Credit Name: CSE 3110 Iterative Algorithms 1

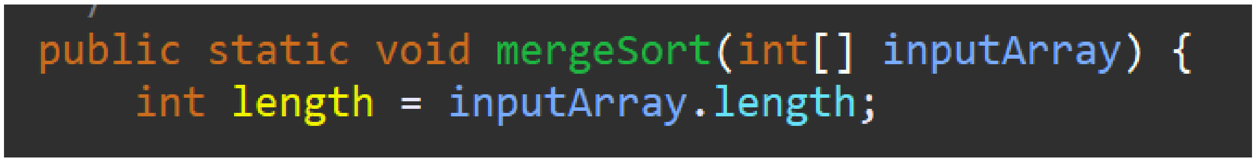
Assignment: ternarySearch

How has your program changed from planning to coding to now? Please Explain

This program consists of 3 classes: a search class which contains the ternary search method, a ternaryMergeSort class which contains a mergeSort method for this program and lastly a ternarySearchLocations class which is a tester.

This main tester class should operate as follows:
Should randomly generate an array of integers. It should sort them in ascending order and display them. After it should prompt the user for a number to find the position of. It should then find that position using a ternarySearch method.

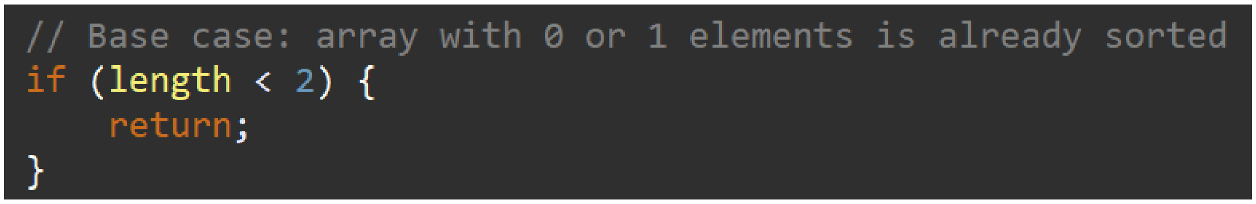
I started by defining the mergeSort method. While any sort algorithm could work the mergeSort is among the most efficient.



```
public static void mergeSort(int[] inputArray) {
    int length = inputArray.length;
```

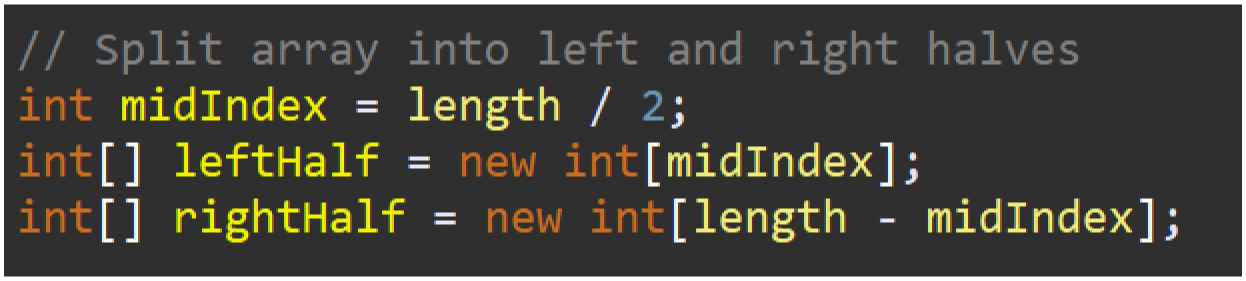
In our method we are not returning anything and we want to call this method by different classes leading to it being public and returning void.

We have an intArray that is a parameter and is the list we will sort.



