Java Collections

The Java platform includes a collections framework. A collection is an object that represents a group of objects (such as the classic Vector class). A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

The primary advantages of a collections framework are that it:

* Reduces programming effort by providing data structures and algorithms so you don't have to write them yourself.
* Increases performance by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
* Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.
* Reduces the effort required to learn APIs by requiring you to learn multiple ad hoc collection APIs.
* Reduces the effort required to design and implement APIs by not requiring you to produce ad hoc collections APIs.
* Fosters software reuse by providing a standard interface for collections and algorithms with which to manipulate them.

The collections framework consists of:

* Collection interfaces. Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.
* General-purpose implementations. Primary implementations of the collection interfaces.
* Legacy implementations. The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.
* Special-purpose implementations. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
* Concurrent implementations. Implementations designed for highly concurrent use.
* Wrapper implementations. Add functionality, such as synchronization, to other implementations.
* Convenience implementations. High-performance "mini-implementations" of the collection interfaces.
* Abstract implementations. Partial implementations of the collection interfaces to facilitate custom implementations.
* Algorithms. Static methods that perform useful functions on collections, such as sorting a list.
* Infrastructure. Interfaces that provide essential support for the collection interfaces.
* Array Utilities. Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

# Collection Interfaces

The collection interfaces are divided into two groups. The most basic interface, java.util.Collection, has the following descendants:

* java.util.Set
* java.util.SortedSet
* java.util.NavigableSet
* java.util.Queue
* java.util.concurrent.BlockingQueue
* java.util.concurrent.TransferQueue
* java.util.Deque
* java.util.concurrent.BlockingDeque

The other collection interfaces are based on java.util.Map and are not true collections. However, these interfaces contain collection-view operations, which enable them to be manipulated as collections. Map has the following offspring:

* java.util.SortedMap
* java.util.NavigableMap
* java.util.concurrent.ConcurrentMap
* java.util.concurrent.ConcurrentNavigableMap

Many of the modification methods in the collection interfaces are labeled optional. Implementations are permitted to not perform one or more of these operations, throwing a runtime exception

(UnsupportedOperationException) if they are attempted. The documentation for each implementation must specify which optional operations are supported. Several terms are introduced to aid in this specification:

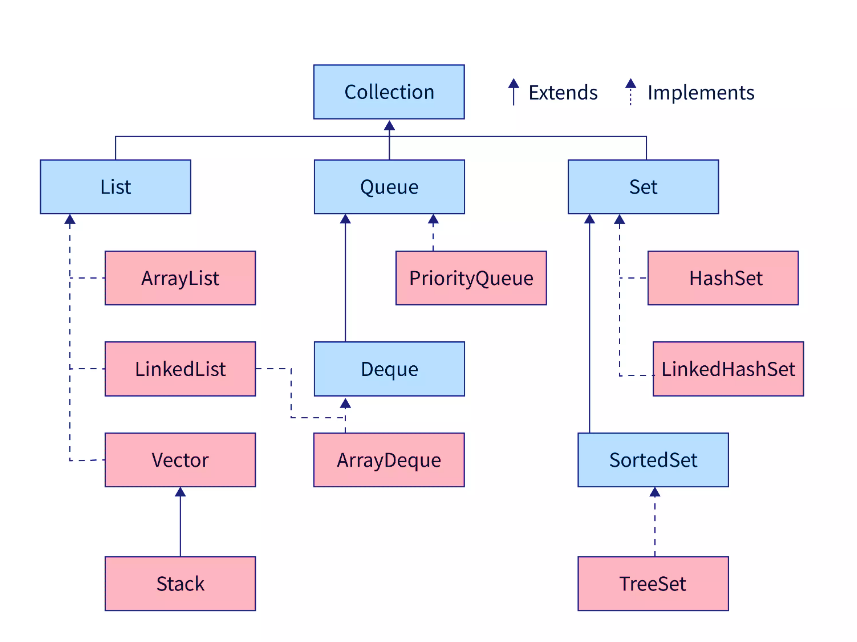
* Collections that do not support modification operations (such as add, remove and clear) are referred to as unmodifiable. Collections that are not unmodifiable are modifiable.
* Collections that additionally guarantee that no change in the Collection object will be visible are referred to as immutable. Collections that are not immutable are mutable.
* Lists that guarantee that their size remains constant even though the elements can change are referred to as fixed-size. Lists that are not fixed-size are referred to as variable-size.
* Lists that support fast (generally constant time) indexed element access are known as random access lists. Lists that do not support fast indexed element access are known as sequential access lists. The RandomAccess marker interface enables lists to advertise the fact that they support random access. This enables generic algorithms to change their behavior to provide good performance when applied to either random or sequential access lists.

