- LINEAR SEARCH

```
//Source: https://www.programiz.com/dsa/linear-search
//Big O Complexity: O(n)
//This means the time it takes increases as the array gets bigger.

public class LinearSearch {
    public static int linearSearch(int array[], int x) {
        int n = array.length;

        // Go through array one element at a time
        for (int i = 0; i < n; i++) {
          //if number is found, return
            if (array[i] == x)
                return i;
        }
        //if we reach the end and don’t find it, return -1
        return -1;
    }
```

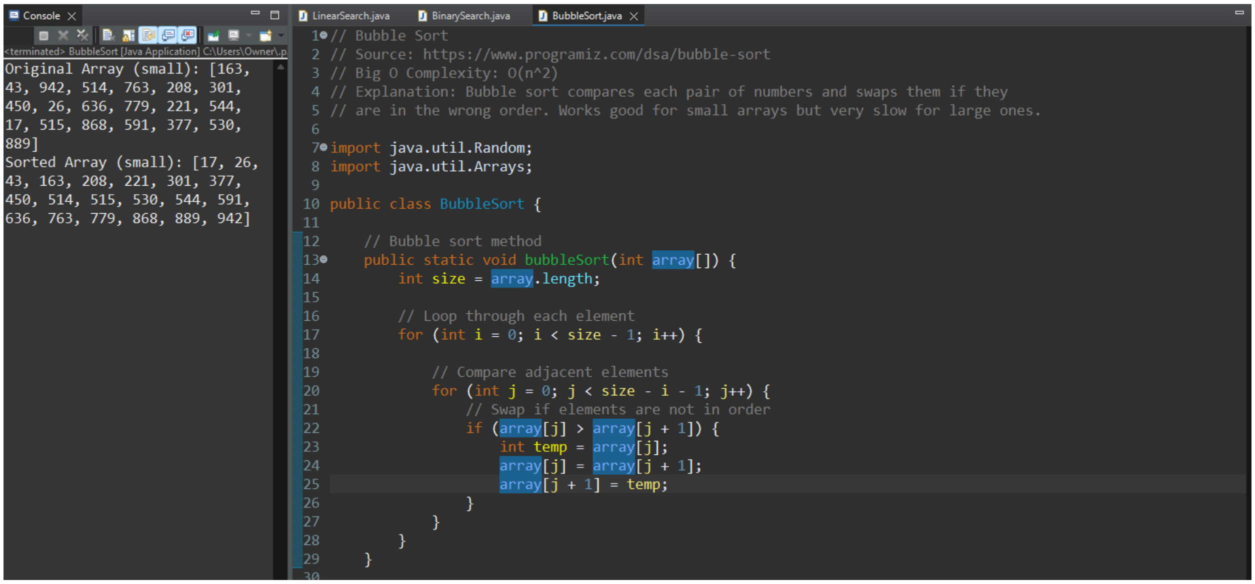
- BINARY SEARCH

```
// BinarySearchLab.java
// Source: https://www.geeksforgeeks.org/dsa/binary-search/
// Big O Complexity: O(log n)
// Explanation: Binary search works on sorted arrays. It divides
// the array in half each time to find the target number. Each step reduces the
//  search by half, making it much faster than linear search for larger arrays.

import java.util.Random;
import java.util.Arrays;

```

```java
public class BinarySearch {

    // Binary Search method
    public static int binarySearch(int arr[], int x) {
        int low = 0, high = arr.length - 1;

        // Keep looking while the search space is not empty
        while (low <= high) {
            int mid = low + (high - low) / 2;

            // Check if x is at mid
            if (arr[mid] == x)
                return mid;

            // If x is bigger, ignore left half
            if (arr[mid] < x)
                low = mid + 1;

            // If x is smaller, ignore right half
            else
                high = mid - 1;
        }

        // If we reach here, x is not in the array
        return -1;
    }
```