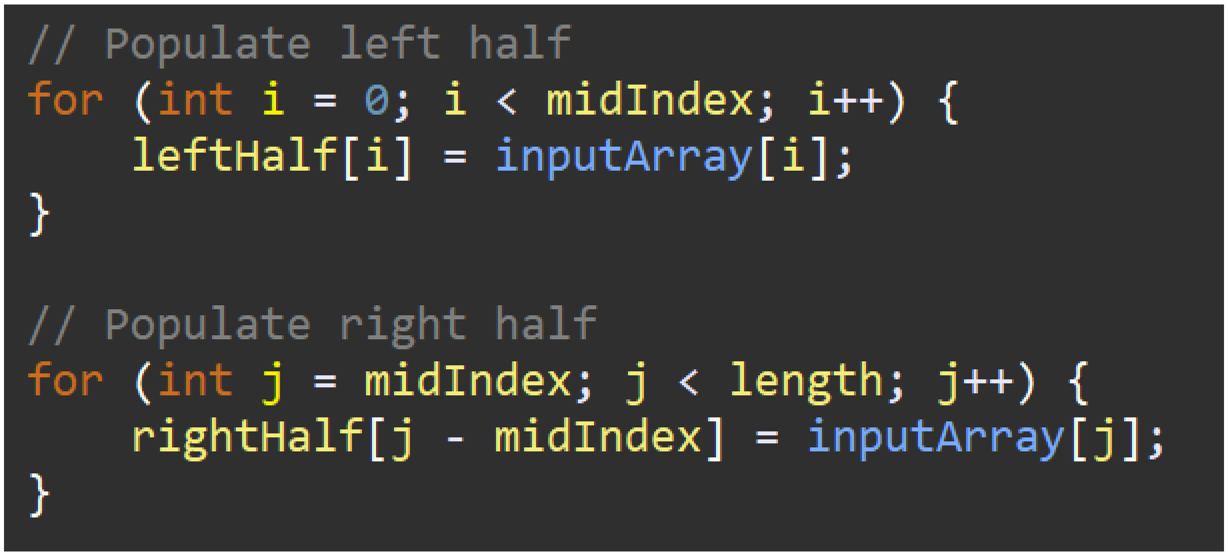
```
// Base case: array with 0 or 1 elements is already sorted
if (length < 2) {
    return;
}
```

If the list has no elements or only 1 element then it is already sorted. And we end this method immediately.



```
// Split array into left and right halves
int midIndex = length / 2;
int[] leftHalf = new int[midIndex];
int[] rightHalf = new int[length - midIndex];
```

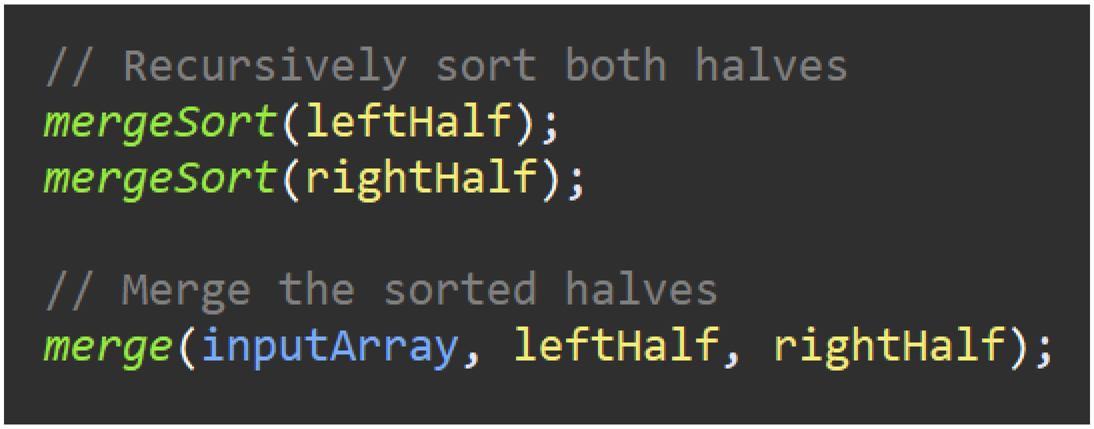
we then define the middle index by dividing by the number of elements by 2. We than define 2 new arrays, one for the left half and another for the right half. The left half has all elements of the list up to the midpoint. And the right half has the remaining elements. The reason we cannot define the rightHalf to also have midIndex elements is to account for lists with odd number of elements.



```
// Populate left half
for (int i = 0; i < midIndex; i++) {
    leftHalf[i] = inputArray[i];
}

// Populate right half
for (int j = midIndex; j < length; j++) {
    rightHalf[j - midIndex] = inputArray[j];
}
```

