java.util

Class Scanner

```
java.lang.Object
  extended by java.util.Scanner
```

All Implemented Interfaces: Iterator<String>

```
public final class Scanner extends Object implements Iterator<String>
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

Constructor Summary
`Scanner``(File source)` Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner``(InputStream source)` Constructs a new `Scanner` that produces values scanned from the specified input stream.
`Scanner``(String source)` Constructs a new `Scanner` that produces values scanned from the specified string.

Method Summary	
`void`	**`close`**`()` Closes this scanner.
`boolean`	**`hasNextXXX`**`()` Returns true if there is another line in the input of this scanner. XXX can be nothing or things like Line, Int, Double, ...
`String`	**`next`**`()` Finds and returns the next complete token from this scanner.
`double`	**`nextDouble`**`()` Scans the next token of the input as a `double`. You can exchange "Double" for Int, Line and so on...
`String`	**`nextLine`**`()` Advances this scanner past the current line and returns the input that was skipped.
`String`	**`toString`**`()` Returns the string representation of this `Scanner`.

java.util

Class Arrays

```
java.lang.Object
  extended by java.util.Arrays
```

```
public class Arrays extends Object
```

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a `NullPointerException` if the specified array reference is null, except where noted.

Method Summary

static int	**binarySearch**(int[] a, int key) Searches the specified array of ints for the specified value using the binary search algorithm. Int can be exchanged for Object, Double and so on
static <T> int	**binarySearch**(T[] a, T key, Comparator<? super T> c) Searches the specified array for the specified object using the binary search algorithm.
static boolean	**equals**(int[] a, int[] a2) Returns `true` if the two specified arrays of ints are *equal* to one another. Int can be exchanged for Double ....
static void	**fill**(int[] a, int val) Assigns the specified int value to each element of the specified array of ints.
static void	**fill**(int[] a, int fromIndex, int toIndex, int val) Assigns the specified int value to each element of the specified range of the specified array of ints.
static int	**hashCode**(int[] a) Returns a hash code based on the contents of the specified array.
static void	**sort**(int[] a) Sorts the specified array of ints into ascending numerical order.
static void	**sort**(int[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of ints into ascending numerical order.
static <T> void	**sort**(T[] a, int fromIndex, int toIndex, Comparator<? super T> c) Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.
static String	**toString**(int[] a) Returns a string representation of the contents of the specified array.

java.util

Interface Collection<E>

All Superinterfaces:

Iterable<E>

```
public interface Collection<E>
extends Iterable<E>
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Bags or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.
...

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `contains(Object o)` method says: "returns `true` if and only if this collection contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`." This specification should *not* be construed to imply that invoking `Collection.contains` with a non-null argument `o` will cause `o.equals(e)` to be invoked for any element `e`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two elements. (The `Object.hashCode()` specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying `Object` methods wherever the implementor deems it appropriate.

Method Summary

`boolean`	`add(E o)` Ensures that this collection contains the specified element (optional operation).
`boolean`	`addAll(Collection<? extends E> c)` Adds all of the elements in the specified collection to this collection (optional operation).
`void`	`clear()` Removes all of the elements from this collection (optional operation).
`boolean`	`contains(Object o)` Returns `true` if this collection contains the specified element.
`boolean`	`containsAll(Collection<?> c)` Returns `true` if this collection contains all of the elements in the specified collection.
`boolean`	`equals(Object o)` Compares the specified object with this collection for equality.
`int`	`hashCode()` Returns the hash code value for this collection.
`boolean`	`isEmpty()` Returns `true` if this collection contains no elements.
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this collection.
`boolean`	`remove(Object o)` Removes a single instance of the specified element from this collection, if it is present (optional operation).
`boolean`	`removeAll(Collection<?> c)` Removes all this collection's elements that are also contained in the specified collection (optional operation).
`boolean`	`retainAll(Collection<?> c)` Retains only the elements in this collection that are contained in the specified collection (optional operation).
`int`	`size()` Returns the number of elements in this collection.
`Object[]`	`toArray()` Returns an array containing all of the elements in this collection.
`<T> T[]`	`toArray(T[] a)` Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

