**Data Structures and Algorithms**

Exercise 2: E-commerce Platform Search Function

**Understanding Asymptotic Notation:**

* **Big – O** notation provides a way to analyse the time or space taken by an algorithm. It describes the asymptotic behaviour or the order of growth of time or space taken by the algorithm. It provides an upper limit on the time or space taken.

If f(n) is a function and g(n) is another function then we can say

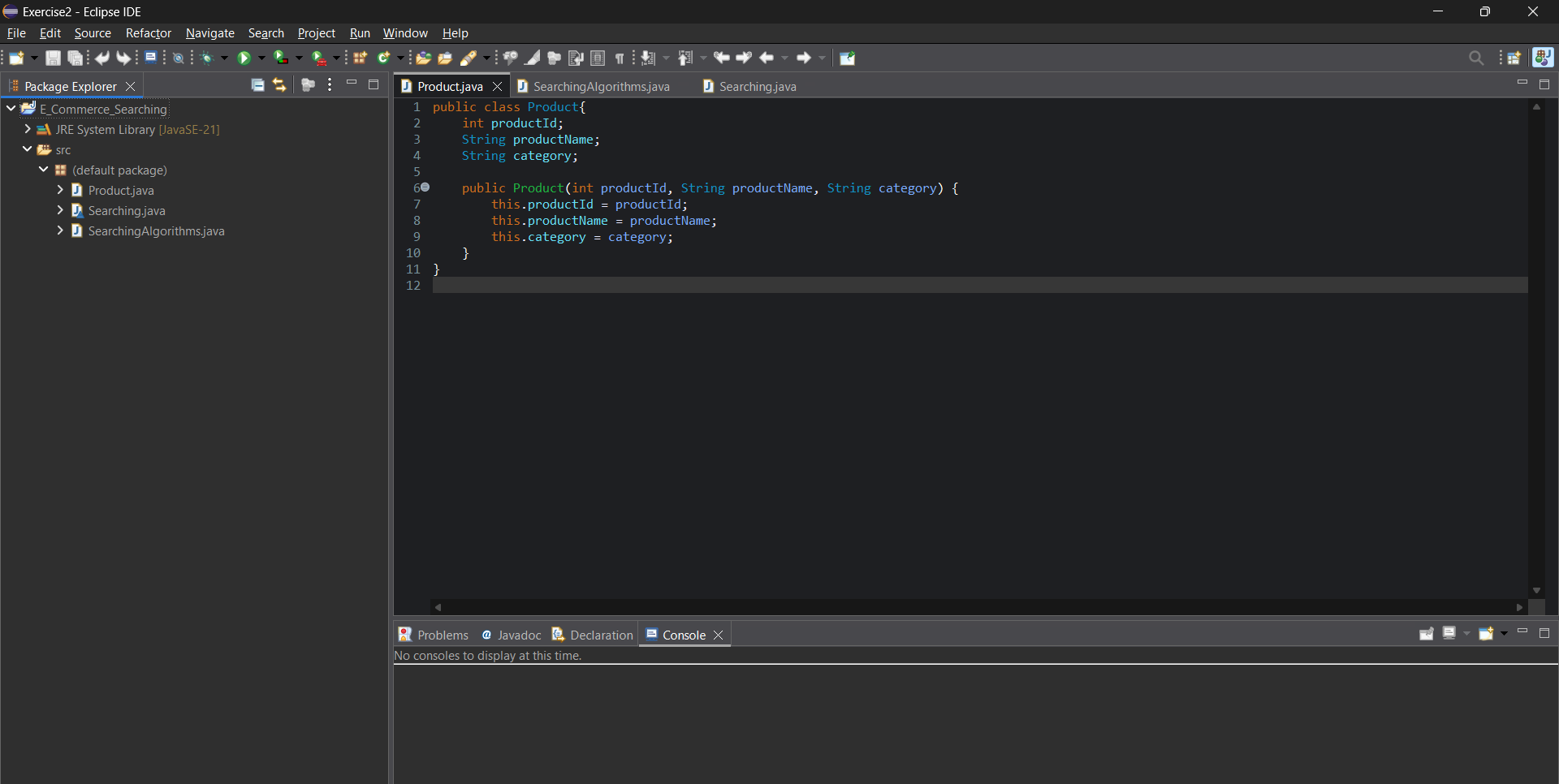
* + f(n) is O(g(n)) if there exists constants c > 0 and n1 > 0 such that
  + f(n) <= c\*g(n) for all n >= n1

The Big-O notation helps in comparing the time complexity or space complexity of different algorithms. It can be used to measure the efficiency of an algorithm. It provides a way to describe how the runtime or space requirements of an algorithm grow with increase in input size.

* Best Case – The best case for a searching algorithm is when it finds the required element in its first comparison. For linear search it occurs when the required element is at the first index and for binary search it occurs when the required element is at the middle of the array.
* Average Case – The average case for a searching algorithm is when it has to compare half of the elements to find the required element.
* Worst Case – The worst case for Linear search is when it has to check all the elements and for Binary Search it is when the element is found by reducing the rage to 0 which is its last iteration. It also occurs when the element is not present.