- BUBBLE SORT

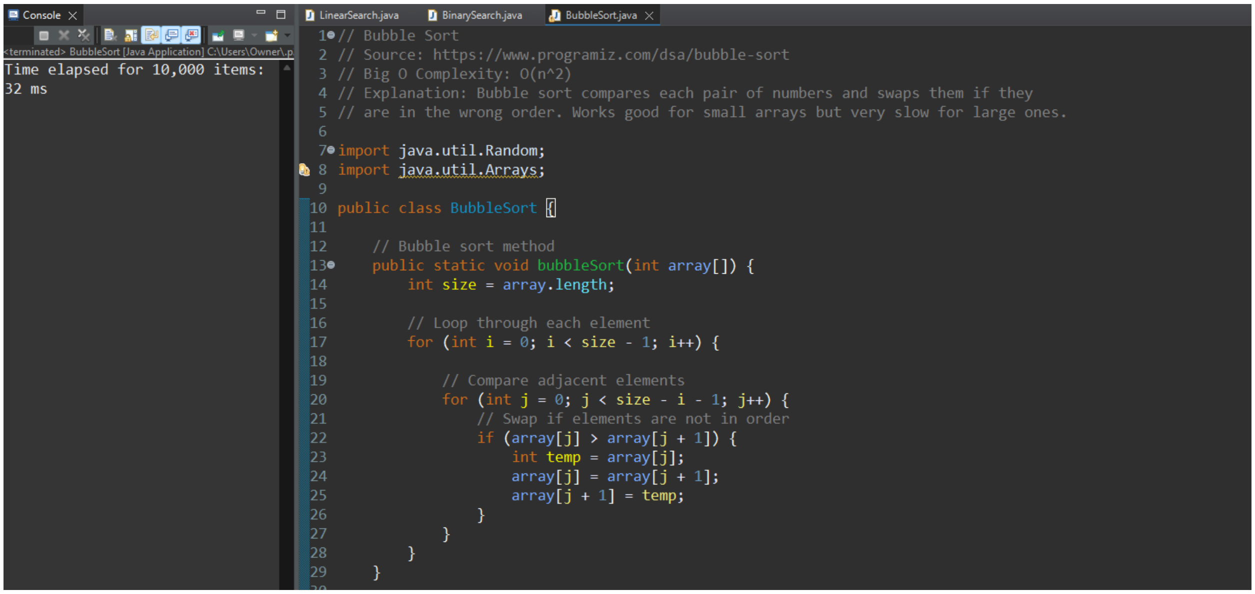
Screenshot 1:


Console
<terminated> BubbleSort [Java Application] C:\Users\Owner\.p
LinearSearch.java
BinarySearch.java
BubbleSort.java

```
Original Array (small): [163,
43, 942, 514, 763, 208, 301,
450, 26, 636, 779, 221, 544,
17, 515, 868, 591, 377, 530,
889]
Sorted Array (small): [17, 26,
43, 163, 208, 221, 301, 377,
450, 514, 515, 530, 544, 591,
636, 763, 779, 868, 889, 942]
```

```java
// Bubble Sort
// Source: https://www.programiz.com/dsa/bubble-sort
// Big O Complexity: O(n^2)
// Explanation: Bubble sort compares each pair of numbers and swaps them if they
// are in the wrong order. Works good for small arrays but very slow for large ones.

import java.util.Random;
import java.util.Arrays;

public class BubbleSort {

    // Bubble sort method
    public static void bubbleSort(int array[]) {
        int size = array.length;

        // Loop through each element
        for (int i = 0; i < size - 1; i++) {

            // Compare adjacent elements
            for (int j = 0; j < size - i - 1; j++) {
                // Swap if elements are not in order
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
```

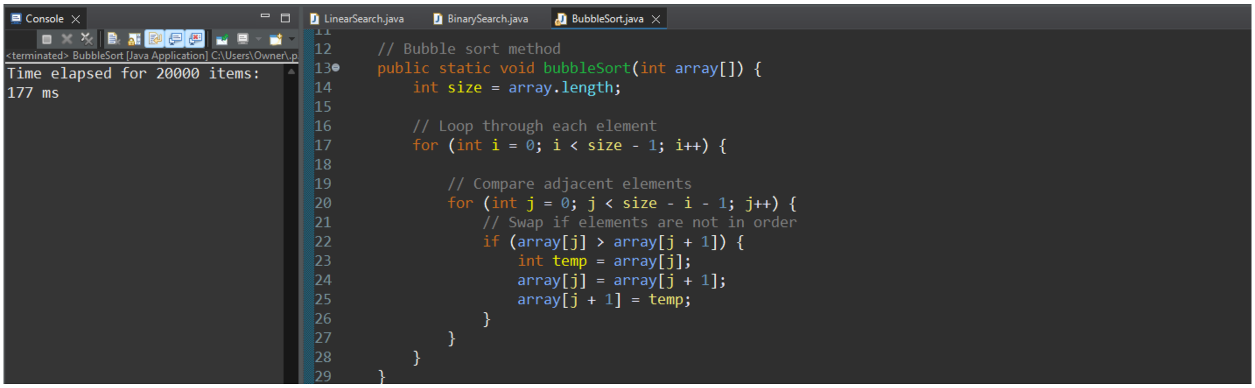
Screenshot 2:


Console
<terminated> BubbleSort [Java Application] C:\Users\Owner\.p
LinearSearch.java
BinarySearch.java
BubbleSort.java

```
Time elapsed for 10,000 items:
32 ms
```

```java
// Bubble Sort
// Source: https://www.programiz.com/dsa/bubble-sort
// Big O Complexity: O(n^2)
// Explanation: Bubble sort compares each pair of numbers and swaps them if they
// are in the wrong order. Works good for small arrays but very slow for large ones.

import java.util.Random;
import java.util.Arrays;

public class BubbleSort {

    // Bubble sort method
    public static void bubbleSort(int array[]) {
        int size = array.length;

        // Loop through each element
        for (int i = 0; i < size - 1; i++) {

            // Compare adjacent elements
            for (int j = 0; j < size - i - 1; j++) {
                // Swap if elements are not in order
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
```

Screenshot 3:

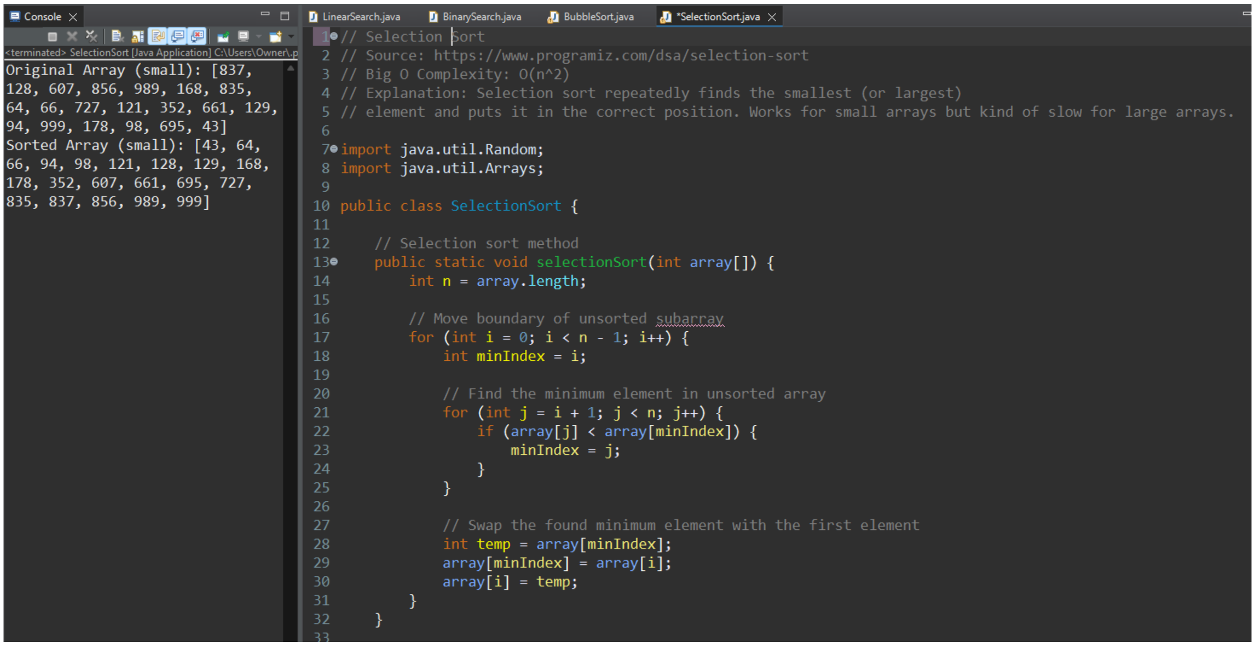


```
<terminated> BubbleSort [Java Application] C:\Users\Owner\.p
Time elapsed for 20000 items:
177 ms
```

```java
    // Bubble sort method
    public static void bubbleSort(int array[]) {
        int size = array.length;

        // Loop through each element
        for (int i = 0; i < size - 1; i++) {

            // Compare adjacent elements
            for (int j = 0; j < size - i - 1; j++) {
                // Swap if elements are not in order
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
```