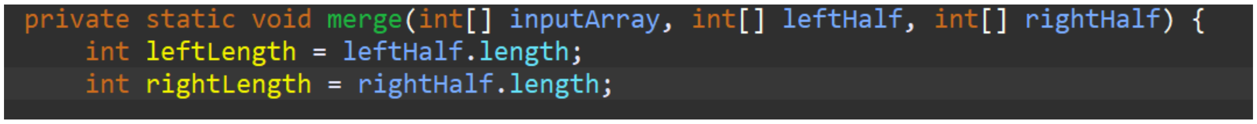
Now we use for loops to add all elements from the original array to its relevant half. This is controlled by the bounds of our for-loop. The index for right half is corrected to start from 0 while input array starts from after the midIndex (b/c the index-system starts from 0).



```java
// Recursively sort both halves
mergeSort(leftHalf);
mergeSort(rightHalf);

// Merge the sorted halves
merge(inputArray, leftHalf, rightHalf);
```

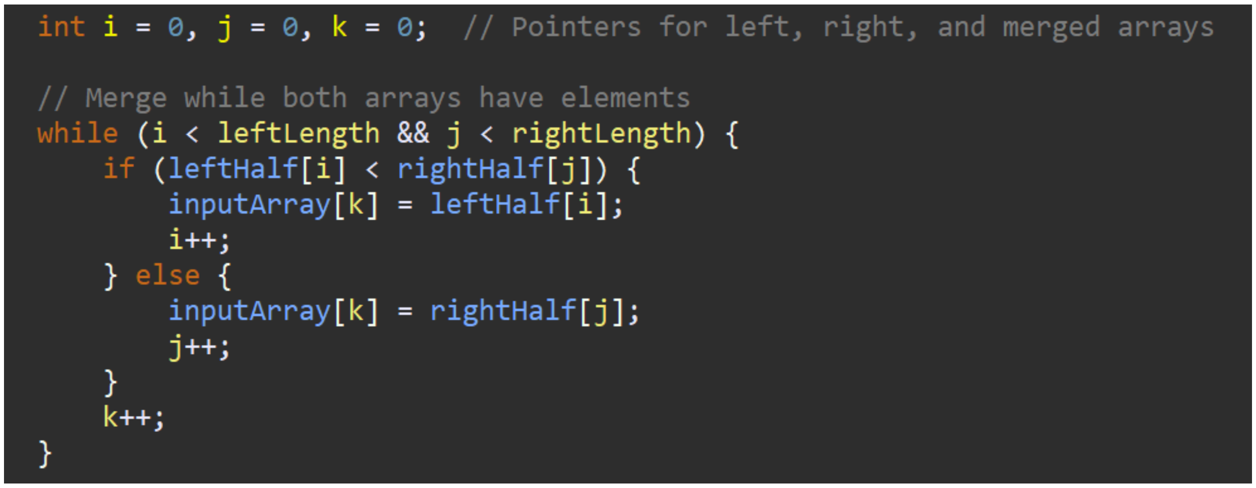
Then we recursively sort both halves by breaking them down into smaller halves. Until they are single elements. Then we run a merge method which combines and sorts the halves until we have recombined AND sorted the entire list.



```java
private static void merge(int[] inputArray, int[] leftHalf, int[] rightHalf) {
    int leftLength = leftHalf.length;
    int rightLength = rightHalf.length;
```

The merge method is defined with 3 parameters; the inputArray we want to override with a sorted version and the left and right halves to be merged together.