Some implementations restrict what elements (or in the case of Maps, keys and values) can be stored. Possible restrictions include requiring elements to:

* Be of a particular type.
* Be not null.
* Obey some arbitrary predicate.

Attempting to add an element that violates an implementation's restrictions results in a runtime exception, typically a ClassCastException, an IllegalArgumentException, or a NullPointerException. Attempting to remove or test for the presence of an element that violates an implementation's restrictions can result in an exception. Some restricted collections permit this usage.

Following diagram shows the collection interface hierarchy:

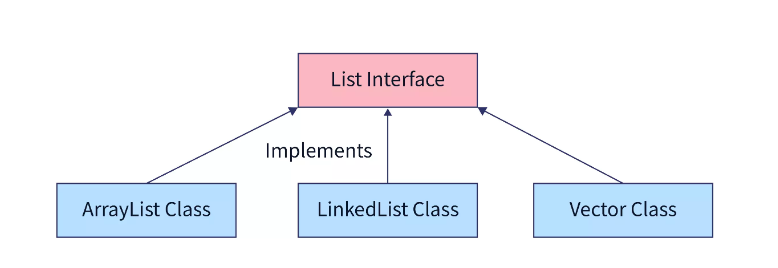


## Collection Interface

The Collection interface is the base of the collection classes and interfaces. The List, Queue, and Set interfaces extends the Collection interface. It provides common methods like add(), remove(), clear(), size(), iterator(), contains() etc.

## List Interface

The list interface extends the collection interface. A list is used to store an ordered collection of data and it may contain duplicates. Ordered collection means the order in which the elements are being inserted and they contain a specific value. The elements present can be accessed or inserted by their position in the list using zero-based indexing. The list interface is implemented by LinkedList, ArrayList, Vectors and Stack classes.



**Stack Class**

### ArrayList class

It is the resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null.

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

**Constructor and Description**

* ArrayList()

Constructs an empty list with an initial capacity of ten.

* ArrayList(Collection<? extends E> c)

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

* ArrayList(int initialCapacity)

Constructs an empty list with the specified initial capacity.

|  |  |  |
| --- | --- | --- |
| **Return type** | **Method** | **Description** |
| boolean | add(E e) | Appends the specified element to the end of this list. |
| void | add(int index, E element) | Inserts the specified element at the specified position in this list. |
| 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. |
| void | clear() | Removes all of the elements from this list. |
| boolean | contains(Object o) | Returns true if this list contains the specified element. |
| void | ensureCapacity(int minCapacity) | If necessary, increase the capacity of this ArrayList instance to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| E | get(int index) | Returns the element at the specified position in this list. |
| int | indexOf(Object o) | Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the 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 of the last occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| E | remove(int index) | Removes the element at the specified position in this list. |
| boolean | remove(Object o) | Removes the first occurrence of the specified element from this list, if it is present. |
| E | set(int index, E element) | Replaces the element at the specified position in this list with the specified element. |
| int | size() | Returns the number of elements in this list. |

### LinkedList class

It is the Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

**Constructor and Description**

* LinkedList()

Constructs an empty list.

