C:\Users\Rich\Documents\NetBeansProjects\Lab12\src\Sort.java

```

import java.util.ArrayList;
import java.util.Arrays;


public class Sort {

//    public static <K> void simpleBubbleSort( K[] data, Comparator<K> comp )
//    {
//        for ( int i = 0; i < data.length; i++ )
//        {
//            for ( int j = 0; j < data.length - 1; j++ )
//            {
//                if ( comp.compare( data[j], data[j+1] ) < 0 )       // neighbor element is greater than prev
//                {
//                    K temp = data[j];
//                    data[j] = data[j+1];
//                    data[j+1] = temp;
//                }
//            }
//        }
//    }

    public static <K> void simpleBubbleSort( K[] data, Comparator<K> comp )
    {
        for(int i = 0; i < data.length-1; i++)
            for(int j = i+1; j < data.length; j++)
            {
                if( comp.compare(data[i], data[j]) < 0) // if data[i](3) >  data[j](2) ---(in comp 3 > 2--> -1 < 0 true for ascending order u swap
                                                // data[i] (Ama) < data[j](Aman) for  for alphabetical
                {
                    K temp = data[i];
                    data[i] = data[j];
                    data[j] = temp;
                }
            }
    }

    public static<K> void selectionSort(K[] data, Comparator<K> comp)
    {

        K temp;                 // temporary location for swap
        int indexOfMax;        // index of the maximum value in subarray

        for(int i = 0; i< data.length; i++)
        {
           // find index of largest value in subarray
            indexOfMax = indexOfLargestElement(data,data.length-i, comp);

            // swap data[indexofMax] and data[data.length-i-1]
            temp = data[indexOfMax];
            data[indexOfMax] = data[data.length-i-1];
            data[data.length-i-1]= temp;
        }

    }

    public static<K> void insertionSort(K[] data, Comparator<K> comp)
    {

        int j;
        K temp;

        for(int i = 1; i<data.length; i++)
        {
            j = i;
            temp = data[i];

            while( j!= 0 && comp.compare(data[j-1], temp) < 0)        //if data[j-1].value > temp.value for ascending order
            {
                data[j] = data[j-1];
                j--;
            }

            data[j] = temp;
        }

    }

    public static<K> void mergeSort(K[] S, Comparator<K> comp)
    {
        int n = S.length;
        if(n < 2) return;                        // base case. array is trivially sorted

        // divide
        int mid = n/2;
        K[] S1 = Arrays.copyOfRange(S, 0, mid);
        K[] S2 = Arrays.copyOfRange(S, mid, n);
        //conquer with recursion
        mergeSort(S1,comp);
        mergeSort(S2,comp);
        //merge results (rom..After all the child call is
        merge(S1,S2,S,comp);                                   // merge sorted halves back into original.
    }
    /**
     *
     * @param <K>
     * @param S
     * @param comp
      * @param a         start of the segment
      * @param b         end of the segment
      */
     public static<K> void quickSortInPlace(K[] S, Comparator<K> comp, int a, int b)     // 1st call Qs(S,0,7) for an array of size 8.
     {
         if(a >= b) return;         // subarray is trivially sorted. Base case
                              // when we have only 1 element into the subarray. (a=0, b=0) or the segment  is invalid

         /*start of partionning logic(where element lesser than the pivot on left of pivot
         and element greater than the pivot on the right of pivot.
         */
         int left = a;              // the index 1st element in subarray
         int right = b-1;           // index element before pivot
         K pivot = S[b];            // last element in orginal segment as pivot
         K temp;                   // temp object for swapping

         while(left <= right)
         {
             // scan until reaching value equal or larger than pivot(or right marker)
             while(left <= right && comp.compare(S[left], pivot) > 0)   //:- (85) > pivot(50) in comp result: -1 > 0..false do not incremnt left index position.
                 left++;
             // scan until reaching value equal or smaller than pivot(or left marker)
             while(left <= right && comp.compare(S[right],pivot) < 0) // (96) > 50 comp: -1 < 0 true
                 right--;

             if(left<= right)    // indices did not strictly cross
             {
                 // swap values and shring range
                 temp = S[left];
                 S[left] = S[right];
                 S[right] = temp;

                 left++; right--;
             }
         }
         //put pivot into its final place(currently marked by left index)
         temp = S[left];
         S[left] = S[b];
         S[b]   = temp;
         //----- End of partitioning---

         //make recursive calls
         quickSortInPlace(S, comp, a, left-1);
         quickSortInPlace(S,comp, left+1, b);
     }
     /**
      *
      * @param data  sequence of element to be sorted
      * @param compList a bag of the comparators in which the highest comparator key
      * should be  put 1st in the bag
      */
     public static<K> void radixSort(K[] data, ArrayBag<Comparator<K>> compList)
     {
         int lowKeyIndex = compList.getCurrentSize() -1;
         mergeSort(data, compList.get(lowKeyIndex));      // name
         mergeSort(data, compList.get(lowKeyIndex-1));    // voted
         mergeSort(data, compList.get(lowKeyIndex-2));   // party comp

    }


     //--------------------------Private Utility --------------------------------
 //    public static<K> K[] arrayClone(K[] parent)
 //    {
 //       int parentSize = parent.length;
 //       K[] clone = (K[]) new Object[parentSize];
 //       for(int i = 0; i < parentSize; ++i)
 //       {
 //          clone[i] = parent[i];
 //       }
 //       return clone;
 //    }
     /**
      *
      * @param <K>
      * @param array the array for which the comparison is to be done on.
      * @param size the size of the subarray.
      * @param comp  comparator on the key of the array
      * @return the index of the largest element key. (index greater int for ID, greater A-Z for alphabetical order)
      */
     private static<K> int indexOfLargestElement(K[]array, int size,Comparator<K> comp)
     {
             int index = 0;
             for(int i = 0; i < size; ++i)
             {
                 if(comp.compare(array[i], array[index]) < 0)        // array[i]< array[index] //< 0 ascending order; A-Z
                                                                 // this is similar if array[i].value(23) > (12) array[index].value, then index = i.
                 {                                                // if you're doing the comp on Id number, then  23 >12 3 ret--> -1< 0...true
                     index = i;
                 }
             }
             return index;
     }
     /**
      * Compare the value at the index of each subarray and
      * copy the smaller of the two.
      * @param <K>
      * @param S1 left subarray
      * @param S2 right subArray for number  greater than the pivot
      * @param S Sequence of element to sort. an Array
      * @param comp comparator
      */
     private static<K> void merge(K[] S1, K[]S2, K[] S,Comparator<K> comp)
     {
         int i = 0; int j = 0;
         while(i+j < S.length)
         {
             if(j== S2.length || (i < S1.length && comp.compare(S1[i],S2[j]) > 0)) // if element at S1(11) > el @ S2(10)---> -1 > 0 false.
                     S[i+j] = S1[i++];           // copy ith element of S1 and increment i;
             else
                     S[i+j] = S2[j++];           // copty jth element and increment j
         }
     }
 }


```