stack overflow, the recursive solution can be optimized by converting it into an **iterative approach**. An iterative loop uses constant stack space and performs the same calculations without the overhead of recursive function calls. Instead of calling the method repeatedly, a for loop can multiply the present value by the growth factor for each year, producing the same result with better performance and lower memory usage. This not only makes the program more efficient for large inputs but also enhances its stability and scalability in real-world applications.

Second Way: Both methods give the same result, but the **iterative version is more efficient** and preferred for large years values.



**OUTPUT:**

****