Java SE 8 Documentation

FAHIM AHMED

Java Documentaion.pdf

Collection (Java Platform SE 8).pdf

Comparator (Java Platform SE 8).pdf

Iterator (Java Platform SE 8).pdf

Deque (Java Platform SE 8).pdf

List (Java Platform SE 8).pdf

ListIterator (Java Platform SE 8).pdf

Map (Java Platform SE 8).pdf

Map.pdf

Queue (Java Platform SE 8).pdf

RandomAccess (Java Platform SE 8).pdf

Set (Java Platform SE 8).pdf

SortedMap (Java Platform SE 8).pdf

SortedSet (Java Platform SE 8).pdf

AbstractCollection (Java Platform SE 8).pdf

AbstractList (Java Platform SE 8).pdf

AbstractMap (Java Platform SE 8).pdf

AbstractMap.pdf

AbstractQueue (Java Platform SE 8).pdf

AbstractSequentialList (Java Platform SE 8).pdf

AbstractSet (Java Platform SE 8).pdf

ArrayDeque (Java Platform SE 8).pdf

ArrayList (Java Platform SE 8).pdf

Arrays (Java Platform SE 8).pdf

BitSet (Java Platform SE 8).pdf

Collections (Java Platform SE 8).pdf

Dictionary (Java Platform SE 8).pdf

HashMap (Java Platform SE 8).pdf

HashSet (Java Platform SE 8).pdf

LinkedList (Java Platform SE 8).pdf

PriorityQueue (Java Platform SE 8).pdf

Stack (Java Platform SE 8).pdf

TreeMap (Java Platform SE 8).pdf

TreeSet (Java Platform SE 8).pdf

Vector (Java Platform SE 8).pdf

Integer (Java Platform SE 8).pdf

Double (Java Platform SE 8).pdf

Character (Java Platform SE 8).pdf

String (Java Platform SE 8).pdf

StringBuffer (Java Platform SE 8).pdf

Math (Java Platform SE 8).pdf

compact1, compact2, compact3

java.util

Interface Collection<E>

Type Parameters:

`E - the type of elements in this collection`

All Superinterfaces:

`Iterable<E>`

All Known Subinterfaces:

`BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>`

All Known Implementing Classes:

`AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentHashMap.KeySetView, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector`

```
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.

All general-purpose `Collection` implementation classes (which typically implement `Collection` indirectly through one of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type `Collection`, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose `Collection` implementations in the Java platform libraries comply.

The "destructive" methods contained in this interface, that is, the methods that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if this collection does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the collection. For example, invoking the `addAll(Collection)` method on an unmodifiable collection may, but is not required to, throw the exception if the collection to be added is empty.

Some collection implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the collection may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

It is up to each collection to determine its own synchronization policy. In the absence of a stronger guarantee by the implementation, undefined behavior may result from the invocation of any method on a collection that is being mutated by another thread; this includes direct invocations, passing the collection to a method that might perform invocations, and using an existing iterator to examine the collection.

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.

Some collection operations which perform recursive traversal of the collection may fail with an exception for self-referential instances where the collection directly or indirectly contains itself. This includes the `clone()`, `equals()`, `hashCode()` and `toString()` methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Implementation Requirements:

`The default method implementations (inherited or otherwise) do not apply any synchronization protocol. If a Collection implementation has a specific synchronization protocol, then it must override default implementations to apply that protocol.`

Since:

`1.2`

See Also:

`Set`, `List`, `Map`, `SortedSet`, `SortedMap`, `HashSet`, `TreeSet`, `ArrayList`, `LinkedList`, `Vector`, `Collections`, `Arrays`, `AbstractCollection`

Method Summary

All Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) 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.
default **Stream**<**E**>	**parallelStream**() Returns a possibly parallel Stream with this collection as its source.
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 of this collection's elements that are also contained in the specified collection (optional operation).
default boolean	**removeIf**(**Predicate**<? super **E**> filter) Removes all of the elements of this collection that satisfy the given predicate.
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.
default **Spliterator**<**E**>	**spliterator**() Creates a **Spliterator** over the elements in this collection.
default **Stream**<**E**>	**stream**() Returns a sequential Stream with this collection as its source.
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.

Methods inherited from interface java.lang.Iterable

forEach

Method Detail

size

```
int size()
```

Returns the number of elements in this collection. If this collection contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Returns:

the number of elements in this collection

isEmpty

```
boolean isEmpty()
```

Returns `true` if this collection contains no elements.

Returns:

`true` if this collection contains no elements

contains

```
boolean contains(Object o)
```

Returns `true` if this collection contains the specified element. More formally, returns `true` if and only if this collection contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

Parameters:

`o` - element whose presence in this collection is to be tested

Returns:

`true` if this collection contains the specified element

Throws:

`ClassCastException` - if the type of the specified element is incompatible with this collection (`optional`)

`NullPointerException` - if the specified element is null and this collection does not permit null elements (`optional`)

iterator

```
Iterator<E> iterator()
```

Returns an iterator over the elements in this collection. There are no guarantees concerning the order in which the elements are returned (unless this collection is an instance of some class that provides a guarantee).

Specified by:

`iterator` in interface `Iterable<E>`

Returns:

an Iterator over the elements in this collection

toArray

`Object[] toArray()`

Returns an array containing all of the elements in this collection. If this collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

The returned array will be "safe" in that no references to it are maintained by this collection. (In other words, this method must allocate a new array even if this collection is backed by an array). The caller is thus free to modify the returned array.

This method acts as bridge between array-based and collection-based APIs.

Returns:

an array containing all of the elements in this collection

toArray

`<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. If the collection fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this collection.

If this collection fits in the specified array with room to spare (i.e., the array has more elements than this collection), the element in the array immediately following the end of the collection is set to `null`. (This is useful in determining the length of this collection *only* if the caller knows that this collection does not contain any `null` elements.)

If this collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

Like the `toArray()` method, this method acts as bridge between array-based and collection-based APIs. Further, this method allows precise control over the runtime type of the output array, and may, under certain circumstances, be used to save allocation costs.

Suppose `x` is a collection known to contain only strings. The following code can be used to dump the collection into a newly allocated array of `String`:

```
String[] y = x.toArray(new String[0]);
```

Note that `toArray(new Object[0])` is identical in function to `toArray()`.

Type Parameters:

T - the runtime type of the array to contain the collection

Parameters:

a - the array into which the elements of this collection are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose.

Returns:

an array containing all of the elements in this collection

Throws:

ArrayStoreException - if the runtime type of the specified array is not a supertype of the runtime type of every element in this collection

NullPointerException - if the specified array is null

add

`boolean add(E e)`

Ensures that this collection contains the specified element (optional operation). Returns `true` if this collection changed as a result of the call. (Returns `false` if this collection does not permit duplicates and already contains the specified element.)

Collections that support this operation may place limitations on what elements may be added to this collection. In particular, some collections will refuse to add `null` elements, and others will impose restrictions on the type of elements that may be added. Collection classes should clearly specify in their documentation any restrictions on what elements may be added.

If a collection refuses to add a particular element for any reason other than that it already contains the element, it *must* throw an exception (rather than returning `false`). This preserves the invariant that a collection always contains the specified element after this call returns.

Parameters:

`e - element whose presence in this collection is to be ensured`

Returns:

`true if this collection changed as a result of the call`

Throws:

`UnsupportedOperationException - if the add operation is not supported by this collection`

`ClassCastException - if the class of the specified element prevents it from being added to this collection`

`NullPointerException - if the specified element is null and this collection does not permit null elements`

`IllegalArgumentException - if some property of the element prevents it from being added to this collection`

`IllegalStateException - if the element cannot be added at this time due to insertion restrictions`

remove

`boolean remove(Object o)`

Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element `e` such that `(o==null ? e==null : o.equals(e))`, if this collection contains one or more such elements. Returns `true` if this collection contained the specified element (or equivalently, if this collection changed as a result of the call).

Parameters:

`o - element to be removed from this collection, if present`

Returns:

`true if an element was removed as a result of this call`

Throws:

`ClassCastException - if the type of the specified element is incompatible with this collection (optional)`

`NullPointerException - if the specified element is null and this collection does not permit null elements (optional)`

UnsupportedOperationException - if the remove operation is not supported by this collection

containsAll

boolean containsAll(Collection<?> c)

Returns true if this collection contains all of the elements in the specified collection.

Parameters:

c - collection to be checked for containment in this collection

Returns:

true if this collection contains all of the elements in the specified collection

Throws:

ClassCastException - if the types of one or more elements in the specified collection are incompatible with this collection (optional)

NullPointerException - if the specified collection contains one or more null elements and this collection does not permit null elements (optional), or if the specified collection is null.

See Also:

contains(Object)

addAll

boolean addAll(Collection<? extends E> c)

Adds all of the elements in the specified collection to this collection (optional operation). The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified collection is this collection, and this collection is nonempty.)

Parameters:

c - collection containing elements to be added to this collection

Returns:

true if this collection changed as a result of the call

Throws:

UnsupportedOperationException - if the addAll operation is not supported by this collection

ClassCastException - if the class of an element of the specified collection prevents it from being added to this collection

NullPointerException - if the specified collection contains a null element and this collection does not permit null elements, or if the specified collection is null

IllegalArgumentException - if some property of an element of the specified collection prevents it from being added to this collection

IllegalStateException - if not all the elements can be added at this time due to insertion restrictions

See Also:

add(Object)

removeAll

`boolean removeAll(Collection<?> c)`

Removes all of this collection's elements that are also contained in the specified collection (optional operation). After this call returns, this collection will contain no elements in common with the specified collection.

Parameters:

`c - collection containing elements to be removed from this collection`

Returns:

`true if this collection changed as a result of the call`

Throws:

`UnsupportedOperationException - if the removeAll method is not supported by this collection`

`ClassCastException - if the types of one or more elements in this collection are incompatible with the specified collection (optional)`

`NullPointerException - if this collection contains one or more null elements and the specified collection does not support null elements (optional), or if the specified collection is null`

See Also:

`remove(Object), contains(Object)`

removeIf

`default boolean removeIf(Predicate<? super E> filter)`

Removes all of the elements of this collection that satisfy the given predicate. Errors or runtime exceptions thrown during iteration or by the predicate are relayed to the caller.

Implementation Requirements:

`The default implementation traverses all elements of the collection using its iterator(). Each matching element is removed using Iterator.remove(). If the collection's iterator does not support removal then an UnsupportedOperationException will be thrown on the first matching element.`

Parameters:

`filter - a predicate which returns true for elements to be removed`

Returns:

`true if any elements were removed`

Throws:

`NullPointerException - if the specified filter is null`

`UnsupportedOperationException - if elements cannot be removed from this collection. Implementations may throw this exception if a matching element cannot be removed or if, in general, removal is not supported.`

Since:

`1.8`

retainAll

`boolean retainAll(Collection<?> c)`

Retains only the elements in this collection that are contained in the specified collection (optional operation). In other words, removes from this collection all of its elements that are not

contained in the specified collection.

Parameters:

`c - collection containing elements to be retained in this collection`

Returns:

`true if this collection changed as a result of the call`

Throws:

`UnsupportedOperationException - if the retainAll operation is not supported by this collection`

`ClassCastException - if the types of one or more elements in this collection are incompatible with the specified collection (optional)`

`NullPointerException - if this collection contains one or more null elements and the specified collection does not permit null elements (optional), or if the specified collection is null`

See Also:

`remove(Object), contains(Object)`

clear

```
void clear()
```

Removes all of the elements from this collection (optional operation). The collection will be empty after this method returns.

Throws:

`UnsupportedOperationException - if the clear operation is not supported by this collection`

equals

```
boolean equals(Object o)
```

Compares the specified object with this collection for equality.

While the `Collection` interface adds no stipulations to the general contract for the `Object.equals`, programmers who implement the `Collection` interface "directly" (in other words, create a class that is a `Collection` but is not a `Set` or a `List`) must exercise care if they choose to override the `Object.equals`. It is not necessary to do so, and the simplest course of action is to rely on `Object`'s implementation, but the implementor may wish to implement a "value comparison" in place of the default "reference comparison." (The `List` and `Set` interfaces mandate such value comparisons.)

The general contract for the `Object.equals` method states that equals must be symmetric (in other words, `a.equals(b)` if and only if `b.equals(a)`). The contracts for `List.equals` and `Set.equals` state that lists are only equal to other lists, and sets to other sets. Thus, a custom `equals` method for a collection class that implements neither the `List` nor `Set` interface must return `false` when this collection is compared to any list or set. (By the same logic, it is not possible to write a class that correctly implements both the `Set` and `List` interfaces.)

Overrides:

`equals in class Object`

Parameters:

`o - object to be compared for equality with this collection`

Returns:

`true` if the specified object is equal to this collection

See Also:

`Object.equals(Object)`, `Set.equals(Object)`, `List.equals(Object)`

hashCode

`int hashCode()`

Returns the hash code value for this collection. While the `Collection` interface adds no stipulations to the general contract for the `Object.hashCode` method, programmers should take note that any class that overrides the `Object.equals` method must also override the `Object.hashCode` method in order to satisfy the general contract for the `Object.hashCode` method. In particular, `c1.equals(c2)` implies that `c1.hashCode()==c2.hashCode()`.

Overrides:

`hashCode` in class `Object`

Returns:

the hash code value for this collection

See Also:

`Object.hashCode()`, `Object.equals(Object)`

spliterator

`default Spliterator<E> spliterator()`

Creates a `Spliterator` over the elements in this collection. Implementations should document characteristic values reported by the spliterator. Such characteristic values are not required to be reported if the spliterator reports `Spliterator.SIZED` and this collection contains no elements.

The default implementation should be overridden by subclasses that can return a more efficient spliterator. In order to preserve expected laziness behavior for the `stream()` and `parallelStream()`} methods, spliterators should either have the characteristic of `IMMUTABLE` or `CONCURRENT`, or be *late-binding*. If none of these is practical, the overriding class should describe the spliterator's documented policy of binding and structural interference, and should override the `stream()` and `parallelStream()` methods to create streams using a `Supplier` of the spliterator, as in:

```
Stream<E> s = StreamSupport.stream(() -> spliterator(), spliteratorCharacteristics)
```

These requirements ensure that streams produced by the `stream()` and `parallelStream()` methods will reflect the contents of the collection as of initiation of the terminal stream operation.

Specified by:

`spliterator` in interface `Iterable<E>`

Implementation Requirements:

The default implementation creates a *late-binding* spliterator from the collections's Iterator. The spliterator inherits the *fail-fast* properties of the collection's iterator.

The created Spliterator reports `Spliterator.SIZED`.

Implementation Note:

The created Spliterator additionally reports Spliterator.SUBSIZED.

If a spliterator covers no elements then the reporting of additional characteristic values, beyond that of SIZED and SUBSIZED, does not aid clients to control, specialize or simplify computation. However, this does enable shared use of an immutable and empty spliterator instance (see Spliterators.emptySpliterator()) for empty collections, and enables clients to determine if such a spliterator covers no elements.

Returns:

a Spliterator over the elements in this collection

Since:

1.8

stream

```
default Stream<E> stream()
```

Returns a sequential Stream with this collection as its source.

This method should be overridden when the spliterator() method cannot return a spliterator that is IMMUTABLE, CONCURRENT, or *late-binding*. (See spliterator() for details.)

Implementation Requirements:

The default implementation creates a sequential Stream from the collection's Spliterator.

Returns:

a sequential Stream over the elements in this collection

Since:

1.8

parallelStream

```
default Stream<E> parallelStream()
```

Returns a possibly parallel Stream with this collection as its source. It is allowable for this method to return a sequential stream.

This method should be overridden when the spliterator() method cannot return a spliterator that is IMMUTABLE, CONCURRENT, or *late-binding*. (See spliterator() for details.)

Implementation Requirements:

The default implementation creates a parallel Stream from the collection's Spliterator.

Returns:

a possibly parallel Stream over the elements in this collection

Since:

1.8

Submit a bug or feature

For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

compact1, compact2, compact3

java.util

Interface Comparator<T>

Type Parameters:

`T - the type of objects that may be compared by this comparator`

All Known Implementing Classes:

`Collator`, `RuleBasedCollator`

Functional Interface:

`This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.`

```
@FunctionalInterface
public interface Comparator<T>
```

A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as `Collections.sort` or `Arrays.sort`) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as `sorted sets` or `sorted maps`), or to provide an ordering for collections of objects that don't have a `natural ordering`.

The ordering imposed by a comparator `c` on a set of elements `S` is said to be *consistent with equals* if and only if `c.compare(e1, e2)==0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` in `S`.

Caution should be exercised when using a comparator capable of imposing an ordering inconsistent with equals to order a sorted set (or sorted map). Suppose a sorted set (or sorted map) with an explicit comparator `c` is used with elements (or keys) drawn from a set `S`. If the ordering imposed by `c` on `S` is inconsistent with equals, the sorted set (or sorted map) will behave "strangely." In particular the sorted set (or sorted map) will violate the general contract for set (or map), which is defined in terms of `equals`.

For example, suppose one adds two elements `a` and `b` such that `(a.equals(b) && c.compare(a, b) != 0)` to an empty `TreeSet` with comparator `c`. The second `add` operation will return true (and the size of the tree set will increase) because `a` and `b` are not equivalent from the tree set's perspective, even though this is contrary to the specification of the `Set.add` method.

Note: It is generally a good idea for comparators to also implement `java.io.Serializable`, as they may be used as ordering methods in serializable data structures (like `TreeSet`, `TreeMap`). In order for the data structure to serialize successfully, the comparator (if provided) must implement `Serializable`.

For the mathematically inclined, the *relation* that defines the *imposed ordering* that a given comparator `c` imposes on a given set of objects `S` is:

```
{(x, y) such that c.compare(x, y) <= 0}.
```

The *quotient* for this total order is:

```
{(x, y) such that c.compare(x, y) == 0}.
```

It follows immediately from the contract for `compare` that the quotient is an *equivalence relation* on `S`, and that the imposed ordering is a *total order* on `S`. When we say that the ordering imposed by `c` on `S` is *consistent with equals*, we mean that the quotient for the ordering is the equivalence relation defined by the objects' `equals(Object)` method(s):

```
{(x, y) such that x.equals(y)}.
```

Unlike `Comparable`, a comparator may optionally permit comparison of null arguments, while maintaining the requirements for an equivalence relation.

This interface is a member of the Java Collections Framework.

Since:

`1.2`

See Also:

`Comparable`, `Serializable`

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
int	compare(T o1, T o2)

	Compares its two arguments for order.
static <T,U extends Comparable<? super U>> Comparator<T>	comparing(Function<? super T,? extends U> keyExtractor) Accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T,U> Comparator<T>	comparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator) Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified Comparator.
static <T> Comparator<T>	comparingDouble(ToDoubleFunction<? super T> keyExtractor) Accepts a function that extracts a double sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> Comparator<T>	comparingInt(ToIntFunction<? super T> keyExtractor) Accepts a function that extracts an int sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> Comparator<T>	comparingLong(ToLongFunction<? super T> keyExtractor) Accepts a function that extracts a long sort key from a type T, and returns a Comparator<T> that compares by that sort key.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this comparator.
static <T extends Comparable<? super T>> Comparator<T>	naturalOrder() Returns a comparator that compares Comparable objects in natural order.
static <T> Comparator<T>	nullsFirst(Comparator<? super T> comparator) Returns a null-friendly comparator that considers null to be less than non-null.
static <T> Comparator<T>	nullsLast(Comparator<? super T> comparator) Returns a null-friendly comparator that considers null to be greater than non-null.
default Comparator<T>	reversed() Returns a comparator that imposes the reverse ordering of this comparator.
static <T extends Comparable<? super T>> Comparator<T>	reverseOrder() Returns a comparator that imposes the reverse of the *natural ordering*.
default Comparator<T>	thenComparing(Comparator<? super T> other) Returns a lexicographic-order comparator with another comparator.
default <U extends Comparable<? super U>> Comparator<T>	thenComparing(Function<? super T,? extends U> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a Comparable sort key.
default <U> Comparator<T>	thenComparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator) Returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator.
default Comparator<T>	thenComparingDouble(ToDoubleFunction<? super T> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a double sort key.
default Comparator<T>	thenComparingInt(ToIntFunction<? super T> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a int sort key.
default Comparator<T>	thenComparingLong(ToLongFunction<? super T> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a long sort key.

Method Detail

compare

```
int compare(T o1,
            T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

The implementor must ensure that `sgn(compare(x, y)) == -sgn(compare(y, x))` for all `x` and `y`. (This implies that `compare(x, y)` must throw an exception if and only if `compare(y, x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `((compare(x, y)>0) && (compare(y, z)>0))` implies `compare(x, z)>0`.

Finally, the implementor must ensure that `compare(x, y)==0` implies that `sgn(compare(x, z))==sgn(compare(y, z))` for all `z`.

It is generally the case, but *not* strictly required that `(compare(x, y)==0) == (x.equals(y))`. Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

Parameters:

`o1 - the first object to be compared.`

`o2 - the second object to be compared.`

Returns:

`a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.`

Throws:

`NullPointerException - if an argument is null and this comparator does not permit null arguments`

`ClassCastException - if the arguments' types prevent them from being compared by this comparator.`

equals

`boolean equals(Object obj)`

Indicates whether some other object is "equal to" this comparator. This method must obey the general contract of `Object.equals(Object)`. Additionally, this method can return `true` *only* if the specified object is also a comparator and it imposes the same ordering as this comparator. Thus, `comp1.equals(comp2)` implies that `sgn(comp1.compare(o1, o2))==sgn(comp2.compare(o1, o2))` for every object reference `o1` and `o2`.

Note that it is *always* safe *not* to override `Object.equals(Object)`. However, overriding this method may, in some cases, improve performance by allowing programs to determine that two distinct comparators impose the same order.

Overrides:

`equals in class Object`

Parameters:

`obj - the reference object with which to compare.`

Returns:

`true only if the specified object is also a comparator and it imposes the same ordering as this comparator.`

See Also:

`Object.equals(Object), Object.hashCode()`

reversed

`default Comparator<T> reversed()`

Returns a comparator that imposes the reverse ordering of this comparator.

Returns:

`a comparator that imposes the reverse ordering of this comparator.`

Since:

`1.8`

thenComparing

`default Comparator<T> thenComparing(Comparator<? super T> other)`

Returns a lexicographic-order comparator with another comparator. If this `Comparator` considers two elements equal, i.e. `compare(a, b) == 0`, `other` is used to determine the order.

The returned comparator is serializable if the specified comparator is also serializable.

API Note:

`For example, to sort a collection of String based on the length and then case-insensitive natural ordering, the comparator can be composed using following code,`

```
Comparator<String> cmp = Comparator.comparingInt(String::length)
        .thenComparing(String.CASE_INSENSITIVE_ORDER);
```

Parameters:

other - the other comparator to be used when this comparator compares two objects that are equal.

Returns:

a lexicographic-order comparator composed of this and then the other comparator

Throws:

NullPointerException - if the argument is null.

Since:

1.8

thenComparing

```
default <U> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor,
                                        Comparator<? super U> keyComparator)
```

Returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator.

Implementation Requirements:

This default implementation behaves as if thenComparing(comparing(keyExtractor, cmp)).

Type Parameters:

U - the type of the sort key

Parameters:

keyExtractor - the function used to extract the sort key

keyComparator - the Comparator used to compare the sort key

Returns:

a lexicographic-order comparator composed of this comparator and then comparing on the key extracted by the keyExtractor function

Throws:

NullPointerException - if either argument is null.

Since:

1.8

See Also:

comparing(Function, Comparator), thenComparing(Comparator)

thenComparing

```
default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor)
```

Returns a lexicographic-order comparator with a function that extracts a Comparable sort key.

Implementation Requirements:

This default implementation behaves as if thenComparing(comparing(keyExtractor)).

Type Parameters:

U - the type of the Comparable sort key

Parameters:

keyExtractor - the function used to extract the Comparable sort key

Returns:

a lexicographic-order comparator composed of this and then the Comparable sort key.

Throws:

NullPointerException - if the argument is null.

Since:

1.8

See Also:

comparing(Function), thenComparing(Comparator)

thenComparingInt

```
default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)
```

Returns a lexicographic-order comparator with a function that extracts a int sort key.

Implementation Requirements:
This default implementation behaves as if thenComparing(comparingInt(keyExtractor)).

Parameters:
keyExtractor - the function used to extract the integer sort key

Returns:
a lexicographic-order comparator composed of this and then the int sort key

Throws:
NullPointerException - if the argument is null.

Since:
1.8

See Also:
comparingInt(ToIntFunction), thenComparing(Comparator)

thenComparingLong

```
default Comparator<T> thenComparingLong(ToLongFunction<? super T> keyExtractor)
```

Returns a lexicographic-order comparator with a function that extracts a long sort key.

Implementation Requirements:
This default implementation behaves as if thenComparing(comparingLong(keyExtractor)).

Parameters:
keyExtractor - the function used to extract the long sort key

Returns:
a lexicographic-order comparator composed of this and then the long sort key

Throws:
NullPointerException - if the argument is null.

Since:
1.8

See Also:
comparingLong(ToLongFunction), thenComparing(Comparator)

thenComparingDouble

```
default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor)
```

Returns a lexicographic-order comparator with a function that extracts a double sort key.

Implementation Requirements:
This default implementation behaves as if thenComparing(comparingDouble(keyExtractor)).

Parameters:
keyExtractor - the function used to extract the double sort key

Returns:
a lexicographic-order comparator composed of this and then the double sort key

Throws:
NullPointerException - if the argument is null.

Since:
1.8

See Also:
comparingDouble(ToDoubleFunction), thenComparing(Comparator)

reverseOrder

```
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

Returns a comparator that imposes the reverse of the *natural ordering*.

The returned comparator is serializable and throws NullPointerException when comparing null.

Type Parameters:
T - the Comparable type of element to be compared

Returns:
a comparator that imposes the reverse of the *natural ordering* on Comparable objects.

Since:

1.8

See Also:

Comparable

naturalOrder

```
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
```

Returns a comparator that compares Comparable objects in natural order.

The returned comparator is serializable and throws NullPointerException when comparing null.

Type Parameters:

T - the Comparable type of element to be compared

Returns:

a comparator that imposes the *natural ordering* on Comparable objects.

Since:

1.8

See Also:

Comparable

nullsFirst

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)
```

Returns a null-friendly comparator that considers null to be less than non-null. When both are null, they are considered equal. If both are non-null, the specified Comparator is used to determine the order. If the specified comparator is null, then the returned comparator considers all non-null values to be equal.

The returned comparator is serializable if the specified comparator is serializable.

Type Parameters:

T - the type of the elements to be compared

Parameters:

comparator - a Comparator for comparing non-null values

Returns:

a comparator that considers null to be less than non-null, and compares non-null objects with the supplied Comparator.

Since:

1.8

nullsLast

```
static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)
```

Returns a null-friendly comparator that considers null to be greater than non-null. When both are null, they are considered equal. If both are non-null, the specified Comparator is used to determine the order. If the specified comparator is null, then the returned comparator considers all non-null values to be equal.

The returned comparator is serializable if the specified comparator is serializable.

Type Parameters:

T - the type of the elements to be compared

Parameters:

comparator - a Comparator for comparing non-null values

Returns:

a comparator that considers null to be greater than non-null, and compares non-null objects with the supplied Comparator.

Since:

1.8

comparing

```
static <T,U> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor,
                                     Comparator<? super U> keyComparator)
```

Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified Comparator.

The returned comparator is serializable if the specified function and comparator are both serializable.

API Note:

For example, to obtain a Comparator that compares Person objects by their last name ignoring case differences,

```
Comparator<Person> cmp = Comparator.comparing(
        Person::getLastName,
        String.CASE_INSENSITIVE_ORDER);
```

Type Parameters:

T - the type of element to be compared

U - the type of the sort key

Parameters:

keyExtractor - the function used to extract the sort key

keyComparator - the Comparator used to compare the sort key

Returns:

a comparator that compares by an extracted key using the specified Comparator

Throws:

NullPointerException - if either argument is null

Since:

1.8

comparing

```
static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor)
```

Accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator<T> that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

API Note:

For example, to obtain a Comparator that compares Person objects by their last name,

```
Comparator<Person> byLastName = Comparator.comparing(Person::getLastName);
```

Type Parameters:

T - the type of element to be compared

U - the type of the Comparable sort key

Parameters:

keyExtractor - the function used to extract the Comparable sort key

Returns:

a comparator that compares by an extracted key

Throws:

NullPointerException - if the argument is null

Since:

1.8

comparingInt

```
static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)
```

Accepts a function that extracts an int sort key from a type T, and returns a Comparator<T> that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

Type Parameters:

T - the type of element to be compared

Parameters:

keyExtractor - the function used to extract the integer sort key

Returns:

a comparator that compares by an extracted key

Throws:

NullPointerException - if the argument is null

Since:

1.8

See Also:

comparing(Function)

comparingLong

static <T> Comparator<T> comparingLong(ToLongFunction<? super T> keyExtractor)

Accepts a function that extracts a long sort key from a type T, and returns a Comparator<T> that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

Type Parameters:

T - the type of element to be compared

Parameters:

keyExtractor - the function used to extract the long sort key

Returns:

a comparator that compares by an extracted key

Throws:

NullPointerException - if the argument is null

Since:

1.8

See Also:

comparing(Function)

comparingDouble

static <T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> keyExtractor)

Accepts a function that extracts a double sort key from a type T, and returns a Comparator<T> that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

Type Parameters:

T - the type of element to be compared

Parameters:

keyExtractor - the function used to extract the double sort key

Returns:

a comparator that compares by an extracted key

Throws:

NullPointerException - if the argument is null

Since:

1.8

See Also:

comparing(Function)

compact1, compact2, compact3

java.util

Interface Iterator<E>

Type Parameters:

`E - the type of elements returned by this iterator`

All Known Subinterfaces:

`ListIterator<E>, PrimitiveIterator<T,T_CONS>, PrimitiveIterator.OfDouble, PrimitiveIterator.OfInt, PrimitiveIterator.OfLong, XMLEventReader`

All Known Implementing Classes:

`BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner`

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

An iterator over a collection. `Iterator` takes the place of `Enumeration` in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since:

`1.2`

See Also:

`Collection, ListIterator, Iterable`

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`default void`	**`forEachRemaining`**`(Consumer<? super E> action)` Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
`boolean`	**`hasNext`**`()` Returns `true` if the iteration has more elements.
`E`	**`next`**`()` Returns the next element in the iteration.
`default void`	**`remove`**`()`

	Removes from the underlying collection the last element returned by this iterator (optional operation).