Then we define values for our left half length and right half length.

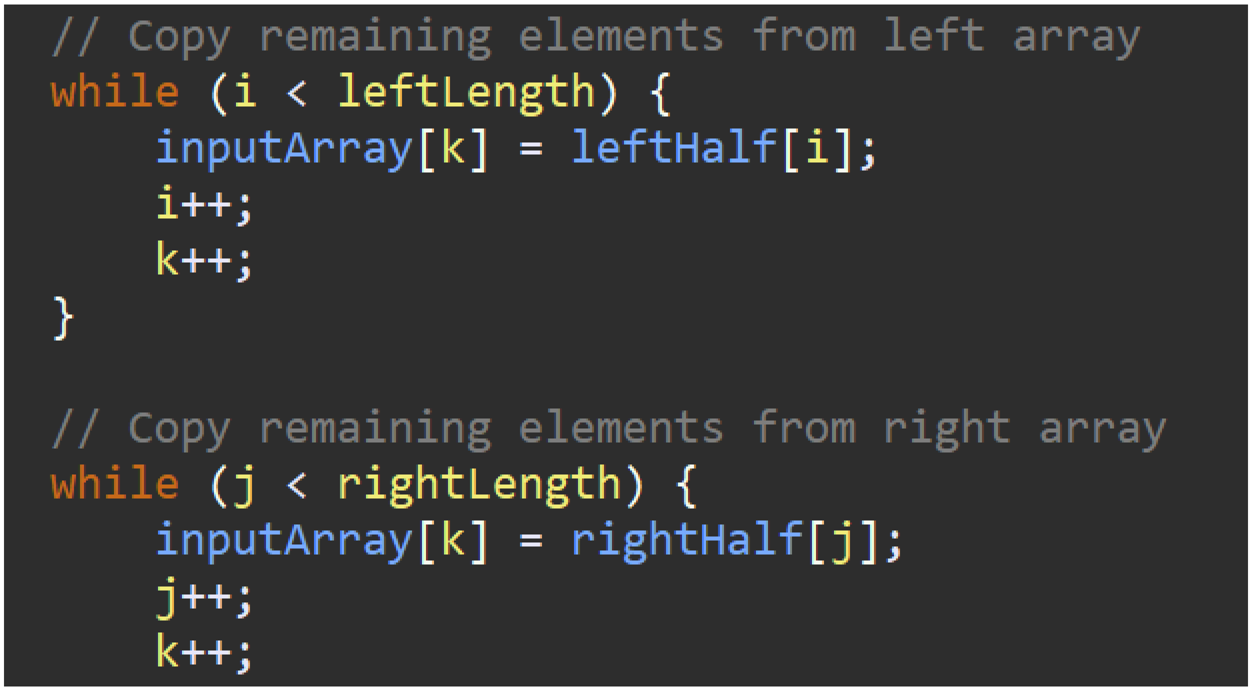


```java
int i = 0, j = 0, k = 0;  // Pointers for left, right, and merged arrays

// Merge while both arrays have elements
while (i < leftLength && j < rightLength) {
    if (leftHalf[i] < rightHalf[j]) {
        inputArray[k] = leftHalf[i];
        i++;
    } else {
        inputArray[k] = rightHalf[j];
        j++;
    }
    k++;
}
```

Then we make pointers for left and right arrays as well as the merged arrays (i,j and k respectively).

Then while both halves of the list still contain elements (our while loop) we compare an elements from both halves and we assign the lower value to the input array and then increment the value of the index in the input array AND in the half which had that lower value.



