CompareTo

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Person implements Comparable<Person> {
    private String name;
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter methods for name and age
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // Implementation of the compareTo method from the Comparable interface
    @Override
    public int compareTo(Person other) {
        return this.age - other.age; // Ascending order by age
    }

    // Overriding toString method for better output readability
    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + '}';
    }

    public static void main(String[] args) {
        // Creating a list of Person objects
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 22));
        people.add(new Person("Bob", 30));
        people.add(new Person("Charlie", 25));
        people.add(new Person("Diana", 20));

        // Sorting the list using Collections.sort()
        Collections.sort(people);

        // Printing the sorted list of people
        System.out.println("Sorted list of people by age:");
        for (Person person : people) {
            System.out.println(person);
        }
    }
}
```

Output

```
Sorted list of people by age:

Person{name='Diana', age=20}

Person{name='Alice', age=22}

Person{name='Charlie', age=25}

Person{name='Bob', age=30}
```

Selection Sort

```
public class SelectionSort {

    public static void sort(int[] array) {

        for (int i = 0; i < array.length - 1; i++) {

            int minIndex = i; // Start by assuming the first element is the minimum


            // Find the smallest element in the unsorted section of the array for

            (int j = i + 1; j < array.length; j++) {

                if (array[j] < array[minIndex]) {

                    minIndex = j; // Update the index of the minimum element

                }

            }

            // Swap the found minimum element with the first element of the unsorted section int

            temp = array[minIndex];

            array[minIndex] = array[i]; array[i] =

            temp;

        }

    }

    public static void main(String[] args) {

        int[] data = {64, 25, 12, 22, 11};
        sort(data);
        for (int num : data) {
            System.out.print(num + " ");
        }
    }
}
```

OUTPUT→11 12 22 25 64

Insertion Sort

```
public class InsertionSort {
    public static void sort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int current = arr[i];
            int j = i - 1;

            // Move elements of arr[0..i-1], that are greater than current,
            // to one position ahead of their current position
            while (j >= 0 && arr[j] > current) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = current;
        }
    }

    public static void main(String[] args) {
        int[] arr = {9, 5, 1, 4, 3};
        sort(arr);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

OUTPUT→1 3 4 5 9

Recursive Insertion Sort

```
public class RecursiveInsertionSort {
    public static void recursiveSort(int[] arr, int n) {
        // Base case
        if (n <= 1) {
            return;
        }

        // Sort first n-1 elements
        recursiveSort(arr, n - 1);

        // Insert last element at its correct position in sorted array.
        int last = arr[n - 1];
        int j = n - 2;

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

    public static void main(String[] args) {
        int[] arr = {9, 5, 1, 4, 3};
        recursiveSort(arr, arr.length);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

OUTPUT→1 3 4 5 9

Bubble Sort

```
public class BubbleSort {
    public static void sort(int[] arr) {
        boolean swapped;
        int n = arr.length;
        // Perform passes through the array
        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            // Compare adjacent elements and swap if necessary
            for (int j = 0; j < n - 1 - i; j++) { // The '-i' optimizes by reducing the comparisons in each subsequent pass
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j + 1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true; // Mark as swapped to know if we should continue
                }
            }
            // If no two elements were swapped by inner loop, then break
            if (!swapped) {
                break;
            }
        }
    }

    // Utility method to print array elements
    public static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    // Main method to test the bubble sort algorithm
    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};
        System.out.println("Original array:");
        printArray(arr);

        sort(arr);

        System.out.println("Sorted array:");
        printArray(arr);
    }
}
```

Merge Sort

```
public class MergeSort {

    // Main method to test the merge sort algorithm
    public static void main(String[] args) {
        int[] arr = {38, 27, 43, 3, 9, 82, 10};
        System.out.println("Original array:");
        printArray(arr);

        mergeSort(arr, 0, arr.length - 1);

        System.out.println("Sorted array:");
        printArray(arr);
    }

    // Merge Sort function: recursively divides and sorts the array
```

```
public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        // Find the middle point
        int mid = (left + right) / 2;

        // Recursively sort first half
        mergeSort(arr, left, mid);
        // Recursively sort second half
        mergeSort(arr, mid + 1, right);

        // Merge the two sorted halves
        merge(arr, left, mid, right);
    }
}

// Merge function: combines two sorted subarrays into one sorted array
public static void merge(int[] arr, int left, int mid, int right) {
    // Find sizes of two subarrays to be merged
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int[] L = new int[n1];
    int[] R = new int[n2];

    // Copy data into temporary arrays
    for (int i = 0; i < n1; i++) {
```

```
    L[i] = arr[left + i];
}
for (int j = 0; j < n2; j++) {
    R[j] = arr[mid + 1 + j];
}

// Merge the temporary arrays back into arr[left..right]
int i = 0, j = 0, k = left;
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

// Copy remaining elements of L[] if any
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

// Copy remaining elements of R[] if any
```

```
        while (j < n2) {

            arr[k] = R[j];

            j++;

            k++;

        }

    }


    // Utility method to print the elements of an array

    public static void printArray(int[] arr) {

        for (int num : arr) {

            System.out.print(num + " ");

        }

        System.out.println();

    }

}
```

QuickSort

```
public class QuickSort {


    // QuickSort function: sorts arr[low..high]

    public static void quickSort(int[] arr, int low, int high) {

        if (low < high) {

            // Partition the array and get the pivot index

            int pivotIndex = partition(arr, low, high);


            // Recursively sort elements before pivot

            quickSort(arr, low, pivotIndex - 1);
        // Recursively sort elements after pivot
        quickSort(arr, pivotIndex + 1, high);
    }
}

// Partition function: rearranges the elements around the pivot
public static int partition(int[] arr, int low, int high) {
    // Choose the pivot (using the last element in this case)
    int pivot = arr[high];
    // Index for the smaller element
    int i = low - 1;

    // Compare each element with the pivot
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++; // Move index for smaller element
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Place the pivot in its correct sorted position
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
```

```
        // Return the pivot index
        return i + 1;
    }

    // Utility method to print the array
    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    // Main method to test QuickSort
    public static void main(String[] args) {
        int[] arr = {10, 7, 8, 9, 1, 5};
        System.out.println("Original array:");
        printArray(arr);

        quickSort(arr, 0, arr.length - 1);

        System.out.println("Sorted array:");
        printArray(arr);
    }
}
```