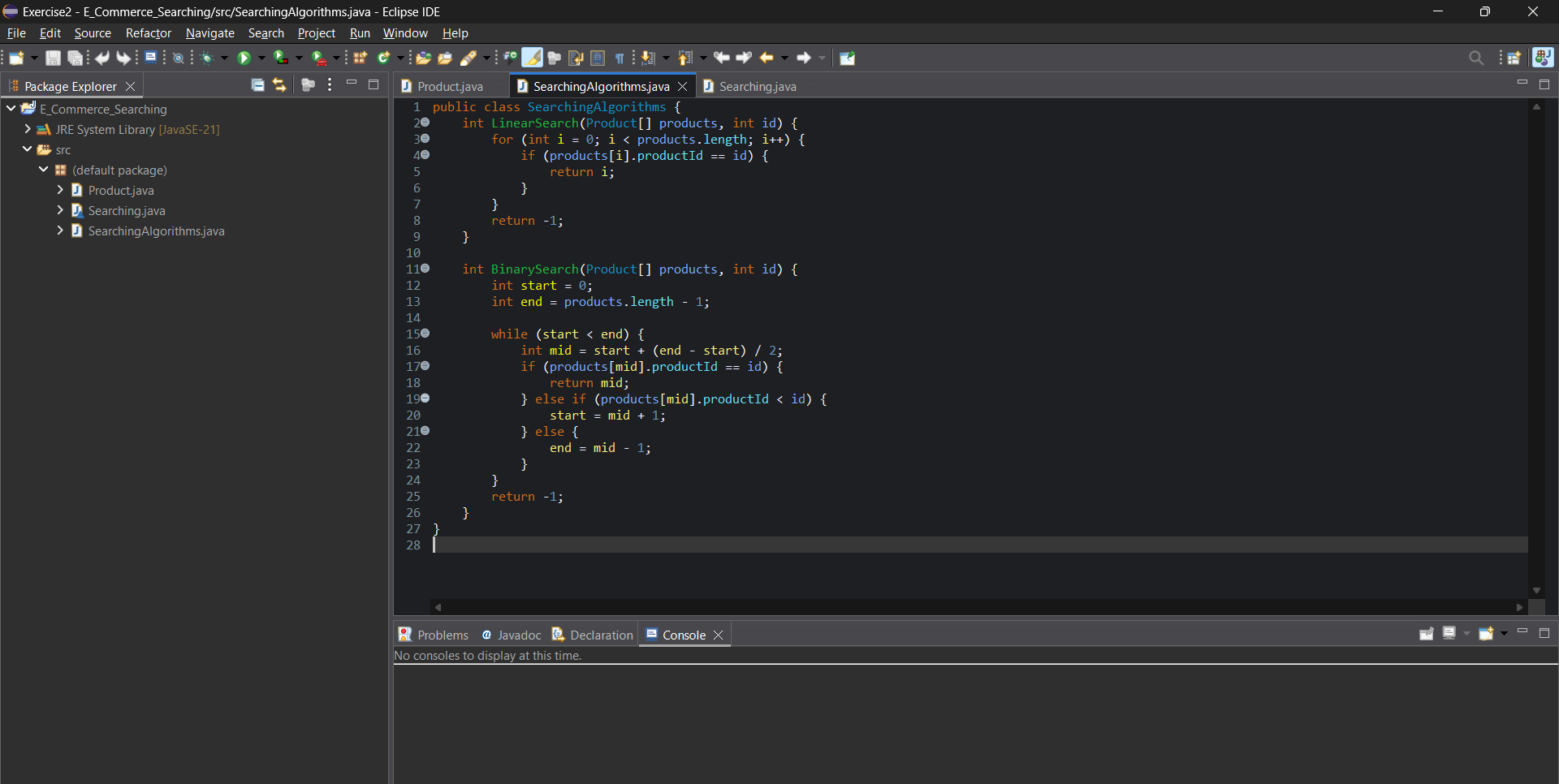
**Setup**

) Product.java

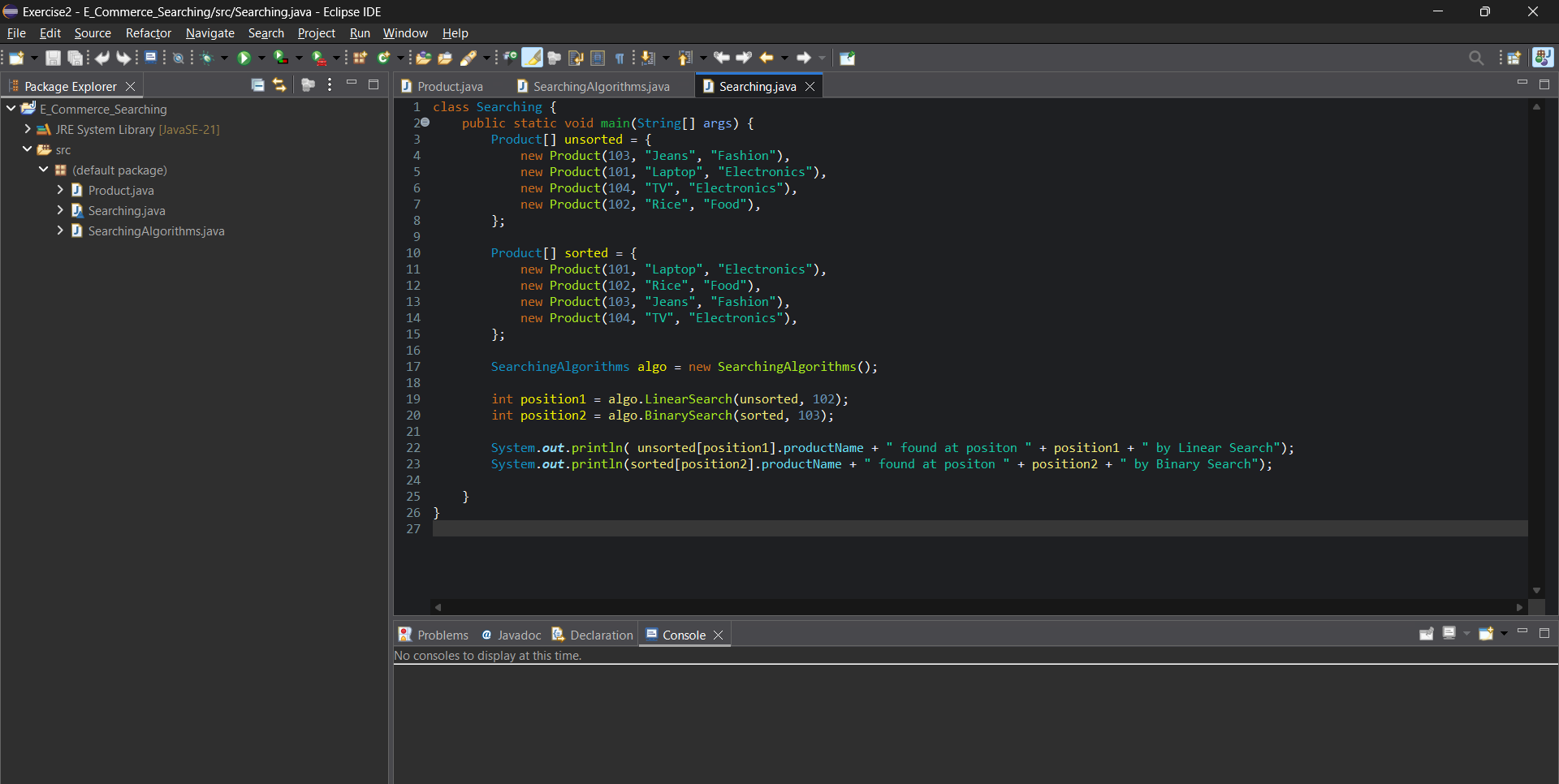


**Implementation**

) SearchingAlgorithms.java

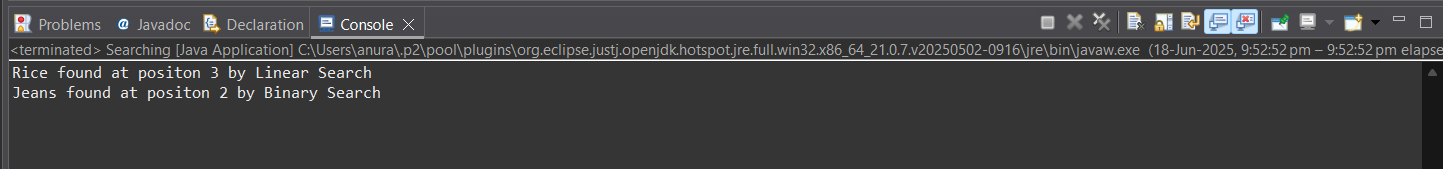


) Searching.java



**Output**

) Terminal



**Analysis**

1. Linear Search has a linear time complexity. Its runtime increases with increase in input size. It can be represented as **O(n)**. It works better with small datasets but does not perform in case of large datasets.
2. Binary Search has a logarithmic time complexity. It can be represented as **O (log n).** It may not work faster with small datasets but works much better with larger datasets, compared to linear search.