- SELECTION SORT

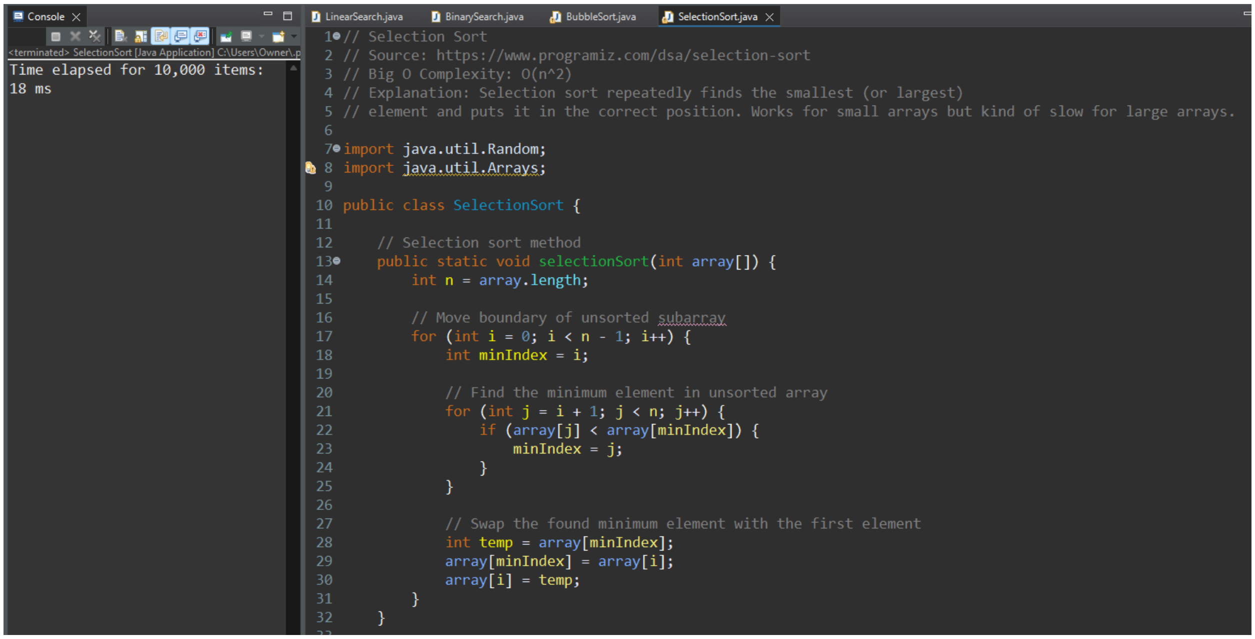
Screenshot 4:



```
<terminated> SelectionSort [Java Application] C:\Users\Owner\.p
Original Array (small): [837,
128, 607, 856, 989, 168, 835,
64, 66, 727, 121, 352, 661, 129,
94, 999, 178, 98, 695, 43]
Sorted Array (small): [43, 64,
66, 94, 98, 121, 128, 129, 168,
178, 352, 607, 661, 695, 727,
835, 837, 856, 989, 999]
```

```java
// Selection Sort
// Source: https://www.programiz.com/dsa/selection-sort
// Big O Complexity: O(n^2)
// Explanation: Selection sort repeatedly finds the smallest (or largest)
// element and puts it in the correct position. Works for small arrays but kind of slow for large arrays.

import java.util.Random;
import java.util.Arrays;

public class SelectionSort {

    // Selection sort method
    public static void selectionSort(int array[]) {
        int n = array.length;

        // Move boundary of unsorted subarray
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;

            // Find the minimum element in unsorted array
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum element with the first element
            int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
    }
```