java.util

Interface List<E>

All Superinterfaces:
Collection<E>, Iterable<E>

`public interface List<E> extends Collection<E>`

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

Method Summary

`boolean`	**`add`**`(E o)` Appends the specified element to the end of this list (optional operation).
`void`	**`add`**`(int index, E element)` Inserts the specified element at the specified position in this list (optional operation).
`boolean`	**`addAll`**`(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
`boolean`	**`addAll`**`(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
`void`	**`clear`**`()` Removes all of the elements from this list (optional operation).
`boolean`	**`contains`**`(Object o)` Returns `true` if this list contains the specified element.
`boolean`	**`containsAll`**`(Collection<?> c)` Returns `true` if this list contains all of the elements of the specified collection.
`boolean`	**`equals`**`(Object o)` Compares the specified object with this list for equality.
`E`	**`get`**`(int index)` Returns the element at the specified position in this list.
`int`	**`hashCode`**`()` Returns the hash code value for this list.
`int`	**`indexOf`**`(Object o)` Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.
`boolean`	**`isEmpty`**`()` Returns `true` if this list contains no elements.
`Iterator<E>`	**`iterator`**`()` Returns an iterator over the elements in this list in proper sequence.
`int`	**`lastIndexOf`**`(Object o)` Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element.
`ListIterator<E>`	**`listIterator`**`()` Returns a list iterator of the elements in this list (in proper sequence).
`ListIterator<E>`	**`listIterator`**`(int index)` Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.

E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence in this list of the specified element (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this list all the elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

---

java.util

Class Stack<E>

```
java.lang.Object
  extended by java.util.AbstractCollection<E>
      extended by java.util.AbstractList<E>
          extended by java.util.Vector<E>
              extended by java.util.Stack<E>
```

All Implemented Interfaces: Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

---

`public class Stack<E> extends Vector<E>`

The `Stack` class represents a last-in-first-out (LIFO) stack of objects. It extends class `Vector` with five operations that allow a vector to be treated as a stack. The usual `push` and `pop` operations are provided, as well as a method to `peek` at the top item on the stack, a method to test for whether the stack is `empty`, and a method to `search` the stack for an item and discover how far it is from the top.
When a stack is first created, it contains no items.

Constructor Summary

Stack() Creates an empty Stack.

Method Summary

boolean	**empty**() Tests if this stack is empty.

E	peek() Looks at the object at the top of this stack without removing it from the stack.
E	pop() Removes the object at the top of this stack and returns that object as the value of this function.
E	push(E item) Pushes an item onto the top of this stack.
int	search(Object o) Returns the 1-based position where an object is on this stack.

some of the Methods inherited from class java.util.Vector
clear, clone, equals, hashCode, toArray, toString,

java.util

Interface Queue<E>

Type Parameters: `E` - the type of elements held in this collection

All Superinterfaces: Collection<E>, Iterable<E>

```
public interface Queue<E> extends Collection<E>
```

A collection designed for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the *head* of the queue is that element which would be removed by a call to `remove()` or `poll()`. In a FIFO queue, all new elements are inserted at the *tail* of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.

Method Summary	
E	element() Retrieves, but does not remove, the head of this queue.
boolean	offer(E o) Inserts the specified element into this queue, if possible.
E	peek() Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
E	poll() Retrieves and removes the head of this queue, or `null` if this queue is empty.
E	remove() Retrieves and removes the head of this queue.

Methods inherited from interface java.util.Collection
add, addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, remove, removeAll, retainAll, size, toArray, toArray

java.util

Class PriorityQueue<E>

```
java.lang.Object
  extended by java.util.AbstractCollection<E>
      extended by java.util.AbstractQueue<E>
          extended by java.util.PriorityQueue<E>
```

Type Parameters: `E` - the type of elements held in this collection

All Implemented Interfaces: Serializable, Iterable<E>, Collection<E>, Queue<E>

`public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable`

An unbounded priority queue based on a priority heap. This queue orders elements according to an order specified at construction time, which is specified either according to their *natural order* (see `Comparable`), or according to a `Comparator`, depending on which constructor is used. A priority queue does not permit `null` elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

A priority queue is unbounded, but has an internal *capacity* governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

Implementation note: this implementation provides O(log(n)) time for the insertion methods (`offer`, `poll`, `remove()` and `add`) methods; linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

Constructor Summary
`PriorityQueue``()` Creates a `PriorityQueue` with the default initial capacity (11) that orders its elements according to their natural ordering (using `Comparable`).
`PriorityQueue``(Collection<? extends E> c)` Creates a `PriorityQueue` containing the elements in the specified collection.
`PriorityQueue``(int initialCapacity, Comparator<? super E> comparator)` Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to the specified comparator.