For the searching purpose in an e-commerce platform, using **Binary Search** will be the better option as there may be a large number of products from which the desired product has to be searched. Therefore, Binary Search is suitable for our platform.

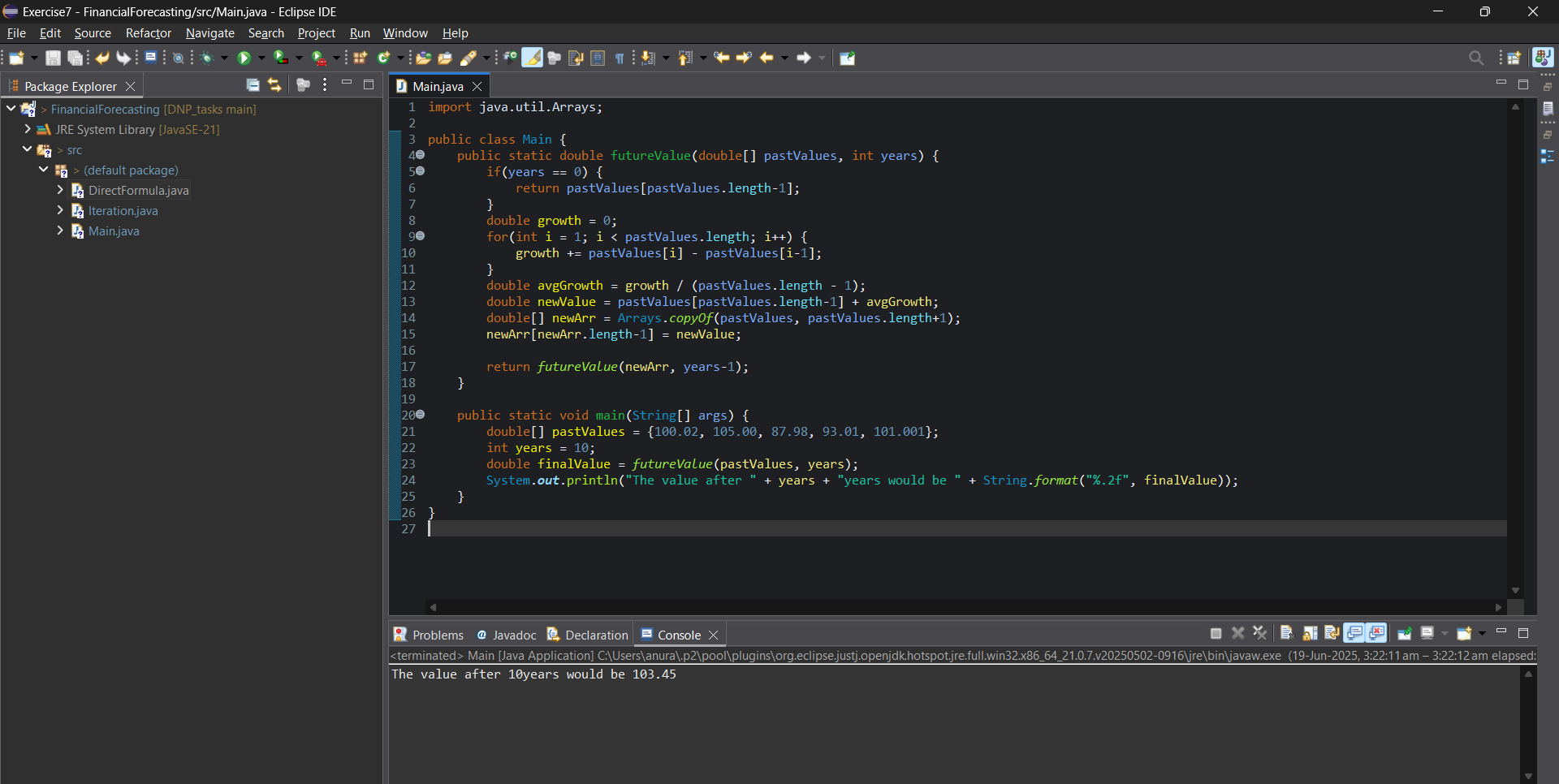
Exercise 7: Financial Forecasting

**Understanding Recursive Algorithms:**

* The scenario in which a function calls itself in its implementation, either directly or indirectly is called recursion and the function is called a recursive function.
* Recursions can simplify certain problems like tower of Hanoi, finding Fibonacci series, calculating factorial of a number, by breaking down the problem into simpler sub-problems similar to the global problem and then executing the sub-problems, this simplifies the problem.