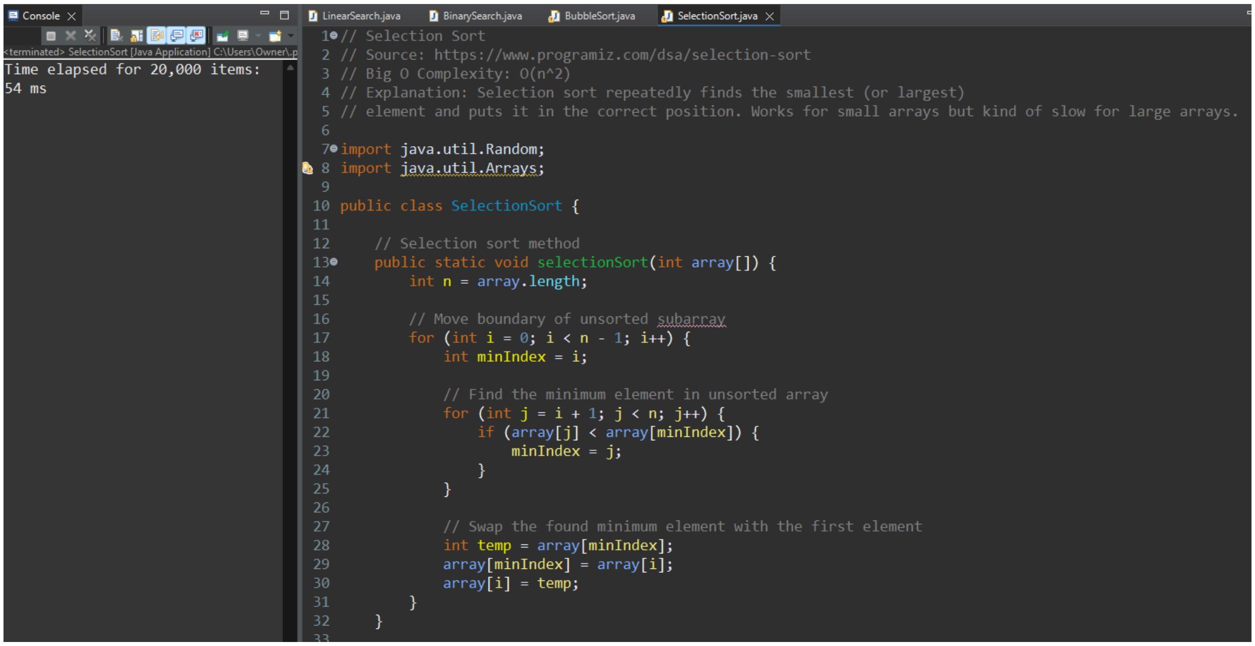
Screenshot 5:



```
<terminated> SelectionSort [Java Application] C:\Users\Owner\.p
Time elapsed for 10,000 items:
18 ms
```

```java
// Selection Sort
// Source: https://www.programiz.com/dsa/selection-sort
// Big O Complexity: O(n^2)
// Explanation: Selection sort repeatedly finds the smallest (or largest)
// element and puts it in the correct position. Works for small arrays but kind of slow for large arrays.

import java.util.Random;
import java.util.Arrays;

public class SelectionSort {

    // Selection sort method
    public static void selectionSort(int array[]) {
        int n = array.length;

        // Move boundary of unsorted subarray
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;

            // Find the minimum element in unsorted array
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum element with the first element
            int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
    }
```

Screenshot 6:

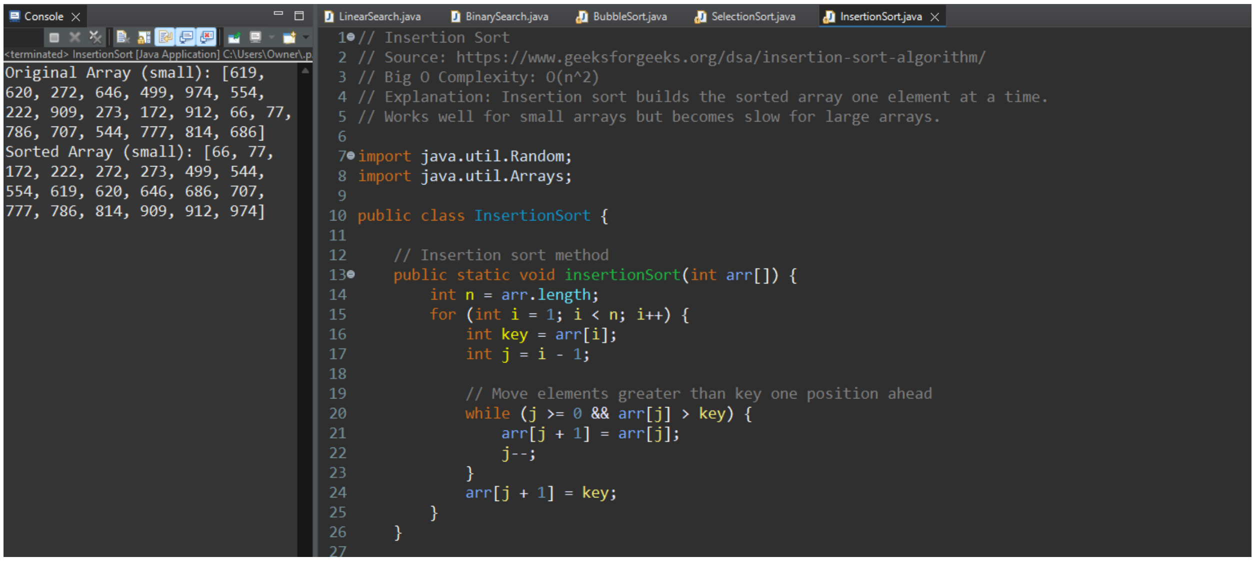


```
<terminated> SelectionSort [Java Application] C:\Users\Owner\.p
Time elapsed for 20,000 items:
54 ms
```

```java
// Selection Sort
// Source: https://www.programiz.com/dsa/selection-sort
// Big O Complexity: O(n^2)
// Explanation: Selection sort repeatedly finds the smallest (or largest)
// element and puts it in the correct position. Works for small arrays but kind of slow for large arrays.

import java.util.Random;
import java.util.Arrays;

public class SelectionSort {

    // Selection sort method
    public static void selectionSort(int array[]) {
        int n = array.length;

        // Move boundary of unsorted subarray
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;

            // Find the minimum element in unsorted array
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum element with the first element
            int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
    }
```