Method Detail

hasNext

```
boolean hasNext()
```

Returns `true` if the iteration has more elements. (In other words, returns `true` if `next()` would return an element rather than throwing an exception.)

Returns:

`true if the iteration has more elements`

next

```
E next()
```

Returns the next element in the iteration.

Returns:

`the next element in the iteration`

Throws:

`NoSuchElementException - if the iteration has no more elements`

remove

```
default void remove()
```

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to `next()`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Implementation Requirements:

`The default implementation throws an instance of UnsupportedOperationException and performs no other action.`

Throws:

`UnsupportedOperationException - if the remove operation is not supported by this iterator`

`IllegalStateException - if the next method has not yet been called, or the remove method has already been called after the last call to the next method`

forEachRemaining

```
default void forEachRemaining(Consumer<? super E> action)
```

Performs the given action for each remaining element until all elements have been processed or the action throws an exception. Actions are performed in the order of iteration, if that order is specified. Exceptions thrown by the action are relayed to the caller.

Implementation Requirements:

The default implementation behaves as if:

```
while (hasNext())
    action.accept(next());
```

Parameters:

`action` - The action to be performed for each element

Throws:

`NullPointerException` - if the specified action is null

Since:

1.8

compact1, compact2, compact3

java.util

Interface Deque<E>

Type Parameters:

`E - the type of elements held in this collection`

All Superinterfaces:

`Collection<E>, Iterable<E>, Queue<E>`

All Known Subinterfaces:

`BlockingDeque<E>`

All Known Implementing Classes:

`ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, LinkedList`

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

A linear collection that supports element insertion and removal at both ends. The name *deque* is short for "double ended queue" and is usually pronounced "deck". Most `Deque` implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element. 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 `Deque` implementations; in most implementations, insert operations cannot fail.

The twelve methods described above are summarized in the following table:

Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

This interface extends the `Queue` interface. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. Elements are added at the end of the deque and removed from the beginning. The methods inherited from the `Queue` interface are precisely equivalent to `Deque` methods as indicated in the following table:

Comparison of Queue and Deque methods

Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Deques can also be used as LIFO (Last-In-First-Out) stacks. This interface should be used in preference to the legacy Stack class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque. Stack methods are precisely equivalent to Deque methods as indicated in the table below:

Comparison of Stack and Deque methods

Stack Method	Equivalent Deque Method
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()

Note that the peek method works equally well when a deque is used as a queue or a stack; in either case, elements are drawn from the beginning of the deque.

This interface provides two methods to remove interior elements, removeFirstOccurrence and removeLastOccurrence.

Unlike the List interface, this interface does not provide support for indexed access to elements.

While Deque implementations are not strictly required to prohibit the insertion of null elements, they are strongly encouraged to do so. Users of any Deque implementations that do allow null elements are strongly encouraged *not* to take advantage of the ability to insert nulls. This is so because null is used as a special return value by various methods to indicated that the deque is empty.

Deque implementations generally do not define element-based versions of the equals and hashCode methods, but instead inherit the identity-based versions from class Object.

This interface is a member of the Java Collections Framework.

Since:

1.6

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to

	deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
void	**addFirst**(**E** e) Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions, throwing an `IllegalStateException` if no space is currently available.
void	**addLast**(**E** e) Inserts the specified element at the end of this deque if it is possible to do so immediately without violating capacity restrictions, throwing an `IllegalStateException` if no space is currently available.
boolean	**contains**(**Object** o) Returns `true` if this deque contains the specified element.
Iterator<**E**>	**descendingIterator**() Returns an iterator over the elements in this deque in reverse sequential order.
E	**element**() Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of 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.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this deque in proper sequence.
boolean	**offer**(**E** e) Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available.
boolean	**offerFirst**(**E** e) Inserts the specified element at the front of this deque unless it would violate capacity restrictions.
boolean	**offerLast**(**E** e) Inserts the specified element at the end of this deque unless it would violate capacity restrictions.
E	**peek**() Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of

	represented by this deque (in other words, the first element of 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 (in other words, at the head of this deque) if it is possible to do so immediately without violating capacity restrictions, throwing an `IllegalStateException` if no space is currently available.
`E`	**`remove()`** Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque).
`boolean`	**`remove(Object o)`** Removes the first occurrence 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 from this deque.
`E`	**`removeLast()`** Retrieves and removes the last element of this deque.
`boolean`	**`removeLastOccurrence(Object o)`** Removes the last occurrence of the specified element from this deque.
`int`	**`size()`** Returns the number of elements in this deque.

compact1, compact2, compact3

java.util

Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

`Collection<E>, Iterable<E>`

All Known Implementing Classes:

`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

```
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.

The `List` interface places additional stipulations, beyond those specified in the `Collection` interface, on the contracts of the `iterator`, `add`, `remove`, `equals`, and `hashCode` methods. Declarations for other inherited methods are also included here for convenience.

The `List` interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the `LinkedList` class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

The `List` interface provides a special iterator, called a `ListIterator`, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the `Iterator` interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The `List` interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

The `List` interface provides two methods to efficiently insert and remove multiple elements at an arbitrary point in the list.

Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a list.

Some list implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the list may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `Set`, `ArrayList`, `LinkedList`, `Vector`, `Arrays.asList(Object[])`, `Collections.nCopies(int, Object)`, `Collections.EMPTY_LIST`, `AbstractList`, `AbstractSequentialList`

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) 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 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.
ListIterator<**E**>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<**E**>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the 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 of the specified element from this list, if it is present (optional operation).
boolean	**removeAll**(**Collection**<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
default void	**replaceAll**(**UnaryOperator**<**E**> operator) Replaces each element of this list with the result of applying the operator to that element.
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.
default void	**sort**(**Comparator**<? super **E**> c) Sorts this list according to the order induced by the specified **Comparator**.
default **Spliterator**<**E**>	**spliterator**() Creates a **Spliterator** over the 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 (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Methods inherited from interface java.util.Collection

parallelStream, removeIf, stream

Methods inherited from interface java.lang.Iterable

forEach

Method Detail

size

```
int size()
```

Returns the number of elements in this list. If this list contains more than Integer.MAX_VALUE elements, returns Integer.MAX_VALUE.

Specified by:

size in interface Collection<E>

Returns:

the number of elements in this list

isEmpty

```
boolean isEmpty()
```

compact1, compact2, compact3

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()`. An iterator for a list of length `n` has `n+1` possible cursor positions, as illustrated by the carets (^) below:

```
                      Element(0)   Element(1)   Element(2)   ... Element(n-1)
 cursor positions:  ^            ^            ^            ^                  ^
```

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()`.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

Collection, List, Iterator, Enumeration, List.listIterator()

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	**add**(E e) 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.

traversing the list in the reverse direction.

E	**next**() Returns the next element in the list and advances the cursor position.
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 and moves the cursor position backwards.
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** e) Replaces the last element returned by **next()** or **previous()** with the specified element (optional operation).

Methods inherited from interface java.util.Iterator

forEachRemaining

Method Detail

hasNext

boolean hasNext()

Returns `true` if this list iterator has more elements when traversing the list in the forward direction. (In other words, returns `true` if `next()` would return an element rather than throwing an exception.)

Specified by:

`hasNext` in interface `Iterator<E>`

Returns:

`true if the list iterator has more elements when traversing the list in the forward direction`

next

E next()

Returns the next element in the list and advances the cursor position. This method may be called repeatedly to iterate through the list, or intermixed with calls to `previous()`

compact1, compact2, compact3

java.util

Interface Map<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Known Subinterfaces:

`Bindings`, `ConcurrentMap<K,V>`, `ConcurrentNavigableMap<K,V>`, `LogicalMessageContext`, `MessageContext`, `NavigableMap<K,V>`, `SOAPMessageContext`, `SortedMap<K,V>`

All Known Implementing Classes:

`AbstractMap`, `Attributes`, `AuthProvider`, `ConcurrentHashMap`, `ConcurrentSkipListMap`, `EnumMap`, `HashMap`, `Hashtable`, `IdentityHashMap`, `LinkedHashMap`, `PrinterStateReasons`, `Properties`, `Provider`, `RenderingHints`, `SimpleBindings`, `TabularDataSupport`, `TreeMap`, `UIDefaults`, `WeakHashMap`

```
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.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

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.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw `UnsupportedOperationException` if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the map. For example, invoking the `putAll(Map)` method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `containsKey(Object key)` method says: "returns `true` if and only if this map contains a mapping for a key `k` such that `(key==null ? k==null : key.equals(k))`." This specification should *not* be construed to imply that invoking `Map.containsKey` with a non-null argument `key` will cause `key.equals(k)` to be invoked for any key `k`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two keys. (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.

Some map operations which perform recursive traversal of the map may fail with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the `clone()`, `equals()`, `hashCode()` and `toString()` methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`HashMap`, `TreeMap`, `Hashtable`, `SortedMap`, `Collection`, `Set`

Nested Class Summary

Nested Classes

Modifier and Type	Interface and Description
static interface	**Map.Entry<K,V>** A map entry (key-value pair).

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
void	**clear**() Removes all of the mappings from this map (optional operation).
default V	**compute**(K key, **BiFunction**<? super K,? super V,? extends V> remappingFunction) Attempts to compute a mapping for the specified key and its current mapped value (or `null` if there is no current mapping).
default V	**computeIfAbsent**(K key, **Function**<? super K,? extends V> mappingFunction)

	If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function and enters it into this map unless `null`.
default **V**	**computeIfPresent**(**K** key, **BiFunction**<? super **K**,? super **V**,? extends **V**> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
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.
default void	**forEach**(**BiConsumer**<? super **K**,? super **V**> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
default **V**	**getOrDefault**(**Object** key, **V** defaultValue) Returns the value to which the specified key is mapped, or `defaultValue` if this map contains no mapping for the 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.
default **V**	**merge**(**K** key, **V** value, **BiFunction**<? super **V**,? super **V**,? extends **V**> remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
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**> m) Copies all of the mappings from the specified map to this map (optional operation).
default **V**	**putIfAbsent**(**K** key, **V** value) If the specified key is not already associated with a value (or is mapped to `null`) associates it with the given value and returns `null`, else returns the current value.
V	**remove**(**Object** key) Removes the mapping for a key from this map if it is present (optional operation).

default boolean	**remove**(**Object** key, **Object** value) Removes the entry for the specified key only if it is currently mapped to the specified value.
default **V**	**replace**(**K** key, **V** value) Replaces the entry for the specified key only if it is currently mapped to some value.
default boolean	**replace**(**K** key, **V** oldValue, **V** newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
default void	**replaceAll**(**BiFunction**<? super **K**,? super **V**,? extends **V**> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
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.

Method Detail

size

```
int size()
```

Returns the number of key-value mappings in this map. If the map contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Returns:

the number of key-value mappings in this map

isEmpty

```
boolean isEmpty()
```

Returns `true` if this map contains no key-value mappings.

Returns:

true if this map contains no key-value mappings

containsKey

```
boolean containsKey(Object key)
```

Returns `true` if this map contains a mapping for the specified key. More formally, returns `true` if and only if this map contains a mapping for a key `k` such that `(key==null ? k==null : key.equals(k))`. (There can be at most one such mapping.)

Parameters:

key - key whose presence in this map is to be tested

Returns:

compact1, compact2, compact3

java.util

Interface Map.Entry<K,V>

All Known Implementing Classes:

AbstractMap.SimpleEntry, AbstractMap.SimpleImmutableEntry

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.

Since:

1.2

See Also:

Map.entrySet()

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>>	comparingByKey() Returns a comparator that compares Map.Entry in natural order on key.
static <K,V> Comparator<Map.Entry<K,V>>	comparingByKey(Comparator<? super K> cmp) Returns a comparator that compares Map.Entry by key using the given Comparator.
static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>>	comparingByValue() Returns a comparator that compares Map.Entry in natural order on value.
static <K,V> Comparator<Map.Entry<K,V>>	comparingByValue(Comparator<? super V> cmp) Returns a comparator that compares Map.Entry by value using the given Comparator.
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).

Method Detail

getKey

```
K getKey()
```

Returns the key corresponding to this entry.

Returns:

the key corresponding to this entry

Throws:

IllegalStateException - implementations may, but are not required to, throw this exception if the entry has been removed from the backing map.

getValue

```
V getValue()
```

Returns the value corresponding to this entry. If the mapping has been removed from the backing map (by the iterator's `remove` operation), the results of this call are undefined.

Returns:

the value corresponding to this entry

Throws:

IllegalStateException - implementations may, but are not required to, throw this exception if the entry has been removed from the backing map.

setValue

```
V setValue(V value)
```

Replaces the value corresponding to this entry with the specified value (optional operation). (Writes through to the map.) The behavior of this call is undefined if the mapping has already been removed from the map (by the iterator's `remove` operation).

Parameters:

compact1, compact2, compact3

java.util

Interface Queue<E>

Type Parameters:

`E - the type of elements held in this collection`

All Superinterfaces:

`Collection`<E>, `Iterable`<E>

All Known Subinterfaces:

`BlockingDeque`<E>, `BlockingQueue`<E>, `Deque`<E>, `TransferQueue`<E>

All Known Implementing Classes:

`AbstractQueue`, `ArrayBlockingQueue`, `ArrayDeque`, `ConcurrentLinkedDeque`, `ConcurrentLinkedQueue`, `DelayQueue`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `LinkedList`, `LinkedTransferQueue`, `PriorityBlockingQueue`, `PriorityQueue`, `SynchronousQueue`

```
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. 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.

The `offer` method inserts an element if possible, otherwise returning `false`. This differs from the `Collection.add` method, which can fail to add an element only by throwing an unchecked exception. The `offer` method is designed for use when failure is a normal, rather than exceptional occurrence, for example, in fixed-capacity (or "bounded") queues.

The `remove()` and `poll()` methods remove and return the head of the queue. Exactly which element is removed from the queue is a function of the queue's ordering policy, which differs from implementation to implementation. The `remove()` and `poll()` methods differ only in their behavior when the queue is empty: the `remove()` method throws an exception, while the `poll()` method returns `null`.

The `element()` and `peek()` methods return, but do not remove, the head of the queue.

The `Queue` interface does not define the *blocking queue methods*, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the `BlockingQueue` interface, which extends this interface.

`Queue` implementations generally do not allow insertion of `null` elements, although some implementations, such as `LinkedList`, do not prohibit insertion of `null`. Even in the implementations that permit it, `null` should not be inserted into a `Queue`, as `null` is also used as a special return value by the `poll` method to indicate that the queue contains no elements.

`Queue` implementations generally do not define element-based versions of methods `equals` and `hashCode` but instead inherit the identity based versions from class `Object`, because element-based equality is not always well-defined for queues with the same elements but different ordering properties.

This interface is a member of the Java Collections Framework.

Since:

1.5

See Also:

`Collection`, `LinkedList`, `PriorityQueue`, `LinkedBlockingQueue`, `BlockingQueue`, `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
boolean	**add**(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
E	**element**() Retrieves, but does not remove, the head of this queue.
boolean	**offer**(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
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.

this queue is empty.

E	**remove**() Retrieves and removes the head of this queue.

Methods inherited from interface java.util.Collection

addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, parallelStream, remove, removeAll, removeIf, retainAll, size, spliterator, stream, toArray, toArray

Methods inherited from interface java.lang.Iterable

forEach

Method Detail

add

boolean add(E e)

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.

Specified by:

add in interface Collection<E>

Parameters:

e - the element to add

Returns:

true (as specified by Collection.add(E))

Throws:

IllegalStateException - if the element cannot be added at this time due to capacity restrictions

ClassCastException - if the class of the specified element prevents it from being added to this queue

NullPointerException - if the specified element is null and this queue does not permit null elements

IllegalArgumentException - if some property of this element prevents it from being added to this queue

offer

boolean offer(E e)

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted queue, this

method is generally preferable to add(E), which can fail to insert an element only by throwing an exception.

Parameters:

e - the element to add

Returns:

true if the element was added to this queue, else false

Throws:

ClassCastException - if the class of the specified element prevents it from being added to this queue

NullPointerException - if the specified element is null and this queue does not permit null elements

IllegalArgumentException - if some property of this element prevents it from being added to this queue

remove

E remove()

Retrieves and removes the head of this queue. This method differs from poll only in that it throws an exception if this queue is empty.

Returns:

the head of this queue

Throws:

NoSuchElementException - if this queue is empty

poll

E poll()

Retrieves and removes the head of this queue, or returns null if this queue is empty.

Returns:

the head of this queue, or null if this queue is empty

element

E element()

Retrieves, but does not remove, the head of this queue. This method differs from peek only in that it throws an exception if this queue is empty.

Returns:

the head of this queue

Throws:

NoSuchElementException - if this queue is empty

peek

E peek()

Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

Returns:

the head of this queue, or null if this queue is empty

compact1, compact2, compact3

java.util

Interface RandomAccess

All Known Implementing Classes:

ArrayList, AttributeList, CopyOnWriteArrayList, RoleList, RoleUnresolvedList, Stack, Vector

---

public interface **RandomAccess**

Marker interface used by `List` implementations to indicate that they support fast (generally constant time) random access. The primary purpose of this interface is to allow generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access lists.

The best algorithms for manipulating random access lists (such as `ArrayList`) can produce quadratic behavior when applied to sequential access lists (such as `LinkedList`). Generic list algorithms are encouraged to check whether the given list is an `instanceof` this interface before applying an algorithm that would provide poor performance if it were applied to a sequential access list, and to alter their behavior if necessary to guarantee acceptable performance.

It is recognized that the distinction between random and sequential access is often fuzzy. For example, some `List` implementations provide asymptotically linear access times if they get huge, but constant access times in practice. Such a `List` implementation should generally implement this interface. As a rule of thumb, a `List` implementation should implement this interface if, for typical instances of the class, this loop:

```
for (int i=0, n=list.size(); i < n; i++)
    list.get(i);
```

runs faster than this loop:

```
for (Iterator i=list.iterator(); i.hasNext(); )
    i.next();
```

This interface is a member of the Java Collections Framework.

Since:

1.4

Submit a bug or feature

For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

compact1, compact2, compact3

java.util

Interface Set<E>

Type Parameters:

`E - the type of elements maintained by this set`

All Superinterfaces:

`Collection<E>, Iterable<E>`

All Known Subinterfaces:

`NavigableSet<E>, SortedSet<E>`

All Known Implementing Classes:

`AbstractSet, ConcurrentHashMap.KeySetView, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet`

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

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.

The `Set` interface places additional stipulations, beyond those inherited from the `Collection` interface, on the contracts of all constructors and on the contracts of the `add`, `equals` and `hashCode` methods. Declarations for other inherited methods are also included here for convenience. (The specifications accompanying these declarations have been tailored to the `Set` interface, but they do not contain any additional stipulations.)

The additional stipulation on constructors is, not surprisingly, that all constructors must create a set that contains no duplicate elements (as defined above).

Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

Some set implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the set may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

Collection, List, SortedSet, HashSet, TreeSet, AbstractSet, Collections.singleton(java.lang.Object), Collections.EMPTY_SET

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(**Collection**<? extends **E**> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(**Object** o) Returns `true` if this set contains the specified element.
boolean	**containsAll**(**Collection**<?> c) Returns `true` if this set contains all of the elements of the specified collection.
boolean	**equals**(**Object** o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
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 (optional operation).
boolean	**removeAll**(**Collection**<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).

	the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
default **Spliterator**<E>	**spliterator**() Creates a Spliterator over the elements in this set.
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Methods inherited from interface java.util.Collection

parallelStream, removeIf, stream

Methods inherited from interface java.lang.Iterable

forEach

Method Detail

size

```
int size()
```

Returns the number of elements in this set (its cardinality). If this set contains more than Integer.MAX_VALUE elements, returns Integer.MAX_VALUE.

Specified by:

size in interface Collection<E>

Returns:

the number of elements in this set (its cardinality)

isEmpty

```
boolean isEmpty()
```

Returns true if this set contains no elements.

Specified by:

isEmpty in interface Collection<E>

Returns:

true if this set contains no elements

contains

compact1, compact2, compact3

java.util

Interface SortedMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Superinterfaces:

`Map<K,V>`

All Known Subinterfaces:

`ConcurrentNavigableMap<K,V>`, `NavigableMap<K,V>`

All Known Implementing Classes:

`ConcurrentSkipListMap`, `TreeMap`

```
public interface SortedMap<K,V>
extends Map<K,V>
```

A `Map` that further provides a *total ordering* on its keys. The map is ordered according to the natural ordering of its keys, or by a `Comparator` typically provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the `entrySet`, `keySet` and `values` methods). Several additional operations are provided to take advantage of the ordering. (This interface is the map analogue of `SortedSet`.)

All keys inserted into a sorted map must implement the `Comparable` interface (or be accepted by the specified comparator). Furthermore, all such keys must be *mutually comparable*: `k1.compareTo(k2)` (or `comparator.compare(k1, k2)`) must not throw a `ClassCastException` for any keys `k1` and `k2` in the sorted map. 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 map (whether or not an explicit comparator is provided) must be *consistent with equals* if the sorted map is to correctly implement the `Map` interface. (See the `Comparable` interface or `Comparator` interface for a precise definition of *consistent with equals*.) This is so because the `Map` interface is defined in terms of the `equals` operation, but a sorted map performs all key comparisons using its `compareTo` (or `compare`) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a tree map *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the `Map` interface.

All general-purpose sorted map implementation classes should provide four "standard" constructors. It is not possible to enforce this recommendation though as required constructors cannot be specified by interfaces. The expected "standard" constructors for all sorted map implementations are:

1. A void (no arguments) constructor, which creates an empty sorted map sorted according to the natural ordering of its keys.
2. A constructor with a single argument of type `Comparator`, which creates an empty sorted map sorted according to the specified comparator.
3. A constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument, sorted according to the keys' natural ordering.
4. A constructor with a single argument of type `SortedMap`, which creates a new sorted map with the same key-value mappings and the same ordering as the input sorted map.

Note: several methods return submaps with restricted key 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 key type allows for calculation of the successor of a given key, merely request the subrange from `lowEndpoint` to `successor(highEndpoint)`. For example, suppose that `m` is a map whose keys are strings. The following idiom obtains a view containing all of the key-value mappings in `m` whose keys are between `low` and `high`, inclusive:

```
SortedMap<String, V> sub = m.subMap(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 key-value mappings in `m` whose keys are between `low` and `high`, exclusive:

```
SortedMap<String, V> sub = m.subMap(low+"\0", high);
```

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Map`, `TreeMap`, `SortedSet`, `Comparator`, `Comparable`, `Collection`, `ClassCastException`

Nested Class Summary

Nested classes/interfaces inherited from interface java.util.Map

`Map.Entry<K,V>`

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
`Comparator<? super K>`	**`comparator()`** Returns the comparator used to order the keys in this map, or `null` if this map uses the **natural ordering** of its keys.
`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 map.
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 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.

Methods inherited from interface java.util.Map

clear, compute, computeIfAbsent, computeIfPresent, containsKey, containsValue, equals, forEach, get, getOrDefault, hashCode, isEmpty, merge, put, putAll, putIfAbsent, remove, remove, replace, replace, replaceAll, size

Method Detail

comparator

```
Comparator<? super K> comparator()
```

Returns the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys.

Returns:

the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys

subMap

```
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. (If fromKey and toKey are equal, the returned map is empty.) The returned map is backed by this map, so changes in the returned map are reflected in

compact1, compact2, compact3

java.util

Interface SortedSet<E>

Type Parameters:

`E - the type of elements maintained by this set`

All Superinterfaces:

`Collection<E>, Iterable<E>, Set<E>`

All Known Subinterfaces:

`NavigableSet<E>`

All Known Implementing Classes:

`ConcurrentSkipListSet, TreeSet`

```
public interface SortedSet<E>
extends Set<E>
```

A `Set` that further provides a *total ordering* on its elements. The elements are ordered using their natural ordering, or by a `Comparator` 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`.)

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)` (or `comparator.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 `Comparable` interface 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);
```

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Set`, `TreeSet`, `SortedMap`, `Collection`, `Comparable`, `Comparator`, `ClassCastException`

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`Comparator<? super E>`	**`comparator()`** Returns the comparator used to order the elements in this set, or `null` if this set uses the **natural ordering** of its elements.
`E`	**`first()`** Returns the first (lowest) element currently in this set.
`SortedSet<E>`	**`headSet(E toElement)`** Returns a view of the portion of this set whose elements are strictly less than `toElement`.
`E`	**`last()`** Returns the last (highest) element currently in this set.
`default Spliterator<E>`	**`spliterator()`** Creates a `Spliterator` over the elements in this sorted set.
`SortedSet<E>`	**`subSet(E fromElement, E toElement)`** Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.
`SortedSet<E>`	**`tailSet(E fromElement)`**

Returns a view of the portion of this set whose elements are greater than or equal to `fromElement`.

Methods inherited from interface java.util.Set

`add`, `addAll`, `clear`, `contains`, `containsAll`, `equals`, `hashCode`, `isEmpty`, `iterator`, `remove`, `removeAll`, `retainAll`, `size`, `toArray`, `toArray`

Methods inherited from interface java.util.Collection

`parallelStream`, `removeIf`, `stream`

Methods inherited from interface java.lang.Iterable

`forEach`

Method Detail

comparator

```
Comparator<? super E> comparator()
```

Returns the comparator used to order the elements in this set, or `null` if this set uses the natural ordering of its elements.

Returns:

the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements

subSet

```
SortedSet<E> subSet(E fromElement,
                    E toElement)
```

Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive. (If `fromElement` and `toElement` are equal, the returned set is empty.) The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa. The returned set supports all optional set operations that this set supports.

The returned set will throw an `IllegalArgumentException` on an attempt to insert an element outside its range.

Parameters:

fromElement - low endpoint (inclusive) of the returned set

toElement - high endpoint (exclusive) of the returned set

Returns:

a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive

compact1, compact2, compact3

java.util

Class AbstractCollection<E>

java.lang.Object
    java.util.AbstractCollection<E>

All Implemented Interfaces:

Iterable<E>, Collection<E>

Direct Known Subclasses:

AbstractList, AbstractQueue, AbstractSet, ArrayDeque, ConcurrentLinkedDeque

---

```
public abstract class AbstractCollection<E>
extends Object
implements Collection<E>
```

This class provides a skeletal implementation of the `Collection` interface, to minimize the effort required to implement this interface.

To implement an unmodifiable collection, the programmer needs only to extend this class and provide implementations for the `iterator` and `size` methods. (The iterator returned by the `iterator` method must implement `hasNext` and `next`.)

To implement a modifiable collection, the programmer must additionally override this class's `add` method (which otherwise throws an `UnsupportedOperationException`), and the iterator returned by the `iterator` method must additionally implement its `remove` method.

The programmer should generally provide a void (no argument) and `Collection` constructor, as per the recommendation in the `Collection` interface specification.

The documentation for each non-abstract method in this class describes its implementation in detail. Each of these methods may be overridden if the collection being implemented admits a more efficient implementation.

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

Collection

Constructor Summary

Constructors

Modifier	Constructor and Description

protected	**AbstractCollection**() Sole constructor.

Method Summary

All Methods | Instance Methods | Abstract Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) 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	**isEmpty**() Returns `true` if this collection contains no elements.
abstract **Iterator**<**E**>	**iterator**() Returns an iterator over the elements contained 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 of 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).
abstract 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.
`String`	**`toString`**`()` Returns a string representation of this collection.

Methods inherited from class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

Methods inherited from interface java.util.Collection

`equals`, `hashCode`, `parallelStream`, `removeIf`, `spliterator`, `stream`

Methods inherited from interface java.lang.Iterable

`forEach`

Constructor Detail

AbstractCollection

```
protected AbstractCollection()
```

Sole constructor. (For invocation by subclass constructors, typically implicit.)

Method Detail

iterator

```
public abstract Iterator<E> iterator()
```

Returns an iterator over the elements contained in this collection.

Specified by:

`iterator` in interface `Iterable<E>`

Specified by:

`iterator` in interface `Collection<E>`

Returns:

an iterator over the elements contained in this collection

size

```
public abstract int size()
```

compact1, compact2, compact3

java.util

Class AbstractList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>

All Implemented Interfaces:

`Iterable<E>, Collection<E>, List<E>`

Direct Known Subclasses:

`AbstractSequentialList, ArrayList, Vector`

```
public abstract class AbstractList<E>
extends AbstractCollection<E>
implements List<E>
```

This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array). For sequential access data (such as a linked list), `AbstractSequentialList` should be used in preference to this class.

To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the `get(int)` and `size()` methods.

To implement a modifiable list, the programmer must additionally override the `set(int, E)` method (which otherwise throws an `UnsupportedOperationException`). If the list is variable-size the programmer must additionally override the `add(int, E)` and `remove(int)` methods.

The programmer should generally provide a void (no argument) and collection constructor, as per the recommendation in the `Collection` interface specification.

Unlike the other abstract collection implementations, the programmer does *not* have to provide an iterator implementation; the iterator and list iterator are implemented by this class, on top of the "random access" methods: `get(int)`, `set(int, E)`, `add(int, E)` and `remove(int)`.

The documentation for each non-abstract method in this class describes its implementation in detail. Each of these methods may be overridden if the collection being implemented admits a more efficient implementation.

This class is a member of the Java Collections Framework.

Since:

1.2

Field Summary

Fields

Modifier and Type	Field and Description
protected int	**modCount** The number of times this list has been *structurally modified*.

Constructor Summary

Constructors

Modifier	Constructor and Description
protected	**AbstractList**() Sole constructor.