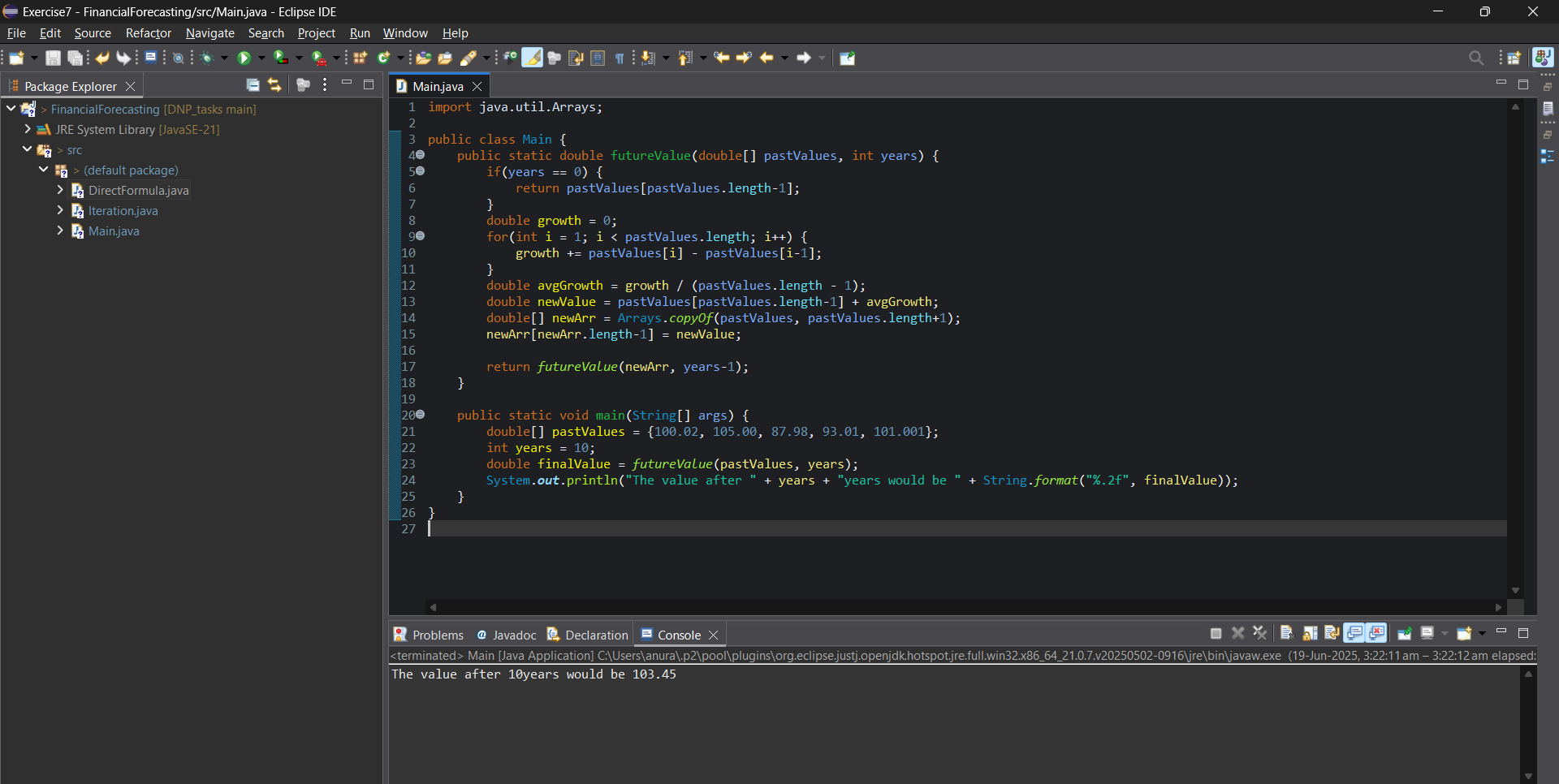
**Setup & Implementation**

) Main.java



**Output**

) Terminal



**Analysis**

1. Initial size of the array = n, number of years to predict = k
2. Every call has
   1. One loop to calculate the growth = **O (present size of array)**
   2. Copies the array into a new array = **O (present size of the array)**
   3. Adds one new element = **O (1)**
   4. Calls itself again
3. If we look at the size of the array at each call we get
   1. At first call size of array is **n**
   2. At second call it is **n+1**
   3. At third it is **n+2**
   4. At fourth it is **n+3** and so on
4. Therefore, at kth call which is the final call, the size of the array would be **n + k – 1**
5. So, if we analyse the time complexity we get,