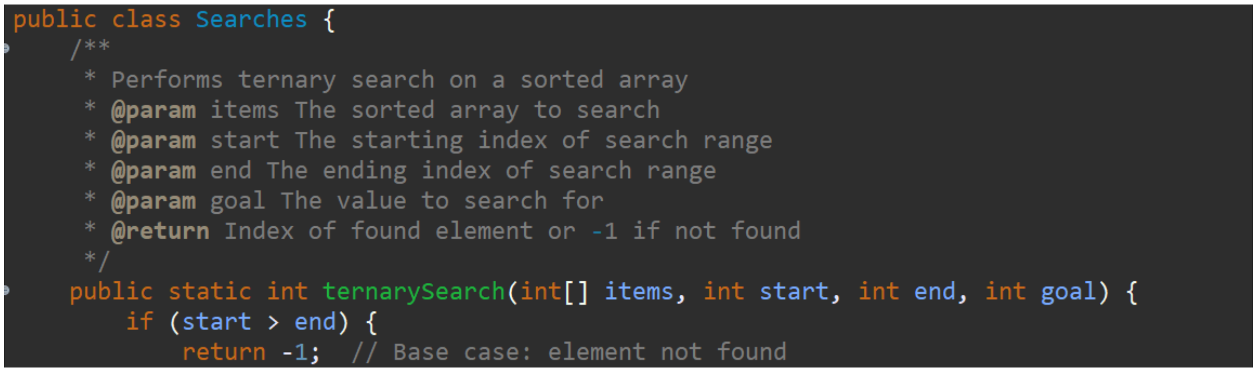
```
// Copy remaining elements from left array
while (i < leftLength) {
    inputArray[k] = leftHalf[i];
    i++;
    k++;
}

// Copy remaining elements from right array
while (j < rightLength) {
    inputArray[k] = rightHalf[j];
    j++;
    k++;
```

After one of the lists is empty we add the remaining elements from whichever half they remain again incrementing its index value and the input arrays index. We don't need to compare values because the 2 lists we are merging are sorted and any remaining values in a single array are already sorted.

This merge method works with the recursive calls to build back the sorted array from the broken down elements, recombining them to form a single sorted array.

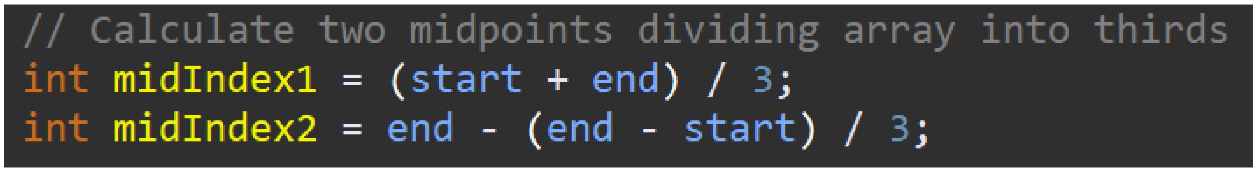
Then I made the class for the ternary search method. To make this method i started with the binary search and used some logic and planning to adapt it to break it down into 3 sections rather than 2.



```
public class Searches {
    /**
     * Performs ternary search on a sorted array
     * @param items The sorted array to search
     * @param start The starting index of search range
     * @param end The ending index of search range
     * @param goal The value to search for
     * @return Index of found element or -1 if not found
     */
    public static int ternarySearch(int[] items, int start, int end, int goal) {
        if (start > end) {
            return -1;  // Base case: element not found
```

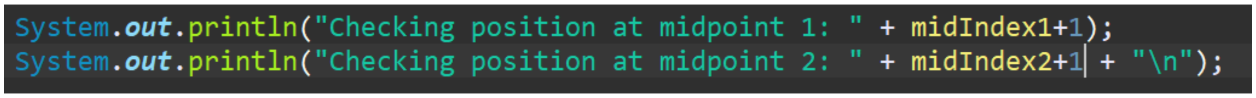
This public method returns an int for the index at which an element is found. At the end I added 1 to this returned value for a index system starting at 1 which is easier to use. This method has 4 parameters, the list which we will find the element in, a start and end index

and the goal, which is an integer because we have an integer list. If our start index is greater than our end index,we return -1 to show the element wasn't found.