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Concrete Methods**

Modifier and Type	Method and Description
boolean	**add**(**E** e) 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**(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	**equals**(**Object** o) Compares the specified object with this list for equality.
abstract **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 of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this list in proper sequence

	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.
ListIterator<**E**>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<**E**>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
E	**set**(int index, **E** element) Replaces the element at the specified position in this list with the specified element (optional operation).
List<**E**>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

Methods inherited from class java.util.AbstractCollection

addAll, contains, containsAll, isEmpty, remove, removeAll, retainAll, size, toArray, toArray, toString

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.List

addAll, contains, containsAll, isEmpty, remove, removeAll, replaceAll, retainAll, size, sort, spliterator, toArray, toArray

Methods inherited from interface java.util.Collection

parallelStream, removeIf, stream

Methods inherited from interface java.lang.Iterable

forEach

compact1, compact2, compact3

java.util

Class AbstractMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Implemented Interfaces:

Map<K,V>

Direct Known Subclasses:

ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, IdentityHashMap, TreeMap, WeakHashMap

---

```
public abstract class AbstractMap<K,V>
extends Object
implements Map<K,V>
```

This class provides a skeletal implementation of the `Map` interface, to minimize the effort required to implement this interface.

To implement an unmodifiable map, the programmer needs only to extend this class and provide an implementation for the `entrySet` method, which returns a set-view of the map's mappings. Typically, the returned set will, in turn, be implemented atop `AbstractSet`. This set should not support the `add` or `remove` methods, and its iterator should not support the `remove` method.

To implement a modifiable map, the programmer must additionally override this class's `put` method (which otherwise throws an `UnsupportedOperationException`), and the iterator returned by `entrySet().iterator()` must additionally implement its `remove` method.

The programmer should generally provide a void (no argument) and map constructor, as per the recommendation in the `Map` interface specification.

The documentation for each non-abstract method in this class describes its implementation in detail. Each of these methods may be overridden if the map being implemented admits a more efficient implementation.

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

Map, Collection

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
static class	**AbstractMap.SimpleEntry<K,V>** An Entry maintaining a key and a value.
static class	**AbstractMap.SimpleImmutableEntry<K,V>** An Entry maintaining an immutable key and value.

Nested classes/interfaces inherited from interface java.util.Map

Map.Entry<K,V>

Constructor Summary

Constructors

Modifier	Constructor and Description
protected	**AbstractMap**() Sole constructor.

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Concrete Methods**

Modifier and Type	Method and Description
void	**clear**() Removes all of the mappings from this map (optional operation).
protected Object	**clone**() Returns a shallow copy of this AbstractMap 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.
abstract Set<Map.Entry<K,V>>	**entrySet**() Returns a **Set** view of the mappings contained in this map

	map.
boolean	**equals**(**Object** o) Compares the specified object with this map for equality.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the 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**> m) Copies all of the mappings from the specified map to this map (optional operation).
V	**remove**(**Object** key) Removes the mapping for a key from this map if it is present (optional operation).
int	**size**() Returns the number of key-value mappings in this map.
String	**toString**() Returns a string representation of this map.
Collection<**V**>	**values**() Returns a **Collection** view of the values contained in this map.

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Map

compute, computeIfAbsent, computeIfPresent, forEach, getOrDefault, merge, putIfAbsent, remove, replace, replace, replaceAll

compact1, compact2, compact3

java.util

Class AbstractMap.SimpleEntry<K,V>

java.lang.Object
    java.util.AbstractMap.SimpleEntry<K,V>

All Implemented Interfaces:

Serializable, Map.Entry<K,V>

Enclosing class:

AbstractMap<K,V>

```
public static class AbstractMap.SimpleEntry<K,V>
extends Object
implements Map.Entry<K,V>, Serializable
```

An Entry maintaining a key and a value. The value may be changed using the `setValue` method. This class facilitates the process of building custom map implementations. For example, it may be convenient to return arrays of `SimpleEntry` instances in method `Map.entrySet().toArray`.

Since:

1.6

See Also:

Serialized Form

Constructor Summary

Constructors

Constructor and Description
SimpleEntry(**K** key, **V** value) Creates an entry representing a mapping from the specified key to the specified value.
SimpleEntry(**Map.Entry**<? extends **K**,? extends **V**> entry) Creates an entry representing the same mapping as the specified entry.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
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.
String	**toString**() Returns a String representation of this map entry.

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Map.Entry

comparingByKey, comparingByKey, comparingByValue, comparingByValue

Constructor Detail

SimpleEntry

```
public SimpleEntry(K key,
                   V value)
```

Creates an entry representing a mapping from the specified key to the specified value.

Parameters:

key - the key represented by this entry

value - the value represented by this entry

SimpleEntry

```
public SimpleEntry(Map.Entry<? extends K,? extends V> entry)
```

Creates an entry representing the same mapping as the specified entry.

Parameters:

entry - the entry to copy

compact1, compact2, compact3

java.util

Class AbstractQueue<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractQueue<E>

Type Parameters:

`E - the type of elements held in this collection`

All Implemented Interfaces:

`Iterable<E>, Collection<E>, Queue<E>`

Direct Known Subclasses:

`ArrayBlockingQueue, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue`

```
public abstract class AbstractQueue<E>
extends AbstractCollection<E>
implements Queue<E>
```

This class provides skeletal implementations of some `Queue` operations. The implementations in this class are appropriate when the base implementation does *not* allow `null` elements. Methods `add`, `remove`, and `element` are based on `offer`, `poll`, and `peek`, respectively, but throw exceptions instead of indicating failure via `false` or `null` returns.

A `Queue` implementation that extends this class must minimally define a method `Queue.offer(E)` which does not permit insertion of `null` elements, along with methods `Queue.peek()`, `Queue.poll()`, `Collection.size()`, and `Collection.iterator()`. Typically, additional methods will be overridden as well. If these requirements cannot be met, consider instead subclassing `AbstractCollection`.

This class is a member of the Java Collections Framework.

Since:

`1.5`

Constructor Summary

Constructors

Modifier	Constructor and Description
protected	**AbstractQueue**() Constructor for use by subclasses.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
boolean	**addAll**(**Collection**<? extends **E**> c) Adds all of the elements in the specified collection to this queue.
void	**clear**() Removes all of the elements from this queue.
E	**element**() Retrieves, but does not remove, the head of this queue.
E	**remove**() Retrieves and removes the head of this queue.

Methods inherited from class java.util.AbstractCollection

contains, containsAll, isEmpty, iterator, remove, removeAll, retainAll, size, toArray, toArray, toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Queue

offer, peek, poll

Methods inherited from interface java.util.Collection

contains, containsAll, equals, hashCode, isEmpty, iterator, parallelStream, remove, removeAll, removeIf, retainAll, size, spliterator, stream, toArray, toArray

Methods inherited from interface java.lang.Iterable

forEach

Constructor Detail

AbstractQueue

compact1, compact2, compact3

java.util

Class AbstractSequentialList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.AbstractSequentialList<E>

All Implemented Interfaces:

Iterable<E>, Collection<E>, List<E>

Direct Known Subclasses:

LinkedList

```
public abstract class AbstractSequentialList<E>
extends AbstractList<E>
```

This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list). For random access data (such as an array), `AbstractList` should be used in preference to this class.

This class is the opposite of the `AbstractList` class in the sense that it implements the "random access" methods (`get(int index)`, `set(int index, E element)`, `add(int index, E element)` and `remove(int index)`) on top of the list's list iterator, instead of the other way around.

To implement a list the programmer needs only to extend this class and provide implementations for the `listIterator` and `size` methods. For an unmodifiable list, the programmer need only implement the list iterator's `hasNext`, `next`, `hasPrevious`, `previous` and `index` methods.

For a modifiable list the programmer should additionally implement the list iterator's `set` method. For a variable-size list the programmer should additionally implement the list iterator's `remove` and `add` methods.

The programmer should generally provide a void (no argument) and collection constructor, as per the recommendation in the `Collection` interface specification.

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

Collection, List, AbstractList, AbstractCollection

Field Summary

Fields inherited from class java.util.AbstractList

modCount

Constructor Summary

Constructors

Modifier	Constructor and Description
protected	**AbstractSequentialList**() Sole constructor.

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Concrete Methods**

Modifier and Type	Method and Description
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list (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).
E	**get**(int index) Returns the element at the specified position in this list.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this list (in proper sequence).
abstract **ListIterator**<**E**>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence).
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
E	**set**(int index, **E** element) Replaces the element at the specified position in this list with the specified element (optional operation).

Methods inherited from class java.util.AbstractList

add, clear, equals, hashCode, indexOf, lastIndexOf, listIterator, removeRange, subList

Methods inherited from class java.util.AbstractCollection

addAll, contains, containsAll, isEmpty, remove, removeAll, retainAll, size, toArray, toArray, toString

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.List

addAll, contains, containsAll, isEmpty, remove, removeAll, replaceAll, retainAll, size, sort, spliterator, toArray, toArray

Methods inherited from interface java.util.Collection

parallelStream, removeIf, stream

Methods inherited from interface java.lang.Iterable

forEach

Constructor Detail

AbstractSequentialList

```
protected AbstractSequentialList()
```

Sole constructor. (For invocation by subclass constructors, typically implicit.)

Method Detail

get

```
public E get(int index)
```

Returns the element at the specified position in this list.

This implementation first gets a list iterator pointing to the indexed element (with `listIterator(index)`). Then, it gets the element using `ListIterator.next` and returns it.

Specified by:

`get` in interface `List<E>`

Specified by:

`get` in class `AbstractList<E>`

Parameters:

compact1, compact2, compact3

java.util

Class AbstractSet<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractSet<E>

Type Parameters:

`E - the type of elements maintained by this set`

All Implemented Interfaces:

`Iterable<E>, Collection<E>, Set<E>`

Direct Known Subclasses:

`ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, TreeSet`

```
public abstract class AbstractSet<E>
extends AbstractCollection<E>
implements Set<E>
```

This class provides a skeletal implementation of the `Set` interface to minimize the effort required to implement this interface.

The process of implementing a set by extending this class is identical to that of implementing a Collection by extending AbstractCollection, except that all of the methods and constructors in subclasses of this class must obey the additional constraints imposed by the `Set` interface (for instance, the add method must not permit addition of multiple instances of an object to a set).

Note that this class does not override any of the implementations from the `AbstractCollection` class. It merely adds implementations for `equals` and `hashCode`.

This class is a member of the Java Collections Framework.

Since:

`1.2`

See Also:

`Collection, AbstractCollection, Set`

Constructor Summary

Constructors

Modifier	Constructor and Description
protected	**AbstractSet**() Sole constructor.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	**equals**(**Object** o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**removeAll**(**Collection**<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).

Methods inherited from class java.util.AbstractCollection

add, addAll, clear, contains, containsAll, isEmpty, iterator, remove, retainAll, size, toArray, toArray, toString

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Set

add, addAll, clear, contains, containsAll, isEmpty, iterator, remove, retainAll, size, spliterator, toArray, toArray

Methods inherited from interface java.util.Collection

parallelStream, removeIf, stream

Methods inherited from interface java.lang.Iterable

forEach

Constructor Detail

AbstractSet

```
protected AbstractSet()
```

Sole constructor. (For invocation by subclass constructors, typically implicit.)

Method Detail

compact1, compact2, compact3

java.util

Class ArrayDeque<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.ArrayDeque<E>

Type Parameters:

`E - the type of elements held in this collection`

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `Deque<E>`, `Queue<E>`

```
public class ArrayDeque<E>
extends AbstractCollection<E>
implements Deque<E>, Cloneable, Serializable
```

Resizable-array implementation of the `Deque` interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads. 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.

The iterators returned by this class's `iterator` method are *fail-fast*: If the deque is modified at any time after the iterator is created, in any way except through the iterator's own `remove` method, the iterator will generally throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class and its iterator implement all of the *optional* methods of the `Collection` and `Iterator` interfaces.

This class is a member of the Java Collections Framework.

Since:

`1.6`

See Also:

Serialized Form

Constructor Summary

Constructors

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.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and 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.
Iterator<**E**>	**descendingIterator**() Returns an iterator over the elements in this deque in reverse sequential order.
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.

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.
Spliterator<E>	**spliterator**() Creates a ***late-binding*** and *fail-fast* **Spliterator** over the 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.

Methods inherited from class java.util.AbstractCollection

addAll, containsAll, removeAll, retainAll, toString

Methods inherited from class java.lang.Object

equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Collection

addAll, containsAll, equals, hashCode, parallelStream, removeAll, removeIf, retainAll, stream

Methods inherited from interface java.lang.Iterable

forEach

Constructor Detail

ArrayDeque

compact1, compact2, compact3

java.util

Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

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.

Note that this implementation is not synchronized. If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `List`, `LinkedList`, `Vector`, `Serialized Form`

Field Summary

Fields inherited from class java.util.AbstractList

`modCount`

Constructor Summary

Constructors

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.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and 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.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this ArrayList instance.
boolean	**contains**(**Object** o) Returns true if this list contains the specified element.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
void	**forEach**(**Consumer**<? super **E**> action) Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
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.
ListIterator<**E**>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<**E**>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the 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.
boolean	**removeAll**(**Collection**<?> c) Removes from this list all of its elements that are contained in the specified collection.
boolean	**removeIf**(**Predicate**<? super **E**> filter) Removes all of the elements of this collection that satisfy the given predicate.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
void	**replaceAll**(**UnaryOperator**<**E**> operator) Replaces each element of this list with the result of applying the operator to that element.
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this list that are contained in the specified collection.
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.
void	**sort**(**Comparator**<? super **E**> c) Sorts this list according to the order induced by the specified **Comparator**.
Spliterator<**E**>	**spliterator**() Creates a ***late-binding*** and *fail-fast* **Spliterator** over the 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 (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
void	**trimToSize**()

Trims the capacity of this `ArrayList` instance to be the list's current size.

Methods inherited from class java.util.AbstractList

`equals`, `hashCode`

Methods inherited from class java.util.AbstractCollection

`containsAll`, `toString`

Methods inherited from class java.lang.Object

`finalize`, `getClass`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

Methods inherited from interface java.util.List

`containsAll`, `equals`, `hashCode`

Methods inherited from interface java.util.Collection

`parallelStream`, `stream`

Constructor Detail

ArrayList

```
public ArrayList(int initialCapacity)
```

Constructs an empty list with the specified initial capacity.

Parameters:

`initialCapacity` - the initial capacity of the list

Throws:

`IllegalArgumentException` - if the specified initial capacity is negative

ArrayList

```
public ArrayList()
```

Constructs an empty list with an initial capacity of ten.

ArrayList

```
public 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.

compact1, compact2, compact3

java.util

Class Arrays

java.lang.Object
    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.

The documentation for the methods contained in this class includes briefs description of the *implementations*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort(Object[])` does not have to be a MergeSort, but it does have to be *stable*.)

This class is a member of the Java Collections Framework.

Since:

1.2

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method and Description
`static <T> List<T>`	**`asList`**`(T... a)` Returns a fixed-size list backed by the specified array.
`static int`	**`binarySearch`**`(byte[] a, byte key)` Searches the specified array of bytes for the specified value using the binary search algorithm.
`static int`	**`binarySearch`**`(byte[] a, int fromIndex, int toIndex, byte key)` Searches a range of the specified array of bytes for the specified value using the binary search algorithm.

static int	**binarySearch**(char[] a, char key) Searches the specified array of chars for the specified value using the binary search algorithm.
static int	**binarySearch**(char[] a, int fromIndex, int toIndex, char key) Searches a range of the specified array of chars for the specified value using the binary search algorithm.
static int	**binarySearch**(double[] a, double key) Searches the specified array of doubles for the specified value using the binary search algorithm.
static int	**binarySearch**(double[] a, int fromIndex, int toIndex, double key) Searches a range of the specified array of doubles for the specified value using the binary search algorithm.
static int	**binarySearch**(float[] a, float key) Searches the specified array of floats for the specified value using the binary search algorithm.
static int	**binarySearch**(float[] a, int fromIndex, int toIndex, float key) Searches a range of the specified array of floats for the specified value using the binary search algorithm.
static int	**binarySearch**(int[] a, int key) Searches the specified array of ints for the specified value using the binary search algorithm.
static int	**binarySearch**(int[] a, int fromIndex, int toIndex, int key) Searches a range of the specified array of ints for the specified value using the binary search algorithm.
static int	**binarySearch**(long[] a, int fromIndex, int toIndex, long key) Searches a range of the specified array of longs for the specified value using the binary search algorithm.

static int	**binarySearch**(long[] a, long key) Searches the specified array of longs for the specified value using the binary search algorithm.
static int	**binarySearch**(**Object**[] a, int fromIndex, int toIndex, **Object** key) Searches a range of the specified array for the specified object using the binary search algorithm.
static int	**binarySearch**(**Object**[] a, **Object** key) Searches the specified array for the specified object using the binary search algorithm.
static int	**binarySearch**(short[] a, int fromIndex, int toIndex, short key) Searches a range of the specified array of shorts for the specified value using the binary search algorithm.
static int	**binarySearch**(short[] a, short key) Searches the specified array of shorts for the specified value using the binary search algorithm.
static <T> int	**binarySearch**(T[] a, int fromIndex, int toIndex, T key, **Comparator**<? super T> c) Searches a range of the specified array for the specified object using the binary search algorithm.
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[]	**copyOf**(boolean[] original, int newLength) Copies the specified array, truncating or padding with `false` (if necessary) so the copy has the specified length.
static byte[]	**copyOf**(byte[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.

static char[]	**copyOf**(char[] original, int newLength) Copies the specified array, truncating or padding with null characters (if necessary) so the copy has the specified length.
static double[]	**copyOf**(double[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static float[]	**copyOf**(float[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static int[]	**copyOf**(int[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static long[]	**copyOf**(long[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static short[]	**copyOf**(short[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static <T> T[]	**copyOf**(T[] original, int newLength) Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
static <T,U> T[]	**copyOf**(U[] original, int newLength, **Class**<? extends T[]> newType) Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
static boolean[]	**copyOfRange**(boolean[] original, int from, int to) Copies the specified range of the specified array into a new array.
static byte[]	

	copyOfRange(byte[] original, int from, int to) Copies the specified range of the specified array into a new array.
static char[]	**copyOfRange**(char[] original, int from, int to) Copies the specified range of the specified array into a new array.
static double[]	**copyOfRange**(double[] original, int from, int to) Copies the specified range of the specified array into a new array.
static float[]	**copyOfRange**(float[] original, int from, int to) Copies the specified range of the specified array into a new array.
static int[]	**copyOfRange**(int[] original, int from, int to) Copies the specified range of the specified array into a new array.
static long[]	**copyOfRange**(long[] original, int from, int to) Copies the specified range of the specified array into a new array.
static short[]	**copyOfRange**(short[] original, int from, int to) Copies the specified range of the specified array into a new array.
static <T> T[]	**copyOfRange**(T[] original, int from, int to) Copies the specified range of the specified array into a new array.
static <T,U> T[]	**copyOfRange**(U[] original, int from, int to, **Class**<? extends T[]> newType) Copies the specified range of the specified array into a new array.
static boolean	**deepEquals**(**Object**[] a1, **Object**[] a2) Returns true if the two specified arrays are *deeply equal* to one another.
static int	**deepHashCode**(**Object**[] a) Returns a hash code based on the "deep contents" of the specified array.

static **String**	**deepToString**(**Object**[] a) Returns a string representation of the "deep contents" of the specified array.
static boolean	**equals**(boolean[] a, boolean[] a2) Returns `true` if the two specified arrays of booleans are *equal* to one another.
static boolean	**equals**(byte[] a, byte[] a2) Returns `true` if the two specified arrays of bytes are *equal* to one another.
static boolean	**equals**(char[] a, char[] a2) Returns `true` if the two specified arrays of chars are *equal* to one another.
static boolean	**equals**(double[] a, double[] a2) Returns `true` if the two specified arrays of doubles are *equal* to one another.
static boolean	**equals**(float[] a, float[] a2) Returns `true` if the two specified arrays of floats are *equal* to one another.
static boolean	**equals**(int[] a, int[] a2) Returns `true` if the two specified arrays of ints are *equal* to one another.
static boolean	**equals**(long[] a, long[] a2) Returns `true` if the two specified arrays of longs are *equal* to one another.
static boolean	**equals**(**Object**[] a, **Object**[] a2) Returns `true` if the two specified arrays of Objects are *equal* to one another.
static boolean	**equals**(short[] a, short[] a2) Returns `true` if the two specified arrays of shorts are *equal* to one another.
static void	**fill**(boolean[] a, boolean val) Assigns the specified boolean value to each element of the specified array of booleans.
static void	**fill**(boolean[] a, int fromIndex, int toIndex, boolean val)

	Assigns the specified boolean value to each element of the specified range of the specified array of booleans.
static void	**fill**(byte[] a, byte val) Assigns the specified byte value to each element of the specified array of bytes.
static void	**fill**(byte[] a, int fromIndex, int toIndex, byte val) Assigns the specified byte value to each element of the specified range of the specified array of bytes.
static void	**fill**(char[] a, char val) Assigns the specified char value to each element of the specified array of chars.
static void	**fill**(char[] a, int fromIndex, int toIndex, char val) Assigns the specified char value to each element of the specified range of the specified array of chars.
static void	**fill**(double[] a, double val) Assigns the specified double value to each element of the specified array of doubles.
static void	**fill**(double[] a, int fromIndex, int toIndex, double val) Assigns the specified double value to each element of the specified range of the specified array of doubles.
static void	**fill**(float[] a, float val) Assigns the specified float value to each element of the specified array of floats.
static void	**fill**(float[] a, int fromIndex, int toIndex, float val) Assigns the specified float value to each element of the specified range of the specified array of floats.
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 void	**fill**(long[] a, int fromIndex, int toIndex, long val) Assigns the specified long value to each element of the specified range of the specified array of longs.
static void	**fill**(long[] a, long val) Assigns the specified long value to each element of the specified array of longs.
static void	**fill**(**Object**[] a, int fromIndex, int toIndex, **Object** val) Assigns the specified Object reference to each element of the specified range of the specified array of Objects.
static void	**fill**(**Object**[] a, **Object** val) Assigns the specified Object reference to each element of the specified array of Objects.
static void	**fill**(short[] a, int fromIndex, int toIndex, short val) Assigns the specified short value to each element of the specified range of the specified array of shorts.
static void	**fill**(short[] a, short val) Assigns the specified short value to each element of the specified array of shorts.
static int	**hashCode**(boolean[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(byte[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(char[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(double[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(float[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(int[] a)

	Returns a hash code based on the contents of the specified array.
static int	**hashCode**(long[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(**Object**[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(short[] a) Returns a hash code based on the contents of the specified array.
static void	**parallelPrefix**(double[] array, **DoubleBinaryOperator** op) Cumulates, in parallel, each element of the given array in place, using the supplied function.
static void	**parallelPrefix**(double[] array, int fromIndex, int toIndex, **DoubleBinaryOperator** op) Performs **parallelPrefix(double[], DoubleBinaryOperator)** for the given subrange of the array.
static void	**parallelPrefix**(int[] array, **IntBinaryOperator** op) Cumulates, in parallel, each element of the given array in place, using the supplied function.
static void	**parallelPrefix**(int[] array, int fromIndex, int toIndex, **IntBinaryOperator** op) Performs **parallelPrefix(int[], IntBinaryOperator)** for the given subrange of the array.
static void	**parallelPrefix**(long[] array, int fromIndex, int toIndex, **LongBinaryOperator** op) Performs **parallelPrefix(long[], LongBinaryOperator)** for the given subrange of the array.
static void	**parallelPrefix**(long[] array, **LongBinaryOperator** op) Cumulates, in parallel, each element of the given array in place, using the supplied function.
static <T> void	**parallelPrefix**(T[] array, **BinaryOperator**<T> op)

	Cumulates, in parallel, each element of the given array in place, using the supplied function.
static <T> void	**parallelPrefix**(T[] array, int fromIndex, int toIndex, **BinaryOperator**<T> op) Performs **parallelPrefix(Object[], BinaryOperator)** for the given subrange of the array.
static void	**parallelSetAll**(double[] array, **IntToDoubleFunction** generator) Set all elements of the specified array, in parallel, using the provided generator function to compute each element.
static void	**parallelSetAll**(int[] array, **IntUnaryOperator** generator) Set all elements of the specified array, in parallel, using the provided generator function to compute each element.
static void	**parallelSetAll**(long[] array, **IntToLongFunction** generator) Set all elements of the specified array, in parallel, using the provided generator function to compute each element.
static <T> void	**parallelSetAll**(T[] array, **IntFunction**<? extends T> generator) Set all elements of the specified array, in parallel, using the provided generator function to compute each element.
static void	**parallelSort**(byte[] a) Sorts the specified array into ascending numerical order.
static void	**parallelSort**(byte[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending numerical order.
static void	**parallelSort**(char[] a) Sorts the specified array into ascending numerical order.
static void	**parallelSort**(char[] a, int fromIndex, int toIndex)

	Sorts the specified range of the array into ascending numerical order.
static void	**parallelSort**(double[] a) Sorts the specified array into ascending numerical order.
static void	**parallelSort**(double[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending numerical order.
static void	**parallelSort**(float[] a) Sorts the specified array into ascending numerical order.
static void	**parallelSort**(float[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending numerical order.
static void	**parallelSort**(int[] a) Sorts the specified array into ascending numerical order.
static void	**parallelSort**(int[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending numerical order.
static void	**parallelSort**(long[] a) Sorts the specified array into ascending numerical order.
static void	**parallelSort**(long[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending numerical order.
static void	**parallelSort**(short[] a) Sorts the specified array into ascending numerical order.
static void	**parallelSort**(short[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending numerical order.
static <T extends **Comparable**<? super T>> void	**parallelSort**(T[] a) Sorts the specified array of objects into ascending order, according to the **natural ordering** of its elements.
static <T> void	**parallelSort**(T[] a, **Comparator**<? super T> cmp)

	Sorts the specified array of objects according to the order induced by the specified comparator.
static <T extends **Comparable**<? super T>> void	**parallelSort**(T[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of objects into ascending order, according to the **natural ordering** of its elements.
static <T> void	**parallelSort**(T[] a, int fromIndex, int toIndex, **Comparator**<? super T> cmp) Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.
static void	**setAll**(double[] array, **IntToDoubleFunction** generator) Set all elements of the specified array, using the provided generator function to compute each element.
static void	**setAll**(int[] array, **IntUnaryOperator** generator) Set all elements of the specified array, using the provided generator function to compute each element.
static void	**setAll**(long[] array, **IntToLongFunction** generator) Set all elements of the specified array, using the provided generator function to compute each element.
static <T> void	**setAll**(T[] array, **IntFunction**<? extends T> generator) Set all elements of the specified array, using the provided generator function to compute each element.
static void	**sort**(byte[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(byte[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(char[] a) Sorts the specified array into ascending numerical order.

static void	**sort**(char[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(double[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(double[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(float[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(float[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(int[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(int[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(long[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(long[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(**Object**[] a) Sorts the specified array of objects into ascending order, according to the **natural ordering** of its elements.
static void	**sort**(**Object**[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of objects into ascending order, according to the **natural ordering** of its elements.
static void	**sort**(short[] a) Sorts the specified array into ascending numerical order.

static void	**sort**(short[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static <T> void	**sort**(T[] a, **Comparator**<? super T> c) Sorts the specified array of objects according to the order induced by the specified comparator.
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 **Spliterator.OfDouble**	**spliterator**(double[] array) Returns a **Spliterator.OfDouble** covering all of the specified array.
static **Spliterator.OfDouble**	**spliterator**(double[] array, int startInclusive, int endExclusive) Returns a **Spliterator.OfDouble** covering the specified range of the specified array.
static **Spliterator.OfInt**	**spliterator**(int[] array) Returns a **Spliterator.OfInt** covering all of the specified array.
static **Spliterator.OfInt**	**spliterator**(int[] array, int startInclusive, int endExclusive) Returns a **Spliterator.OfInt** covering the specified range of the specified array.
static **Spliterator.OfLong**	**spliterator**(long[] array) Returns a **Spliterator.OfLong** covering all of the specified array.
static **Spliterator.OfLong**	**spliterator**(long[] array, int startInclusive, int endExclusive) Returns a **Spliterator.OfLong** covering the specified range of the specified array.
static <T> **Spliterator**<T>	**spliterator**(T[] array) Returns a **Spliterator** covering all of the specified array.

static <T> **Spliterator**<T>	**spliterator**(T[] array, int startInclusive, int endExclusive) Returns a **Spliterator** covering the specified range of the specified array.
static **DoubleStream**	**stream**(double[] array) Returns a sequential **DoubleStream** with the specified array as its source.
static **DoubleStream**	**stream**(double[] array, int startInclusive, int endExclusive) Returns a sequential **DoubleStream** with the specified range of the specified array as its source.
static **IntStream**	**stream**(int[] array) Returns a sequential **IntStream** with the specified array as its source.
static **IntStream**	**stream**(int[] array, int startInclusive, int endExclusive) Returns a sequential **IntStream** with the specified range of the specified array as its source.
static **LongStream**	**stream**(long[] array) Returns a sequential **LongStream** with the specified array as its source.
static **LongStream**	**stream**(long[] array, int startInclusive, int endExclusive) Returns a sequential **LongStream** with the specified range of the specified array as its source.
static <T> **Stream**<T>	**stream**(T[] array) Returns a sequential **Stream** with the specified array as its source.
static <T> **Stream**<T>	**stream**(T[] array, int startInclusive, int endExclusive) Returns a sequential **Stream** with the specified range of the specified array as its source.
static **String**	**toString**(boolean[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(byte[] a)

	Returns a string representation of the contents of the specified array.
static String	**toString**(char[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(double[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(float[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(int[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(long[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(Object[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(short[] a) Returns a string representation of the contents of the specified array.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

sort

```
public static void sort(int[] a)
```

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

a - the array to be sorted

compact1, compact2, compact3

java.util

Class BitSet

java.lang.Object
    java.util.BitSet

All Implemented Interfaces:

Serializable, Cloneable