**T(k) = O(n) + O(n+1) + O(n+2) + … + O (n + k – 1)**

This is an Arithmetic Progression, so the sum can be computed as following,

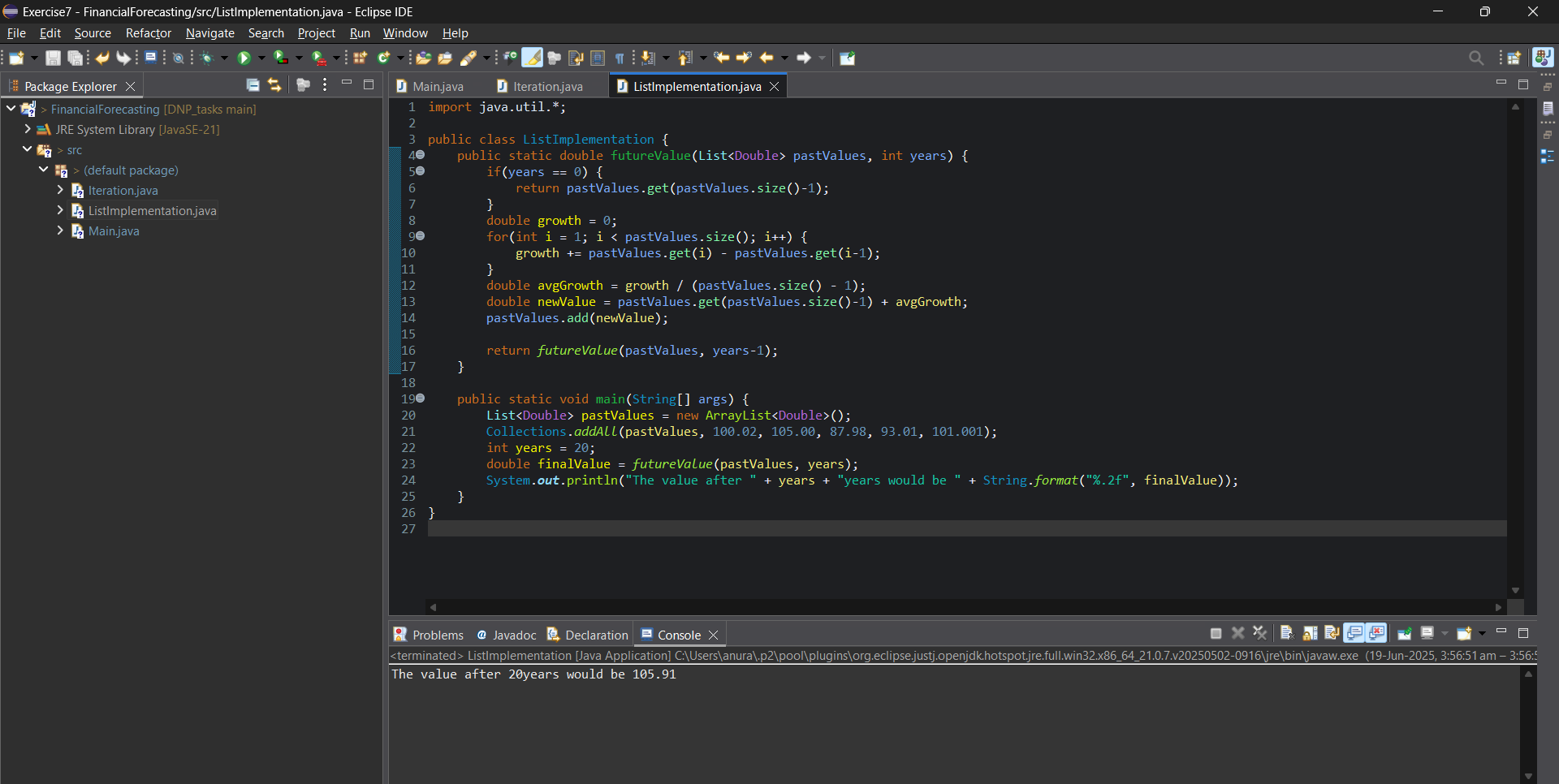
T(k) = {2n + (k – 1) \*1} \*k/2

Or, T(k) = n\*k + k(k-1)/2

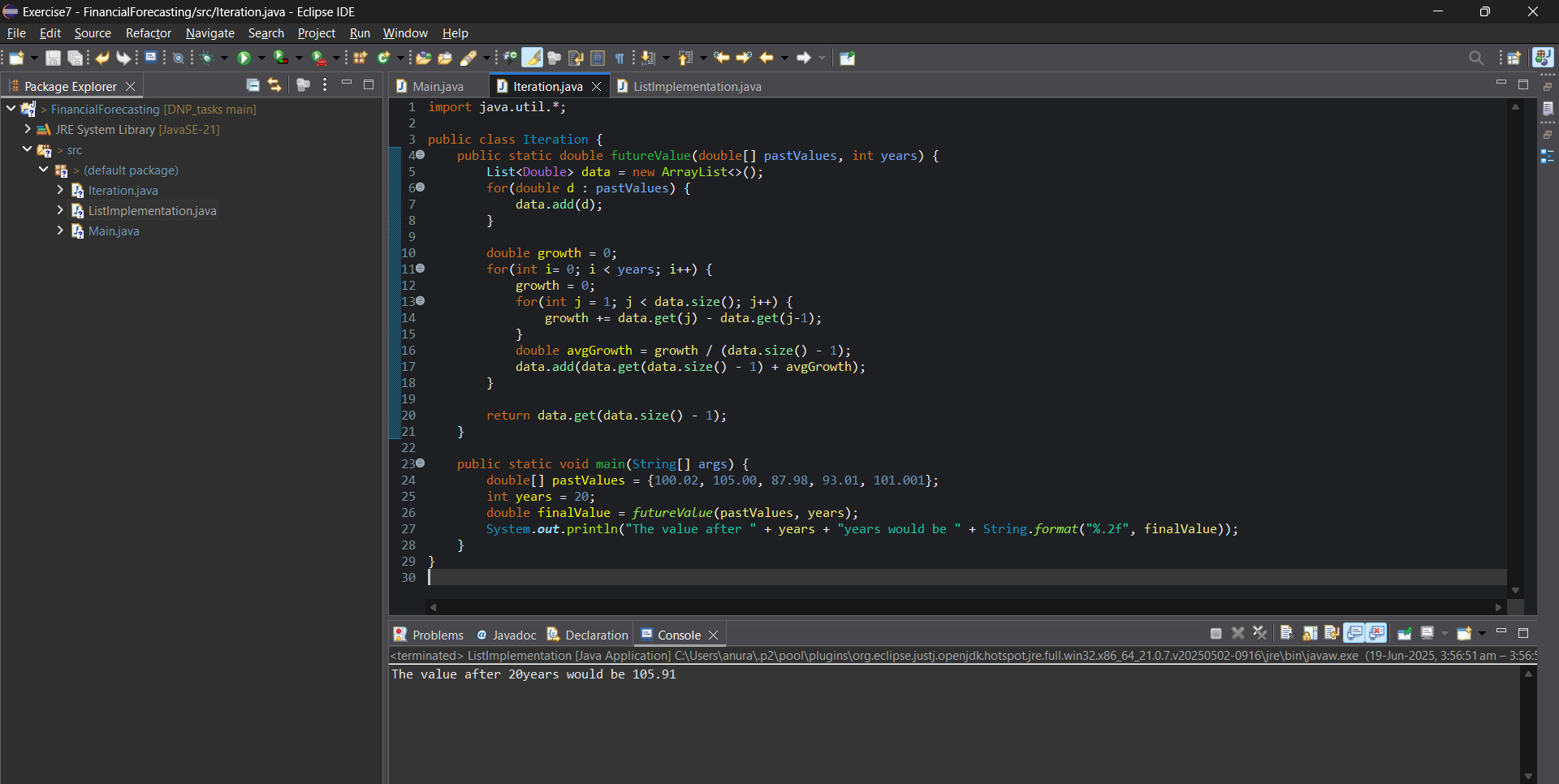
Or, **T(k) = O (nk + k2)**

To avoid the excessive computation, we can go for two other approaches

1. Using a list



1. Using Iterative approach



* Using a list removes the burden of copying the list again and again, by making the size of the array dynamic.
* Theoretically using a list will have same impact on both iterative and recursive approach but using the iterative approach is practically an optimization because everything is under a single function frame, no passing of parameters and returning values, also there is no function call stack.