- INSERTION SORT

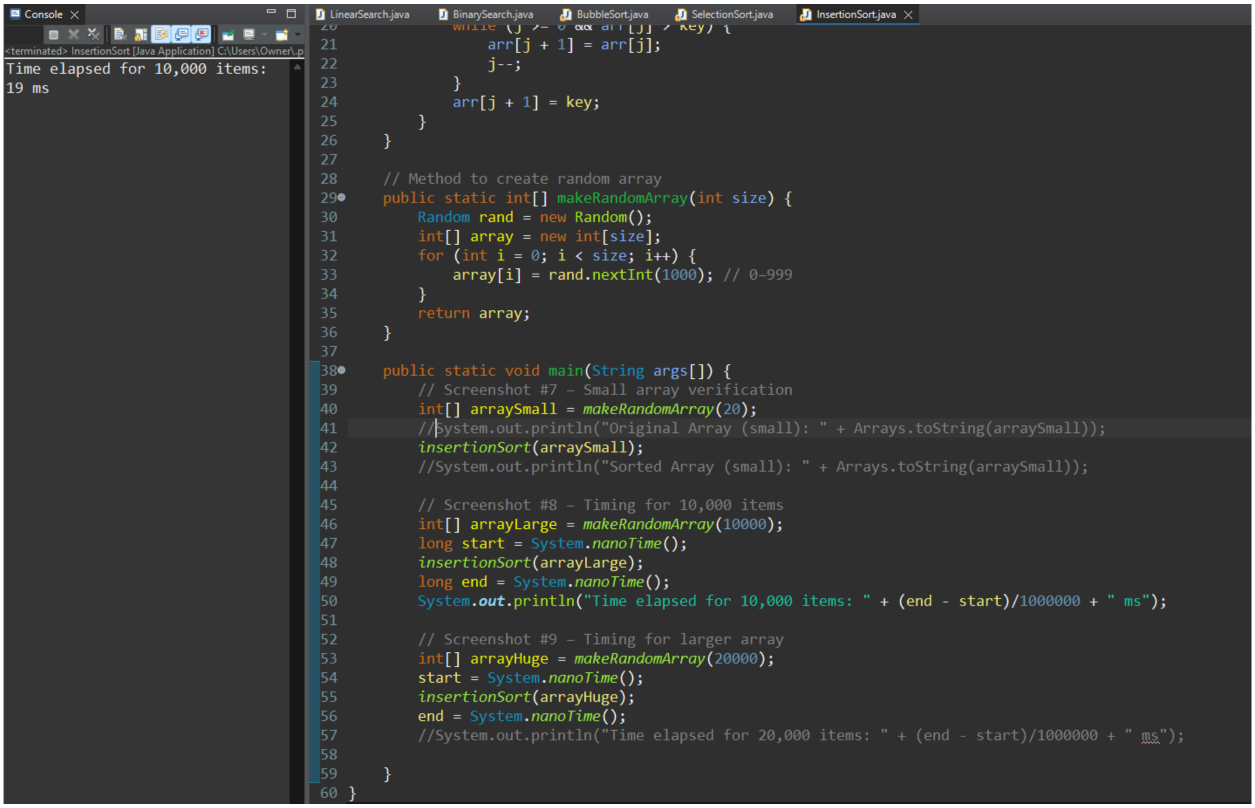
Screenshot 7:



```
<terminated> InsertionSort [Java Application] C:\Users\Owner\.p
Original Array (small): [619,
620, 272, 646, 499, 974, 554,
222, 909, 273, 172, 912, 66, 77,
786, 707, 544, 777, 814, 686]
Sorted Array (small): [66, 77,
172, 222, 272, 273, 499, 544,
554, 619, 620, 646, 686, 707,
777, 786, 814, 909, 912, 974]
```

```java
// Insertion Sort
// Source: https://www.geeksforgeeks.org/dsa/insertion-sort-algorithm/
// Big O Complexity: O(n^2)
// Explanation: Insertion sort builds the sorted array one element at a time.
// Works well for small arrays but becomes slow for large arrays.

import java.util.Random;
import java.util.Arrays;

public class InsertionSort {

    // Insertion sort method
    public static void insertionSort(int arr[]) {
        int n = arr.length;
        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;

            // Move elements greater than key one position ahead
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }
```