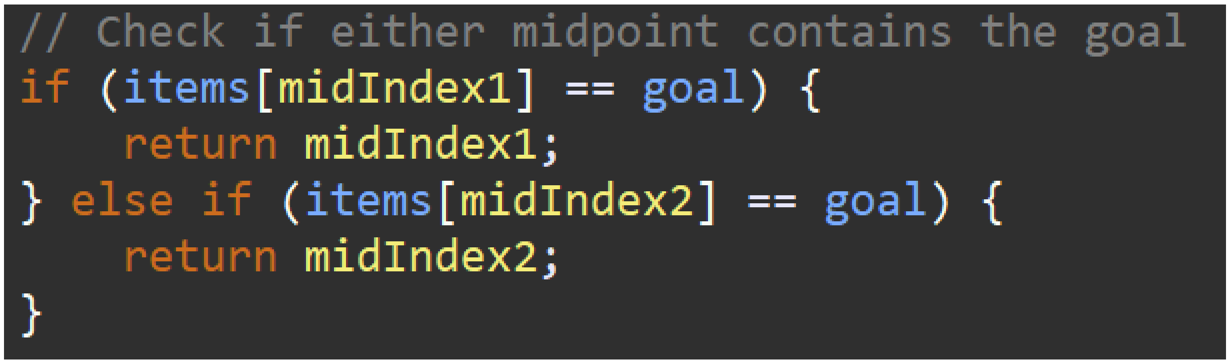
```java
// Calculate two midpoints dividing array into thirds
int midIndex1 = (start + end) / 3;
int midIndex2 = end - (end - start) / 3;
```

I defined the 2 midIndexes accounting for the fact that int division is taking place. So the first one is defined as going a third from the beginning of the list while the second one is defined as going a third from the end of the list.



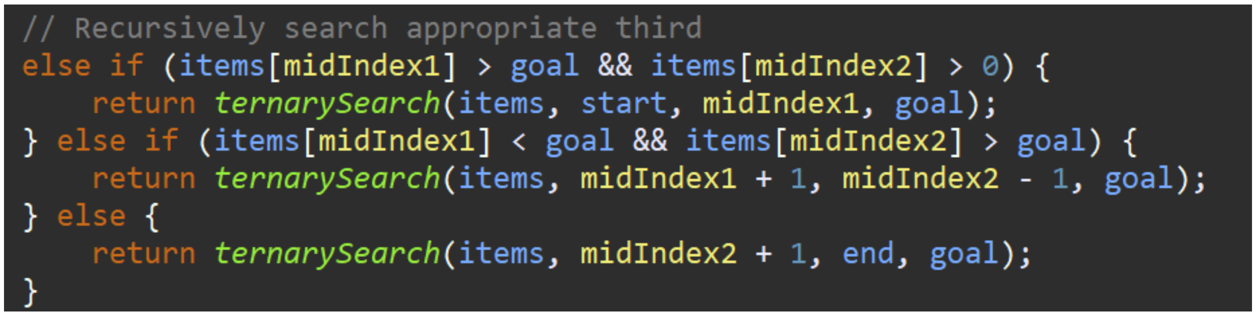
```java
System.out.println("Checking position at midpoint 1: " + midIndex1+1);
System.out.println("Checking position at midpoint 2: " + midIndex2+1 + "\n");
```

I printed out indicators to help the user see what values are checked and again this uses indexes starting at 1 (not 0).



```java
// Check if either midpoint contains the goal
if (items[midIndex1] == goal) {
    return midIndex1;
} else if (items[midIndex2] == goal) {
    return midIndex2;
}
```

Similar to the binary search, if the value of the list at either midIndex is the goal we just return the index we found it at.