* LinkedList(Collection<? extends E> c)

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Method** | **Description** |
| boolean | add(E e) | Appends the specified element to the end of this list. |
| void | add(int index, E element) | Inserts the specified element at the specified position in this list. |
| 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. |
| void | addFirst(E e) | Inserts the specified element at the beginning of this list. |
| void | addLast(E e) | Appends the specified element to the end of this list. |
| void | clear() | Removes all of the elements from this list. |
| boolean | contains(Object o) | Returns true if this list contains the specified element. |
| E | element() | Retrieves, but does not remove, the head (first element) of this list. |
| E | get(int index) | Returns the element at the specified position in this list. |
| E | getFirst() | Returns the first element in this list. |
| E | getLast() | Returns the last element in this list. |
| int | indexOf(Object o) | Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| int | lastIndexOf(Object o) | Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| boolean | offer(E e) | Adds the specified element as the tail (last element) of this list. |
| boolean | offerFirst(E e) | Inserts the specified element at the front of this list. |
| boolean | offerLast(E e) | Inserts the specified element at the end of this list. |
| E | peek() | Retrieves, but does not remove, the head (first element) of this list. |
| E | peekFirst() | Retrieves, but does not remove, the first element of this list, or returns null if this list is empty. |
| E | peekLast() | Retrieves, but does not remove, the last element of this list, or returns null if this list is empty. |
| E | poll() | Retrieves and removes the head (first element) of this list. |
| E | pollFirst() | Retrieves and removes the first element of this list, or returns null if this list is empty. |
| E | pollLast() | Retrieves and removes the last element of this list, or returns null if this list is empty. |
| E | pop() | Pops an element from the stack represented by this list. |
| void | push(E e) | Pushes an element onto the stack represented by this list. |
| E | remove() | Retrieves and removes the head (first element) of this list. |
| E | remove(int index) | Removes the element at the specified position in this list. |
| boolean | remove(Object o) | Removes the first occurrence of the specified element from this list, if it is present. |
| E | removeFirst() | Removes and returns the first element from this list. |
| boolean | removeFirstOccurrence(Object o) | Removes the first occurrence of the specified element in this list (when traversing the list from head to tail). |
| E | removeLast() | Removes and returns the last element from this list. |
| boolean | removeLastOccurrence(Object o) | Removes the last occurrence of the specified element in this list (when traversing the list from head to tail). |
| E | set(int index, E element) | Replaces the element at the specified position in this list with the specified element. |
| int | size() | Returns the number of elements in this list. |

### Stack Class

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.

A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class.

**Constructor and Description**

Stack()

Creates an empty Stack.

|  |  |  |
| --- | --- | --- |
| **Return type** | **Method** | **Decription** |
| 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. |

## Queue interface

A collection designed for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations; in most implementations, insert operations cannot fail.

Summary of Queue methods

|  |  |  |
| --- | --- | --- |
|  | **Throws exception** | **Returns special value** |
| Insert | add(e) | offer(e) |
| Remove | remove() | poll() |
| Examine | element() | peek() |

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.

### PriorityQueue class

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, 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.

This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces. The Iterator provided in method iterator() is not guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays.sort(pq.toArray()).

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

**Constructor and Description**

* PriorityQueue()

Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.

* PriorityQueue(Collection<? extends E> c)

Creates a PriorityQueue containing the elements in the specified collection.

* PriorityQueue(Comparator<? super E> comparator)

Creates a PriorityQueue with the default initial capacity and whose elements are ordered according to the specified comparator.