Method Summary	
`boolean`	**`add`**`(E o)` Adds the specified element to this queue.
`void`	**`clear`**`()` Removes all elements from the priority queue.
`Comparator<? super E>`	**`comparator`**`()` Returns the comparator used to order this collection, or `null` if this collection is sorted according to its elements natural ordering (using `Comparable`).
`Iterator<E>`	**`iterator`**`()` Returns an iterator over the elements in this queue.
`boolean`	**`offer`**`(E o)` Inserts the specified element into this priority queue.
`E`	**`peek`**`()` Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
`E`	**`poll`**`()` Retrieves and removes the head of this queue, or `null` if this queue is empty.

boolean	remove(Object o) Removes a single instance of the specified element from this queue, if it is present.
int	size() Returns the number of elements in this collection.

java.util

Interface Iterator<E>

All Known Subinterfaces: ListIterator<E>

`public interface` **`Iterator<E>`**

An iterator over a collection.

Method Summary	
boolean	hasNext() Returns `true` if the iteration has more elements.
E	next() Returns the next element in the iteration.
void	remove() Removes from the underlying collection the last element returned by the iterator (optional operation).

java.util

Interface Map<K,V>

All Known Subinterfaces: >ConcurrentMap<K,V>, SortedMap<K,V>

All Known Implementing Classes: HashMap, Hashtable, LinkedHashMap, TreeMap

`public interface` **`Map<K,V>`**

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Nested Class Summary	
static interface	Map.Entry<K,V> A map entry (key-value pair).

Method Summary	
void	clear() Removes all mappings from this map (optional operation).

`boolean`	`containsKey(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	`containsValue(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a set view of the mappings contained in this map.
`boolean`	`equals(Object o)` Compares the specified object with this map for equality.
`V`	`get(Object key)` Returns the value to which this map maps the specified key.
`int`	`hashCode()` Returns the hash code value for this map.
`boolean`	`isEmpty()` Returns `true` if this map contains no key-value mappings.
`Set<K>`	`keySet()` Returns a set view of the keys contained in this map.
`V`	`put(K key, V value)` Associates the specified value with the specified key in this map (optional operation).
`void`	`putAll(Map<? extends K,? extends V> t)` Copies all of the mappings from the specified map to this map (optional operation).
`V`	`remove(Object key)` Removes the mapping for this key from this map if it is present (optional operation).
`int`	`size()` Returns the number of key-value mappings in this map.
`Collection<V>`	`values()` Returns a collection view of the values contained in this map.

java.util

Interface Map.Entry<K,V>

Enclosing interface: Map<K,V>

```
public static interface Map.Entry<K,V>
```

A map entry (key-value pair). The `Map.entrySet` method returns a collection-view of the map, whose elements are of this class. The *only* way to obtain a reference to a map entry is from the iterator of this collection-view. These `Map.Entry` objects are valid *only* for the duration of the iteration; more formally, the behavior of a map entry is undefined if the backing map has been modified after the entry was returned by the iterator, except through the `setValue` operation on the map entry.

Method Summary

`boolean`	`equals(Object o)` Compares the specified object with this entry for equality.
`K`	`getKey()` Returns the key corresponding to this entry.
`V`	`getValue()` Returns the value corresponding to this entry.
`int`	`hashCode()` Returns the hash code value for this map entry.
`V`	`setValue(V value)` Replaces the value corresponding to this entry with the specified value (optional operation).

java.util

Interface ListIterator<E>

All Superinterfaces: Iterator<E>

```
public interface ListIterator<E> extends Iterator<E>
```

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. A `ListIterator` has no current element; its *cursor position* always lies between the element that would be returned by a call to `previous()` and the element that would be returned by a call to `next()`. In a list of length `n`, there are `n+1` valid index values, from `0` to `n`, inclusive.

```
          Element(0)   Element(1)   Element(2)   ... Element(n)
        ^            ^            ^            ^               ^
 Index: 0            1            2            3               n+1tre
```

Note that the `remove()` and `set(Object)` methods are *not* defined in terms of the cursor position; they are defined to operate on the last element returned by a call to `next()` or `previous()`.

Method Summary	
`void`	**`add`**`(E o)` Inserts the specified element into the list (optional operation).
`boolean`	**`hasNext`**`()` Returns `true` if this list iterator has more elements when traversing the list in the forward direction.
`boolean`	**`hasPrevious`**`()` Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
`E`	**`next`**`()` Returns the next element in the list.
`int`	**`nextIndex`**`()` Returns the index of the element that would be returned by a subsequent call to `next`.
`E`	**`previous`**`()` Returns the previous element in the list.
`int`	**`previousIndex`**`()` Returns the index of the element that would be returned by a subsequent call to `previous`.
`void`	**`remove`**`()` Removes from the list the last element that was returned by `next` or `previous` (optional operation).
`void`	**`set`**`(E o)` Replaces the last element returned by `next` or `previous` with the specified element (optional operation).