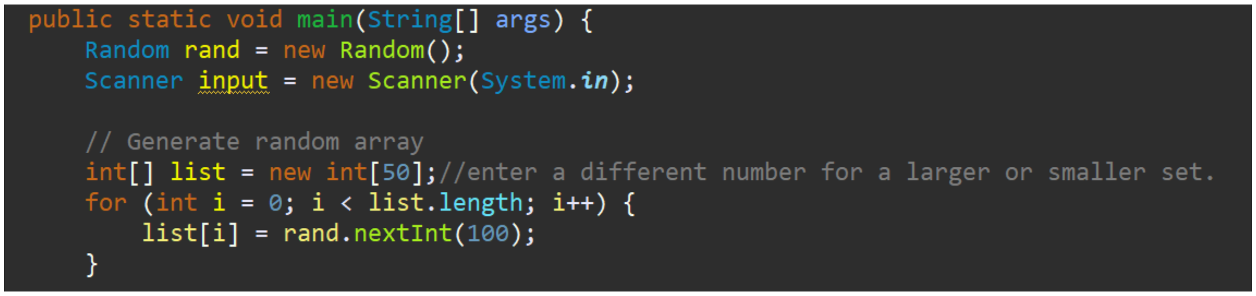


```java
// Recursively search appropriate third
else if (items[midIndex1] > goal && items[midIndex2] > 0) {
    return ternarySearch(items, start, midIndex1, goal);
} else if (items[midIndex1] < goal && items[midIndex2] > goal) {
    return ternarySearch(items, midIndex1 + 1, midIndex2 - 1, goal);
} else {
    return ternarySearch(items, midIndex2 + 1, end, goal);
}
```

If not, then we need to recursively break this array down until we reach the goal. This uses a very similar approach to binary search but we now have 3 regions to search for instead (if the goal was below midIndex1, between both midIndexes, or above midIndex2). We create if-elseif-else statements for these conditions and recursively called the ternarySearch method for these specific regions. This will keep breaking down

the list to a region closer and closer to the value until either MidIndex lands on that number.

Lastly, lets define our tester class ternarySearchLocations.

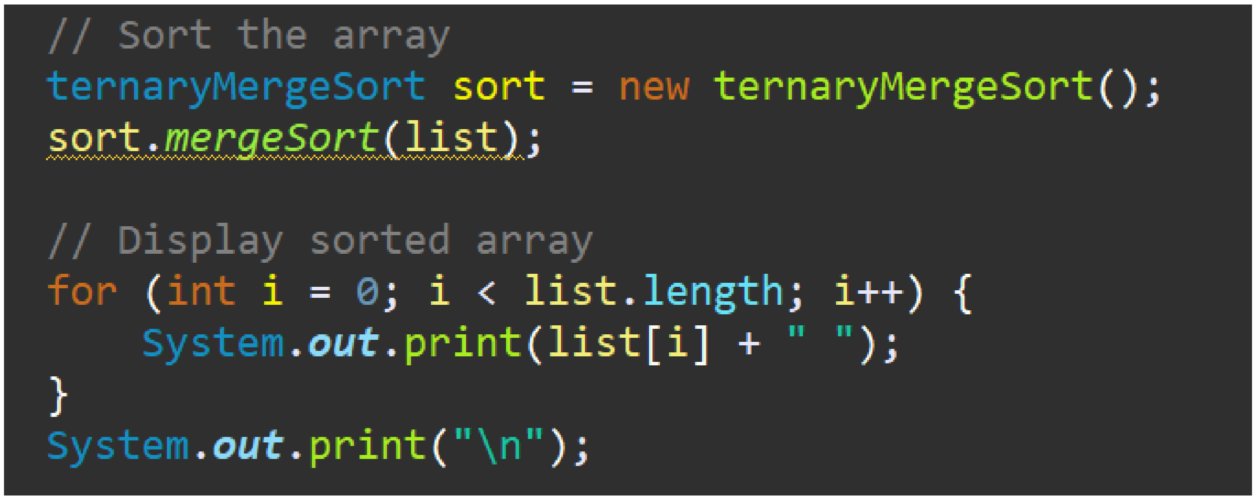


```java
public static void main(String[] args) {
    Random rand = new Random();
    Scanner input = new Scanner(System.in);

    // Generate random array
    int[] list = new int[50];//enter a different number for a larger or smaller set.
    for (int i = 0; i < list.length; i++) {
        list[i] = rand.nextInt(100);
    }
```

We define the main method and and intialize Scanner and Random objects for user input and generating random numbers.