```
public class BitSet
extends Object
implements Cloneable, Serializable
```

This class implements a vector of bits that grows as needed. Each component of the bit set has a `boolean` value. The bits of a `BitSet` are indexed by nonnegative integers. Individual indexed bits can be examined, set, or cleared. One `BitSet` may be used to modify the contents of another `BitSet` through logical AND, logical inclusive OR, and logical exclusive OR operations.

By default, all bits in the set initially have the value `false`.

Every bit set has a current size, which is the number of bits of space currently in use by the bit set. Note that the size is related to the implementation of a bit set, so it may change with implementation. The length of a bit set relates to logical length of a bit set and is defined independently of implementation.

Unless otherwise noted, passing a null parameter to any of the methods in a `BitSet` will result in a `NullPointerException`.

A `BitSet` is not safe for multithreaded use without external synchronization.

Since:

`JDK1.0`

See Also:

Serialized Form

Constructor Summary

Constructors

Constructor and Description
BitSet() Creates a new bit set.
BitSet(int nbits)

Creates a bit set whose initial size is large enough to explicitly represent bits with indices in the range `0` through `nbits-1`.

Method Summary

All Methods	Static Methods	Instance Methods	Concrete Methods

Modifier and Type	Method and Description
void	**and**(**BitSet** set) Performs a logical **AND** of this target bit set with the argument bit set.
void	**andNot**(**BitSet** set) Clears all of the bits in this `BitSet` whose corresponding bit is set in the specified `BitSet`.
int	**cardinality**() Returns the number of bits set to `true` in this `BitSet`.
void	**clear**() Sets all of the bits in this BitSet to `false`.
void	**clear**(int bitIndex) Sets the bit specified by the index to `false`.
void	**clear**(int fromIndex, int toIndex) Sets the bits from the specified `fromIndex` (inclusive) to the specified `toIndex` (exclusive) to `false`.
Object	**clone**() Cloning this `BitSet` produces a new `BitSet` that is equal to it.
boolean	**equals**(**Object** obj) Compares this object against the specified object.
void	**flip**(int bitIndex) Sets the bit at the specified index to the complement of its current value.
void	**flip**(int fromIndex, int toIndex) Sets each bit from the specified `fromIndex` (inclusive) to the specified `toIndex` (exclusive) to the complement of its current value.
boolean	**get**(int bitIndex) Returns the value of the bit with the specified index.
BitSet	**get**(int fromIndex, int toIndex) Returns a new `BitSet` composed of bits from this `BitSet` from `fromIndex` (inclusive) to `toIndex` (exclusive).
int	**hashCode**() Returns the hash code value for this bit set.
boolean	**intersects**(**BitSet** set)

	Returns true if the specified BitSet has any bits set to true that are also set to true in this BitSet.
boolean	**isEmpty**() Returns true if this BitSet contains no bits that are set to true.
int	**length**() Returns the "logical size" of this BitSet: the index of the highest set bit in the BitSet plus one.
int	**nextClearBit**(int fromIndex) Returns the index of the first bit that is set to false that occurs on or after the specified starting index.
int	**nextSetBit**(int fromIndex) Returns the index of the first bit that is set to true that occurs on or after the specified starting index.
void	**or**(**BitSet** set) Performs a logical **OR** of this bit set with the bit set argument.
int	**previousClearBit**(int fromIndex) Returns the index of the nearest bit that is set to false that occurs on or before the specified starting index.
int	**previousSetBit**(int fromIndex) Returns the index of the nearest bit that is set to true that occurs on or before the specified starting index.
void	**set**(int bitIndex) Sets the bit at the specified index to true.
void	**set**(int bitIndex, boolean value) Sets the bit at the specified index to the specified value.
void	**set**(int fromIndex, int toIndex) Sets the bits from the specified fromIndex (inclusive) to the specified toIndex (exclusive) to true.
void	**set**(int fromIndex, int toIndex, boolean value) Sets the bits from the specified fromIndex (inclusive) to the specified toIndex (exclusive) to the specified value.
int	**size**() Returns the number of bits of space actually in use by this BitSet to represent bit values.
IntStream	**stream**() Returns a stream of indices for which this BitSet contains a bit in the set state.
byte[]	**toByteArray**() Returns a new byte array containing all the bits in this bit set.
long[]	**toLongArray**() Returns a new long array containing all the bits in this bit set.

String	**toString**() Returns a string representation of this bit set.
static BitSet	**valueOf**(byte[] bytes) Returns a new bit set containing all the bits in the given byte array.
static BitSet	**valueOf**(ByteBuffer bb) Returns a new bit set containing all the bits in the given byte buffer between its position and limit.
static BitSet	**valueOf**(long[] longs) Returns a new bit set containing all the bits in the given long array.
static BitSet	**valueOf**(LongBuffer lb) Returns a new bit set containing all the bits in the given long buffer between its position and limit.
void	**xor**(BitSet set) Performs a logical **XOR** of this bit set with the bit set argument.

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Constructor Detail

BitSet

```
public BitSet()
```

Creates a new bit set. All bits are initially `false`.

BitSet

```
public BitSet(int nbits)
```

Creates a bit set whose initial size is large enough to explicitly represent bits with indices in the range `0` through `nbits-1`. All bits are initially `false`.

Parameters:

`nbits - the initial size of the bit set`

Throws:

`NegativeArraySizeException - if the specified initial size is negative`

Method Detail

valueOf

compact1, compact2, compact3

java.util

Class Collections

java.lang.Object
    java.util.Collections

```
public class Collections
extends Object
```

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The methods of this class all throw a `NullPointerException` if the collections or class objects provided to them are null.

The documentation for the polymorphic algorithms contained in this class generally includes a brief description of the *implementation*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort` does not have to be a mergesort, but it does have to be *stable*.)

The "destructive" algorithms contained in this class, that is, the algorithms that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if the collection does not support the appropriate mutation primitive(s), such as the `set` method. These algorithms may, but are not required to, throw this exception if an invocation would have no effect on the collection. For example, invoking the `sort` method on an unmodifiable list that is already sorted may or may not throw `UnsupportedOperationException`.

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `Set`, `List`, `Map`

Field Summary

Fields

Modifier and Type	Field and Description
static List	EMPTY_LIST The empty list (immutable).
static Map	EMPTY_MAP The empty map (immutable).
static Set	EMPTY_SET The empty set (immutable).

Method Summary

All Methods **Static Methods** **Concrete Methods**

Modifier and Type	Method and Description

static <T> boolean	**addAll**(**Collection**<? super T> c, T... elements) Adds all of the specified elements to the specified collection.
static <T> **Queue**<T>	**asLifoQueue**(**Deque**<T> deque) Returns a view of a **Deque** as a Last-in-first-out (Lifo) **Queue**.
static <T> int	**binarySearch**(**List**<? extends **Comparable**<? super T>> list, T key) Searches the specified list for the specified object using the binary search algorithm.
static <T> int	**binarySearch**(**List**<? extends T> list, T key, **Comparator**<? super T> c) Searches the specified list for the specified object using the binary search algorithm.
static <E> **Collection**<E>	**checkedCollection**(**Collection**<E> c, **Class**<E> type) Returns a dynamically typesafe view of the specified collection.
static <E> **List**<E>	**checkedList**(**List**<E> list, **Class**<E> type) Returns a dynamically typesafe view of the specified list.
static <K,V> **Map**<K,V>	**checkedMap**(**Map**<K,V> m, **Class**<K> keyType, **Class**<V> valueType) Returns a dynamically typesafe view of the specified map.
static <K,V> **NavigableMap**<K,V>	**checkedNavigableMap**(**NavigableMap**<K,V> m, **Class**<K> keyType, **Class**<V> valueType) Returns a dynamically typesafe view of the specified navigable map.
static <E> **NavigableSet**<E>	**checkedNavigableSet**(**NavigableSet**<E> s, **Class**<E> type) Returns a dynamically typesafe view of the specified navigable set.
static <E> **Queue**<E>	**checkedQueue**(**Queue**<E> queue, **Class**<E> type) Returns a dynamically typesafe view of the specified queue.
static <E> **Set**<E>	**checkedSet**(**Set**<E> s, **Class**<E> type) Returns a dynamically typesafe view of the specified set.
static <K,V> **SortedMap**<K,V>	**checkedSortedMap**(**SortedMap**<K,V> m, **Class**<K> keyType, **Class**<V> valueType) Returns a dynamically typesafe view of the specified sorted map.
static <E> **SortedSet**<E>	**checkedSortedSet**(**SortedSet**<E> s, **Class**<E> type) Returns a dynamically typesafe view of the specified sorted set.
static <T> void	**copy**(**List**<? super T> dest, **List**<? extends T> src) Copies all of the elements from one list into another.

static boolean	**disjoint**(**Collection**<?> c1, **Collection**<?> c2) Returns true if the two specified collections have no elements in common.
static <T> **Enumeration**<T>	**emptyEnumeration**() Returns an enumeration that has no elements.
static <T> **Iterator**<T>	**emptyIterator**() Returns an iterator that has no elements.
static <T> **List**<T>	**emptyList**() Returns an empty list (immutable).
static <T> **ListIterator**<T>	**emptyListIterator**() Returns a list iterator that has no elements.
static <K,V> **Map**<K,V>	**emptyMap**() Returns an empty map (immutable).
static <K,V> **NavigableMap**<K,V>	**emptyNavigableMap**() Returns an empty navigable map (immutable).
static <E> **NavigableSet**<E>	**emptyNavigableSet**() Returns an empty navigable set (immutable).
static <T> **Set**<T>	**emptySet**() Returns an empty set (immutable).
static <K,V> **SortedMap**<K,V>	**emptySortedMap**() Returns an empty sorted map (immutable).
static <E> **SortedSet**<E>	**emptySortedSet**() Returns an empty sorted set (immutable).
static <T> **Enumeration**<T>	**enumeration**(**Collection**<T> c) Returns an enumeration over the specified collection.
static <T> void	**fill**(**List**<? super T> list, T obj) Replaces all of the elements of the specified list with the specified element.
static int	**frequency**(**Collection**<?> c, **Object** o) Returns the number of elements in the specified collection equal to the specified object.
static int	**indexOfSubList**(**List**<?> source, **List**<?> target) Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static int	**lastIndexOfSubList**(**List**<?> source, **List**<?> target) Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static <T> **ArrayList**<T>	**list**(**Enumeration**<T> e) Returns an array list containing the elements returned by the specified enumeration in the order they are returned by the enumeration.
static <T extends **Object** & **Comparable**<? super T>> T	**max**(**Collection**<? extends T> coll) Returns the maximum element of the given collection, according to the *natural ordering* of its elements.

static <T> T	**max**(**Collection**<? extends T> coll, **Comparator**<? super T> comp) Returns the maximum element of the given collection, according to the order induced by the specified comparator.
static <T extends **Object** & **Comparable**<? super T>> T	**min**(**Collection**<? extends T> coll) Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**min**(**Collection**<? extends T> coll, **Comparator**<? super T> comp) Returns the minimum element of the given collection, according to the order induced by the specified comparator.
static <T> **List**<T>	**nCopies**(int n, T o) Returns an immutable list consisting of n copies of the specified object.
static <E> **Set**<E>	**newSetFromMap**(**Map**<E,**Boolean**> map) Returns a set backed by the specified map.
static <T> boolean	**replaceAll**(**List**<T> list, T oldVal, T newVal) Replaces all occurrences of one specified value in a list with another.
static void	**reverse**(**List**<?> list) Reverses the order of the elements in the specified list.
static <T> **Comparator**<T>	**reverseOrder**() Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface.
static <T> **Comparator**<T>	**reverseOrder**(**Comparator**<T> cmp) Returns a comparator that imposes the reverse ordering of the specified comparator.
static void	**rotate**(**List**<?> list, int distance) Rotates the elements in the specified list by the specified distance.
static void	**shuffle**(**List**<?> list) Randomly permutes the specified list using a default source of randomness.
static void	**shuffle**(**List**<?> list, **Random** rnd) Randomly permute the specified list using the specified source of randomness.
static <T> **Set**<T>	**singleton**(T o) Returns an immutable set containing only the specified object.
static <T> **List**<T>	**singletonList**(T o) Returns an immutable list containing only the specified object.
static <K,V> **Map**<K,V>	**singletonMap**(K key, V value) Returns an immutable map, mapping only the specified key to the specified value.
static <T extends **Comparable**<? super T>>	**sort**(**List**<T> list)

void	Sorts the specified list into ascending order, according to the **natural ordering** of its elements.
static <T> void	**sort**(List<T> list, Comparator<? super T> c) Sorts the specified list according to the order induced by the specified comparator.
static void	**swap**(List<?> list, int i, int j) Swaps the elements at the specified positions in the specified list.
static <T> Collection<T>	**synchronizedCollection**(Collection<T> c) Returns a synchronized (thread-safe) collection backed by the specified collection.
static <T> List<T>	**synchronizedList**(List<T> list) Returns a synchronized (thread-safe) list backed by the specified list.
static <K,V> Map<K,V>	**synchronizedMap**(Map<K,V> m) Returns a synchronized (thread-safe) map backed by the specified map.
static <K,V> NavigableMap<K,V>	**synchronizedNavigableMap**(NavigableMap<K,V> m) Returns a synchronized (thread-safe) navigable map backed by the specified navigable map.
static <T> NavigableSet<T>	**synchronizedNavigableSet**(NavigableSet<T> s) Returns a synchronized (thread-safe) navigable set backed by the specified navigable set.
static <T> Set<T>	**synchronizedSet**(Set<T> s) Returns a synchronized (thread-safe) set backed by the specified set.
static <K,V> SortedMap<K,V>	**synchronizedSortedMap**(SortedMap<K,V> m) Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
static <T> SortedSet<T>	**synchronizedSortedSet**(SortedSet<T> s) Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.
static <T> Collection<T>	**unmodifiableCollection**(Collection<? extends T> c) Returns an unmodifiable view of the specified collection.
static <T> List<T>	**unmodifiableList**(List<? extends T> list) Returns an unmodifiable view of the specified list.
static <K,V> Map<K,V>	**unmodifiableMap**(Map<? extends K,? extends V> m) Returns an unmodifiable view of the specified map.
static <K,V> NavigableMap<K,V>	**unmodifiableNavigableMap**(NavigableMap<K,? extends V> m) Returns an unmodifiable view of the specified navigable map.
static <T> NavigableSet<T>	**unmodifiableNavigableSet**(NavigableSet<T> s) Returns an unmodifiable view of the specified navigable set.
static <T> Set<T>	**unmodifiableSet**(Set<? extends T> s) Returns an unmodifiable view of the specified set.

static <K,V> SortedMap<K,V>	**unmodifiableSortedMap**(SortedMap<K,? extends V> m) Returns an unmodifiable view of the specified sorted map.
static <T> SortedSet<T>	**unmodifiableSortedSet**(SortedSet<T> s) Returns an unmodifiable view of the specified sorted set.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

EMPTY_SET

```
public static final Set EMPTY_SET
```

The empty set (immutable). This set is serializable.

See Also:

emptySet()

EMPTY_LIST

```
public static final List EMPTY_LIST
```

The empty list (immutable). This list is serializable.

See Also:

emptyList()

EMPTY_MAP

```
public static final Map EMPTY_MAP
```

The empty map (immutable). This map is serializable.

Since:

1.3

See Also:

emptyMap()

Method Detail

sort

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the Comparable interface. Furthermore, all elements in the list must be *mutually comparable* (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the list).

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

compact1, compact2, compact3

java.util

Class Dictionary<K,V>

java.lang.Object
    java.util.Dictionary<K,V>

Direct Known Subclasses:

Hashtable

```
public abstract class Dictionary<K,V>
extends Object
```

The `Dictionary` class is the abstract parent of any class, such as `Hashtable`, which maps keys to values. Every key and every value is an object. In any one `Dictionary` object, every key is associated with at most one value. Given a `Dictionary` and a key, the associated element can be looked up. Any non-`null` object can be used as a key and as a value.

As a rule, the `equals` method should be used by implementations of this class to decide if two keys are the same.

NOTE: This class is obsolete. New implementations should implement the Map interface, rather than extending this class.

Since:

`JDK1.0`

See Also:

`Map`, `Object.equals(java.lang.Object)`, `Object.hashCode()`, `Hashtable`

Constructor Summary

Constructors

Constructor and Description
Dictionary() Sole constructor.

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
abstract Enumeration<V>	**elements**()

	Returns an enumeration of the values in this dictionary.
abstract V	**get**(Object key) Returns the value to which the key is mapped in this dictionary.
abstract boolean	**isEmpty**() Tests if this dictionary maps no keys to value.
abstract Enumeration<K>	**keys**() Returns an enumeration of the keys in this dictionary.
abstract V	**put**(K key, V value) Maps the specified key to the specified value in this dictionary.
abstract V	**remove**(Object key) Removes the key (and its corresponding value) from this dictionary.
abstract int	**size**() Returns the number of entries (distinct keys) in this dictionary.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Dictionary

```
public Dictionary()
```

Sole constructor. (For invocation by subclass constructors, typically implicit.)

Method Detail

size

```
public abstract int size()
```

Returns the number of entries (distinct keys) in this dictionary.

Returns:

the number of keys in this dictionary.

isEmpty

```
public abstract boolean isEmpty()
```

Tests if this dictionary maps no keys to value. The general contract for the `isEmpty` method is that the result is true if and only if this dictionary contains no entries.

Returns:

`true if this dictionary maps no keys to values; false otherwise.`

keys

```
public abstract Enumeration<K> keys()
```

Returns an enumeration of the keys in this dictionary. The general contract for the keys method is that an `Enumeration` object is returned that will generate all the keys for which this dictionary contains entries.

Returns:

`an enumeration of the keys in this dictionary.`

See Also:

`elements()`, `Enumeration`

elements

```
public abstract Enumeration<V> elements()
```

Returns an enumeration of the values in this dictionary. The general contract for the `elements` method is that an `Enumeration` is returned that will generate all the elements contained in entries in this dictionary.

Returns:

`an enumeration of the values in this dictionary.`

See Also:

`keys()`, `Enumeration`

get

```
public abstract V get(Object key)
```

Returns the value to which the key is mapped in this dictionary. The general contract for the `isEmpty` method is that if this dictionary contains an entry for the specified key, the associated value is returned; otherwise, `null` is returned.

Parameters:

`key - a key in this dictionary. null if the key is not mapped to any value in this dictionary.`

Returns:

`the value to which the key is mapped in this dictionary;`

Throws:

`NullPointerException - if the key is null.`

See Also:

put(java.lang.Object, java.lang.Object)

put

```
public abstract V put(K key,
                      V value)
```

Maps the specified key to the specified value in this dictionary. Neither the key nor the value can be null.

If this dictionary already contains an entry for the specified key, the value already in this dictionary for that key is returned, after modifying the entry to contain the new element.

If this dictionary does not already have an entry for the specified key, an entry is created for the specified key and value, and null is returned.

The value can be retrieved by calling the get method with a key that is equal to the original key.

Parameters:

key - the hashtable key.

value - the value.

Returns:

the previous value to which the key was mapped in this dictionary, or null if the key did not have a previous mapping.

Throws:

NullPointerException - if the key or value is null.

See Also:

Object.equals(java.lang.Object), get(java.lang.Object)

remove

```
public abstract V remove(Object key)
```

Removes the key (and its corresponding value) from this dictionary. This method does nothing if the key is not in this dictionary.

Parameters:

key - the key that needs to be removed.

Returns:

the value to which the key had been mapped in this dictionary, or null if the key did not have a mapping.

Throws:

NullPointerException - if key is null.

Submit a bug or feature

For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

compact1, compact2, compact3

java.util

Class HashMap<K,V>

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

Type Parameters:

`K - the type of keys maintained by this map`

`V - the type of mapped values`

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 hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

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. Note that using many keys with the same `hashCode()` is a sure way to slow

down performance of any hash table. To ameliorate impact, when keys are `Comparable`, this class may use comparison order among keys to help break ties.

Note that this implementation is not synchronized. If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the `Collections.synchronizedMap` method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` method, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Object.hashCode()`, `Collection`, `Map`, `TreeMap`, `Hashtable`, `Serialized Form`

Nested Class Summary

Nested classes/interfaces inherited from class java.util.AbstractMap

`AbstractMap.SimpleEntry<K,V>`, `AbstractMap.SimpleImmutableEntry<K,V>`

Nested classes/interfaces inherited from interface java.util.Map

`Map.Entry<K,V>`

Constructor Summary

Constructors

Constructor and Description
`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

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
void	**clear**() Removes all of the mappings from this map.
Object	**clone**() Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
V	**compute**(**K** key, **BiFunction**<? super **K**,? super **V**,? extends **V**> remappingFunction) Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V	**computeIfAbsent**(**K** key, **Function**<? super **K**,? extends **V**> mappingFunction) If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
V	**computeIfPresent**(**K** key, **BiFunction**<? super **K**,? super **V**,? extends **V**> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
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.
void	**forEach**(**BiConsumer**<? super **K**,? super **V**> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
V	**getOrDefault**(**Object** key, **V** defaultValue) Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the 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	**merge**(**K** key, **V** value, **BiFunction**<? super **V**,? super **V**,? extends **V**> remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
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.
V	**putIfAbsent**(**K** key, **V** value) If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
V	**remove**(**Object** key) Removes the mapping for the specified key from this map if present.
boolean	**remove**(**Object** key, **Object** value) Removes the entry for the specified key only if it is currently mapped to the specified value.
V	**replace**(**K** key, **V** value) Replaces the entry for the specified key only if it is currently mapped to some value.
boolean	**replace**(**K** key, **V** oldValue, **V** newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
void	**replaceAll**(**BiFunction**<? super **K**,? super **V**,? extends **V**> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
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.

Methods inherited from class java.util.AbstractMap

equals, hashCode, toString

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Map

equals, hashCode

Constructor Detail

compact1, compact2, compact3

java.util

Class HashSet<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractSet<E>
            java.util.HashSet<E>

Type Parameters:

`E - the type of elements maintained by this set`

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `Set<E>`

Direct Known Subclasses:

`JobStateReasons`, `LinkedHashSet`

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

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.

Note that this implementation is not synchronized. If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the `Collections.synchronizedSet` method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

The iterators returned by this class's `iterator` method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own `remove` method, the Iterator throws a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `Set`, `TreeSet`, `HashMap`, `Serialized Form`

Constructor Summary

Constructors

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.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`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).
Spliterator<**E**>	**spliterator**() Creates a ***late-binding*** and *fail-fast* **Spliterator** over the elements in this set.

Methods inherited from class java.util.AbstractSet

equals, hashCode, removeAll

Methods inherited from class java.util.AbstractCollection

addAll, containsAll, retainAll, toArray, toArray, toString

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Set

addAll, containsAll, equals, hashCode, removeAll, retainAll, toArray, toArray

Methods inherited from interface java.util.Collection

parallelStream, removeIf, stream

Methods inherited from interface java.lang.Iterable

forEach

Constructor Detail

HashSet

```
public HashSet()
```

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

HashSet

```
public HashSet(Collection<? extends E> c)
```

Constructs a new set containing the elements in the specified collection. The `HashMap` is created with default load factor (0.75) and an initial capacity sufficient to contain the elements in the specified collection.

Parameters:

`c - the collection whose elements are to be placed into this set`

Throws:

`NullPointerException - if the specified collection is null`

HashSet

```
public HashSet(int initialCapacity,
               float loadFactor)
```

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

Parameters:

`initialCapacity - the initial capacity of the hash map`

`loadFactor - the load factor of the hash map`

Throws:

`IllegalArgumentException - if the initial capacity is less than zero, or if the load factor is nonpositive`

HashSet

```
public HashSet(int initialCapacity)
```

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

Parameters:

`initialCapacity - the initial capacity of the hash table`

Throws:

`IllegalArgumentException - if the initial capacity is less than zero`

Method Detail

iterator

```
public Iterator<E> iterator()
```

compact1, compact2, compact3

java.util

Class LinkedList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.AbstractSequentialList<E>
                java.util.LinkedList<E>

Type Parameters:

`E - the type of elements held in this collection`

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `Deque<E>`, `List<E>`, `Queue<E>`

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

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.

Note that this implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new LinkedList(...));
```

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this

exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

List, ArrayList, Serialized Form

Field Summary

Fields inherited from class java.util.AbstractList

modCount

Constructor Summary

Constructors

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.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and 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.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
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.
Object	**clone**() Returns a shallow copy of this `LinkedList`.
boolean	**contains**(**Object** o) Returns `true` if this list contains the specified element.
Iterator<**E**>	**descendingIterator**() Returns an iterator over the elements in this deque in reverse sequential order.
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.
ListIterator<**E**>	**listIterator**(int index) Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
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.

	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.
Spliterator<**E**>	**spliterator**() Creates a ***late-binding*** and *fail-fast* **Spliterator** over the elements in this list.

elements in this list.

`Object[]`	**`toArray`**`()` Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`<T> T[]`	**`toArray`**`(T[] a)` Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Methods inherited from class java.util.AbstractSequentialList

`iterator`

Methods inherited from class java.util.AbstractList

`equals`, `hashCode`, `listIterator`, `removeRange`, `subList`

Methods inherited from class java.util.AbstractCollection

`containsAll`, `isEmpty`, `removeAll`, `retainAll`, `toString`

Methods inherited from class java.lang.Object

`finalize`, `getClass`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

Methods inherited from interface java.util.List

`containsAll`, `equals`, `hashCode`, `isEmpty`, `iterator`, `listIterator`, `removeAll`, `replaceAll`, `retainAll`, `sort`, `subList`

Methods inherited from interface java.util.Deque

`iterator`

Methods inherited from interface java.util.Collection

`parallelStream`, `removeIf`, `stream`

Methods inherited from interface java.lang.Iterable

`forEach`

Constructor Detail

LinkedList

```
public LinkedList()
```

compact1, compact2, compact3

java.util

Class PriorityQueue<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
      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. 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`).

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations `poll`, `remove`, `peek`, and `element` access the element at the head of the queue.

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())`.

Note that this implementation is not synchronized. Multiple threads should not access a `PriorityQueue` instance concurrently if any of the threads modifies the queue. Instead, use the thread-safe `PriorityBlockingQueue` class.

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`).

This class is a member of the Java Collections Framework.

Since:

1.5

See Also:

Serialized Form

Constructor Summary

Constructors

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.
PriorityQueue(int initialCapacity) Creates a PriorityQueue with the specified initial capacity that orders its elements according to their **natural ordering**.
PriorityQueue(int initialCapacity, **Comparator**<? super **E**> comparator) Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
PriorityQueue(**PriorityQueue**<? extends **E**> c) Creates a PriorityQueue containing the elements in the specified priority queue.
PriorityQueue(**SortedSet**<? extends **E**> c) Creates a PriorityQueue containing the elements in the specified sorted set.

Method Summary

All Methods **Instance Methods** **Concrete Methods**

Modifier and Type	Method and Description
boolean	**add**(**E** e) Inserts the specified element into this priority queue.
void	**clear**() Removes all of the elements from this priority queue.
Comparator<? super **E**>	**comparator**() Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the **natural ordering** of its elements.
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.
`Spliterator<E>`	**`spliterator`**`()` Creates a ***late-binding*** and *fail-fast* **`Spliterator`** over the elements in this queue.
`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.

Methods inherited from class java.util.AbstractQueue

`addAll`, `element`, `remove`

Methods inherited from class java.util.AbstractCollection

`containsAll`, `isEmpty`, `removeAll`, `retainAll`, `toString`

Methods inherited from class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

Methods inherited from interface java.util.Collection

`containsAll`, `equals`, `hashCode`, `isEmpty`, `parallelStream`, `removeAll`, `removeIf`, `retainAll`, `stream`

Methods inherited from interface java.lang.Iterable

compact1, compact2, compact3

java.util

Class Stack<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>
        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.

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. For example:

```
   Deque<Integer> stack = new ArrayDeque<Integer>();
```

Since:

JDK1.0

See Also:

Serialized Form

Field Summary

Fields inherited from class java.util.Vector

capacityIncrement, elementCount, elementData

Fields inherited from class java.util.AbstractList

modCount

Constructor Summary

Constructors

Constructor and Description
Stack() Creates an empty Stack.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
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.

Methods inherited from class java.util.Vector

add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, forEach, get, hashCode, indexOf, indexOf, insertElementAt, isEmpty, iterator, lastElement, lastIndexOf, lastIndexOf, listIterator, listIterator, remove, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeIf, removeRange, replaceAll, retainAll, set, setElementAt, setSize, size, sort, spliterator, subList, toArray, toArray, toString, trimToSize

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Collection

parallelStream, stream

compact1, compact2, compact3

java.util

Class TreeMap<K,V>

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

Type Parameters:

`K - the type of keys maintained by this map`

`V - the type of mapped values`

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Map<K,V>`, `NavigableMap<K,V>`, `SortedMap<K,V>`

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

A Red-Black tree based `NavigableMap` implementation. The map is sorted according to the natural ordering of its keys, or by a `Comparator` provided at map 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*.

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

Note that this implementation is not synchronized. If multiple threads access a map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with an existing key is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the `Collections.synchronizedSortedMap` method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```

The iterators returned by the `iterator` method of the collections returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` method, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

All `Map.Entry` pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced. They do **not** support the `Entry.setValue` method. (Note however that it is possible to change mappings in the associated map using `put`.)

This class is a member of the Java Collections Framework.

