<http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html>

Method 1:

Lists (and arrays) of objects that implement this interface can be sorted automatically by **[Collections.sort](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html" \l "sort(java.util.List))** (and **[Arrays.sort](http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html" \l "sort(java.lang.Object[]))**). Objects that implement this interface can be used as keys in a [sorted map](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html) or as elements in a [sorted set](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html), without the need to specify a [**comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html).

Rui: suitable for using a fixed method to compare objects

public class MyObj implements Comparable{

public int value;

public int compareTo(Object o){

MyObj obj = (MyObj)o;

If( this.value > obj.value)

return 1;

else if(this.value == obj.value)

return 0;

else

return -1;

}

}

MyObj [] arr = {…};

Arrays.sort(arr);

Method 2:

specify a [**comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)

Rui: suitable for using different methods to compare objects

public class MyObj{

public int value;

}

public class MyComparator implements Comparator<MyObj> { // <MyObj> must be added

public int compare(MyObj o1, MyObj o2){

If( o1.value > o2.value)

return 1;

else if(o1.value == o2.value)

return 0;

else

return -1;

}

}

MyObj [] arr = {…};

Arrays.sort(arr, new MyComparator() );

Method 1:

public static void sort([Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)[] a)

Sorts the specified array of objects into ascending order, according to the [**natural ordering**](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) of its elements. All elements in the array must implement the [**Comparable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) interface. Furthermore, all elements in the array must be *mutually comparable* (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python ([TimSort](http://svn.python.org/projects/python/trunk/Objects/listsort.txt)). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

public interface **Comparable<T>**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by **[Collections.sort](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html" \l "sort(java.util.List))** (and **[Arrays.sort](http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html" \l "sort(java.lang.Object[]))**). Objects that implement this interface can be used as keys in a [sorted map](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html) or as elements in a [sorted set](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html), without the need to specify a [**comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html).

Method 2:

public static <T> void sort(T[] a,

[Comparator](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)<? super T> c)

Sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be *mutually comparable* by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python ([TimSort](http://svn.python.org/projects/python/trunk/Objects/listsort.txt)). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

**Parameters:**

a - the array to be sorted

c - the comparator to determine the order of the array. A null value indicates that the elements' [natural ordering](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) should be used.

**Throws:**

[ClassCastException](http://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html) - if the array contains elements that are not *mutually comparable* using the specified comparator

[IllegalArgumentException](http://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html) - (optional) if the comparator is found to violate the [Comparator](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) contract