java.util

Class HashMap<K,V>

```
java.lang.Object
  extended by java.util.AbstractMap<K,V>
      extended by java.util.HashMap<K,V>
```

All Implemented Interfaces: Serializable, Cloneable, Map<K,V>
Direct Known Subclasses: LinkedHashMap, PrinterStateReasons

`public class **HashMap<K,V>** extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable`

Hash table based implementation of the `Map` interface. This implementation provides all of the optional map operations, and permits `null` values and the `null` key. (The `HashMap` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (`get` and `put`), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the `HashMap` instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of `HashMap` has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the `rehash` method.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the `HashMap` class, including `get` and `put`). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of `rehash` operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no `rehash` operations will ever occur.

If many mappings are to be stored in a `HashMap` instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

Constructor Summary
`HashMap``()` Constructs an empty `HashMap` with the default initial capacity (16) and the default load factor (0.75).
`HashMap``(int initialCapacity)` Constructs an empty `HashMap` with the specified initial capacity and the default load factor (0.75).
`HashMap``(int initialCapacity, float loadFactor)` Constructs an empty `HashMap` with the specified initial capacity and load factor.
`HashMap``(Map<? extends K,? extends V> m)` Constructs a new `HashMap` with the same mappings as the specified `Map`.

Method Summary	
`void`	**`clear`**`()` Removes all mappings from this map.
`Object`	**`clone`**`()` Returns a shallow copy of this `HashMap` instance: the keys and values themselves are not cloned.
`boolean`	**`containsKey`**`(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	**`containsValue`**`(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	**`entrySet`**`()` Returns a collection view of the mappings contained in this map.

`V`	`get(Object key)` Returns the value to which the specified key is mapped in this identity hash map, or `null` if the map contains no mapping for this key.
`boolean`	`isEmpty()` Returns `true` if this map contains no key-value mappings.
`Set<K>`	`keySet()` Returns a set view of the keys contained in this map.
`V`	`put(K key, V value)` Associates the specified value with the specified key in this map.
`void`	`putAll(Map<? extends K,? extends V> m)` Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
`V`	`remove(Object key)` Removes the mapping for this key from this map if present.
`int`	`size()` Returns the number of key-value mappings in this map.
`Collection<V>`	`values()` Returns a collection view of the values contained in this map.

java.util

Class TreeMap<K,V>

```
java.lang.Object
  extended by java.util.AbstractMap<K,V>
      extended by java.util.TreeMap<K,V>
```

All Implemented Interfaces: Serializable, Cloneable, Map<K,V>, SortedMap<K,V>

`public class TreeMap<K,V> extends AbstractMap<K,V> implements SortedMap<K,V>, Cloneable, Serializable`

Red-Black tree based implementation of the `SortedMap` interface. This class guarantees that the map will be in ascending key order, sorted according to the *natural order* for the key's class (see `Comparable`), or by the comparator provided at creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

Constructor Summary

`TreeMap()` Constructs a new, empty map, sorted according to the keys' natural order.
`TreeMap(Comparator<? super K> c)` Constructs a new, empty map, sorted according to the given comparator.
`TreeMap(Map<? extends K,? extends V> m)` Constructs a new map containing the same mappings as the given map, sorted according to the keys' *natural order*.
`TreeMap(SortedMap<K,? extends V> m)` Constructs a new map containing the same mappings as the given `SortedMap`, sorted according to the same ordering.

Method Summary

`void`	`clear()` Removes all mappings from this TreeMap.

Object	**clone**() Returns a shallow copy of this `TreeMap` instance.
Comparator<? super K>	**comparator**() Returns the comparator used to order this map, or `null` if this map uses its keys' natural order.
boolean	**containsKey**(Object key) Returns `true` if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns `true` if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet**() Returns a set view of the mappings contained in this map.
K	**firstKey**() Returns the first (lowest) key currently in this sorted map.
V	**get**(Object key) Returns the value to which this map maps the specified key.
SortedMap<K,V>	**headMap**(K toKey) Returns a view of the portion of this map whose keys are strictly less than `toKey`.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
K	**lastKey**() Returns the last (highest) key currently in this sorted map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map.
void	**putAll**(Map<? extends K,? extends V> map) Copies all of the mappings from the specified map to this map.
V	**remove**(Object key) Removes the mapping for this key from this TreeMap if present.
int	**size**() Returns the number of key-value mappings in this map.
SortedMap<K,V>	**subMap**(K fromKey, K toKey) Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
SortedMap<K,V>	**tailMap**(K fromKey) Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`.
Collection<V>	**values**() Returns a collection view of the values contained in this map.