Since:

`1.2`

See Also:

`Map`, `HashMap`, `Hashtable`, `Comparable`, `Comparator`, `Collection`, `Serialized Form`

Nested Class Summary

Nested classes/interfaces inherited from class java.util.AbstractMap

`AbstractMap.SimpleEntry<K,V>`, `AbstractMap.SimpleImmutableEntry<K,V>`

Constructor Summary

Constructors

Constructor and Description
`TreeMap()` Constructs a new, empty tree map, using the natural ordering of its keys.
`TreeMap(Comparator<? super K> comparator)` Constructs a new, empty tree map, ordered according to the given comparator.
`TreeMap(Map<? extends K,? extends V> m)` Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys.
`TreeMap(SortedMap<K,? extends V> m)` Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
Map.Entry<K,V>	ceilingEntry(K key) Returns a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.
K	ceilingKey(K key) Returns the least key greater than or equal to the given key, or null if there is no such key.
void	clear() Removes all of the mappings from this map.
Object	clone() Returns a shallow copy of this TreeMap instance.
Comparator<? super K>	comparator() Returns the comparator used to order the keys in this map, or null if this map uses the **natural ordering** of its keys.
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.
NavigableSet<K>	descendingKeySet() Returns a reverse order NavigableSet view of the keys contained in this map.
NavigableMap<K,V>	descendingMap() Returns a reverse order view of the mappings contained in this map.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
Map.Entry<K,V>	firstEntry() Returns a key-value mapping associated with the least key in this map, or null if the map is empty.
K	firstKey() Returns the first (lowest) key currently in this map.
Map.Entry<K,V>	floorEntry(K key) Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.
K	floorKey(K key)

	Returns the greatest key less than or equal to the given key, or `null` if there is no such key.
`void`	**`forEach`**`(`**`BiConsumer`**`<? super K,? super V> action)` Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
`V`	**`get`**`(`**`Object`** `key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`SortedMap``<K,V>`	**`headMap`**`(K toKey)` Returns a view of the portion of this map whose keys are strictly less than `toKey`.
`NavigableMap``<K,V>`	**`headMap`**`(K toKey, boolean inclusive)` Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`.
`Map.Entry``<K,V>`	**`higherEntry`**`(K key)` Returns a key-value mapping associated with the least key strictly greater than the given key, or `null` if there is no such key.
`K`	**`higherKey`**`(K key)` Returns the least key strictly greater than the given key, or `null` if there is no such key.
`Set``<K>`	**`keySet`**`()` Returns a **`Set`** view of the keys contained in this map.
`Map.Entry``<K,V>`	**`lastEntry`**`()` Returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
`K`	**`lastKey`**`()` Returns the last (highest) key currently in this map.
`Map.Entry``<K,V>`	**`lowerEntry`**`(K key)` Returns a key-value mapping associated with the greatest key strictly less than the given key, or `null` if there is no such key.
`K`	**`lowerKey`**`(K key)` Returns the greatest key strictly less than the given key, or `null` if there is no such key.
`NavigableSet``<K>`	**`navigableKeySet`**`()` Returns a **`NavigableSet`** view of the keys contained in this map.
`Map.Entry``<K,V>`	**`pollFirstEntry`**`()` Removes and returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
`Map.Entry``<K,V>`	**`pollLastEntry`**`()` Removes and returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.

	Removes and returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
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.
V	**replace**(**K** key, **V** value) Replaces the entry for the specified key only if it is currently mapped to some value.
boolean	**replace**(**K** key, **V** oldValue, **V** newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
void	**replaceAll**(**BiFunction**<? super **K**,? super **V**,? extends **V**> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
int	**size**() Returns the number of key-value mappings in this map.
NavigableMap<**K**,**V**>	**subMap**(**K** fromKey, boolean fromInclusive, **K** toKey, boolean toInclusive) Returns a view of the portion of this map whose keys range from `fromKey` to `toKey`.
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`.
NavigableMap<**K**,**V**>	**tailMap**(**K** fromKey, boolean inclusive) Returns a view of the portion of this map whose keys are greater than (or equal to, if `inclusive` is true) `fromKey`.
Collection<**V**>	**values**() Returns a **Collection** view of the values contained in this map.

Methods inherited from class java.util.AbstractMap

equals, hashCode, isEmpty, toString

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Map

compute, computeIfAbsent, computeIfPresent, equals, getOrDefault, hashCode, isEmpty, merge, putIfAbsent, remove

Constructor Detail

TreeMap

```
public TreeMap()
```

Constructs a new, empty tree map, using the natural ordering of its keys. All keys inserted into the map must implement the Comparable interface. Furthermore, all such keys must be *mutually comparable*: k1.compareTo(k2) must not throw a ClassCastException for any keys k1 and k2 in the map. If the user attempts to put a key into the map that violates this constraint (for example, the user attempts to put a string key into a map whose keys are integers), the put(Object key, Object value) call will throw a ClassCastException.

TreeMap

```
public TreeMap(Comparator<? super K> comparator)
```

Constructs a new, empty tree map, ordered according to the given comparator. All keys inserted into the map must be *mutually comparable* by the given comparator: comparator.compare(k1, k2) must not throw a ClassCastException for any keys k1 and k2 in the map. If the user attempts to put a key into the map that violates this constraint, the put(Object key, Object value) call will throw a ClassCastException.

Parameters:

comparator - the comparator that will be used to order this map. If null, the natural ordering of the keys will be used.

TreeMap

```
public TreeMap(Map<? extends K,? extends V> m)
```

Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys. All keys inserted into the new map must implement the Comparable interface. Furthermore, all such keys must be *mutually comparable*: k1.compareTo(k2) must not throw a ClassCastException for any keys k1 and k2 in the map. This method runs in n*log(n) time.

Parameters:

m - the map whose mappings are to be placed in this map

Throws:

`ClassCastException` - if the keys in m are not `Comparable`, or are not mutually comparable

`NullPointerException` - if the specified map is null

TreeMap

```
public TreeMap(SortedMap<K,? extends V> m)
```

Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map. This method runs in linear time.

Parameters:

`m` - the sorted map whose mappings are to be placed in this map, and whose comparator is to be used to sort this map

Throws:

`NullPointerException` - if the specified map is null

Method Detail

size

```
public int size()
```

Returns the number of key-value mappings in this map.

Specified by:

`size` in interface `Map<K,V>`

Overrides:

`size` in class `AbstractMap<K,V>`

Returns:

the number of key-value mappings in this map

containsKey

```
public boolean containsKey(Object key)
```

Returns `true` if this map contains a mapping for the specified key.

Specified by:

`containsKey` in interface `Map<K,V>`

Overrides:

`containsKey` in class `AbstractMap<K,V>`

Parameters:

`key` - key whose presence in this map is to be tested

Returns:

true if this map contains a mapping for the specified key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

containsValue

public boolean containsValue(Object value)

Returns true if this map maps one or more keys to the specified value. More formally, returns true if and only if this map contains at least one mapping to a value v such that (value==null ? v==null : value.equals(v)). This operation will probably require time linear in the map size for most implementations.

Specified by:

containsValue in interface Map<K,V>

Overrides:

containsValue in class AbstractMap<K,V>

Parameters:

value - value whose presence in this map is to be tested

Returns:

true if a mapping to value exists; false otherwise

Since:

1.2

get

public V get(Object key)

Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

More formally, if this map contains a mapping from a key k to a value v such that key compares equal to k according to the map's ordering, then this method returns v; otherwise it returns null. (There can be at most one such mapping.)

A return value of null does not *necessarily* indicate that the map contains no mapping for the key; it's also possible that the map explicitly maps the key to null. The containsKey operation may be used to distinguish these two cases.

Specified by:

get in interface Map<K,V>

Overrides:

get in class AbstractMap<K,V>

Parameters:

key - the key whose associated value is to be returned

Returns:

the value to which the specified key is mapped, or null if this map contains no mapping for the key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

comparator

```
public Comparator<? super K> comparator()
```

Description copied from interface: SortedMap

Returns the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys.

Specified by:

comparator in interface SortedMap<K,V>

Returns:

the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys

firstKey

```
public K firstKey()
```

Description copied from interface: SortedMap

Returns the first (lowest) key currently in this map.

Specified by:

firstKey in interface SortedMap<K,V>

Returns:

the first (lowest) key currently in this map

Throws:

NoSuchElementException - if this map is empty

lastKey

```
public K lastKey()
```

Description copied from interface: SortedMap

Returns the last (highest) key currently in this map.

Specified by:

lastKey in interface SortedMap<K,V>

Returns:

the last (highest) key currently in this map

Throws:

NoSuchElementException - if this map is empty

putAll

```
public void putAll(Map<? extends K,? extends V> map)
```

Copies all of the mappings from the specified map to this map. These mappings replace any mappings that this map had for any of the keys currently in the specified map.

Specified by:

putAll in interface Map<K,V>

Overrides:

putAll in class AbstractMap<K,V>

Parameters:

map - mappings to be stored in this map

Throws:

ClassCastException - if the class of a key or value in the specified map prevents it from being stored in this map

NullPointerException - if the specified map is null or the specified map contains a null key and this map does not permit null keys

put

```
public V put(K key,
             V value)
```

Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.

Specified by:

put in interface Map<K,V>

Overrides:

put in class AbstractMap<K,V>

Parameters:

key - key with which the specified value is to be associated

value - value to be associated with the specified key

Returns:

the previous value associated with key, or null if there was no mapping for key. (A null return can also indicate that the map previously associated null with key.)

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

remove

```
public V remove(Object key)
```

Removes the mapping for this key from this TreeMap if present.

Specified by:

remove in interface Map<K,V>

Overrides:

remove in class AbstractMap<K,V>

Parameters:

key - key for which mapping should be removed

Returns:

the previous value associated with key, or null if there was no mapping for key. (A null return can also indicate that the map previously associated null with key.)

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

clear

```
public void clear()
```

Removes all of the mappings from this map. The map will be empty after this call returns.

Specified by:

clear in interface Map<K,V>

Overrides:

clear in class AbstractMap<K,V>

clone

```
public Object clone()
```

Returns a shallow copy of this TreeMap instance. (The keys and values themselves are not cloned.)

Overrides:

clone in class AbstractMap<K,V>

Returns:

a shallow copy of this map

See Also:

Cloneable

firstEntry

```
public Map.Entry<K,V> firstEntry()
```

Description copied from interface: NavigableMap
Returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.

Specified by:
`firstEntry` in interface `NavigableMap<K,V>`

Returns:
an entry with the least key, or null if this map is empty

Since:
1.6

lastEntry

```
public Map.Entry<K,V> lastEntry()
```

Description copied from interface: NavigableMap
Returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.

Specified by:
`lastEntry` in interface `NavigableMap<K,V>`

Returns:
an entry with the greatest key, or null if this map is empty

Since:
1.6

pollFirstEntry

```
public Map.Entry<K,V> pollFirstEntry()
```

Description copied from interface: NavigableMap
Removes and returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.

Specified by:
`pollFirstEntry` in interface `NavigableMap<K,V>`

Returns:
the removed first entry of this map, or null if this map is empty

Since:
1.6

pollLastEntry

public Map.Entry<K,V> pollLastEntry()

Description copied from interface: NavigableMap

Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.

Specified by:

pollLastEntry in interface NavigableMap<K,V>

Returns:

the removed last entry of this map, or null if this map is empty

Since:

1.6

lowerEntry

public Map.Entry<K,V> lowerEntry(K key)

Description copied from interface: NavigableMap

Returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.

Specified by:

lowerEntry in interface NavigableMap<K,V>

Parameters:

key - the key

Returns:

an entry with the greatest key less than key, or null if there is no such key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

Since:

1.6

lowerKey

public K lowerKey(K key)

Description copied from interface: NavigableMap

Returns the greatest key strictly less than the given key, or null if there is no such key.

Specified by:

lowerKey in interface NavigableMap<K,V>

Parameters:

key - the key

Returns:

the greatest key less than key, or null if there is no such key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

Since:

1.6

floorEntry

```
public Map.Entry<K,V> floorEntry(K key)
```

Description copied from interface: NavigableMap

Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.

Specified by:

floorEntry in interface NavigableMap<K,V>

Parameters:

key - the key

Returns:

an entry with the greatest key less than or equal to key, or null if there is no such key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

Since:

1.6

floorKey

```
public K floorKey(K key)
```

Description copied from interface: NavigableMap

Returns the greatest key less than or equal to the given key, or null if there is no such key.

Specified by:

floorKey in interface NavigableMap<K,V>

Parameters:

key - the key

Returns:

the greatest key less than or equal to key, or null if there is no such key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

Since:

1.6

ceilingEntry

```
public Map.Entry<K,V> ceilingEntry(K key)
```

Description copied from interface: NavigableMap

Returns a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.

Specified by:

ceilingEntry in interface NavigableMap<K,V>

Parameters:

key - the key

Returns:

an entry with the least key greater than or equal to key, or null if there is no such key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

Since:

1.6

ceilingKey

```
public K ceilingKey(K key)
```

Description copied from interface: NavigableMap

Returns the least key greater than or equal to the given key, or null if there is no such key.

Specified by:

ceilingKey in interface NavigableMap<K,V>

Parameters:

key - the key

Returns:

the least key greater than or equal to key, or null if there is no such key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

Since:

1.6

higherEntry

```
public Map.Entry<K,V> higherEntry(K key)
```

Description copied from interface: NavigableMap

Returns a key-value mapping associated with the least key strictly greater than the given key, or `null` if there is no such key.

Specified by:

higherEntry in interface NavigableMap<K,V>

Parameters:

key - the key

Returns:

an entry with the least key greater than key, or null if there is no such key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

Since:

1.6

higherKey

```
public K higherKey(K key)
```

Description copied from interface: NavigableMap

Returns the least key strictly greater than the given key, or `null` if there is no such key.

Specified by:

higherKey in interface NavigableMap<K,V>

Parameters:

key - the key

Returns:

the least key greater than key, or null if there is no such key

Throws:

ClassCastException - if the specified key cannot be compared with the keys currently in the map

NullPointerException - if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

Since:

1.6

keySet

```
public Set<K> keySet()
```

Returns a Set view of the keys contained in this map.

The set's iterator returns the keys in ascending order. The set's spliterator is *late-binding*, *fail-fast*, and additionally reports Spliterator.SORTED and Spliterator.ORDERED with an encounter order that is ascending key order. The spliterator's comparator (see Spliterator.getComparator()) is null if the tree map's comparator (see comparator()) is null. Otherwise, the spliterator's comparator is the same as or imposes the same total ordering as the tree map's comparator.

The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

Specified by:

keySet in interface Map<K,V>

Specified by:

keySet in interface SortedMap<K,V>

Overrides:

keySet in class AbstractMap<K,V>

Returns:

a set view of the keys contained in this map

navigableKeySet

```
public NavigableSet<K> navigableKeySet()
```

Description copied from interface: NavigableMap

Returns a NavigableSet view of the keys contained in this map. The set's iterator returns the keys in ascending order. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

Specified by:

navigableKeySet in interface NavigableMap<K,V>

Returns:

a navigable set view of the keys in this map

Since:

1.6

descendingKeySet

public NavigableSet<K> descendingKeySet()

Description copied from interface: NavigableMap

Returns a reverse order NavigableSet view of the keys contained in this map. The set's iterator returns the keys in descending order. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

Specified by:

descendingKeySet in interface NavigableMap<K,V>

Returns:

a reverse order navigable set view of the keys in this map

Since:

1.6

values

public Collection<V> values()

Returns a Collection view of the values contained in this map.

The collection's iterator returns the values in ascending order of the corresponding keys. The collection's spliterator is *late-binding*, *fail-fast*, and additionally reports Spliterator.ORDERED with an encounter order that is ascending order of the corresponding keys.

The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa. If the map is modified while an iteration over the collection is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Collection.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

Specified by:

values in interface Map<K,V>

Specified by:

values in interface SortedMap<K,V>

Overrides:

values in class AbstractMap<K,V>

Returns:

a collection view of the values contained in this map

entrySet

public Set<Map.Entry<K,V>> entrySet()

Returns a Set view of the mappings contained in this map.

The set's iterator returns the entries in ascending key order. The sets's spliterator is *late-binding*, *fail-fast*, and additionally reports Spliterator.SORTED and Spliterator.ORDERED with an encounter order that is ascending key order.

The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation, or through the setValue operation on a map entry returned by the iterator) the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

Specified by:

entrySet in interface Map<K,V>

Specified by:

entrySet in interface SortedMap<K,V>

Specified by:

entrySet in class AbstractMap<K,V>

Returns:

a set view of the mappings contained in this map

descendingMap

public NavigableMap<K,V> descendingMap()

Description copied from interface: NavigableMap

Returns a reverse order view of the mappings contained in this map. The descending map is backed by this map, so changes to the map are reflected in the descending map, and vice-versa. If either map is modified while an iteration over a collection view of either map is in progress (except through the iterator's own remove operation), the results of the iteration are undefined.

The returned map has an ordering equivalent to Collections.reverseOrder(comparator()). The expression m.descendingMap().descendingMap() returns a view of m essentially equivalent to m.

Specified by:

descendingMap in interface NavigableMap<K,V>

Returns:

a reverse order view of this map

Since:

1.6

subMap

```
public NavigableMap<K,V> subMap(K fromKey,
                                boolean fromInclusive,
                                K toKey,
                                boolean toInclusive)
```

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys range from `fromKey` to `toKey`. If `fromKey` and `toKey` are equal, the returned map is empty unless `fromInclusive` and `toInclusive` are both true. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a key outside of its range, or to construct a submap either of whose endpoints lie outside its range.

Specified by:

`subMap` in interface `NavigableMap<K,V>`

Parameters:

`fromKey` - low endpoint of the keys in the returned map

`fromInclusive` - true if the low endpoint is to be included in the returned view

`toKey` - high endpoint of the keys in the returned map

`toInclusive` - true if the high endpoint is to be included in the returned view

Returns:

a view of the portion of this map whose keys range from fromKey to toKey

Throws:

`ClassCastException` - if fromKey and toKey cannot be compared to one another using this map's comparator (or, if the map has no comparator, using natural ordering). Implementations may, but are not required to, throw this exception if fromKey or toKey cannot be compared to keys currently in the map.

`NullPointerException` - if fromKey or toKey is null and this map uses natural ordering, or its comparator does not permit null keys

`IllegalArgumentException` - if fromKey is greater than toKey; or if this map itself has a restricted range, and fromKey or toKey lies outside the bounds of the range

Since:

1.6

headMap

```
public NavigableMap<K,V> headMap(K toKey,
                                 boolean inclusive)
```

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a key outside its range.

Specified by:

`headMap` in interface `NavigableMap<K,V>`

Parameters:

`toKey` - high endpoint of the keys in the returned map

`inclusive` - true if the high endpoint is to be included in the returned view

Returns:

a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey

Throws:

`ClassCastException` - if toKey is not compatible with this map's comparator (or, if the map has no comparator, if toKey does not implement `Comparable`). Implementations may, but are not required to, throw this exception if toKey cannot be compared to keys currently in the map.

`NullPointerException` - if toKey is null and this map uses natural ordering, or its comparator does not permit null keys

`IllegalArgumentException` - if this map itself has a restricted range, and toKey lies outside the bounds of the range

Since:

1.6

tailMap

```
public NavigableMap<K,V> tailMap(K fromKey,
                                 boolean inclusive)
```

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys are greater than (or equal to, if `inclusive` is true) `fromKey`. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a key outside its range.

Specified by:

tailMap in interface NavigableMap<K,V>

Parameters:

fromKey - low endpoint of the keys in the returned map

inclusive - true if the low endpoint is to be included in the returned view

Returns:

a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey

Throws:

ClassCastException - if fromKey is not compatible with this map's comparator (or, if the map has no comparator, if fromKey does not implement Comparable). Implementations may, but are not required to, throw this exception if fromKey cannot be compared to keys currently in the map.

NullPointerException - if fromKey is null and this map uses natural ordering, or its comparator does not permit null keys

IllegalArgumentException - if this map itself has a restricted range, and fromKey lies outside the bounds of the range

Since:

1.6

subMap

```
public SortedMap<K,V> subMap(K fromKey,
                             K toKey)
```

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive. (If fromKey and toKey are equal, the returned map is empty.) The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an IllegalArgumentException on an attempt to insert a key outside its range.

Equivalent to subMap(fromKey, true, toKey, false).

Specified by:

subMap in interface NavigableMap<K,V>

Specified by:

subMap in interface SortedMap<K,V>

Parameters:

fromKey - low endpoint (inclusive) of the keys in the returned map

toKey - high endpoint (exclusive) of the keys in the returned map

Returns:

a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive

Throws:

ClassCastException - if fromKey and toKey cannot be compared to one another using this map's comparator (or, if the map has no comparator, using natural ordering). Implementations may, but are not required to, throw this exception if fromKey or toKey cannot be compared to keys currently in the map.

NullPointerException - if fromKey or toKey is null and this map uses natural ordering, or its comparator does not permit null keys

IllegalArgumentException - if fromKey is greater than toKey; or if this map itself has a restricted range, and fromKey or toKey lies outside the bounds of the range

headMap

public SortedMap<K,V> headMap(K toKey)

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys are strictly less than toKey. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an IllegalArgumentException on an attempt to insert a key outside its range.

Equivalent to headMap(toKey, false).

Specified by:

headMap in interface NavigableMap<K,V>

Specified by:

headMap in interface SortedMap<K,V>

Parameters:

toKey - high endpoint (exclusive) of the keys in the returned map

Returns:

a view of the portion of this map whose keys are strictly less than toKey

Throws:

ClassCastException - if toKey is not compatible with this map's comparator (or, if the map has no comparator, if toKey does not implement Comparable). Implementations may, but are not required to, throw this exception if toKey cannot be compared to keys currently in the map.

NullPointerException - if toKey is null and this map uses natural ordering, or its comparator does not permit null keys

IllegalArgumentException - if this map itself has a restricted range, and toKey lies outside the bounds of the range

tailMap

public SortedMap<K,V> tailMap(K fromKey)

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a key outside its range.

Equivalent to `tailMap(fromKey, true)`.

Specified by:

`tailMap` in interface `NavigableMap<K,V>`

Specified by:

`tailMap` in interface `SortedMap<K,V>`

Parameters:

`fromKey - low endpoint (inclusive) of the keys in the returned map`

Returns:

`a view of the portion of this map whose keys are greater than or equal to fromKey`

Throws:

`ClassCastException - if fromKey is not compatible with this map's comparator (or, if the map has no comparator, if fromKey does not implement Comparable). Implementations may, but are not required to, throw this exception if fromKey cannot be compared to keys currently in the map.`

`NullPointerException - if fromKey is null and this map uses natural ordering, or its comparator does not permit null keys`

`IllegalArgumentException - if this map itself has a restricted range, and fromKey lies outside the bounds of the range`

replace

```
public boolean replace(K key,
                       V oldValue,
                       V newValue)
```

Description copied from interface: Map

Replaces the entry for the specified key only if currently mapped to the specified value.

Specified by:

`replace` in interface `Map<K,V>`

Parameters:

`key - key with which the specified value is associated`

`oldValue - value expected to be associated with the specified key`

`newValue - value to be associated with the specified key`

Returns:

`true if the value was replaced`

replace

```
public V replace(K key,
                 V value)
```

Description copied from interface: Map

Replaces the entry for the specified key only if it is currently mapped to some value.

Specified by:

replace in interface Map<K,V>

Parameters:

key - key with which the specified value is associated

value - value to be associated with the specified key

Returns:

the previous value associated with the specified key, or null if there was no mapping for the key. (A null return can also indicate that the map previously associated null with the key, if the implementation supports null values.)

forEach

```
public void forEach(BiConsumer<? super K,? super V> action)
```

Description copied from interface: Map

Performs the given action for each entry in this map until all entries have been processed or the action throws an exception. Unless otherwise specified by the implementing class, actions are performed in the order of entry set iteration (if an iteration order is specified.) Exceptions thrown by the action are relayed to the caller.

Specified by:

forEach in interface Map<K,V>

Parameters:

action - The action to be performed for each entry

replaceAll

```
public void replaceAll(BiFunction<? super K,? super V,? extends V> function)
```

Description copied from interface: Map

Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. Exceptions thrown by the function are relayed to the caller.

Specified by:

replaceAll in interface Map<K,V>

Parameters:

function - the function to apply to each entry

Submit a bug or feature

For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

compact1, compact2, compact3

java.util

Class TreeSet<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractSet<E>
            java.util.TreeSet<E>

Type Parameters:

`E - the type of elements maintained by this set`

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `NavigableSet<E>`, `Set<E>`, `SortedSet<E>`

```
public class TreeSet<E>
extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, Serializable
```

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`).

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be *consistent with equals* if it is to correctly implement the `Set` interface. (See `Comparable` or `Comparator` 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 `TreeSet` instance 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 set, equal. The behavior of a set *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the `Set` interface.

Note that this implementation is not synchronized. If multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the `Collections.synchronizedSortedSet` method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));
```

The iterators returned by this class's `iterator` method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own `remove` method, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of

concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `Set`, `HashSet`, `Comparable`, `Comparator`, `TreeMap`, Serialized Form

Constructor Summary

Constructors

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.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and 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

	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.
`Comparator``<? super E>`	**`comparator`**`()` Returns the comparator used to order the elements in this set, or `null` if this set uses the **natural ordering** of its elements.
`boolean`	**`contains`**`(Object o)` Returns `true` if this set contains the specified element.
`Iterator``<E>`	**`descendingIterator`**`()` Returns an iterator over the elements in this set in descending order.
`NavigableSet``<E>`	**`descendingSet`**`()` Returns a reverse order view of the elements contained in this set.
`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.
`SortedSet``<E>`	**`headSet`**`(E toElement)` Returns a view of the portion of this set whose elements are strictly less than `toElement`.
`NavigableSet``<E>`	**`headSet`**`(E toElement, boolean inclusive)` Returns a view of the portion of this set whose elements are less than (or equal to, if `inclusive` is true) `toElement`.
`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).
Spliterator<**E**>	**spliterator**() Creates a ***late-binding*** and *fail-fast* **Spliterator** over the elements in this set.
NavigableSet<**E**>	**subSet**(**E** fromElement, boolean fromInclusive, **E** toElement, boolean toInclusive) Returns a view of the portion of this set whose elements range from `fromElement` to `toElement`.
SortedSet<**E**>	**subSet**(**E** fromElement, **E** toElement) Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.
SortedSet<**E**>	**tailSet**(**E** fromElement) Returns a view of the portion of this set whose elements are greater than or equal to `fromElement`.
NavigableSet<**E**>	**tailSet**(**E** fromElement, boolean inclusive) Returns a view of the portion of this set whose elements are greater than (or equal to, if `inclusive` is true) `fromElement`.

Methods inherited from class java.util.AbstractSet

equals, hashCode, removeAll

Methods inherited from class java.util.AbstractCollection

containsAll, retainAll, toArray, toArray, toString

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Set

containsAll, equals, hashCode, removeAll, retainAll, toArray, toArray

Methods inherited from interface java.util.Collection

parallelStream, removeIf, stream

Methods inherited from interface java.lang.Iterable

forEach

Constructor Detail

TreeSet

```
public TreeSet()
```

Constructs a new, empty tree set, sorted according to the natural ordering of its elements. All elements inserted into the set must implement the Comparable interface. Furthermore, all such elements must be *mutually comparable*: e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the set. If the user attempts to add an element to the set that violates this constraint (for example, the user attempts to add a string element to a set whose elements are integers), the add call will throw a ClassCastException.

TreeSet

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

Constructs a new, empty tree set, sorted according to the specified comparator. All elements inserted into the set must be *mutually comparable* by the specified comparator: comparator.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the set. If the user attempts to add an element to the set that violates this constraint, the add call will throw a ClassCastException.

Parameters:

comparator - the comparator that will be used to order this set. If null, the natural ordering of the elements will be used.

TreeSet

```
public 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. All elements inserted into the set must implement the Comparable interface. Furthermore, all such elements must be *mutually comparable*: e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the set.

Parameters:

c - collection whose elements will comprise the new set

Throws:

ClassCastException - if the elements in c are not Comparable, or are not mutually comparable

NullPointerException - if the specified collection is null

TreeSet

```
public TreeSet(SortedSet<E> s)
```

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

Parameters:

`s - sorted set whose elements will comprise the new set`

Throws:

`NullPointerException - if the specified sorted set is null`

Method Detail

iterator

```
public Iterator<E> iterator()
```

Returns an iterator over the elements in this set in ascending order.

Specified by:

`iterator in interface Iterable<E>`

Specified by:

`iterator in interface Collection<E>`

Specified by:

`iterator in interface NavigableSet<E>`

Specified by:

`iterator in interface Set<E>`

Specified by:

`iterator in class AbstractCollection<E>`

Returns:

`an iterator over the elements in this set in ascending order`

descendingIterator

```
public Iterator<E> descendingIterator()
```

Returns an iterator over the elements in this set in descending order.

Specified by:

`descendingIterator in interface NavigableSet<E>`

Returns:

`an iterator over the elements in this set in descending order`

Since:

`1.6`

descendingSet

```
public NavigableSet<E> descendingSet()
```

Description copied from interface: NavigableSet

Returns a reverse order view of the elements contained in this set. The descending set is backed by this set, so changes to the set are reflected in the descending set, and vice-versa. If either set is modified while an iteration over either set is in progress (except through the iterator's own `remove` operation), the results of the iteration are undefined.

The returned set has an ordering equivalent to `Collections.reverseOrder(comparator())`. The expression `s.descendingSet().descendingSet()` returns a view of `s` essentially equivalent to `s`.

Specified by:

`descendingSet` in interface `NavigableSet<E>`

Returns:

a reverse order view of this set

Since:

1.6

size

```
public int size()
```

Returns the number of elements in this set (its cardinality).

Specified by:

`size` in interface `Collection<E>`

Specified by:

`size` in interface `Set<E>`

Specified by:

`size` in class `AbstractCollection<E>`

Returns:

the number of elements in this set (its cardinality)

isEmpty

```
public boolean isEmpty()
```

Returns `true` if this set contains no elements.

Specified by:

`isEmpty` in interface `Collection<E>`

Specified by:

`isEmpty` in interface `Set<E>`

Overrides:

`isEmpty` in class `AbstractCollection<E>`

Returns:

true if this set contains no elements

contains

public boolean contains(Object o)

Returns true if this set contains the specified element. More formally, returns true if and only if this set contains an element e such that (o==null ? e==null : o.equals(e)).

Specified by:

contains in interface Collection<E>

Specified by:

contains in interface Set<E>

Overrides:

contains in class AbstractCollection<E>

Parameters:

o - object to be checked for containment in this set

Returns:

true if this set contains the specified element

Throws:

ClassCastException - if the specified object cannot be compared with the elements currently in the set

NullPointerException - if the specified element is null and this set uses natural ordering, or its comparator does not permit null elements

add

public boolean add(E e)

Adds the specified element to this set if it is not already present. More formally, adds the specified element e to this set if the set contains no element e2 such that (e==null ? e2==null : e.equals(e2)). If this set already contains the element, the call leaves the set unchanged and returns false.