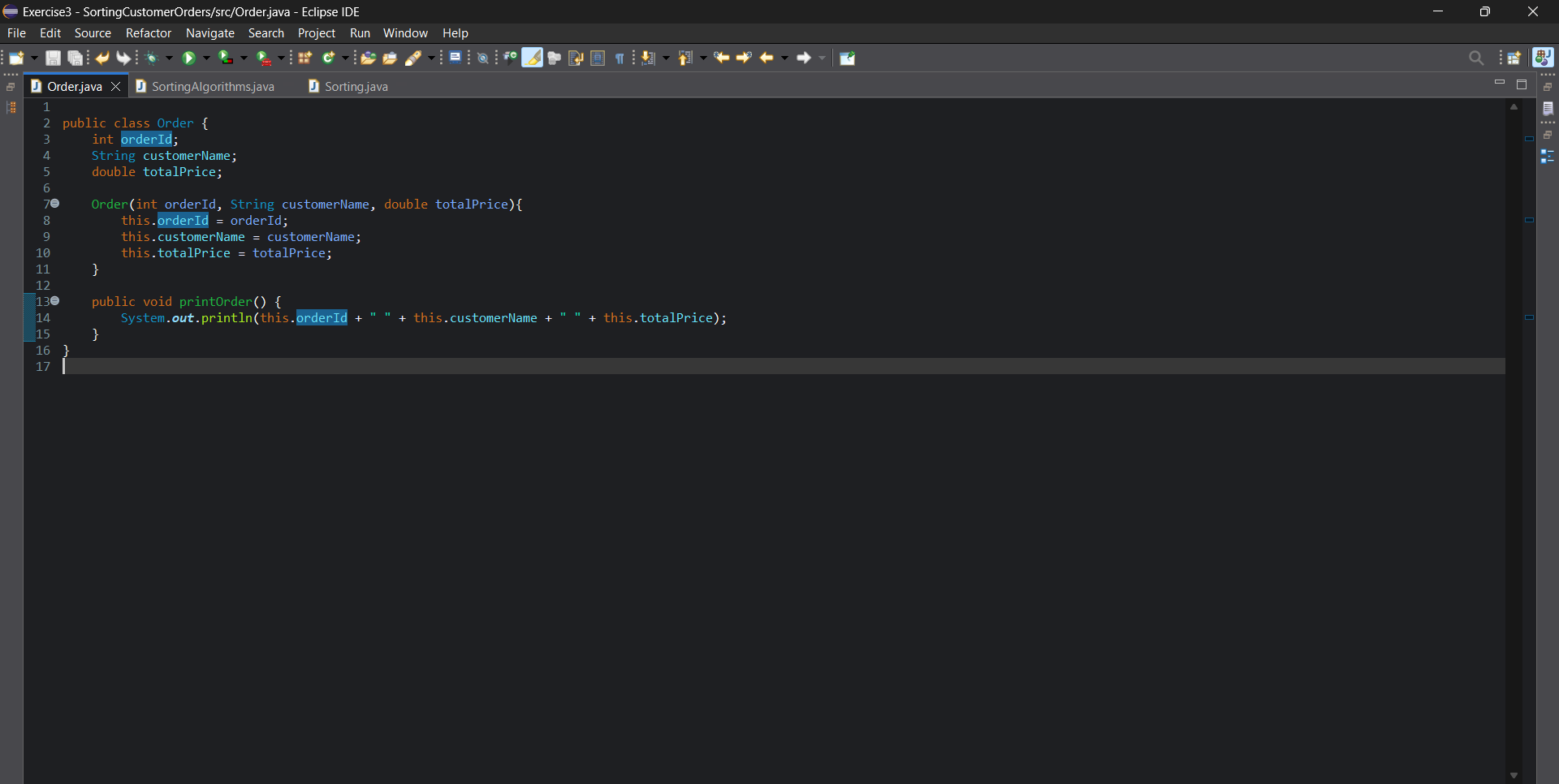
Exercise 3: Sorting Customer Orders

**Understanding Sorting Algorithms**

* Bubble Sort and Insertion Sort algorithms are based on comparing every element and swapping them to find the correct position of one element per pass
* Merge Sort and Quick Sort algorithms are based on Divide and conquer algorithm where the total problem is divided into sub-problems and then combined to form the desired answer.
* In **Bubble Sort**, in every pass we compare each neighboring element and swap them if needed. One element reaches its correct index at every pass so the focusing size of the array reduces by 1 after every pass.
* In **Insertion Sort**, similar to bubble sort we compare every element to each other but we gradually increase the focusing size of the array starting from size=1. In every pass we consider one key element and then traverse the array to find its correct position by swapping positions of elements. The focusing size of the array increases by 1 after every pass.
* In **Merge Sort** we use the divide and conquer algorithm in which we use a recursive function which first divides the array into equal halves until single element arrays are formed and then they are merged and sorted one by one thus forming the desired sorted array at the end of merging. We require some extra sub-arrays to form the sorted full array. Elements from the sub-arrays are compared and accordingly added into the new array to make it sorted.
* In **Quick Sort** also we use divide and conquer algorithm in which we use a recursive function which divides the array. However, here we don’t use extra sub-arrays, instead we use a partitioning function. The partitioning function takes a random pivot element (either starting or ending element or any element from the array), and then sorts the array around that pivot thus finding its correct position. After each such pass the array is divided into sub-arrays consisting of the left elements of the pivot and the elements on the right side of the pivot.