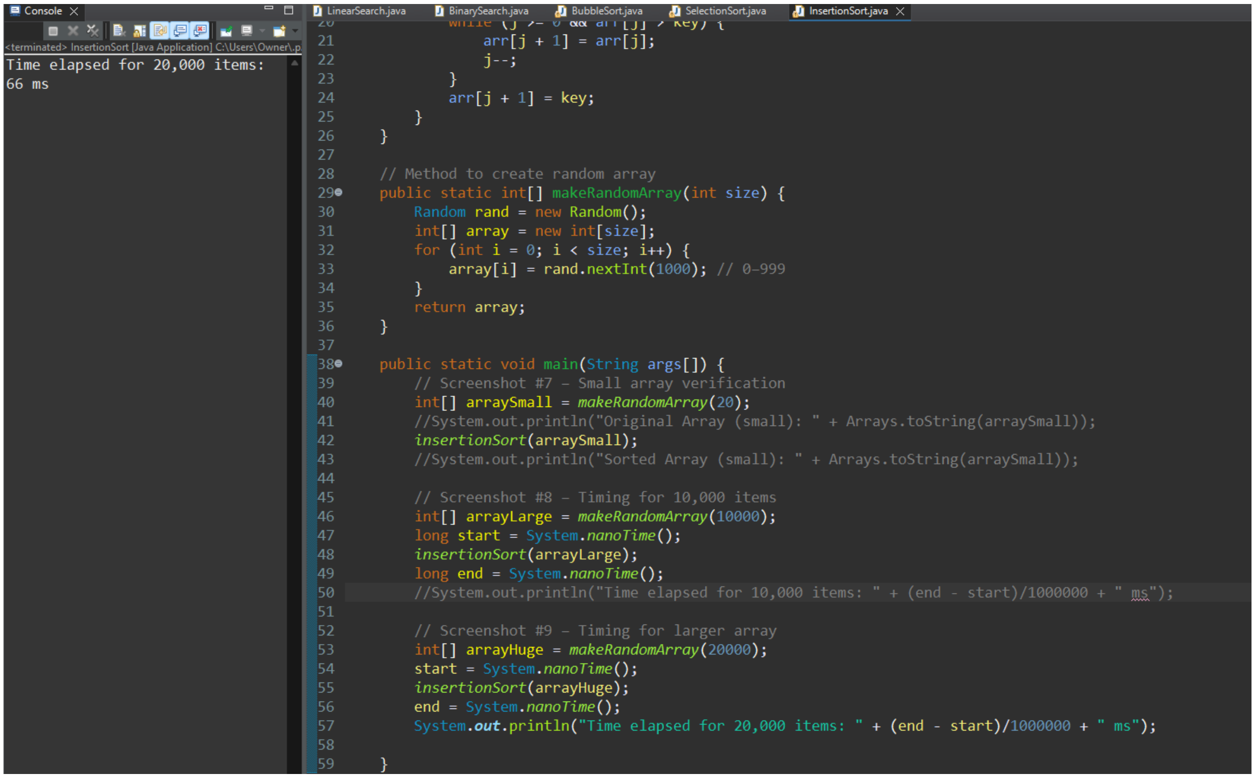
Screenshot 8:



```
<terminated> InsertionSort [Java Application] C:\Users\Owner\.p
Time elapsed for 10,000 items:
19 ms
```

```java
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    // Method to create random array
    public static int[] makeRandomArray(int size) {
        Random rand = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = rand.nextInt(1000); // 0-999
        }
        return array;
    }

    public static void main(String args[]) {
        // Screenshot #7 - Small array verification
        int[] arraySmall = makeRandomArray(20);
        //System.out.println("Original Array (small): " + Arrays.toString(arraySmall));
        insertionSort(arraySmall);
        //System.out.println("Sorted Array (small): " + Arrays.toString(arraySmall));

        // Screenshot #8 - Timing for 10,000 items
        int[] arrayLarge = makeRandomArray(10000);
        long start = System.nanoTime();
        insertionSort(arrayLarge);
        long end = System.nanoTime();
        System.out.println("Time elapsed for 10,000 items: " + (end - start)/1000000 + " ms");

        // Screenshot #9 - Timing for larger array
        int[] arrayHuge = makeRandomArray(20000);
        start = System.nanoTime();
        insertionSort(arrayHuge);
        end = System.nanoTime();
        //System.out.println("Time elapsed for 20,000 items: " + (end - start)/1000000 + " ms");

    }
}
```

Screenshot 9:



```
<terminated> InsertionSort [Java Application] C:\Users\Owner\.p
Time elapsed for 20,000 items:
66 ms
```

```java
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    // Method to create random array
    public static int[] makeRandomArray(int size) {
        Random rand = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = rand.nextInt(1000); // 0-999
        }
        return array;
    }

    public static void main(String args[]) {
        // Screenshot #7 - Small array verification
        int[] arraySmall = makeRandomArray(20);
        //System.out.println("Original Array (small): " + Arrays.toString(arraySmall));
        insertionSort(arraySmall);
        //System.out.println("Sorted Array (small): " + Arrays.toString(arraySmall));

        // Screenshot #8 - Timing for 10,000 items
        int[] arrayLarge = makeRandomArray(10000);
        long start = System.nanoTime();
        insertionSort(arrayLarge);
        long end = System.nanoTime();
        //System.out.println("Time elapsed for 10,000 items: " + (end - start)/1000000 + " ms");

        // Screenshot #9 - Timing for larger array
        int[] arrayHuge = makeRandomArray(20000);
        start = System.nanoTime();
        insertionSort(arrayHuge);
        end = System.nanoTime();
        System.out.println("Time elapsed for 20,000 items: " + (end - start)/1000000 + " ms");

    }
```

- SUMMARY AND CONCLUSIONS
    - For this lab, I tested bubble selection, and insertion sort. On small arrays, all three sorted correctly and quickly, but with larger arrays, timing shows big differences. Bubble sorts were the slowest, then selection then insertions being the fastest. All three algorithms have a Big O complexity of O(n^2), meaning their time grows quickly as the array gets bigger. Using arrays with 10,000 elements or more was necessary to see some differences and see how the algorithms efficiency affects performance.