Specified by:

add in interface Collection<E>

Specified by:

add in interface Set<E>

Overrides:

add in class AbstractCollection<E>

Parameters:

e - element to be added to this set

Returns:

true if this set did not already contain the specified element

Throws:

ClassCastException - if the specified object cannot be compared with the elements currently in this set

NullPointerException - if the specified element is null and this set uses natural ordering, or its comparator does not permit null elements

remove

```
public boolean remove(Object o)
```

Removes the specified element from this set if it is present. More formally, removes an element e such that (o==null ? e==null : o.equals(e)), if this set contains such an element. Returns true if this set contained the element (or equivalently, if this set changed as a result of the call). (This set will not contain the element once the call returns.)

Specified by:

remove in interface Collection<E>

Specified by:

remove in interface Set<E>

Overrides:

remove in class AbstractCollection<E>

Parameters:

o - object to be removed from this set, if present

Returns:

true if this set contained the specified element

Throws:

ClassCastException - if the specified object cannot be compared with the elements currently in this set

NullPointerException - if the specified element is null and this set uses natural ordering, or its comparator does not permit null elements

clear

```
public void clear()
```

Removes all of the elements from this set. The set will be empty after this call returns.

Specified by:

clear in interface Collection<E>

Specified by:

clear in interface Set<E>

Overrides:

clear in class AbstractCollection<E>

addAll

```
public boolean addAll(Collection<? extends E> c)
```

Adds all of the elements in the specified collection to this set.

Specified by:

addAll in interface Collection<E>

Specified by:

addAll in interface Set<E>

Overrides:

addAll in class AbstractCollection<E>

Parameters:

c - collection containing elements to be added to this set

Returns:

true if this set changed as a result of the call

Throws:

ClassCastException - if the elements provided cannot be compared with the elements currently in the set

NullPointerException - if the specified collection is null or if any element is null and this set uses natural ordering, or its comparator does not permit null elements

See Also:

AbstractCollection.add(Object)

subSet

```
public NavigableSet<E> subSet(E fromElement,
                              boolean fromInclusive,
                              E toElement,
                              boolean toInclusive)
```

Description copied from interface: NavigableSet

Returns a view of the portion of this set whose elements range from fromElement to toElement. If fromElement and toElement are equal, the returned set is empty unless fromInclusive and toInclusive are both true. The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa. The returned set supports all optional set operations that this set supports.

The returned set will throw an IllegalArgumentException on an attempt to insert an element outside its range.

Specified by:

subSet in interface NavigableSet<E>

Parameters:

fromElement - low endpoint of the returned set

fromInclusive - true if the low endpoint is to be included in the returned view

toElement - high endpoint of the returned set

toInclusive - true if the high endpoint is to be included in the returned view

Returns:

a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive

Throws:

ClassCastException - if fromElement and toElement cannot be compared to one another using this set's comparator (or, if the set has no comparator, using natural ordering). Implementations may, but are not required to, throw this exception if fromElement or toElement cannot be compared to elements currently in the set.

NullPointerException - if fromElement or toElement is null and this set uses natural ordering, or its comparator does not permit null elements

IllegalArgumentException - if fromElement is greater than toElement; or if this set itself has a restricted range, and fromElement or toElement lies outside the bounds of the range.

Since:

1.6

headSet

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

Description copied from interface: NavigableSet

Returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) toElement. The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa. The returned set supports all optional set operations that this set supports.

The returned set will throw an IllegalArgumentException on an attempt to insert an element outside its range.

Specified by:

headSet in interface NavigableSet<E>

Parameters:

toElement - high endpoint of the returned set

inclusive - true if the high endpoint is to be included in the returned view

Returns:

a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) toElement

Throws:

ClassCastException - if toElement is not compatible with this set's comparator (or, if the set has no comparator, if toElement does not implement Comparable). Implementations may, but are not required to, throw this exception if toElement cannot be compared to elements currently in the set.

NullPointerException - if toElement is null and this set uses natural ordering, or its comparator does not permit null elements

IllegalArgumentException - if this set itself has a restricted range, and toElement lies outside the bounds of the range

Since:

1.6

tailSet

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

Description copied from interface: NavigableSet

Returns a view of the portion of this set whose elements are greater than (or equal to, if inclusive is true) fromElement. The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa. The returned set supports all optional set operations that this set supports.

The returned set will throw an IllegalArgumentException on an attempt to insert an element outside its range.

Specified by:

tailSet in interface NavigableSet<E>

Parameters:

fromElement - low endpoint of the returned set

inclusive - true if the low endpoint is to be included in the returned view

Returns:

a view of the portion of this set whose elements are greater than or equal to fromElement

Throws:

ClassCastException - if fromElement is not compatible with this set's comparator (or, if the set has no comparator, if fromElement does not implement Comparable). Implementations may, but are not required to, throw this exception if fromElement cannot be compared to elements currently in the set.

NullPointerException - if fromElement is null and this set uses natural ordering, or its comparator does not permit null elements

IllegalArgumentException - if this set itself has a restricted range, and fromElement lies outside the bounds of the range

Since:

1.6

subSet

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

Description copied from interface: NavigableSet

Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive. (If `fromElement` and `toElement` are equal, the returned set is empty.) The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa. The returned set supports all optional set operations that this set supports.

The returned set will throw an `IllegalArgumentException` on an attempt to insert an element outside its range.

Equivalent to `subSet(fromElement, true, toElement, false)`.

Specified by:

`subSet` in interface `NavigableSet<E>`

Specified by:

`subSet` in interface `SortedSet<E>`

Parameters:

`fromElement` - low endpoint (inclusive) of the returned set

`toElement` - high endpoint (exclusive) of the returned set

Returns:

a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive

Throws:

`ClassCastException` - if fromElement and toElement cannot be compared to one another using this set's comparator (or, if the set has no comparator, using natural ordering). Implementations may, but are not required to, throw this exception if fromElement or toElement cannot be compared to elements currently in the set.

`NullPointerException` - if fromElement or toElement is null and this set uses natural ordering, or its comparator does not permit null elements

`IllegalArgumentException` - if fromElement is greater than toElement; or if this set itself has a restricted range, and fromElement or toElement lies outside the bounds of the range

headSet

```
public SortedSet<E> headSet(E toElement)
```

Description copied from interface: NavigableSet

Returns a view of the portion of this set whose elements are strictly less than `toElement`. The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa. The returned set supports all optional set operations that this set supports.

The returned set will throw an `IllegalArgumentException` on an attempt to insert an element outside its range.

Equivalent to `headSet(toElement, false)`.

Specified by:

headSet in interface NavigableSet<E>

Specified by:

headSet in interface SortedSet<E>

Parameters:

toElement - high endpoint (exclusive) of the returned set

Returns:

a view of the portion of this set whose elements are strictly less than toElement

Throws:

ClassCastException - if toElement is not compatible with this set's comparator (or, if the set has no comparator, if toElement does not implement Comparable). Implementations may, but are not required to, throw this exception if toElement cannot be compared to elements currently in the set.

NullPointerException - if toElement is null and this set uses natural ordering, or its comparator does not permit null elements

IllegalArgumentException - if this set itself has a restricted range, and toElement lies outside the bounds of the range

tailSet

public SortedSet<E> tailSet(E fromElement)

Description copied from interface: NavigableSet

Returns a view of the portion of this set whose elements are greater than or equal to fromElement. The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa. The returned set supports all optional set operations that this set supports.

The returned set will throw an IllegalArgumentException on an attempt to insert an element outside its range.

Equivalent to tailSet(fromElement, true).

Specified by:

tailSet in interface NavigableSet<E>

Specified by:

tailSet in interface SortedSet<E>

Parameters:

fromElement - low endpoint (inclusive) of the returned set

Returns:

a view of the portion of this set whose elements are greater than or equal to fromElement

Throws:

ClassCastException - if fromElement is not compatible with this set's comparator (or, if the set has no comparator, if fromElement does not implement Comparable). Implementations may, but are not required to, throw

this exception if fromElement cannot be compared to elements currently in the set.

NullPointerException - if fromElement is null and this set uses natural ordering, or its comparator does not permit null elements

IllegalArgumentException - if this set itself has a restricted range, and fromElement lies outside the bounds of the range

comparator

```
public Comparator<? super E> comparator()
```

Description copied from interface: SortedSet

Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.

Specified by:

comparator in interface SortedSet<E>

Returns:

the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements

first

```
public E first()
```

Description copied from interface: SortedSet

Returns the first (lowest) element currently in this set.

Specified by:

first in interface SortedSet<E>

Returns:

the first (lowest) element currently in this set

Throws:

NoSuchElementException - if this set is empty

last

```
public E last()
```

Description copied from interface: SortedSet

Returns the last (highest) element currently in this set.

Specified by:

last in interface SortedSet<E>

Returns:

the last (highest) element currently in this set

Throws:

NoSuchElementException - if this set is empty

lower

public E lower(E e)

Description copied from interface: NavigableSet

Returns the greatest element in this set strictly less than the given element, or null if there is no such element.

Specified by:

lower in interface NavigableSet<E>

Parameters:

e - the value to match

Returns:

the greatest element less than e, or null if there is no such element

Throws:

ClassCastException - if the specified element cannot be compared with the elements currently in the set

NullPointerException - if the specified element is null and this set uses natural ordering, or its comparator does not permit null elements

Since:

1.6

floor

public E floor(E e)

Description copied from interface: NavigableSet

Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.

Specified by:

floor in interface NavigableSet<E>

Parameters:

e - the value to match

Returns:

the greatest element less than or equal to e, or null if there is no such element

Throws:

ClassCastException - if the specified element cannot be compared with the elements currently in the set

NullPointerException - if the specified element is null and this set uses natural ordering, or its comparator does not permit null elements

Since:

1.6

ceiling

```
public E ceiling(E e)
```

Description copied from interface: NavigableSet

Returns the least element in this set greater than or equal to the given element, or null if there is no such element.

Specified by:

ceiling in interface NavigableSet<E>

Parameters:

e - the value to match

Returns:

the least element greater than or equal to e, or null if there is no such element

Throws:

ClassCastException - if the specified element cannot be compared with the elements currently in the set

NullPointerException - if the specified element is null and this set uses natural ordering, or its comparator does not permit null elements

Since:

1.6

higher

```
public E higher(E e)
```

Description copied from interface: NavigableSet

Returns the least element in this set strictly greater than the given element, or null if there is no such element.

Specified by:

higher in interface NavigableSet<E>

Parameters:

e - the value to match

Returns:

the least element greater than e, or null if there is no such element

Throws:

ClassCastException - if the specified element cannot be compared with the elements currently in the set

NullPointerException - if the specified element is null and this set uses natural ordering, or its comparator does not permit null elements

Since:

1.6

pollFirst

```
public E pollFirst()
```

Description copied from interface: NavigableSet
Retrieves and removes the first (lowest) element, or returns `null` if this set is empty.

Specified by:
`pollFirst` in interface `NavigableSet<E>`

Returns:
the first element, or null if this set is empty

Since:
1.6

pollLast

```
public E pollLast()
```

Description copied from interface: NavigableSet
Retrieves and removes the last (highest) element, or returns `null` if this set is empty.

Specified by:
`pollLast` in interface `NavigableSet<E>`

Returns:
the last element, or null if this set is empty

Since:
1.6

clone

```
public Object clone()
```

Returns a shallow copy of this `TreeSet` instance. (The elements themselves are not cloned.)

Overrides:
`clone` in class `Object`

Returns:
a shallow copy of this set

See Also:
`Cloneable`

spliterator

```
public Spliterator<E> spliterator()
```

Creates a *late-binding* and *fail-fast* `Spliterator` over the elements in this set.

The Spliterator reports Spliterator.SIZED, Spliterator.DISTINCT, Spliterator.SORTED, and Spliterator.ORDERED. Overriding implementations should document the reporting of additional characteristic values.

The spliterator's comparator (see Spliterator.getComparator()) is null if the tree set's comparator (see comparator()) is null. Otherwise, the spliterator's comparator is the same as or imposes the same total ordering as the tree set's comparator.

Specified by:

spliterator in interface Iterable<E>

Specified by:

spliterator in interface Collection<E>

Specified by:

spliterator in interface Set<E>

Specified by:

spliterator in interface SortedSet<E>

Returns:

a Spliterator over the elements in this set

Since:

1.8

compact1, compact2, compact3

java.util

Class Vector<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.Vector<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

Stack

```
public class Vector<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

The `Vector` class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a `Vector` can grow or shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Each vector tries to optimize storage management by maintaining a `capacity` and a `capacityIncrement`. The `capacity` is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of `capacityIncrement`. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the vector is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. The `Enumerations` returned by the `elements` method are *not* fail-fast.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

As of the Java 2 platform v1.2, this class was retrofitted to implement the `List` interface, making it a member of the Java Collections Framework. Unlike the new collection

implementations, `Vector` is synchronized. If a thread-safe implementation is not needed, it is recommended to use `ArrayList` in place of `Vector`.

Since:

`JDK1.0`

See Also:

`Collection`, `LinkedList`, `Serialized Form`

Field Summary

Fields

Modifier and Type	Field and Description
`protected int`	**`capacityIncrement`** The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.
`protected int`	**`elementCount`** The number of valid components in this `Vector` object.
`protected Object[]`	**`elementData`** The array buffer into which the components of the vector are stored.

Fields inherited from class java.util.AbstractList

`modCount`

Constructor Summary

Constructors

Constructor and Description
`Vector``()` Constructs an empty vector so that its internal data array has size `10` and its standard capacity increment is zero.
`Vector``(Collection<? extends E> c)` Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
`Vector``(int initialCapacity)` Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
`Vector``(int initialCapacity, int capacityIncrement)` Constructs an empty vector with the specified initial capacity and capacity increment.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) Appends the specified element to the end of this Vector.
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this Vector.
boolean	**addAll**(**Collection**<? extends **E**> c) Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified Collection into this Vector at the specified position.
void	**addElement**(**E** obj) Adds the specified component to the end of this vector, increasing its size by one.
int	**capacity**() Returns the current capacity of this vector.
void	**clear**() Removes all of the elements from this Vector.
Object	**clone**() Returns a clone of this vector.
boolean	**contains**(**Object** o) Returns `true` if this vector contains the specified element.
boolean	**containsAll**(**Collection**<?> c) Returns true if this Vector contains all of the elements in the specified Collection.
void	**copyInto**(**Object**[] anArray) Copies the components of this vector into the specified array.
E	**elementAt**(int index) Returns the component at the specified index.
Enumeration<**E**>	**elements**() Returns an enumeration of the components of this vector.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
boolean	**equals**(**Object** o) Compares the specified Object with this Vector for equality.

E	**firstElement**() Returns the first component (the item at index 0) of this vector.
void	**forEach**(**Consumer**<? super **E**> action) Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
E	**get**(int index) Returns the element at the specified position in this Vector.
int	**hashCode**() Returns the hash code value for this Vector.
int	**indexOf**(**Object** o) Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
int	**indexOf**(**Object** o, int index) Returns the index of the first occurrence of the specified element in this vector, searching forwards from index, or returns -1 if the element is not found.
void	**insertElementAt**(**E** obj, int index) Inserts the specified object as a component in this vector at the specified index.
boolean	**isEmpty**() Tests if this vector has no components.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
E	**lastElement**() Returns the last component of the vector.
int	**lastIndexOf**(**Object** o) Returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
int	**lastIndexOf**(**Object** o, int index) Returns the index of the last occurrence of the specified element in this vector, searching backwards from index, or returns -1 if the element is not found.
ListIterator<**E**>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<**E**>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index)

	Removes the element at the specified position in this Vector.
boolean	**remove**(**Object** o) Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
boolean	**removeAll**(**Collection**<?> c) Removes from this Vector all of its elements that are contained in the specified Collection.
void	**removeAllElements**() Removes all components from this vector and sets its size to zero.
boolean	**removeElement**(**Object** obj) Removes the first (lowest-indexed) occurrence of the argument from this vector.
void	**removeElementAt**(int index) Deletes the component at the specified index.
boolean	**removeIf**(**Predicate**<? super **E**> filter) Removes all of the elements of this collection that satisfy the given predicate.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive.
void	**replaceAll**(**UnaryOperator**<**E**> operator) Replaces each element of this list with the result of applying the operator to that element.
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this Vector that are contained in the specified Collection.
E	**set**(int index, **E** element) Replaces the element at the specified position in this Vector with the specified element.
void	**setElementAt**(**E** obj, int index) Sets the component at the specified `index` of this vector to be the specified object.
void	**setSize**(int newSize) Sets the size of this vector.
int	**size**() Returns the number of components in this vector.
void	**sort**(**Comparator**<? super **E**> c) Sorts this list according to the order induced by the specified **Comparator**.
Spliterator<**E**>	**spliterator**()

	Creates a ***late-binding*** and *fail-fast* **Spliterator** over the elements in this list.
List<**E**>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this Vector in the correct order.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
String	**toString**() Returns a string representation of this Vector, containing the String representation of each element.
void	**trimToSize**() Trims the capacity of this vector to be the vector's current size.

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Collection

parallelStream, stream

Field Detail

elementData

```
protected Object[] elementData
```

The array buffer into which the components of the vector are stored. The capacity of the vector is the length of this array buffer, and is at least large enough to contain all the vector's elements.

Any array elements following the last element in the Vector are null.

elementCount

```
protected int elementCount
```

The number of valid components in this `Vector` object. Components `elementData[0]` through `elementData[elementCount-1]` are the actual items.

capacityIncrement

```
protected int capacityIncrement
```

The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity. If the capacity increment is less than or equal to zero, the capacity of the vector is doubled each time it needs to grow.

Constructor Detail

Vector

```
public Vector(int initialCapacity,
              int capacityIncrement)
```

Constructs an empty vector with the specified initial capacity and capacity increment.

Parameters:

`initialCapacity` - the initial capacity of the vector

`capacityIncrement` - the amount by which the capacity is increased when the vector overflows

Throws:

`IllegalArgumentException` - if the specified initial capacity is negative

Vector

```
public Vector(int initialCapacity)
```

Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.

Parameters:

`initialCapacity` - the initial capacity of the vector

Throws:

`IllegalArgumentException` - if the specified initial capacity is negative

Vector

```
public Vector()
```

Constructs an empty vector so that its internal data array has size `10` and its standard capacity increment is zero.

Vector

```
public Vector(Collection<? extends E> c)
```

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

Parameters:

c - the collection whose elements are to be placed into this vector

Throws:

NullPointerException - if the specified collection is null

Since:

1.2

Method Detail

copyInto

public void copyInto(Object[] anArray)

Copies the components of this vector into the specified array. The item at index k in this vector is copied into component k of anArray.

Parameters:

anArray - the array into which the components get copied

Throws:

NullPointerException - if the given array is null

IndexOutOfBoundsException - if the specified array is not large enough to hold all the components of this vector

ArrayStoreException - if a component of this vector is not of a runtime type that can be stored in the specified array

See Also:

toArray(Object[])

trimToSize

public void trimToSize()

Trims the capacity of this vector to be the vector's current size. If the capacity of this vector is larger than its current size, then the capacity is changed to equal the size by replacing its internal data array, kept in the field elementData, with a smaller one. An application can use this operation to minimize the storage of a vector.

ensureCapacity

public void ensureCapacity(int minCapacity)

Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.

If the current capacity of this vector is less than minCapacity, then its capacity is increased by replacing its internal data array, kept in the field elementData, with a larger one. The size of the new data array will be the old size plus capacityIncrement, unless the value of capacityIncrement is less than or equal to zero, in which case the

new capacity will be twice the old capacity; but if this new size is still smaller than `minCapacity`, then the new capacity will be `minCapacity`.

Parameters:

`minCapacity - the desired minimum capacity`

setSize

`public void setSize(int newSize)`

Sets the size of this vector. If the new size is greater than the current size, new `null` items are added to the end of the vector. If the new size is less than the current size, all components at index `newSize` and greater are discarded.

Parameters:

`newSize - the new size of this vector`

Throws:

`ArrayIndexOutOfBoundsException - if the new size is negative`

capacity

`public int capacity()`

Returns the current capacity of this vector.

Returns:

`the current capacity (the length of its internal data array, kept in the field elementData of this vector)`

size

`public int size()`

Returns the number of components in this vector.

Specified by:

`size in interface Collection<E>`

Specified by:

`size in interface List<E>`

Specified by:

`size in class AbstractCollection<E>`

Returns:

`the number of components in this vector`

isEmpty

`public boolean isEmpty()`

Tests if this vector has no components.

Specified by:

`isEmpty` in interface `Collection<E>`

Specified by:

`isEmpty` in interface `List<E>`

Overrides:

`isEmpty` in class `AbstractCollection<E>`

Returns:

`true` if and only if this vector has no components, that is, its size is zero; `false` otherwise.

elements

```
public Enumeration<E> elements()
```

Returns an enumeration of the components of this vector. The returned `Enumeration` object will generate all items in this vector. The first item generated is the item at index `0`, then the item at index `1`, and so on.

Returns:

an enumeration of the components of this vector

See Also:

`Iterator`

contains

```
public boolean contains(Object o)
```

Returns `true` if this vector contains the specified element. More formally, returns `true` if and only if this vector contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

Specified by:

`contains` in interface `Collection<E>`

Specified by:

`contains` in interface `List<E>`

Overrides:

`contains` in class `AbstractCollection<E>`

Parameters:

`o` - element whose presence in this vector is to be tested

Returns:

`true` if this vector contains the specified element

indexOf

```
public int indexOf(Object o)
```

Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element. More formally, returns the lowest index `i` such that `(o==null ? get(i)==null : o.equals(get(i)))`, or -1 if there is no such index.

Specified by:

`indexOf` in interface `List<E>`

Overrides:

`indexOf` in class `AbstractList<E>`

Parameters:

`o` - element to search for

Returns:

the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element

indexOf

```
public int indexOf(Object o,
                   int index)
```

Returns the index of the first occurrence of the specified element in this vector, searching forwards from `index`, or returns -1 if the element is not found. More formally, returns the lowest index `i` such that `(i >= index && (o==null ? get(i)==null : o.equals(get(i))))`, or -1 if there is no such index.

Parameters:

`o` - element to search for

`index` - index to start searching from

Returns:

the index of the first occurrence of the element in this vector at position index or later in the vector; -1 if the element is not found.

Throws:

`IndexOutOfBoundsException` - if the specified index is negative

See Also:

`Object.equals(Object)`

lastIndexOf

```
public int lastIndexOf(Object o)
```

Returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element. More formally, returns the highest index `i` such that `(o==null ? get(i)==null : o.equals(get(i)))`, or -1 if there is no such index.

Specified by:

`lastIndexOf` in interface `List<E>`

Overrides:

`lastIndexOf` in class `AbstractList<E>`

Parameters:

`o - element to search for`

Returns:

`the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element`

lastIndexOf

```
public int lastIndexOf(Object o,
                       int index)
```

Returns the index of the last occurrence of the specified element in this vector, searching backwards from `index`, or returns -1 if the element is not found. More formally, returns the highest index `i` such that `(i <= index && (o==null ? get(i)==null : o.equals(get(i))))`, or -1 if there is no such index.

Parameters:

`o - element to search for`

`index - index to start searching backwards from`

Returns:

`the index of the last occurrence of the element at position less than or equal to index in this vector; -1 if the element is not found.`

Throws:

`IndexOutOfBoundsException - if the specified index is greater than or equal to the current size of this vector`

elementAt

```
public E elementAt(int index)
```

Returns the component at the specified index.

This method is identical in functionality to the `get(int)` method (which is part of the `List` interface).

Parameters:

`index - an index into this vector`

Returns:

`the component at the specified index`

Throws:

`ArrayIndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

firstElement

```
public E firstElement()
```

Returns the first component (the item at index `0`) of this vector.

Returns:

the first component of this vector

Throws:

`NoSuchElementException` - if this vector has no components

lastElement

```
public E lastElement()
```

Returns the last component of the vector.

Returns:

the last component of the vector, i.e., the component at index size() - 1.

Throws:

`NoSuchElementException` - if this vector is empty

setElementAt

```
public void setElementAt(E obj,
                         int index)
```

Sets the component at the specified `index` of this vector to be the specified object. The previous component at that position is discarded.

The index must be a value greater than or equal to `0` and less than the current size of the vector.

This method is identical in functionality to the `set(int, E)` method (which is part of the `List` interface). Note that the `set` method reverses the order of the parameters, to more closely match array usage. Note also that the `set` method returns the old value that was stored at the specified position.

Parameters:

obj - what the component is to be set to

index - the specified index

Throws:

`ArrayIndexOutOfBoundsException` - if the index is out of range (index < 0 || index >= size())

removeElementAt

```
public void removeElementAt(int index)
```

Deletes the component at the specified index. Each component in this vector with an index greater or equal to the specified `index` is shifted downward to have an index one smaller than the value it had previously. The size of this vector is decreased by 1.

The index must be a value greater than or equal to `0` and less than the current size of the vector.

This method is identical in functionality to the `remove(int)` method (which is part of the `List` interface). Note that the `remove` method returns the old value that was stored at the specified position.

Parameters:

`index - the index of the object to remove`

Throws:

`ArrayIndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

insertElementAt

```
public void insertElementAt(E obj,
                            int index)
```

Inserts the specified object as a component in this vector at the specified `index`. Each component in this vector with an index greater or equal to the specified `index` is shifted upward to have an index one greater than the value it had previously.

The index must be a value greater than or equal to `0` and less than or equal to the current size of the vector. (If the index is equal to the current size of the vector, the new element is appended to the Vector.)

This method is identical in functionality to the `add(int, E)` method (which is part of the `List` interface). Note that the `add` method reverses the order of the parameters, to more closely match array usage.

Parameters:

`obj - the component to insert`

`index - where to insert the new component`

Throws:

`ArrayIndexOutOfBoundsException - if the index is out of range (index < 0 || index > size())`

addElement

```
public void addElement(E obj)
```

Adds the specified component to the end of this vector, increasing its size by one. The capacity of this vector is increased if its size becomes greater than its capacity.

This method is identical in functionality to the `add(E)` method (which is part of the `List` interface).

Parameters:

`obj - the component to be added`

removeElement

```
public boolean removeElement(Object obj)
```

Removes the first (lowest-indexed) occurrence of the argument from this vector. If the object is found in this vector, each component in the vector with an index greater or equal to the object's index is shifted downward to have an index one smaller than the value it had previously.

This method is identical in functionality to the `remove(Object)` method (which is part of the `List` interface).

Parameters:

`obj - the component to be removed`

Returns:

`true if the argument was a component of this vector; false otherwise.`

removeAllElements

```
public void removeAllElements()
```

Removes all components from this vector and sets its size to zero.

This method is identical in functionality to the `clear()` method (which is part of the `List` interface).

clone

```
public Object clone()
```

Returns a clone of this vector. The copy will contain a reference to a clone of the internal data array, not a reference to the original internal data array of this `Vector` object.

Overrides:

`clone in class Object`

Returns:

`a clone of this vector`

See Also:

`Cloneable`

toArray

```
public Object[] toArray()
```

Returns an array containing all of the elements in this Vector in the correct order.

Specified by:

`toArray in interface Collection<E>`

Specified by:

`toArray in interface List<E>`

Overrides:

toArray in class AbstractCollection<E>

Returns:

an array containing all of the elements in this collection

Since:

1.2

See Also:

Arrays.asList(Object[])

toArray

public <T> T[] toArray(T[] a)

Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array. If the Vector fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this Vector.

If the Vector fits in the specified array with room to spare (i.e., the array has more elements than the Vector), the element in the array immediately following the end of the Vector is set to null. (This is useful in determining the length of the Vector *only* if the caller knows that the Vector does not contain any null elements.)

Specified by:

toArray in interface Collection<E>

Specified by:

toArray in interface List<E>

Overrides:

toArray in class AbstractCollection<E>

Type Parameters:

T - the runtime type of the array to contain the collection

Parameters:

a - the array into which the elements of the Vector are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose.

Returns:

an array containing the elements of the Vector

Throws:

ArrayStoreException - if the runtime type of a is not a supertype of the runtime type of every element in this Vector

NullPointerException - if the given array is null

Since:

1.2

get

public E get(int index)

Returns the element at the specified position in this Vector.

Specified by:

`get` in interface `List<E>`

Specified by:

`get` in class `AbstractList<E>`

Parameters:

`index - index of the element to return`

Returns:

`object at the specified index`

Throws:

`ArrayIndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

Since:

`1.2`

set

```
public E set(int index,
             E element)
```

Replaces the element at the specified position in this Vector with the specified element.

Specified by:

`set` in interface `List<E>`

Overrides:

`set` in class `AbstractList<E>`

Parameters:

`index - index of the element to replace`

`element - element to be stored at the specified position`

Returns:

`the element previously at the specified position`

Throws:

`ArrayIndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

Since:

`1.2`

add

```
public boolean add(E e)
```

Appends the specified element to the end of this Vector.

Specified by:

`add` in interface `Collection<E>`

Specified by:

`add` in interface `List<E>`

Overrides:

`add` in class `AbstractList<E>`

Parameters:

`e - element to be appended to this Vector`

Returns:

`true (as specified by Collection.add(E))`

Since:

`1.2`

remove

```
public boolean remove(Object o)
```

Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged. More formally, removes the element with the lowest index i such that `(o==null ? get(i)==null : o.equals(get(i)))` (if such an element exists).

Specified by:

`remove` in interface `Collection<E>`

Specified by:

`remove` in interface `List<E>`

Overrides:

`remove` in class `AbstractCollection<E>`

Parameters:

`o - element to be removed from this Vector, if present`

Returns:

`true if the Vector contained the specified element`

Since:

`1.2`

add