Then we initialize a list with a given amount of elements(50 in this case) and we use a for loop to set each value in the list to a random number between 1 and a hundred.

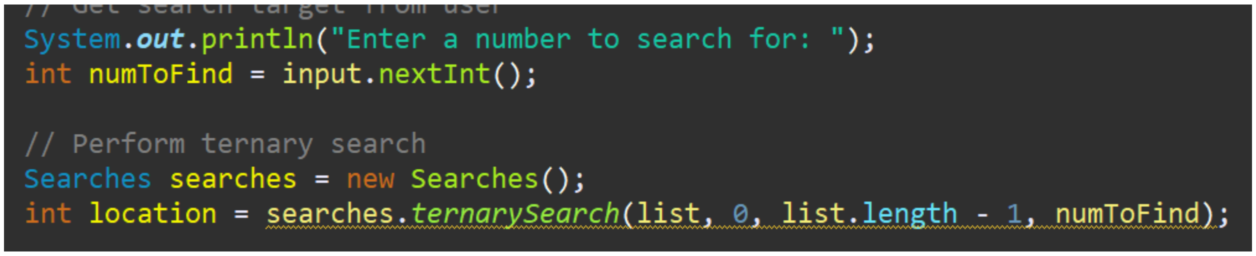


```java
// Sort the array
ternaryMergeSort sort = new ternaryMergeSort();
sort.mergeSort(list);

// Display sorted array
for (int i = 0; i < list.length; i++) {
    System.out.print(list[i] + " ");
}
System.out.print("\n");
```

Then we sort the array by initializing a sort object and then calling and sorting the list defined above.

We then print the list of numbers horizontally using a for loop to print every element.

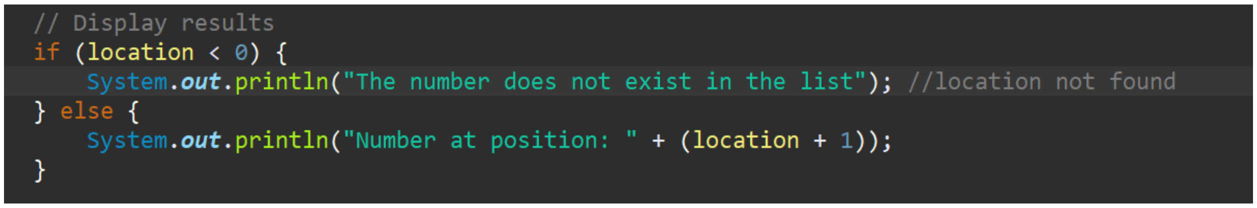


```java
System.out.println("Enter a number to search for: ");
int numToFind = input.nextInt();

// Perform ternary search
Searches searches = new Searches();
int location = searches.ternarySearch(list, 0, list.length - 1, numToFind);
```

We prompt and receive a number to search for from the user.

Then we initialize a searches object which contains the ternarySearch method. We call the ternarySearch method and initialize it to look through the entirety of the loop for the numToFind(user inputted number).



```java
// Display results
if (location < 0) {
    System.out.println("The number does not exist in the list"); //location not found
} else {
    System.out.println("Number at position: " + (location + 1));
}
```

Finally we display results: if the location was -1 and less than 0 then we know the number was not on the list to begin with and we print that. Otherwise we print the index(again based on an index-system that starts from 1).