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Method** | **Description** |
| boolean | add(E e) | Inserts the specified element into this priority queue. |
| void | clear() | Removes all of the elements from this priority queue. |
| boolean | contains(Object o) | Returns true if this queue contains the specified element. |
| Iterator<E> | iterator() | Returns an iterator over the elements in this queue. |
| boolean | offer(E e) | Inserts the specified element into this priority queue. |
| E | peek() | Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |
| E | poll() | Retrieves and removes the head of this queue, or returns 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. |
| Object[] | toArray() | Returns an array containing all of the elements in this queue. |
| <T> T[] | toArray(T[] a) | Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array. |

### ArrayDeque

It is resizable-array implementation of the Deque interface. Array deques have no capacity restrictions; they grow as necessary to support usage. Null elements are prohibited. This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue. Most ArrayDeque operations run in amortized constant time. Exceptions include remove, removeFirstOccurrence, removeLastOccurrence, contains, iterator.remove(), and the bulk operations, all of which run in linear time.

**Constructor and Description**

* ArrayDeque()

Constructs an empty array deque with an initial capacity sufficient to hold 16 elements.

* ArrayDeque(Collection<? extends E> c)

Constructs a deque containing the elements of the specified collection, in the order they are returned by the collection's iterator.

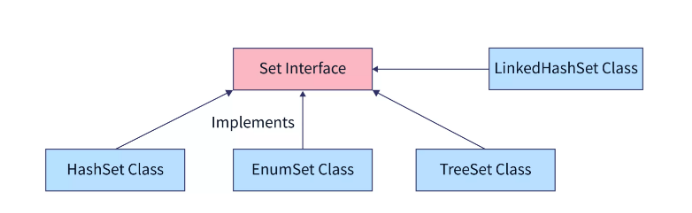
* ArrayDeque(int numElements)

Constructs an empty array deque with an initial capacity sufficient to hold the specified number of elements.

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Method Name** | **Description** |
| boolean | add(E e) | Inserts the specified element at the end of this deque. |
| void | addFirst(E e) | Inserts the specified element at the front of this deque. |
| void | addLast(E e) | Inserts the specified element at the end of this deque. |
| void | clear() | Removes all of the elements from this deque. |
| ArrayDeque<E> | clone() | Returns a copy of this deque. |
| boolean | contains(Object o) | Returns true if this deque contains the specified element. |
| E | element() | Retrieves, but does not remove, the head of the queue represented by this deque. |
| E | getFirst() | Retrieves, but does not remove, the first element of this deque. |
| E | getLast() | Retrieves, but does not remove, the last element of this deque. |
| boolean | isEmpty() | Returns true if this deque contains no elements. |
| Iterator<E> | iterator() | Returns an iterator over the elements in this deque. |
| boolean | offer(E e) | Inserts the specified element at the end of this deque. |
| boolean | offerFirst(E e) | Inserts the specified element at the front of this deque. |
| boolean | offerLast(E e) | Inserts the specified element at the end of this deque. |
| E | peek() | Retrieves, but does not remove, the head of the queue represented by this deque, or returns null if this deque is empty. |
| E | peekFirst() | Retrieves, but does not remove, the first element of this deque, or returns null if this deque is empty. |
| E | peekLast() | Retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty. |
| E | poll() | Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty. |
| E | pollFirst() | Retrieves and removes the first element of this deque, or returns null if this deque is empty. |
| E | pollLast() | Retrieves and removes the last element of this deque, or returns null if this deque is empty. |
| E | pop() | Pops an element from the stack represented by this deque. |
| void | push(E e) | Pushes an element onto the stack represented by this deque. |
| E | remove() | Retrieves and removes the head of the queue represented by this deque. |
| boolean | remove(Object o) | Removes a single instance of the specified element from this deque. |
| E | removeFirst() | Retrieves and removes the first element of this deque. |
| boolean | removeFirstOccurrence(Object o) | Removes the first occurrence of the specified element in this deque (when traversing the deque from head to tail). |
| E | removeLast() | Retrieves and removes the last element of this deque. |
| boolean | removeLastOccurrence(Object o) | Removes the last occurrence of the specified element in this deque (when traversing the deque from head to tail). |
| int | size() | Returns the number of elements in this deque. |
| Object[] | toArray() | Returns an array containing all of the elements in this deque in proper sequence (from first to last element). |
| <T> T[] | toArray(T[] a) | Returns an array containing all of the elements in this deque in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array. |

## Set Interface

A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.



