**HashSet**

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, java.io.Serializable

{

**private transient HashMap<E,Object> map;**

private static final Object PRESENT = new Object();// Dummy value to associate with an Object in the backing Map

public HashSet() {

map = new HashMap<E,Object>();

}

public Iterator<E> iterator() {

return map.keySet().iterator();

}

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

public boolean remove(Object o) {

return map.remove(o)==PRESENT;

}

private void writeObject(java.io.ObjectOutputStream s)

throws java.io.IOException {

// Write out any hidden serialization magic

s.defaultWriteObject();

s.writeInt(map.capacity());// Write out HashMap capacity and load factor

s.writeFloat(map.loadFactor());

s.writeInt(map.size());// Write out size

for (Iterator i=map.keySet().iterator(); i.hasNext(); ) // Write out all elements in the proper order.

s.writeObject(i.next());

}

private void readObject(java.io.ObjectInputStream s)

throws java.io.IOException, ClassNotFoundException {

// Read in any hidden serialization magic

s.defaultReadObject();

int capacity = s.readInt();// Read in HashMap capacity and load factor and create backing HashMap

float loadFactor = s.readFloat();

map = (((HashSet)this) instanceof LinkedHashSet ?

new LinkedHashMap<E,Object>(capacity, loadFactor) :

new HashMap<E,Object>(capacity, loadFactor));

// Read in size

int size = s.readInt();

for (int i=0; i<size; i++) { // Read in all elements in the proper order.

E e = (E) s.readObject();

map.put(e, PRESENT);

}

}

}

**TreeSet**

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, java.io.Serializable

{

**private transient NavigableMap<E,Object> m;**

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();

public TreeSet(Comparator<? super E> comparator) {

this(new TreeMap<E,Object>(comparator));

}

public Iterator<E> iterator() {

return m.navigableKeySet().iterator();

}

public Iterator<E> descendingIterator() {

return m.descendingKeySet().iterator();

}

public NavigableSet<E> descendingSet() {

return new TreeSet(m.descendingMap());

}

public boolean add(E e) {

return m.put(e, PRESENT)==null;

}

public boolean remove(Object o) {

return m.remove(o)==PRESENT;

}

public NavigableSet<E> headSet(E toElement, boolean inclusive) {

return new TreeSet<E>(m.headMap(toElement, inclusive));

}

public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {

return new TreeSet<E>(m.tailMap(fromElement, inclusive));

}

public SortedSet<E> subSet(E fromElement, E toElement) {

return subSet(fromElement, true, toElement, false);

}

public E first() {

return m.firstKey();

}

public E last() {

return m.lastKey();

}

public E lower(E e) {

return m.lowerKey(e);

}

public E floor(E e) {

return m.floorKey(e);

}

public E ceiling(E e) {

return m.ceilingKey(e);

}

public E higher(E e) {

return m.higherKey(e);

}

private void writeObject(java.io.ObjectOutputStream s)

throws java.io.IOException {

// Write out any hidden stuff

s.defaultWriteObject();

// Write out Comparator

s.writeObject(m.comparator());

// Write out size

s.writeInt(m.size());

// Write out all elements in the proper order.

for (Iterator i=m.keySet().iterator(); i.hasNext(); )

s.writeObject(i.next());

}

private void readObject(java.io.ObjectInputStream s)

throws java.io.IOException, ClassNotFoundException {

// Read in any hidden stuff

s.defaultReadObject();

// Read in Comparator

Comparator<? super E> c = (Comparator<? super E>) s.readObject();

// Create backing TreeMap

TreeMap<E,Object> tm;

if (c==null)

tm = new TreeMap<E,Object>();

else

tm = new TreeMap<E,Object>(c);

m = tm;

// Read in size

int size = s.readInt();

tm.readTreeSet(size, s, PRESENT);

}

}

**LinkedHashSet**

public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, java.io.Serializable

{

public LinkedHashSet(int initialCapacity, float loadFactor) {

super(initialCapacity, loadFactor, true);

}

public LinkedHashSet(int initialCapacity) {

super(initialCapacity, .75f, true);

}

public LinkedHashSet() {

super(16, .75f, true); // This internally calls the following of the HashSet class

//HashSet(int initialCapacity, float loadFactor, boolean dummy) {

//map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);

//}

}

}

**SortedSet**

**java.util   
Interface SortedSet<E>**

**Type Parameters:**

E - the type of elements maintained by this set

**All Superinterfaces:**

[Collection](file:///E:\dev\jdk6docs\api\java\util\Collection.html)<E>, [Iterable](file:///E:\\dev\\jdk6docs\\api\\java\\lang\\Iterable.html" \o "interface in java.lang)<E>, [Set](file:///E:\dev\jdk6docs\api\java\util\Set.html)<E>

**All Known Subinterfaces:**

[NavigableSet](file:///E:\dev\jdk6docs\api\java\util\NavigableSet.html)<E>

**All Known Implementing Classes:**

[ConcurrentSkipListSet](file:///E:\dev\jdk6docs\api\java\util\concurrent\ConcurrentSkipListSet.html), [TreeSet](file:///E:\\dev\\jdk6docs\\api\\java\\util\\TreeSet.html" \o "class in java.util)

public interface **SortedSet<E>**

extends [Set](file:///E:\dev\jdk6docs\api\java\util\Set.html)<E>

A [Set](file:///E:\dev\jdk6docs\api\java\util\Set.html) that further provides a *total ordering* on its elements. The elements are ordered using their [natural ordering](file:///E:\dev\jdk6docs\api\java\lang\Comparable.html), or by a [Comparator](file:///E:\dev\jdk6docs\api\java\util\Comparator.html) typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order. Several additional operations are provided to take advantage of the ordering. (This interface is the set analogue of [SortedMap](file:///E:\\dev\\jdk6docs\\api\\java\\util\\SortedMap.html" \o "interface in java.util).)

