**­­// INSERTION SORT (all these methods are inside a class)**

**public static <T extends Comparable<? super T>>**

**void insertionSort(T[] a, int n)**

**{**

**insertionSort(a, 0, n - 1);**

**} // end insertionSort**

**public static <T extends Comparable<? super T>>**

**void insertionSort(T[] a, int first, int last)**

**{**

**for (int unsorted = first + 1; unsorted <= last; unsorted++)**

**{**

**// Assert: a[first]<= a[first + 1]<=..<= a[unsorted - 1]**

**T firstUnsorted = a[unsorted];**

**insertInOrder(firstUnsorted, a, first, unsorted - 1);**

**} // end for**

**} // end insertionSort**

**private static <T extends Comparable<? super T>>**

**void insertInOrder(T anEntry, T[] a, int begin, int end)**

**{**

**int index = end;**

**while ((index >= begin) && (anEntry.compareTo(a[index]) < 0))**

**{**

**a[index + 1] = a[index]; // Make room**

**index--;**

**} // end for**

**// Assertion: a[index + 1] is available**

**a[index + 1] = anEntry; // Insert**

**} // end insertInOrder**

**// QUICK SORT**

**/\*\* Sorts an array into ascending order. Uses quick sort with**

**median-of-three pivot selection for arrays of at least**

**MIN\_SIZE entries, and uses insertion sort for other arrays.**

**MIN\_SIZE is a declared static variable in this class with 10 \*/**

**public static <T extends Comparable<? super T>>**

**void quickSort(T[] array, int n)**

**{**

**quickSort(array, 0, n - 1);**

**} // end quickSort**

**public static <T extends Comparable<? super T>>**

**void quickSort(T[] a, int first, int last)**

**{**

**if (last - first + 1 < MIN\_SIZE)**

**{**

**insertionSort(a, first, last);**

**}**

**else**

**{**

**// Create the partition: Smaller | Pivot | Larger**

**int pivotIndex = partition(a, first, last);**

**// Sort subarrays Smaller and Larger**

**quickSort(a, first, pivotIndex - 1);**

**quickSort(a, pivotIndex + 1, last);**

**} // end if**

**} // end quickSort**

**// Partitions an array as part of quick sort into two subarrays**

**// called Smaller and Larger that are separated by a single**

**// entry called the pivot.**

**// Entries in Smaller are <= pivot and appear before the**

**// pivot in the array.**

**// Entries in Larger are >= pivot and appear after the**

**// pivot in the array.**

**// Parameters:**

**// a An array of Comparable objects.**

**// first The integer index of the first array entry;**

**// first >= 0 and < a.length.**

**// last The integer index of the last array entry;**

**// last - first >= 3; last < a.length.**

**// Returns the index of the pivot.**

**private static <T extends Comparable<? super T>>**

**int partition(T[] a, int first, int last)**

**{**

**int mid = first + (last - first) / 2;**

**sortFirstMiddleLast(a, first, mid, last);**

**// Assertion: The pivot is a[mid]; a[first] <= pivot and**

**// a[last] >= pivot, so do not compare these two**

**// array entries with pivot.**

**// Move pivot to next-to-last position in array**

**swap(a, mid, last - 1);**

**int pivotIndex = last - 1;**

**T pivotValue = a[pivotIndex];**

**// Determine subarrays Smaller = a[first..endSmaller]**

**// and Larger = a[endSmaller+1..last-1]**

**// such that entries in Smaller are <= pivotValue and**

**// entries in Larger are >= pivotValue; initially,**

**// these subarrays are empty**

**int indexFromLeft = first + 1;**

**int indexFromRight = last - 2;**

**boolean done = false;**

**while (!done)**

**{**

**// Starting at beginning of array,**

**// leave entries that are < pivotValue;**

**// locate first entry that is >= pivotValue;**

**// you will find one,**

**// since last entry is >= pivot**

**while (a[indexFromLeft].compareTo(pivotValue) < 0)**

**indexFromLeft++;**

**// Starting at end of array,**

**// leave entries that are > pivot;**

**// locate first entry that is <= pivot; you will find one,**

**// since first entry is <= pivot**

**while (a[indexFromRight].compareTo(pivotValue) > 0)**

**indexFromRight--;**

**assert a[indexFromLeft].compareTo(pivotValue) >= 0 &&**

**a[indexFromRight].compareTo(pivotValue) <= 0;**

**if (indexFromLeft < indexFromRight)**

**{**

**swap(a, indexFromLeft, indexFromRight);**

**indexFromLeft++;**

**indexFromRight--;**

**}**

**else**

**done = true;**

**} // end while**

**// Place pivotValue between the subarrays**

**// Smaller and Larger**

**swap(a, pivotIndex, indexFromLeft);**

**pivotIndex = indexFromLeft;**

**// Assertion:**

**// Smaller = a[first..pivotIndex-1]**

**// Pivot = a[pivotIndex]**

**// Larger = a[pivotIndex+1..last]**

**return pivotIndex;**

**} // end partition**

**// Sorts the first, middle, and last entries**

**// of an array into ascending order.**

**// Parameters:**

**// a An array of Comparable objects.**

**// first The integer index of the first array entry;**

**// first >= 0 and < a.length.**

**// mid The integer index of the middle array entry.**

**// last The integer index of the last array entry;**

**// last - first >= 2, last < a.length.**

**private static <T extends Comparable<? super T>>**

**void sortFirstMiddleLast(T[] a, int first, int mid, int last)**

**{**

**order(a, first, mid); // Make a[first] <= a[mid]**

**order(a, mid, last); // Make a[mid] <= a[last]**

**order(a, first, mid); // Make a[first] <= a[mid]**

**} // end sortFirstMiddleLast**

**// Orders two given array elements into ascending order**

**// so that a[i] <= a[j].**

**private static <T extends Comparable<? super T>>**

**void order(T[] a, int i, int j)**

**{**

**if (a[i].compareTo(a[j]) > 0)**

**swap(a, i, j);**

**} // end order**

**// Swaps the array entries array[i] and array[j].**

**private static void swap(Object[] array, int i, int j)**

**{**

**Object temp = array[i];**

**array[i] = array[j];**

**array[j] = temp;**

**} // end swap**

**// SHELL SORT (all these methods are inside a class)**

**public static <T extends Comparable<? super T>>**

**void shellSort(T[] a, int n)**

**{**

**shellSort(a, 0, n - 1);**

**} // end shellSort**

**/\*\* Sorts equally spaced elements of an array into**

**ascending order.**

**@param a An array of Comparable objects.**

**@param first An integer >= 0 that is the index of the first**

**array element to consider.**

**@param last An integer >= first and < a.length that is the**

**index of the last array element to consider.**

**@param space The difference between the indices of the**

**elements to sort. \*/**

**public static <T extends Comparable<? super T>>**

**void shellSort(T[] a, int first, int last)**

**{**

**int n = last - first + 1; // Number of array elements**

**for (int space = n / 2; space > 0; space = space / 2)**

**{**

**for (int begin = first; begin < first + space; begin++)**

**incrementalInsertionSort(a, begin, last, space);**

**} // end for**

**} // end shellSort**

**// Sorts equally spaced elements of an array**

**// into ascending order.**

**// Parameters:**

**// a An array of Comparable objects.**

**// first The integer index of the first array entry to**

**// consider; first >= 0 and < a.length.**

**// last The integer index of the last array entry to**

**// consider; last >= first and < a.length.**

**// space The difference between the indices of the**

**// entries to sort.**

**private static <T extends Comparable<? super T>>**

**void incrementalInsertionSort(T[] a, int first, int last, int space)**

**{**

**int unsorted, index;**

**for (unsorted = first + space; unsorted <= last;**

**unsorted = unsorted + space)**

**{**

**T nextToInsert = a[unsorted];**

**index = unsorted - space;**

**while ((index >= first) && (nextToInsert.compareTo(a[index]) < 0))**

**{**

**a[index + space] = a[index];**

**index = index - space;**

**} // end while**

**a[index + space] = nextToInsert;**

**} // end for**

**} // end incrementalInsertionSort**