### HashSet class

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

**Constructor and Description**

* HashSet()

Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).

* HashSet(Collection<? extends E> c)

Constructs a new set containing the elements in the specified collection.

* HashSet(int initialCapacity)

Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and default load factor (0.75).

* HashSet(int initialCapacity, float loadFactor)

Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor.

|  |  |  |
| --- | --- | --- |
| boolean | add(E e) | Adds the specified element to this set if it is not already present. |
| void | clear() | Removes all of the elements from this set. |
| Object | clone() | Returns a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| boolean | contains(Object o) | Returns true if this set contains the specified element. |
| boolean | isEmpty() | Returns true if this set contains no elements. |
| Iterator<E> | iterator() | Returns an iterator over the elements in this set. |
| boolean | remove(Object o) | Removes the specified element from this set if it is present. |
| int | size() | Returns the number of elements in this set (its cardinality). |

### LinkedHashSet class

Hash table and linked list implementation of the Set interface, with predictable iteration order. This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order). Note that insertion order is not affected if an element is re-inserted into the set.

This class provides all of the optional Set operations, and permits null elements. Like HashSet, it provides constant-time performance for the basic operations (add, contains and remove), assuming the hash function disperses elements properly among the buckets. Performance is likely to be just slightly below that of HashSet, due to the added expense of maintaining the linked list, with one exception: Iteration over a LinkedHashSet requires time proportional to the size of the set, regardless of its capacity. Iteration over a HashSet is likely to be more expensive, requiring time proportional to its capacity.

**Constructor and Description**

* LinkedHashSet()

Constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75).

* LinkedHashSet(Collection<? extends E> c)

Constructs a new linked hash set with the same elements as the specified collection.

* LinkedHashSet(int initialCapacity)

Constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75).

* LinkedHashSet(int initialCapacity, float loadFactor)

Constructs a new, empty linked hash set with the specified initial capacity and load factor.

### TreeSet class

A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

**Constructor and Description**

* TreeSet()

Constructs a new, empty tree set, sorted according to the natural ordering of its elements.

* TreeSet(Collection<? extends E> c)

Constructs a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements.

* TreeSet(Comparator<? super E> comparator)

Constructs a new, empty tree set, sorted according to the specified comparator.

* TreeSet(SortedSet<E> s)

Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Method** | **Description** |
| boolean | add(E e) | Adds the specified element to this set if it is not already present. |
| boolean | addAll(Collection<? extends E> c) | Adds all of the elements in the specified collection to this set. |
| E | ceiling(E e) | Returns the least element in this set greater than or equal to the given element, or null if there is no such element. |
| void | clear() | Removes all of the elements from this set. |
| Object | clone() | Returns a shallow copy of this TreeSet instance. |
| boolean | contains(Object o) | Returns true if this set contains the specified element. |
| E | first() | Returns the first (lowest) element currently in this set. |
| E | floor(E e) | Returns the greatest element in this set less than or equal to the given element, or null if there is no such element. |
| E | higher(E e) | Returns the least element in this set strictly greater than the given element, or null if there is no such element. |
| boolean | isEmpty() | Returns true if this set contains no elements. |
| Iterator<E> | iterator() | Returns an iterator over the elements in this set in ascending order. |
| E | last() | Returns the last (highest) element currently in this set. |
| E | lower(E e) | Returns the greatest element in this set strictly less than the given element, or null if there is no such element. |
| E | pollFirst() | Retrieves and removes the first (lowest) element, or returns null if this set is empty. |
| E | pollLast() | Retrieves and removes the last (highest) element, or returns null if this set is empty. |
| boolean | remove(Object o) | Removes the specified element from this set if it is present. |
| int | size() | Returns the number of elements in this set (its cardinality). |