All elements inserted into a sorted set must implement the Comparable interface (or be accepted by the specified comparator). Furthermore, all such elements must be *mutually comparable*: e1.compareTo(e2) (orcomparator.compare(e1, e2)) must not throw a ClassCastException for any elements e1 and e2 in the sorted set. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

Note that the ordering maintained by a sorted set (whether or not an explicit comparator is provided) must be *consistent with equals* if the sorted set is to correctly implement the Set interface. (See the Comparableinterface or Comparator interface for a precise definition of *consistent with equals*.) This is so because the Set interface is defined in terms of the equals operation, but a sorted set performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the sorted set, equal. The behavior of a sorted set *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Set interface.

All general-purpose sorted set implementation classes should provide four "standard" constructors: 1) A void (no arguments) constructor, which creates an empty sorted set sorted according to the natural ordering of its elements. 2) A constructor with a single argument of type Comparator, which creates an empty sorted set sorted according to the specified comparator. 3) A constructor with a single argument of type Collection, which creates a new sorted set with the same elements as its argument, sorted according to the natural ordering of the elements. 4) A constructor with a single argument of type SortedSet, which creates a new sorted set with the same elements and the same ordering as the input sorted set. There is no way to enforce this recommendation, as interfaces cannot contain constructors.

Note: several methods return subsets with restricted ranges. Such ranges are *half-open*, that is, they include their low endpoint but not their high endpoint (where applicable). If you need a *closed range* (which includes both endpoints), and the element type allows for calculation of the successor of a given value, merely request the subrange from lowEndpoint to successor(highEndpoint). For example, suppose that s is a sorted set of strings. The following idiom obtains a view containing all of the strings in s from low to high, inclusive:

SortedSet<String> sub = s.subSet(low, high+"\0");

A similar technique can be used to generate an *open range* (which contains neither endpoint). The following idiom obtains a view containing all of the Strings in s from low to high, exclusive:

SortedSet<String> sub = s.subSet(low+"\0", high);

**NavigableSet**

**java.util   
Interface NavigableSet<E>**

**Type Parameters:**

E - the type of elements maintained by this set

**All Superinterfaces:**

[Collection](file:///E:\dev\jdk6docs\api\java\util\Collection.html)<E>, [Iterable](file:///E:\\dev\\jdk6docs\\api\\java\\lang\\Iterable.html" \o "interface in java.lang)<E>, [Set](file:///E:\dev\jdk6docs\api\java\util\Set.html)<E>, [SortedSet](file:///E:\\dev\\jdk6docs\\api\\java\\util\\SortedSet.html" \o "interface in java.util)<E>

**All Known Implementing Classes:**

[ConcurrentSkipListSet](file:///E:\dev\jdk6docs\api\java\util\concurrent\ConcurrentSkipListSet.html), [TreeSet](file:///E:\\dev\\jdk6docs\\api\\java\\util\\TreeSet.html" \o "class in java.util)

public interface **NavigableSet<E>**

extends [SortedSet](file:///E:\dev\jdk6docs\api\java\util\SortedSet.html)<E>

A [SortedSet](file:///E:\\dev\\jdk6docs\\api\\java\\util\\SortedSet.html" \o "interface in java.util) extended with navigation methods reporting closest matches for given search targets. Methods lower, floor, ceiling, and higher return elements respectively less than, less than or equal, greater than or equal, and greater than a given element, returning null if there is no such element. A NavigableSet may be accessed and traversed in either ascending or descending order. The descendingSet method returns a view of the set with the senses of all relational and directional methods inverted. The performance of ascending operations and views is likely to be faster than that of descending ones. This interface additionally defines methodspollFirst and pollLast that return and remove the lowest and highest element, if one exists, else returning null. Methods subSet, headSet, and tailSet differ from the like-named SortedSet methods in accepting additional arguments describing whether lower and upper bounds are inclusive versus exclusive. Subsets of any NavigableSet must implement the NavigableSet interface.

The return values of navigation methods may be ambiguous in implementations that permit null elements. However, even in this case the result can be disambiguated by checking contains(null). To avoid such issues, implementations of this interface are encouraged to not permit insertion of null elements. (Note that sorted sets of [Comparable](file:///E:\dev\jdk6docs\api\java\lang\Comparable.html) elements intrinsically do not permit null.)

Methods [subSet(E, E)](file:///E:\\dev\\jdk6docs\\api\\java\\util\\NavigableSet.html" \l "subSet(E, E)), [headSet(E)](file:///E:\\dev\\jdk6docs\\api\\java\\util\\NavigableSet.html" \l "headSet(E)), and [tailSet(E)](file:///E:\\dev\\jdk6docs\\api\\java\\util\\NavigableSet.html" \l "tailSet(E)) are specified to return SortedSet to allow existing implementations of SortedSet to be compatibly retrofitted to implement NavigableSet, but extensions and implementations of this interface are encouraged to override these methods to return NavigableSet.