```
public void add(int index,
                E element)
```

Inserts the specified element at the specified position in this Vector. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

Specified by:

`add` in interface `List<E>`

Overrides:

`add` in class `AbstractList<E>`

Parameters:

index - index at which the specified element is to be inserted

element - element to be inserted

Throws:

ArrayIndexOutOfBoundsException - if the index is out of range (index < 0 || index > size())

Since:

1.2

remove

public E remove(int index)

Removes the element at the specified position in this Vector. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the Vector.

Specified by:

remove in interface List<E>

Overrides:

remove in class AbstractList<E>

Parameters:

index - the index of the element to be removed

Returns:

element that was removed

Throws:

ArrayIndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())

Since:

1.2

clear

public void clear()

Removes all of the elements from this Vector. The Vector will be empty after this call returns (unless it throws an exception).

Specified by:

clear in interface Collection<E>

Specified by:

clear in interface List<E>

Overrides:

clear in class AbstractList<E>

Since:

1.2

containsAll

public boolean containsAll(Collection<?> c)

Returns true if this Vector contains all of the elements in the specified Collection.

Specified by:

containsAll in interface Collection<E>

Specified by:

containsAll in interface List<E>

Overrides:

containsAll in class AbstractCollection<E>

Parameters:

c - a collection whose elements will be tested for containment in this Vector

Returns:

true if this Vector contains all of the elements in the specified collection

Throws:

NullPointerException - if the specified collection is null

See Also:

AbstractCollection.contains(Object)

addAll

public boolean addAll(Collection<? extends E> c)

Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator. The behavior of this operation is undefined if the specified Collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified Collection is this Vector, and this Vector is nonempty.)

Specified by:

addAll in interface Collection<E>

Specified by:

addAll in interface List<E>

Overrides:

addAll in class AbstractCollection<E>

Parameters:

c - elements to be inserted into this Vector

Returns:

true if this Vector changed as a result of the call

Throws:

NullPointerException - if the specified collection is null

Since:

1.2

See Also:

AbstractCollection.add(Object)

removeAll

public boolean removeAll(Collection<?> c)

Removes from this Vector all of its elements that are contained in the specified Collection.

Specified by:

removeAll in interface Collection<E>

Specified by:

removeAll in interface List<E>

Overrides:

removeAll in class AbstractCollection<E>

Parameters:

c - a collection of elements to be removed from the Vector

Returns:

true if this Vector changed as a result of the call

Throws:

ClassCastException - if the types of one or more elements in this vector are incompatible with the specified collection (optional)

NullPointerException - if this vector contains one or more null elements and the specified collection does not support null elements (optional), or if the specified collection is null

Since:

1.2

See Also:

AbstractCollection.remove(Object), AbstractCollection.contains(Object)

retainAll

public boolean retainAll(Collection<?> c)

Retains only the elements in this Vector that are contained in the specified Collection. In other words, removes from this Vector all of its elements that are not contained in the specified Collection.

Specified by:

retainAll in interface Collection<E>

Specified by:

retainAll in interface List<E>

Overrides:

retainAll in class AbstractCollection<E>

Parameters:

c - a collection of elements to be retained in this Vector (all other elements are removed)

Returns:

true if this Vector changed as a result of the call

Throws:

ClassCastException - if the types of one or more elements in this vector are incompatible with the specified collection (optional)

NullPointerException - if this vector contains one or more null elements and the specified collection does not support null elements (optional), or if the specified collection is null

Since:

1.2

See Also:

AbstractCollection.remove(Object), AbstractCollection.contains(Object)

addAll

```
public boolean addAll(int index,
                      Collection<? extends E> c)
```

Inserts all of the elements in the specified Collection into this Vector at the specified position. Shifts the element currently at that position (if any) and any subsequent elements to the right (increases their indices). The new elements will appear in the Vector in the order that they are returned by the specified Collection's iterator.

Specified by:

addAll in interface List<E>

Overrides:

addAll in class AbstractList<E>

Parameters:

index - index at which to insert the first element from the specified collection

c - elements to be inserted into this Vector

Returns:

true if this Vector changed as a result of the call

Throws:

ArrayIndexOutOfBoundsException - if the index is out of range (index < 0 || index > size())

NullPointerException - if the specified collection is null

Since:

1.2

equals

```
public boolean equals(Object o)
```

Compares the specified Object with this Vector for equality. Returns true if and only if the specified Object is also a List, both Lists have the same size, and all corresponding pairs of elements in the two Lists are *equal*. (Two elements `e1` and `e2` are *equal* if `(e1==null ? e2==null : e1.equals(e2))`.) In other words, two Lists are defined to be equal if they contain the same elements in the same order.

Specified by:

`equals` in interface `Collection<E>`

Specified by:

`equals` in interface `List<E>`

Overrides:

`equals` in class `AbstractList<E>`

Parameters:

`o` - the Object to be compared for equality with this Vector

Returns:

`true` if the specified Object is equal to this Vector

See Also:

`Object.hashCode()`, `HashMap`

hashCode

```
public int hashCode()
```

Returns the hash code value for this Vector.

Specified by:

`hashCode` in interface `Collection<E>`

Specified by:

`hashCode` in interface `List<E>`

Overrides:

`hashCode` in class `AbstractList<E>`

Returns:

the hash code value for this list

See Also:

`Object.equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

toString

```
public String toString()
```

Returns a string representation of this Vector, containing the String representation of each element.

Overrides:

toString in class AbstractCollection<E>

Returns:

a string representation of this collection

subList

```
public List<E> subList(int fromIndex,
                       int toIndex)
```

Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive. (If fromIndex and toIndex are equal, the returned List is empty.) The returned List is backed by this List, so changes in the returned List are reflected in this List, and vice-versa. The returned List supports all of the optional List operations supported by this List.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a List can be used as a range operation by operating on a subList view instead of a whole List. For example, the following idiom removes a range of elements from a List:

```
list.subList(from, to).clear();
```

Similar idioms may be constructed for indexOf and lastIndexOf, and all of the algorithms in the Collections class can be applied to a subList.

The semantics of the List returned by this method become undefined if the backing list (i.e., this List) is *structurally modified* in any way other than via the returned List. (Structural modifications are those that change the size of the List, or otherwise perturb it in such a fashion that iterations in progress may yield incorrect results.)

Specified by:

subList in interface List<E>

Overrides:

subList in class AbstractList<E>

Parameters:

fromIndex - low endpoint (inclusive) of the subList

toIndex - high endpoint (exclusive) of the subList

Returns:

a view of the specified range within this List

Throws:

IndexOutOfBoundsException - if an endpoint index value is out of range (fromIndex < 0 || toIndex > size)

IllegalArgumentException - if the endpoint indices are out of order (fromIndex > toIndex)

removeRange

```
protected void removeRange(int fromIndex,
                           int toIndex)
```

Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the list by `(toIndex - fromIndex)` elements. (If `toIndex==fromIndex`, this operation has no effect.)

Overrides:

`removeRange` in class `AbstractList<E>`

Parameters:

`fromIndex - index of first element to be removed`

`toIndex - index after last element to be removed`

listIterator

```
public ListIterator<E> listIterator(int index)
```

Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list. The specified index indicates the first element that would be returned by an initial call to `next`. An initial call to `previous` would return the element with the specified index minus one.

The returned list iterator is *fail-fast*.

Specified by:

`listIterator` in interface `List<E>`

Overrides:

`listIterator` in class `AbstractList<E>`

Parameters:

`index - index of the first element to be returned from the list iterator (by a call to next)`

Returns:

`a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list`

Throws:

`IndexOutOfBoundsException - if the index is out of range (index < 0 || index > size())`

listIterator

```
public ListIterator<E> listIterator()
```

Returns a list iterator over the elements in this list (in proper sequence).

The returned list iterator is *fail-fast*.

Specified by:

`listIterator` in interface `List<E>`

Overrides:

listIterator in class AbstractList<E>

Returns:

a list iterator over the elements in this list (in proper sequence)

See Also:

listIterator(int)

iterator

```
public Iterator<E> iterator()
```

Returns an iterator over the elements in this list in proper sequence.

The returned iterator is *fail-fast*.

Specified by:

iterator in interface Iterable<E>

Specified by:

iterator in interface Collection<E>

Specified by:

iterator in interface List<E>

Overrides:

iterator in class AbstractList<E>

Returns:

an iterator over the elements in this list in proper sequence

forEach

```
public void forEach(Consumer<? super E> action)
```

Description copied from interface: Iterable

Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception. Unless otherwise specified by the implementing class, actions are performed in the order of iteration (if an iteration order is specified). Exceptions thrown by the action are relayed to the caller.

Specified by:

forEach in interface Iterable<E>

Parameters:

action - The action to be performed for each element

removeIf

```
public boolean removeIf(Predicate<? super E> filter)
```

Description copied from interface: Collection

Removes all of the elements of this collection that satisfy the given predicate. Errors or runtime exceptions thrown during iteration or by the predicate are relayed to the

caller.

Specified by:

`removeIf` in interface `Collection<E>`

Parameters:

`filter - a predicate which returns true for elements to be removed`

Returns:

`true if any elements were removed`

replaceAll

```
public void replaceAll(UnaryOperator<E> operator)
```

Description copied from interface: `List`

Replaces each element of this list with the result of applying the operator to that element. Errors or runtime exceptions thrown by the operator are relayed to the caller.

Specified by:

`replaceAll` in interface `List<E>`

Parameters:

`operator - the operator to apply to each element`

sort

```
public void sort(Comparator<? super E> c)
```

Description copied from interface: `List`

Sorts this list according to the order induced by the specified `Comparator`.

All elements in this list must be *mutually comparable* using the specified comparator (that is, `c.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

If the specified comparator is `null` then all elements in this list must implement the `Comparable` interface and the elements' natural ordering should be used.

This list must be modifiable, but need not be resizable.

Specified by:

`sort` in interface `List<E>`

Parameters:

`c - the Comparator used to compare list elements. A null value indicates that the elements' natural ordering should be used`

spliterator

```
public Spliterator<E> spliterator()
```

Creates a *late-binding* and *fail-fast* `Spliterator` over the elements in this list.

The Spliterator reports Spliterator.SIZED, Spliterator.SUBSIZED, and Spliterator.ORDERED. Overriding implementations should document the reporting of additional characteristic values.

Specified by:

spliterator in interface Iterable<E>

Specified by:

spliterator in interface Collection<E>

Specified by:

spliterator in interface List<E>

Returns:

a Spliterator over the elements in this list

Since:

1.8

compact1, compact2, compact3

java.lang

Class Integer

java.lang.Object
    java.lang.Number
        java.lang.Integer

All Implemented Interfaces:

Serializable, Comparable<Integer>

```
public final class Integer
extends Number
implements Comparable<Integer>
```

The `Integer` class wraps a value of the primitive type `int` in an object. An object of type `Integer` contains a single field whose type is `int`.

In addition, this class provides several methods for converting an `int` to a `String` and a `String` to an `int`, as well as other constants and methods useful when dealing with an `int`.

Implementation note: The implementations of the "bit twiddling" methods (such as `highestOneBit` and `numberOfTrailingZeros`) are based on material from Henry S. Warren, Jr.'s *Hacker's Delight*, (Addison Wesley, 2002).

Since:

`JDK1.0`

See Also:

Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	**BYTES** The number of bytes used to represent a `int` value in two's complement binary form.
`static int`	**MAX_VALUE** A constant holding the maximum value an `int` can have, $2^{31}-1$.
`static int`	**MIN_VALUE** A constant holding the minimum value an `int` can have, -2^{31}.
`static int`	**SIZE**

	The number of bits used to represent an `int` value in two's complement binary form.
`static Class<Integer>`	**`TYPE`** The `Class` instance representing the primitive type `int`.

Constructor Summary

Constructors

Constructor and Description
`Integer``(int value)` Constructs a newly allocated `Integer` object that represents the specified `int` value.
`Integer``(String s)` Constructs a newly allocated `Integer` object that represents the `int` value indicated by the `String` parameter.

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`static int`	**`bitCount`**`(int i)` Returns the number of one-bits in the two's complement binary representation of the specified `int` value.
`byte`	**`byteValue`**`()` Returns the value of this `Integer` as a `byte` after a narrowing primitive conversion.
`static int`	**`compare`**`(int x, int y)` Compares two `int` values numerically.
`int`	**`compareTo`**`(Integer anotherInteger)` Compares two `Integer` objects numerically.
`static int`	**`compareUnsigned`**`(int x, int y)` Compares two `int` values numerically treating the values as unsigned.
`static Integer`	**`decode`**`(String nm)` Decodes a `String` into an `Integer`.
`static int`	**`divideUnsigned`**`(int dividend, int divisor)` Returns the unsigned quotient of dividing the first argument by the second where each argument and the result is interpreted as an unsigned value.
`double`	**`doubleValue`**`()` Returns the value of this `Integer` as a `double` after a widening primitive conversion.

primitive conversion.

boolean	**equals**(**Object** obj) Compares this object to the specified object.
float	**floatValue**() Returns the value of this Integer as a float after a widening primitive conversion.
static **Integer**	**getInteger**(**String** nm) Determines the integer value of the system property with the specified name.
static **Integer**	**getInteger**(**String** nm, int val) Determines the integer value of the system property with the specified name.
static **Integer**	**getInteger**(**String** nm, **Integer** val) Returns the integer value of the system property with the specified name.
int	**hashCode**() Returns a hash code for this Integer.
static int	**hashCode**(int value) Returns a hash code for a int value; compatible with Integer.hashCode().
static int	**highestOneBit**(int i) Returns an int value with at most a single one-bit, in the position of the highest-order ("leftmost") one-bit in the specified int value.
int	**intValue**() Returns the value of this Integer as an int.
long	**longValue**() Returns the value of this Integer as a long after a widening primitive conversion.
static int	**lowestOneBit**(int i) Returns an int value with at most a single one-bit, in the position of the lowest-order ("rightmost") one-bit in the specified int value.
static int	**max**(int a, int b) Returns the greater of two int values as if by calling **Math.max**.
static int	**min**(int a, int b) Returns the smaller of two int values as if by calling **Math.min**.
static int	**numberOfLeadingZeros**(int i) Returns the number of zero bits preceding the highest-order ("leftmost") one-bit in the two's complement binary representation of the specified int value.
static int	**numberOfTrailingZeros**(int i) Returns the number of zero bits following the lowest-order

	Returns the number of zero bits following the lowest-order ("rightmost") one-bit in the two's complement binary representation of the specified `int` value.
static int	**parseInt**(**String** s) Parses the string argument as a signed decimal integer.
static int	**parseInt**(**String** s, int radix) Parses the string argument as a signed integer in the radix specified by the second argument.
static int	**parseUnsignedInt**(**String** s) Parses the string argument as an unsigned decimal integer.
static int	**parseUnsignedInt**(**String** s, int radix) Parses the string argument as an unsigned integer in the radix specified by the second argument.
static int	**remainderUnsigned**(int dividend, int divisor) Returns the unsigned remainder from dividing the first argument by the second where each argument and the result is interpreted as an unsigned value.
static int	**reverse**(int i) Returns the value obtained by reversing the order of the bits in the two's complement binary representation of the specified `int` value.
static int	**reverseBytes**(int i) Returns the value obtained by reversing the order of the bytes in the two's complement representation of the specified `int` value.
static int	**rotateLeft**(int i, int distance) Returns the value obtained by rotating the two's complement binary representation of the specified `int` value left by the specified number of bits.
static int	**rotateRight**(int i, int distance) Returns the value obtained by rotating the two's complement binary representation of the specified `int` value right by the specified number of bits.
short	**shortValue**() Returns the value of this `Integer` as a `short` after a narrowing primitive conversion.
static int	**signum**(int i) Returns the signum function of the specified `int` value.
static int	**sum**(int a, int b) Adds two integers together as per the + operator.
static **String**	**toBinaryString**(int i) Returns a string representation of the integer argument as an unsigned integer in base 2.
static **String**	**toHexString**(int i) Returns a string representation of the integer argument as an

	Returns a string representation of the integer argument as an unsigned integer in base 16.
static String	**toOctalString**(int i) Returns a string representation of the integer argument as an unsigned integer in base 8.
String	**toString**() Returns a String object representing this Integer's value.
static String	**toString**(int i) Returns a String object representing the specified integer.
static String	**toString**(int i, int radix) Returns a string representation of the first argument in the radix specified by the second argument.
static long	**toUnsignedLong**(int x) Converts the argument to a long by an unsigned conversion.
static String	**toUnsignedString**(int i) Returns a string representation of the argument as an unsigned decimal value.
static String	**toUnsignedString**(int i, int radix) Returns a string representation of the first argument as an unsigned integer value in the radix specified by the second argument.
static Integer	**valueOf**(int i) Returns an Integer instance representing the specified int value.
static Integer	**valueOf**(String s) Returns an Integer object holding the value of the specified String.
static Integer	**valueOf**(String s, int radix) Returns an Integer object holding the value extracted from the specified String when parsed with the radix given by the second argument.

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Field Detail

MIN_VALUE

```
@Native
public static final int MIN_VALUE
```

A constant holding the minimum value an int can have, -2^{31}.

compact1, compact2, compact3

java.lang

Class Double

java.lang.Object
    java.lang.Number
        java.lang.Double

All Implemented Interfaces:

Serializable, Comparable<Double>

```
public final class Double
extends Number
implements Comparable<Double>
```

The `Double` class wraps a value of the primitive type `double` in an object. An object of type `Double` contains a single field whose type is `double`.

In addition, this class provides several methods for converting a `double` to a `String` and a `String` to a `double`, as well as other constants and methods useful when dealing with a `double`.

Since:

`JDK1.0`

See Also:

Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	**BYTES** The number of bytes used to represent a `double` value.
`static int`	**MAX_EXPONENT** Maximum exponent a finite `double` variable may have.
`static double`	**MAX_VALUE** A constant holding the largest positive finite value of type `double`, $(2-2^{-52})\cdot 2^{1023}$.
`static int`	**MIN_EXPONENT** Minimum exponent a normalized `double` variable may have.
`static double`	**MIN_NORMAL**

	A constant holding the smallest positive normal value of type `double`, 2^{-1022}.
`static double`	**`MIN_VALUE`** A constant holding the smallest positive nonzero value of type `double`, 2^{-1074}.
`static double`	**`NaN`** A constant holding a Not-a-Number (NaN) value of type `double`.
`static double`	**`NEGATIVE_INFINITY`** A constant holding the negative infinity of type `double`.
`static double`	**`POSITIVE_INFINITY`** A constant holding the positive infinity of type `double`.
`static int`	**`SIZE`** The number of bits used to represent a `double` value.
`static Class<Double>`	**`TYPE`** The `Class` instance representing the primitive type `double`.

Constructor Summary

Constructors

Constructor and Description
`Double``(double value)` Constructs a newly allocated `Double` object that represents the primitive `double` argument.
`Double``(String s)` Constructs a newly allocated `Double` object that represents the floating-point value of type `double` represented by the string.

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`byte`	**`byteValue`**`()` Returns the value of this `Double` as a `byte` after a narrowing primitive conversion.
`static int`	**`compare`**`(double d1, double d2)` Compares the two specified `double` values.
`int`	**`compareTo`**`(Double anotherDouble)` Compares two `Double` objects numerically.
`static long`	**`doubleToLongBits`**`(double value)`

	Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout.
static long	**doubleToRawLongBits**(double value) Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout, preserving Not-a-Number (NaN) values.
double	**doubleValue**() Returns the double value of this Double object.
boolean	**equals**(**Object** obj) Compares this object against the specified object.
float	**floatValue**() Returns the value of this Double as a float after a narrowing primitive conversion.
int	**hashCode**() Returns a hash code for this Double object.
static int	**hashCode**(double value) Returns a hash code for a double value; compatible with Double.hashCode().
int	**intValue**() Returns the value of this Double as an int after a narrowing primitive conversion.
static boolean	**isFinite**(double d) Returns true if the argument is a finite floating-point value; returns false otherwise (for NaN and infinity arguments).
boolean	**isInfinite**() Returns true if this Double value is infinitely large in magnitude, false otherwise.
static boolean	**isInfinite**(double v) Returns true if the specified number is infinitely large in magnitude, false otherwise.
boolean	**isNaN**() Returns true if this Double value is a Not-a-Number (NaN), false otherwise.
static boolean	**isNaN**(double v) Returns true if the specified number is a Not-a-Number (NaN) value, false otherwise.
static double	**longBitsToDouble**(long bits) Returns the double value corresponding to a given bit representation.
long	**longValue**() Returns the value of this Double as a long after a narrowing primitive conversion.

primitive conversion.

static double	**max**(double a, double b) Returns the greater of two double values as if by calling **Math.max**.
static double	**min**(double a, double b) Returns the smaller of two double values as if by calling **Math.min**.
static double	**parseDouble**(String s) Returns a new double initialized to the value represented by the specified String, as performed by the valueOf method of class Double.
short	**shortValue**() Returns the value of this Double as a short after a narrowing primitive conversion.
static double	**sum**(double a, double b) Adds two double values together as per the + operator.
static String	**toHexString**(double d) Returns a hexadecimal string representation of the double argument.
String	**toString**() Returns a string representation of this Double object.
static String	**toString**(double d) Returns a string representation of the double argument.
static Double	**valueOf**(double d) Returns a Double instance representing the specified double value.
static Double	**valueOf**(String s) Returns a Double object holding the double value represented by the argument string s.

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Field Detail

POSITIVE_INFINITY

```
public static final double POSITIVE_INFINITY
```

A constant holding the positive infinity of type double. It is equal to the value returned by Double.longBitsToDouble(0x7ff0000000000000L).

See Also:

compact1, compact2, compact3

java.lang

Class Character

java.lang.Object
    java.lang.Character

All Implemented Interfaces:

Serializable, Comparable<Character>