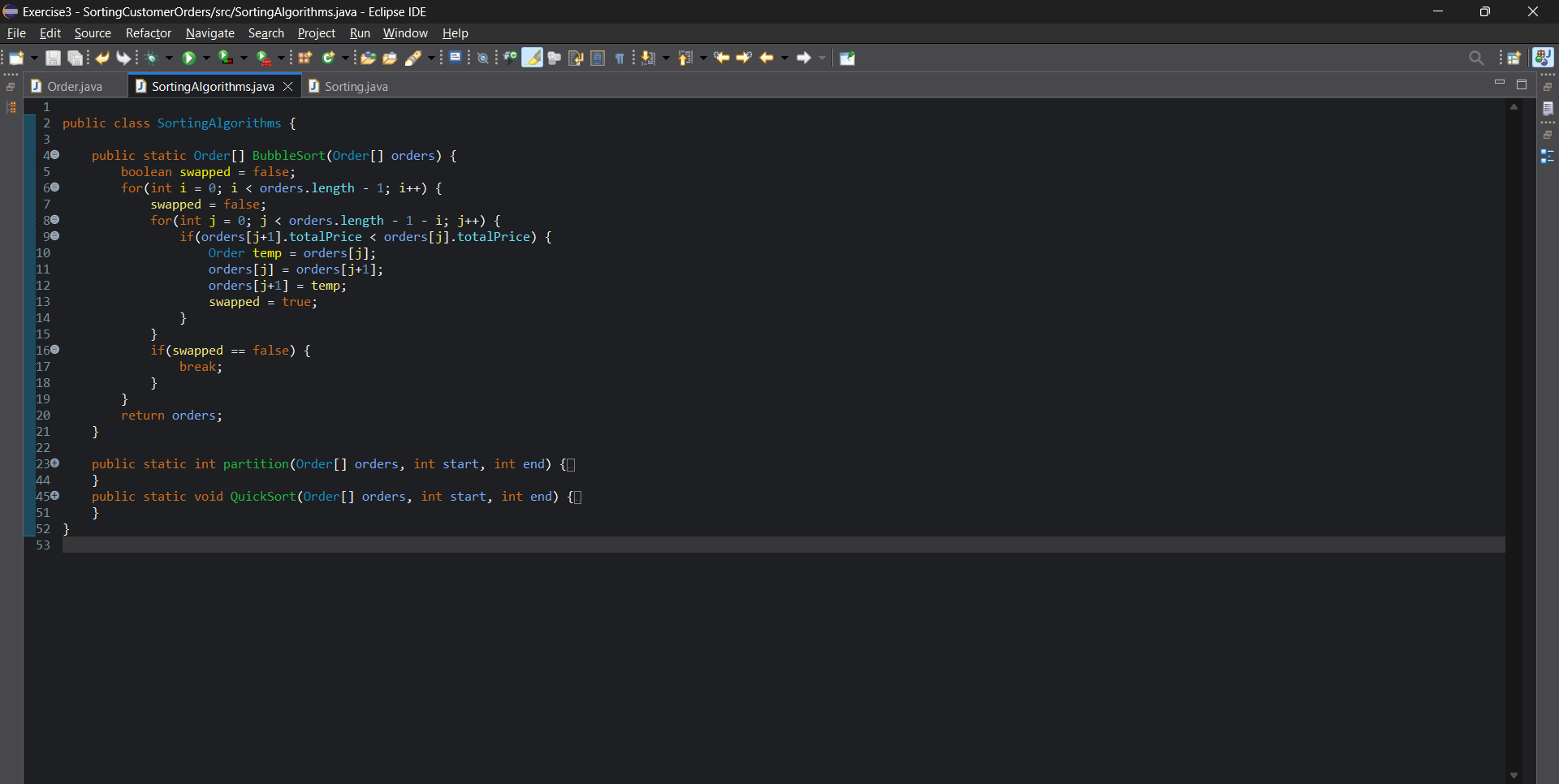
**Setup**

) Order.java

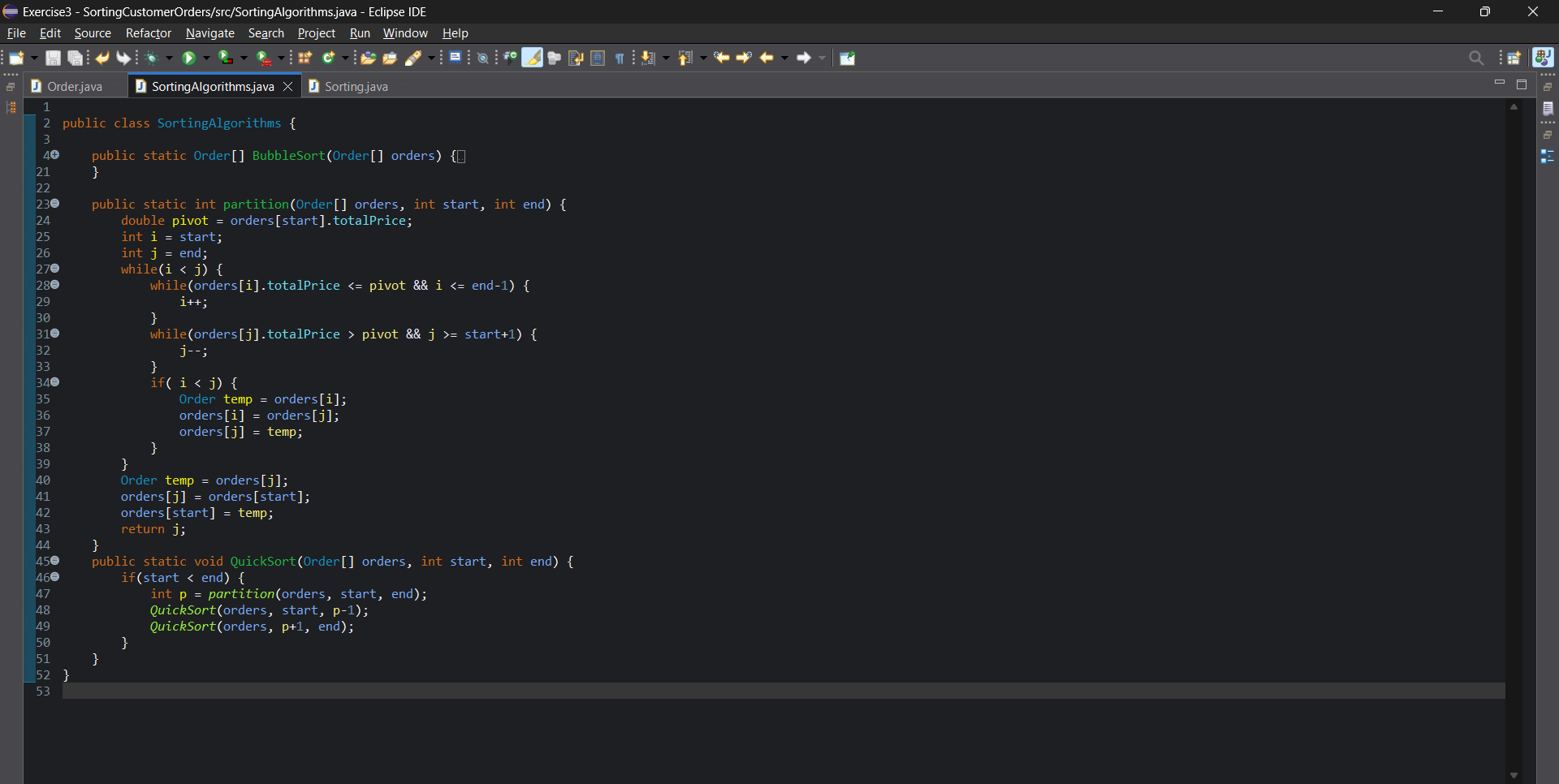


**Implementation**

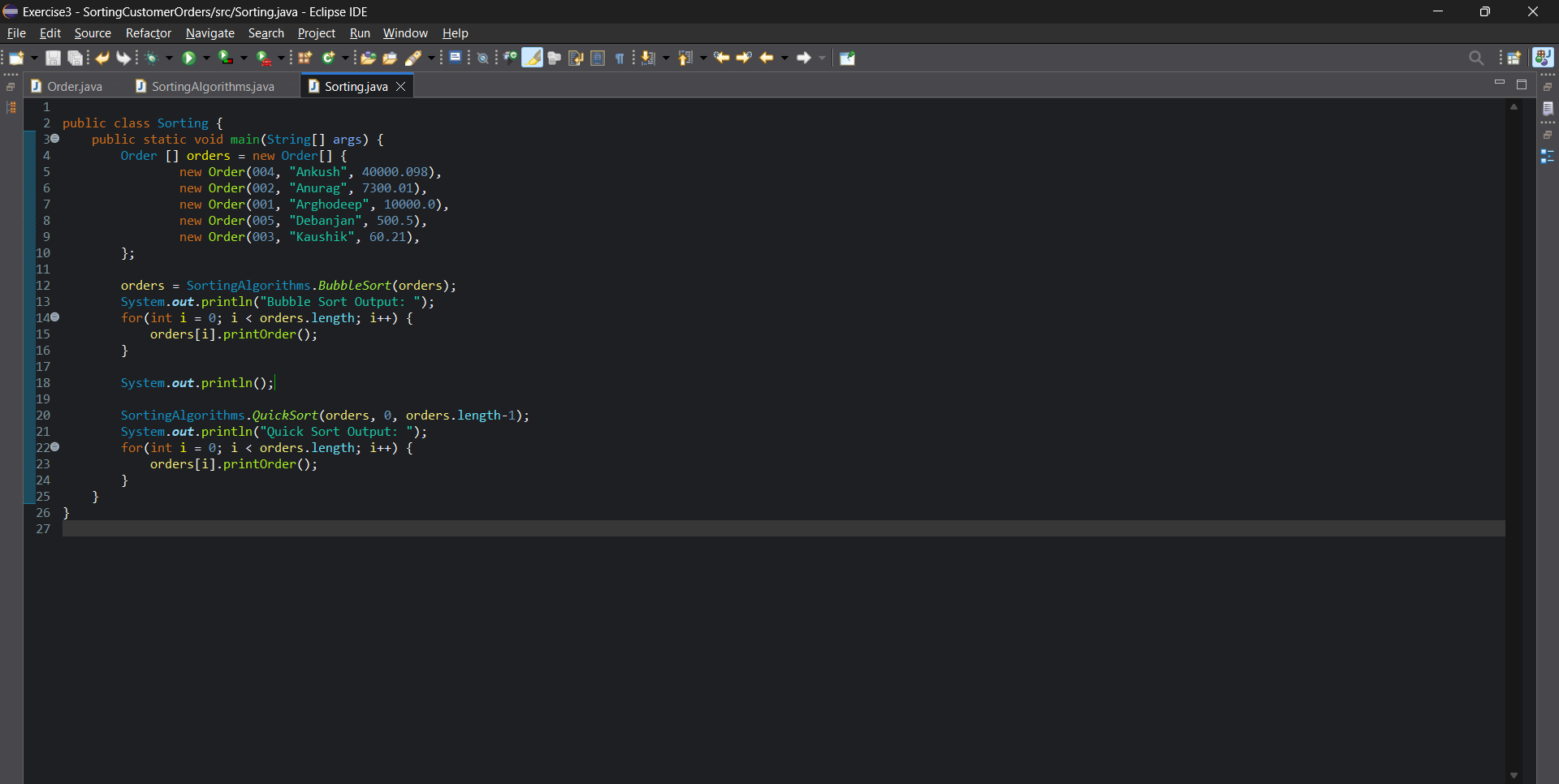
) Bubble Sort



) Quick Sort

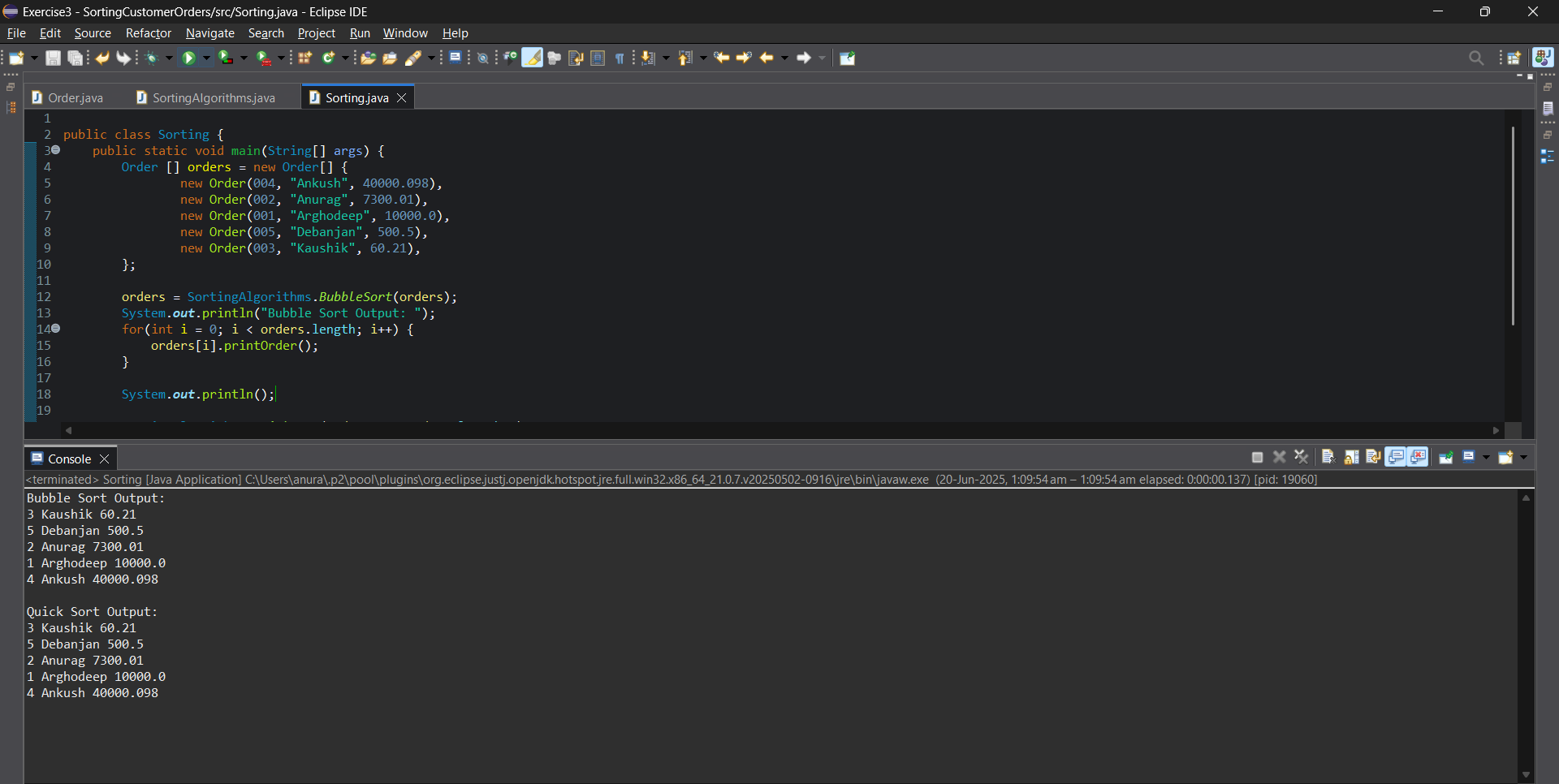


) Sorting



**Output**

) Terminal



**Analysis**

* **Bubble Sort** uses nested loops. There are two loops used one inside another. Its **worst and average case time complexity is O(n2)** and its **best-case time complexity is O(n)** when the array is already sorted.
* **Quick Sort** divides the array into sub-arrays, and also it performs partitioning. Its recurrence relation in best case and average case is

**T(n) = 2T(n/2) + O(n)**

Where O(n) is the complexity of partitioning

On solving the above equation, we get the time complexity of Quick Sort in **best and average cases** as **O (n log n)**

In the **worst case** of quick sort, it divides the array very inefficiently, for example in two halves one including n-1 elements and the other having only 1 element, this occurs when the input array is already sorted. Then the recurrence relation becomes **T(n) = T(n-1) + O(n)** which when evaluated is found to be **O(n2)**

* **Quick Sort is generally quicker** than bubble sort and that is clear from the time complexity analysis as **O (n log n) < O(n2)**.
* **Quick sort is very efficient even on large datasets** but bubble sort on the other hand is very inefficient. Quick sort applies the Divide and Conquer algorithm which makes it much faster than bubble sort which is based on Brute force approach.
* Therefore, **quick sort is faster and more scalable** than bubble sort because of which it is preferred over bubble sort.