```
public final class Character
extends Object
implements Serializable, Comparable<Character>
```

The `Character` class wraps a value of the primitive type `char` in an object. An object of type `Character` contains a single field whose type is `char`.

In addition, this class provides several methods for determining a character's category (lowercase letter, digit, etc.) and for converting characters from uppercase to lowercase and vice versa.

Character information is based on the Unicode Standard, version 6.2.0.

The methods and data of class `Character` are defined by the information in the *UnicodeData* file that is part of the Unicode Character Database maintained by the Unicode Consortium. This file specifies various properties including name and general category for every defined Unicode code point or character range.

The file and its description are available from the Unicode Consortium at:

- http://www.unicode.org

Unicode Character Representations

The `char` data type (and therefore the value that a `Character` object encapsulates) are based on the original Unicode specification, which defined characters as fixed-width 16-bit entities. The Unicode Standard has since been changed to allow for characters whose representation requires more than 16 bits. The range of legal *code point*s is now U+0000 to U+10FFFF, known as *Unicode scalar value*. (Refer to the *definition* of the U+*n* notation in the Unicode Standard.)

The set of characters from U+0000 to U+FFFF is sometimes referred to as the *Basic Multilingual Plane (BMP)*. Characters whose code points are greater than U+FFFF are called *supplementary character*s. The Java platform uses the UTF-16 representation in `char` arrays and in the `String` and `StringBuffer` classes. In this representation, supplementary characters are represented as a pair of `char` values, the first from the *high-surrogates* range, (\uD800-\uDBFF), the second from the *low-surrogates* range (\uDC00-\uDFFF).

A `char` value, therefore, represents Basic Multilingual Plane (BMP) code points, including the surrogate code points, or code units of the UTF-16 encoding. An `int` value represents all

Unicode code points, including supplementary code points. The lower (least significant) 21 bits of `int` are used to represent Unicode code points and the upper (most significant) 11 bits must be zero. Unless otherwise specified, the behavior with respect to supplementary characters and surrogate `char` values is as follows:

- The methods that only accept a `char` value cannot support supplementary characters. They treat `char` values from the surrogate ranges as undefined characters. For example, `Character.isLetter('\uD840')` returns `false`, even though this specific value if followed by any low-surrogate value in a string would represent a letter.
- The methods that accept an `int` value support all Unicode characters, including supplementary characters. For example, `Character.isLetter(0x2F81A)` returns `true` because the code point value represents a letter (a CJK ideograph).

In the Java SE API documentation, *Unicode code point* is used for character values in the range between U+0000 and U+10FFFF, and *Unicode code unit* is used for 16-bit `char` values that are code units of the *UTF-16* encoding. For more information on Unicode terminology, refer to the Unicode Glossary.

Since:

`1.0`

See Also:

`Serialized Form`

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
`static class`	**`Character.Subset`** Instances of this class represent particular subsets of the Unicode character set.
`static class`	**`Character.UnicodeBlock`** A family of character subsets representing the character blocks in the Unicode specification.
`static class`	**`Character.UnicodeScript`** A family of character subsets representing the character scripts defined in the ***Unicode Standard Annex #24: Script Names***.

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	**`BYTES`** The number of bytes used to represent a `char` value in unsigned binary form.
`static byte`	**`COMBINING_SPACING_MARK`** General category "Mc" in the Unicode specification.

static byte	**CONNECTOR_PUNCTUATION** General category "Pc" in the Unicode specification.
static byte	**CONTROL** General category "Cc" in the Unicode specification.
static byte	**CURRENCY_SYMBOL** General category "Sc" in the Unicode specification.
static byte	**DASH_PUNCTUATION** General category "Pd" in the Unicode specification.
static byte	**DECIMAL_DIGIT_NUMBER** General category "Nd" in the Unicode specification.
static byte	**DIRECTIONALITY_ARABIC_NUMBER** Weak bidirectional character type "AN" in the Unicode specification.
static byte	**DIRECTIONALITY_BOUNDARY_NEUTRAL** Weak bidirectional character type "BN" in the Unicode specification.
static byte	**DIRECTIONALITY_COMMON_NUMBER_SEPARATOR** Weak bidirectional character type "CS" in the Unicode specification.
static byte	**DIRECTIONALITY_EUROPEAN_NUMBER** Weak bidirectional character type "EN" in the Unicode specification.
static byte	**DIRECTIONALITY_EUROPEAN_NUMBER_SEPARATOR** Weak bidirectional character type "ES" in the Unicode specification.
static byte	**DIRECTIONALITY_EUROPEAN_NUMBER_TERMINATOR** Weak bidirectional character type "ET" in the Unicode specification.
static byte	**DIRECTIONALITY_LEFT_TO_RIGHT** Strong bidirectional character type "L" in the Unicode specification.
static byte	**DIRECTIONALITY_LEFT_TO_RIGHT_EMBEDDING** Strong bidirectional character type "LRE" in the Unicode specification.
static byte	**DIRECTIONALITY_LEFT_TO_RIGHT_OVERRIDE** Strong bidirectional character type "LRO" in the Unicode specification.
static byte	**DIRECTIONALITY_NONSPACING_MARK** Weak bidirectional character type "NSM" in the Unicode specification.
static byte	**DIRECTIONALITY_OTHER_NEUTRALS**

	Neutral bidirectional character type "ON" in the Unicode specification.
static byte	**DIRECTIONALITY_PARAGRAPH_SEPARATOR** Neutral bidirectional character type "B" in the Unicode specification.
static byte	**DIRECTIONALITY_POP_DIRECTIONAL_FORMAT** Weak bidirectional character type "PDF" in the Unicode specification.
static byte	**DIRECTIONALITY_RIGHT_TO_LEFT** Strong bidirectional character type "R" in the Unicode specification.
static byte	**DIRECTIONALITY_RIGHT_TO_LEFT_ARABIC** Strong bidirectional character type "AL" in the Unicode specification.
static byte	**DIRECTIONALITY_RIGHT_TO_LEFT_EMBEDDING** Strong bidirectional character type "RLE" in the Unicode specification.
static byte	**DIRECTIONALITY_RIGHT_TO_LEFT_OVERRIDE** Strong bidirectional character type "RLO" in the Unicode specification.
static byte	**DIRECTIONALITY_SEGMENT_SEPARATOR** Neutral bidirectional character type "S" in the Unicode specification.
static byte	**DIRECTIONALITY_UNDEFINED** Undefined bidirectional character type.
static byte	**DIRECTIONALITY_WHITESPACE** Neutral bidirectional character type "WS" in the Unicode specification.
static byte	**ENCLOSING_MARK** General category "Me" in the Unicode specification.
static byte	**END_PUNCTUATION** General category "Pe" in the Unicode specification.
static byte	**FINAL_QUOTE_PUNCTUATION** General category "Pf" in the Unicode specification.
static byte	**FORMAT** General category "Cf" in the Unicode specification.
static byte	**INITIAL_QUOTE_PUNCTUATION** General category "Pi" in the Unicode specification.
static byte	**LETTER_NUMBER** General category "Nl" in the Unicode specification.
static byte	**LINE_SEPARATOR**

	General category "Zl" in the Unicode specification.
static byte	**LOWERCASE_LETTER** General category "Ll" in the Unicode specification.
static byte	**MATH_SYMBOL** General category "Sm" in the Unicode specification.
static int	**MAX_CODE_POINT** The maximum value of a **Unicode code point**, constant `U+10FFFF`.
static char	**MAX_HIGH_SURROGATE** The maximum value of a **Unicode high-surrogate code unit** in the UTF-16 encoding, constant `'\uDBFF'`.
static char	**MAX_LOW_SURROGATE** The maximum value of a **Unicode low-surrogate code unit** in the UTF-16 encoding, constant `'\uDFFF'`.
static int	**MAX_RADIX** The maximum radix available for conversion to and from strings.
static char	**MAX_SURROGATE** The maximum value of a Unicode surrogate code unit in the UTF-16 encoding, constant `'\uDFFF'`.
static char	**MAX_VALUE** The constant value of this field is the largest value of type `char`, `'\uFFFF'`.
static int	**MIN_CODE_POINT** The minimum value of a **Unicode code point**, constant `U+0000`.
static char	**MIN_HIGH_SURROGATE** The minimum value of a **Unicode high-surrogate code unit** in the UTF-16 encoding, constant `'\uD800'`.
static char	**MIN_LOW_SURROGATE** The minimum value of a **Unicode low-surrogate code unit** in the UTF-16 encoding, constant `'\uDC00'`.
static int	**MIN_RADIX** The minimum radix available for conversion to and from strings.
static int	**MIN_SUPPLEMENTARY_CODE_POINT** The minimum value of a **Unicode supplementary code point**, constant `U+10000`.
static char	**MIN_SURROGATE** The minimum value of a Unicode surrogate code unit in the UTF-16 encoding, constant `'\uD800'`.
static char	**MIN_VALUE**

	The constant value of this field is the smallest value of type `char`, `'\u0000'`.
static byte	**MODIFIER_LETTER** General category "Lm" in the Unicode specification.
static byte	**MODIFIER_SYMBOL** General category "Sk" in the Unicode specification.
static byte	**NON_SPACING_MARK** General category "Mn" in the Unicode specification.
static byte	**OTHER_LETTER** General category "Lo" in the Unicode specification.
static byte	**OTHER_NUMBER** General category "No" in the Unicode specification.
static byte	**OTHER_PUNCTUATION** General category "Po" in the Unicode specification.
static byte	**OTHER_SYMBOL** General category "So" in the Unicode specification.
static byte	**PARAGRAPH_SEPARATOR** General category "Zp" in the Unicode specification.
static byte	**PRIVATE_USE** General category "Co" in the Unicode specification.
static int	**SIZE** The number of bits used to represent a `char` value in unsigned binary form, constant 16.
static byte	**SPACE_SEPARATOR** General category "Zs" in the Unicode specification.
static byte	**START_PUNCTUATION** General category "Ps" in the Unicode specification.
static byte	**SURROGATE** General category "Cs" in the Unicode specification.
static byte	**TITLECASE_LETTER** General category "Lt" in the Unicode specification.
static Class<Character>	**TYPE** The `Class` instance representing the primitive type `char`.
static byte	**UNASSIGNED** General category "Cn" in the Unicode specification.
static byte	**UPPERCASE_LETTER** General category "Lu" in the Unicode specification.

Constructor Summary

Constructors

Constructor and Description
Character(char value) Constructs a newly allocated Character object that represents the specified char value.

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method and Description
static int	**charCount**(int codePoint) Determines the number of char values needed to represent the specified character (Unicode code point).
char	**charValue**() Returns the value of this Character object.
static int	**codePointAt**(char[] a, int index) Returns the code point at the given index of the char array.
static int	**codePointAt**(char[] a, int index, int limit) Returns the code point at the given index of the char array, where only array elements with index less than limit can be used.
static int	**codePointAt**(**CharSequence** seq, int index) Returns the code point at the given index of the CharSequence.
static int	**codePointBefore**(char[] a, int index) Returns the code point preceding the given index of the char array.
static int	**codePointBefore**(char[] a, int index, int start) Returns the code point preceding the given index of the char array, where only array elements with index greater than or equal to start can be used.
static int	**codePointBefore**(**CharSequence** seq, int index) Returns the code point preceding the given index of the CharSequence.
static int	**codePointCount**(char[] a, int offset, int count) Returns the number of Unicode code points in a subarray of the char array argument.
static int	**codePointCount**(**CharSequence** seq, int beginIndex, int endIndex) Returns the number of Unicode code points in the text range of the specified char sequence.

static int	**compare**(char x, char y) Compares two char values numerically.
int	**compareTo**(**Character** anotherCharacter) Compares two Character objects numerically.
static int	**digit**(char ch, int radix) Returns the numeric value of the character ch in the specified radix.
static int	**digit**(int codePoint, int radix) Returns the numeric value of the specified character (Unicode code point) in the specified radix.
boolean	**equals**(**Object** obj) Compares this object against the specified object.
static char	**forDigit**(int digit, int radix) Determines the character representation for a specific digit in the specified radix.
static byte	**getDirectionality**(char ch) Returns the Unicode directionality property for the given character.
static byte	**getDirectionality**(int codePoint) Returns the Unicode directionality property for the given character (Unicode code point).
static **String**	**getName**(int codePoint) Returns the Unicode name of the specified character codePoint, or null if the code point is **unassigned**.
static int	**getNumericValue**(char ch) Returns the int value that the specified Unicode character represents.
static int	**getNumericValue**(int codePoint) Returns the int value that the specified character (Unicode code point) represents.
static int	**getType**(char ch) Returns a value indicating a character's general category.
static int	**getType**(int codePoint) Returns a value indicating a character's general category.
int	**hashCode**() Returns a hash code for this Character; equal to the result of invoking charValue().
static int	**hashCode**(char value) Returns a hash code for a char value; compatible with Character.hashCode().
static char	**highSurrogate**(int codePoint)

	Returns the leading surrogate (a **high surrogate code unit**) of the **surrogate pair** representing the specified supplementary character (Unicode code point) in the UTF-16 encoding.
static boolean	**isAlphabetic**(int codePoint) Determines if the specified character (Unicode code point) is an alphabet.
static boolean	**isBmpCodePoint**(int codePoint) Determines whether the specified character (Unicode code point) is in the **Basic Multilingual Plane (BMP)**.
static boolean	**isDefined**(char ch) Determines if a character is defined in Unicode.
static boolean	**isDefined**(int codePoint) Determines if a character (Unicode code point) is defined in Unicode.
static boolean	**isDigit**(char ch) Determines if the specified character is a digit.
static boolean	**isDigit**(int codePoint) Determines if the specified character (Unicode code point) is a digit.
static boolean	**isHighSurrogate**(char ch) Determines if the given char value is a **Unicode high-surrogate code unit** (also known as *leading-surrogate code unit*).
static boolean	**isIdentifierIgnorable**(char ch) Determines if the specified character should be regarded as an ignorable character in a Java identifier or a Unicode identifier.
static boolean	**isIdentifierIgnorable**(int codePoint) Determines if the specified character (Unicode code point) should be regarded as an ignorable character in a Java identifier or a Unicode identifier.
static boolean	**isIdeographic**(int codePoint) Determines if the specified character (Unicode code point) is a CJKV (Chinese, Japanese, Korean and Vietnamese) ideograph, as defined by the Unicode Standard.
static boolean	**isISOControl**(char ch) Determines if the specified character is an ISO control character.
static boolean	**isISOControl**(int codePoint) Determines if the referenced character (Unicode code point) is an ISO control character.
static boolean	**isJavaIdentifierPart**(char ch) Determines if the specified character may be part of a Java identifier as other than the first character.

static boolean	**isJavaIdentifierPart**(int codePoint) Determines if the character (Unicode code point) may be part of a Java identifier as other than the first character.
static boolean	**isJavaIdentifierStart**(char ch) Determines if the specified character is permissible as the first character in a Java identifier.
static boolean	**isJavaIdentifierStart**(int codePoint) Determines if the character (Unicode code point) is permissible as the first character in a Java identifier.
static boolean	**isJavaLetter**(char ch) **Deprecated.** Replaced by isJavaIdentifierStart(char).
static boolean	**isJavaLetterOrDigit**(char ch) **Deprecated.** Replaced by isJavaIdentifierPart(char).
static boolean	**isLetter**(char ch) Determines if the specified character is a letter.
static boolean	**isLetter**(int codePoint) Determines if the specified character (Unicode code point) is a letter.
static boolean	**isLetterOrDigit**(char ch) Determines if the specified character is a letter or digit.
static boolean	**isLetterOrDigit**(int codePoint) Determines if the specified character (Unicode code point) is a letter or digit.
static boolean	**isLowerCase**(char ch) Determines if the specified character is a lowercase character.
static boolean	**isLowerCase**(int codePoint) Determines if the specified character (Unicode code point) is a lowercase character.
static boolean	**isLowSurrogate**(char ch) Determines if the given char value is a **Unicode low-surrogate code unit** (also known as *trailing-surrogate code unit*).
static boolean	**isMirrored**(char ch) Determines whether the character is mirrored according to the Unicode specification.
static boolean	**isMirrored**(int codePoint) Determines whether the specified character (Unicode code point) is mirrored according to the Unicode specification.
static boolean	**isSpace**(char ch) **Deprecated.** Replaced by isWhitespace(char).

static boolean	**isSpaceChar**(char ch) Determines if the specified character is a Unicode space character.
static boolean	**isSpaceChar**(int codePoint) Determines if the specified character (Unicode code point) is a Unicode space character.
static boolean	**isSupplementaryCodePoint**(int codePoint) Determines whether the specified character (Unicode code point) is in the **supplementary character** range.
static boolean	**isSurrogate**(char ch) Determines if the given char value is a Unicode *surrogate code unit*.
static boolean	**isSurrogatePair**(char high, char low) Determines whether the specified pair of char values is a valid **Unicode surrogate pair**.
static boolean	**isTitleCase**(char ch) Determines if the specified character is a titlecase character.
static boolean	**isTitleCase**(int codePoint) Determines if the specified character (Unicode code point) is a titlecase character.
static boolean	**isUnicodeIdentifierPart**(char ch) Determines if the specified character may be part of a Unicode identifier as other than the first character.
static boolean	**isUnicodeIdentifierPart**(int codePoint) Determines if the specified character (Unicode code point) may be part of a Unicode identifier as other than the first character.
static boolean	**isUnicodeIdentifierStart**(char ch) Determines if the specified character is permissible as the first character in a Unicode identifier.
static boolean	**isUnicodeIdentifierStart**(int codePoint) Determines if the specified character (Unicode code point) is permissible as the first character in a Unicode identifier.
static boolean	**isUpperCase**(char ch) Determines if the specified character is an uppercase character.
static boolean	**isUpperCase**(int codePoint) Determines if the specified character (Unicode code point) is an uppercase character.
static boolean	**isValidCodePoint**(int codePoint) Determines whether the specified code point is a valid **Unicode code point value**.
static boolean	**isWhitespace**(char ch)

	Determines if the specified character is white space according to Java.
static boolean	**isWhitespace**(int codePoint) Determines if the specified character (Unicode code point) is white space according to Java.
static char	**lowSurrogate**(int codePoint) Returns the trailing surrogate (a **low surrogate code unit**) of the **surrogate pair** representing the specified supplementary character (Unicode code point) in the UTF-16 encoding.
static int	**offsetByCodePoints**(char[] a, int start, int count, int index, int codePointOffset) Returns the index within the given char subarray that is offset from the given index by codePointOffset code points.
static int	**offsetByCodePoints**(**CharSequence** seq, int index, int codePointOffset) Returns the index within the given char sequence that is offset from the given index by codePointOffset code points.
static char	**reverseBytes**(char ch) Returns the value obtained by reversing the order of the bytes in the specified char value.
static char[]	**toChars**(int codePoint) Converts the specified character (Unicode code point) to its UTF-16 representation stored in a char array.
static int	**toChars**(int codePoint, char[] dst, int dstIndex) Converts the specified character (Unicode code point) to its UTF-16 representation.
static int	**toCodePoint**(char high, char low) Converts the specified surrogate pair to its supplementary code point value.
static char	**toLowerCase**(char ch) Converts the character argument to lowercase using case mapping information from the UnicodeData file.
static int	**toLowerCase**(int codePoint) Converts the character (Unicode code point) argument to lowercase using case mapping information from the UnicodeData file.
String	**toString**() Returns a String object representing this Character's value.
static **String**	**toString**(char c) Returns a String object representing the specified char.
static char	**toTitleCase**(char ch) Converts the character argument to titlecase using case mapping information from the UnicodeData file.

static int	**toTitleCase**(int codePoint) Converts the character (Unicode code point) argument to titlecase using case mapping information from the UnicodeData file.
static char	**toUpperCase**(char ch) Converts the character argument to uppercase using case mapping information from the UnicodeData file.
static int	**toUpperCase**(int codePoint) Converts the character (Unicode code point) argument to uppercase using case mapping information from the UnicodeData file.
static Character	**valueOf**(char c) Returns a Character instance representing the specified char value.

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Field Detail

MIN_RADIX

```
public static final int MIN_RADIX
```

The minimum radix available for conversion to and from strings. The constant value of this field is the smallest value permitted for the radix argument in radix-conversion methods such as the digit method, the forDigit method, and the toString method of class Integer.

See Also:

digit(char, int), forDigit(int, int), Integer.toString(int, int), Integer.valueOf(String), Constant Field Values

MAX_RADIX

```
public static final int MAX_RADIX
```

The maximum radix available for conversion to and from strings. The constant value of this field is the largest value permitted for the radix argument in radix-conversion methods such as the digit method, the forDigit method, and the toString method of class Integer.

See Also:

digit(char, int), forDigit(int, int), Integer.toString(int, int), Integer.valueOf(String), Constant Field Values

MIN_VALUE

compact1, compact2, compact3

java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

The class `String` includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the `Character` class.

The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. String concatenation is implemented through the `StringBuilder`(or `StringBuffer`) class and its `append` method. String conversions are implemented through the method `toString`, defined by `Object` and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, *The Java Language Specification*.

Unless otherwise noted, passing a `null` argument to a constructor or method in this class will cause a `NullPointerException` to be thrown.

A `String` represents a string in the UTF-16 format in which *supplementary characters* are represented by *surrogate pairs* (see the section Unicode Character Representations in the `Character` class for more information). Index values refer to `char` code units, so a supplementary character uses two positions in a `String`.

The `String` class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., `char` values).

Since:

`JDK1.0`

See Also:

`Object.toString()`, `StringBuffer`, `StringBuilder`, `Charset`, `Serialized Form`

Field Summary

Fields

Modifier and Type	Field and Description
`static Comparator<String>`	**`CASE_INSENSITIVE_ORDER`** A Comparator that orders `String` objects as by `compareToIgnoreCase`.

Constructor Summary

Constructors

Constructor and Description
`String``()` Initializes a newly created `String` object so that it represents an empty character sequence.
`String``(byte[] bytes)` Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.
`String``(byte[] bytes, Charset charset)` Constructs a new `String` by decoding the specified array of bytes using the specified **`charset`**.
`String``(byte[] ascii, int hibyte)` **Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a **`Charset`**, charset name, or that use the platform's default charset.
`String``(byte[] bytes, int offset, int length)` Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.

String(byte[] bytes, int offset, int length, **Charset** charset)
Constructs a new String by decoding the specified subarray of bytes using the specified **charset**.

String(byte[] ascii, int hibyte, int offset, int count)
Deprecated.
This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a **Charset**, charset name, or that use the platform's default charset.

String(byte[] bytes, int offset, int length, **String** charsetName)
Constructs a new String by decoding the specified subarray of bytes using the specified charset.

String(byte[] bytes, **String** charsetName)
Constructs a new String by decoding the specified array of bytes using the specified **charset**.

String(char[] value)
Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

String(char[] value, int offset, int count)
Allocates a new String that contains characters from a subarray of the character array argument.

String(int[] codePoints, int offset, int count)
Allocates a new String that contains characters from a subarray of the **Unicode code point** array argument.

String(**String** original)
Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

String(**StringBuffer** buffer)
Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

String(**StringBuilder** builder)
Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Concrete Methods** | **Deprecated Methods**

Modifier and Type	Method and Description
char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index)

	Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	**compareTo**(**String** anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(**String** str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(**String** str) Concatenates the specified string to the end of this string.
boolean	**contains**(**CharSequence** s) Returns true if and only if this string contains the specified sequence of char values.
boolean	**contentEquals**(**CharSequence** cs) Compares this string to the specified CharSequence.
boolean	**contentEquals**(**StringBuffer** sb) Compares this string to the specified StringBuffer.
static **String**	**copyValueOf**(char[] data) Equivalent to **valueOf(char[])**.
static **String**	**copyValueOf**(char[] data, int offset, int count) Equivalent to **valueOf(char[], int, int)**.
boolean	**endsWith**(**String** suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(**Object** anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(**String** anotherString) Compares this String to another String, ignoring case considerations.
static **String**	**format**(**Locale** l, **String** format, **Object**... args) Returns a formatted string using the specified locale, format string, and arguments.
static **String**	**format**(**String** format, **Object**... args) Returns a formatted string using the specified format string and arguments.
byte[]	**getBytes**()

	Encodes this `String` into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
`byte[]`	**`getBytes`**`(`**`Charset`** `charset)` Encodes this `String` into a sequence of bytes using the given **`charset`**, storing the result into a new byte array.
`void`	**`getBytes`**`(int srcBegin, int srcEnd, byte[] dst, int dstBegin)` **Deprecated.** This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the **`getBytes()`** method, which uses the platform's default charset.
`byte[]`	**`getBytes`**`(`**`String`** `charsetName)` Encodes this `String` into a sequence of bytes using the named charset, storing the result into a new byte array.
`void`	**`getChars`**`(int srcBegin, int srcEnd, char[] dst, int dstBegin)` Copies characters from this string into the destination character array.
`int`	**`hashCode`**`()` Returns a hash code for this string.
`int`	**`indexOf`**`(int ch)` Returns the index within this string of the first occurrence of the specified character.
`int`	**`indexOf`**`(int ch, int fromIndex)` Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
`int`	**`indexOf`**`(`**`String`** `str)` Returns the index within this string of the first occurrence of the specified substring.
`int`	**`indexOf`**`(`**`String`** `str, int fromIndex)` Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
`String`	**`intern`**`()` Returns a canonical representation for the string object.
`boolean`	**`isEmpty`**`()` Returns `true` if, and only if, **`length()`** is `0`.
`static` **`String`**	**`join`**`(`**`CharSequence`** `delimiter, `**`CharSequence`**`... elements)` Returns a new String composed of copies of the `CharSequence` `elements` joined together with a copy of the specified `delimiter`.
`static` **`String`**	**`join`**`(`**`CharSequence`** `delimiter, `**`Iterable`**`<? extends `**`CharSequence`**`> elements)`

	Returns a new `String` composed of copies of the `CharSequence` `elements` joined together with a copy of the specified `delimiter`.
int	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character.
int	**lastIndexOf**(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	**lastIndexOf**(**String** str) Returns the index within this string of the last occurrence of the specified substring.
int	**lastIndexOf**(**String** str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	**length**() Returns the length of this string.
boolean	**matches**(**String** regex) Tells whether or not this string matches the given **regular expression**.
int	**offsetByCodePoints**(int index, int codePointOffset) Returns the index within this `String` that is offset from the given `index` by `codePointOffset` code points.
boolean	**regionMatches**(boolean ignoreCase, int toffset, **String** other, int ooffset, int len) Tests if two string regions are equal.
boolean	**regionMatches**(int toffset, **String** other, int ooffset, int len) Tests if two string regions are equal.
String	**replace**(char oldChar, char newChar) Returns a string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
String	**replace**(**CharSequence** target, **CharSequence** replacement) Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
String	**replaceAll**(**String** regex, **String** replacement) Replaces each substring of this string that matches the given **regular expression** with the given replacement.
String	**replaceFirst**(**String** regex, **String** replacement) Replaces the first substring of this string that matches the given **regular expression** with the given replacement.

String[]	**split**(String regex) Splits this string around matches of the given **regular expression**.
String[]	**split**(String regex, int limit) Splits this string around matches of the given **regular expression**.
boolean	**startsWith**(String prefix) Tests if this string starts with the specified prefix.
boolean	**startsWith**(String prefix, int toffset) Tests if the substring of this string beginning at the specified index starts with the specified prefix.
CharSequence	**subSequence**(int beginIndex, int endIndex) Returns a character sequence that is a subsequence of this sequence.
String	**substring**(int beginIndex) Returns a string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a string that is a substring of this string.
char[]	**toCharArray**() Converts this string to a new character array.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toLowerCase**(Locale locale) Converts all of the characters in this String to lower case using the rules of the given Locale.
String	**toString**() This object (which is already a string!) is itself returned.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**toUpperCase**(Locale locale) Converts all of the characters in this String to upper case using the rules of the given Locale.
String	**trim**() Returns a string whose value is this string, with any leading and trailing whitespace removed.
static String	**valueOf**(boolean b) Returns the string representation of the boolean argument.
static String	**valueOf**(char c) Returns the string representation of the char argument.

static String	**valueOf**(char[] data) Returns the string representation of the char array argument.
static String	**valueOf**(char[] data, int offset, int count) Returns the string representation of a specific subarray of the char array argument.
static String	**valueOf**(double d) Returns the string representation of the double argument.
static String	**valueOf**(float f) Returns the string representation of the float argument.
static String	**valueOf**(int i) Returns the string representation of the int argument.
static String	**valueOf**(long l) Returns the string representation of the long argument.
static String	**valueOf**(Object obj) Returns the string representation of the Object argument.

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.lang.CharSequence

chars, codePoints

Field Detail

CASE_INSENSITIVE_ORDER

```
public static final Comparator<String> CASE_INSENSITIVE_ORDER
```

A Comparator that orders String objects as by compareToIgnoreCase. This comparator is serializable.

Note that this Comparator does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The java.text package provides *Collators* to allow locale-sensitive ordering.

Since:

1.2

See Also:

Collator.compare(String, String)

Constructor Detail

compact1, compact2, compact3

java.lang

Class StringBuffer

java.lang.Object
    java.lang.StringBuffer

All Implemented Interfaces:

Serializable, Appendable, CharSequence

```
public final class StringBuffer
extends Object
implements Serializable, CharSequence
```

A thread-safe, mutable sequence of characters. A string buffer is like a `String`, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

The principal operations on a `StringBuffer` are the `append` and `insert` methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string buffer. The `append` method always adds these characters at the end of the buffer; the `insert` method adds the characters at a specified point.

For example, if `z` refers to a string buffer object whose current contents are `"start"`, then the method call `z.append("le")` would cause the string buffer to contain `"startle"`, whereas `z.insert(4, "le")` would alter the string buffer to contain `"starlet"`.

In general, if sb refers to an instance of a `StringBuffer`, then `sb.append(x)` has the same effect as `sb.insert(sb.length(), x)`.

Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence), this class synchronizes only on the string buffer performing the operation, not on the source. Note that while `StringBuffer` is designed to be safe to use concurrently from multiple threads, if the constructor or the `append` or `insert` operation is passed a source sequence that is shared across threads, the calling code must ensure that the operation has a consistent and unchanging view of the source sequence for the duration of the operation. This could be satisfied by the caller holding a lock during the operation's call, by using an immutable source sequence, or by not sharing the source sequence across threads.

Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger.

Unless otherwise noted, passing a `null` argument to a constructor or method in this class will cause a `NullPointerException` to be thrown.

As of release JDK 5, this class has been supplemented with an equivalent class designed for use by a single thread, `StringBuilder`. The `StringBuilder` class should generally be used in preference to this one, as it supports all of the same operations but it is faster, as it performs no synchronization.

Since:

`JDK1.0`

See Also:

`StringBuilder`, `String`, `Serialized Form`

Constructor Summary

Constructors

Constructor and Description
`StringBuffer``()` Constructs a string buffer with no characters in it and an initial capacity of 16 characters.
`StringBuffer``(`**`CharSequence`** `seq)` Constructs a string buffer that contains the same characters as the specified `CharSequence`.
`StringBuffer``(int capacity)` Constructs a string buffer with no characters in it and the specified initial capacity.
`StringBuffer``(`**`String`** `str)` Constructs a string buffer initialized to the contents of the specified string.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`StringBuffer`	**`append`**`(boolean b)` Appends the string representation of the `boolean` argument to the sequence.
`StringBuffer`	**`append`**`(char c)` Appends the string representation of the `char` argument to this sequence.
`StringBuffer`	**`append`**`(char[] str)` Appends the string representation of the `char` array argument to this sequence.
`StringBuffer`	**`append`**`(char[] str, int offset, int len)` Appends the string representation of a subarray of the `char` array argument to this sequence.

array argument to this sequence.

StringBuffer	**append**(**CharSequence** s) Appends the specified CharSequence to this sequence.
StringBuffer	**append**(**CharSequence** s, int start, int end) Appends a subsequence of the specified CharSequence to this sequence.
StringBuffer	**append**(double d) Appends the string representation of the double argument to this sequence.
StringBuffer	**append**(float f) Appends the string representation of the float argument to this sequence.
StringBuffer	**append**(int i) Appends the string representation of the int argument to this sequence.
StringBuffer	**append**(long lng) Appends the string representation of the long argument to this sequence.
StringBuffer	**append**(**Object** obj) Appends the string representation of the Object argument.
StringBuffer	**append**(**String** str) Appends the specified string to this character sequence.
StringBuffer	**append**(**StringBuffer** sb) Appends the specified StringBuffer to this sequence.
StringBuffer	**appendCodePoint**(int codePoint) Appends the string representation of the codePoint argument to this sequence.
int	**capacity**() Returns the current capacity.
char	**charAt**(int index) Returns the char value in this sequence at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this sequence.
StringBuffer	**delete**(int start, int end) Removes the characters in a substring of this

	Removes the characters in a substring of this sequence.
StringBuffer	**deleteCharAt**(int index) Removes the char at the specified position in this sequence.
void	**ensureCapacity**(int minimumCapacity) Ensures that the capacity is at least equal to the specified minimum.
void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Characters are copied from this sequence into the destination character array dst.
int	**indexOf**(**String** str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(**String** str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
StringBuffer	**insert**(int offset, boolean b) Inserts the string representation of the boolean argument into this sequence.
StringBuffer	**insert**(int offset, char c) Inserts the string representation of the char argument into this sequence.
StringBuffer	**insert**(int offset, char[] str) Inserts the string representation of the char array argument into this sequence.
StringBuffer	**insert**(int index, char[] str, int offset, int len) Inserts the string representation of a subarray of the str array argument into this sequence.
StringBuffer	**insert**(int dstOffset, **CharSequence** s) Inserts the specified CharSequence into this sequence.
StringBuffer	**insert**(int dstOffset, **CharSequence** s, int start, int end) Inserts a subsequence of the specified CharSequence into this sequence.
StringBuffer	**insert**(int offset, double d) Inserts the string representation of the double argument into this sequence.
StringBuffer	**insert**(int offset, float f) Inserts the string representation of the float argument into this sequence.
StringBuffer	**insert**(int offset, int i) Inserts the string representation of the second int argument into this sequence.

into this sequence.

StringBuffer	**insert**(int offset, long l) Inserts the string representation of the long argument into this sequence.
StringBuffer	**insert**(int offset, **Object** obj) Inserts the string representation of the Object argument into this character sequence.
StringBuffer	**insert**(int offset, **String** str) Inserts the string into this character sequence.
int	**lastIndexOf**(**String** str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	**lastIndexOf**(**String** str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring.
int	**length**() Returns the length (character count).
int	**offsetByCodePoints**(int index, int codePointOffset) Returns the index within this sequence that is offset from the given index by codePointOffset code points.
StringBuffer	**replace**(int start, int end, **String** str) Replaces the characters in a substring of this sequence with characters in the specified String.
StringBuffer	**reverse**() Causes this character sequence to be replaced by the reverse of the sequence.
void	**setCharAt**(int index, char ch) The character at the specified index is set to ch.
void	**setLength**(int newLength) Sets the length of the character sequence.
CharSequence	**subSequence**(int start, int end) Returns a new character sequence that is a subsequence of this sequence.
String	**substring**(int start) Returns a new String that contains a subsequence of characters currently contained in this character sequence.
String	**substring**(int start, int end) Returns a new String that contains a subsequence of characters currently contained in this sequence.
String	**toString**() Returns a string representing the data in this sequence.
void	**trimToSize**()

Attempts to reduce storage used for the character sequence.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.lang.CharSequence

chars, codePoints

Constructor Detail

StringBuffer

```
public StringBuffer()
```

Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

StringBuffer

```
public StringBuffer(int capacity)
```

Constructs a string buffer with no characters in it and the specified initial capacity.

Parameters:

`capacity - the initial capacity.`

Throws:

`NegativeArraySizeException - if the capacity argument is less than 0.`

StringBuffer

```
public StringBuffer(String str)
```

Constructs a string buffer initialized to the contents of the specified string. The initial capacity of the string buffer is 16 plus the length of the string argument.

Parameters:

`str - the initial contents of the buffer.`

StringBuffer

```
public StringBuffer(CharSequence seq)
```

Constructs a string buffer that contains the same characters as the specified `CharSequence`. The initial capacity of the string buffer is 16 plus the length of the `CharSequence` argument.

compact1, compact2, compact3

java.lang

Class Math

java.lang.Object
    java.lang.Math

```
public final class Math
extends Object
```

The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Unlike some of the numeric methods of class `StrictMath`, all implementations of the equivalent functions of class `Math` are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.

By default many of the `Math` methods simply call the equivalent method in `StrictMath` for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide higher-performance implementations of `Math` methods. Such higher-performance implementations still must conform to the specification for `Math`.

The quality of implementation specifications concern two properties, accuracy of the returned result and monotonicity of the method. Accuracy of the floating-point `Math` methods is measured in terms of *ulps*, units in the last place. For a given floating-point format, an ulp of a specific real number value is the distance between the two floating-point values bracketing that numerical value. When discussing the accuracy of a method as a whole rather than at a specific argument, the number of ulps cited is for the worst-case error at any argument. If a method always has an error less than 0.5 ulps, the method always returns the floating-point number nearest the exact result; such a method is *correctly rounded*. A correctly rounded method is generally the best a floating-point approximation can be; however, it is impractical for many floating-point methods to be correctly rounded. Instead, for the `Math` class, a larger error bound of 1 or 2 ulps is allowed for certain methods. Informally, with a 1 ulp error bound, when the exact result is a representable number, the exact result should be returned as the computed result; otherwise, either of the two floating-point values which bracket the exact result may be returned. For exact results large in magnitude, one of the endpoints of the bracket may be infinite. Besides accuracy at individual arguments, maintaining proper relations between the method at different arguments is also important. Therefore, most methods with more than 0.5 ulp errors are required to be *semi-monotonic*: whenever the mathematical function is non-decreasing, so is the floating-point approximation, likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 ulp accuracy will automatically meet the monotonicity requirements.

The platform uses signed two's complement integer arithmetic with int and long primitive types. The developer should choose the primitive type to ensure that arithmetic operations consistently produce correct results, which in some cases means the operations will not

overflow the range of values of the computation. The best practice is to choose the primitive type and algorithm to avoid overflow. In cases where the size is `int` or `long` and overflow errors need to be detected, the methods `addExact`, `subtractExact`, `multiplyExact`, and `toIntExact` throw an `ArithmeticException` when the results overflow. For other arithmetic operations such as divide, absolute value, increment, decrement, and negation overflow occurs only with a specific minimum or maximum value and should be checked against the minimum or maximum as appropriate.

Since:

`JDK1.0`

Field Summary

Fields

Modifier and Type	Field and Description
`static double`	**`E`** The `double` value that is closer than any other to *e*, the base of the natural logarithms.
`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`static double`	**`abs`**`(double a)` Returns the absolute value of a `double` value.
`static float`	**`abs`**`(float a)` Returns the absolute value of a `float` value.
`static int`	**`abs`**`(int a)` Returns the absolute value of an `int` value.
`static long`	**`abs`**`(long a)` Returns the absolute value of a `long` value.
`static double`	**`acos`**`(double a)` Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.
`static int`	**`addExact`**`(int x, int y)` Returns the sum of its arguments, throwing an exception if the result overflows an `int`.
`static long`	**`addExact`**`(long x, long y)` Returns the sum of its arguments, throwing an exception if the result overflows a `long`.

static double	**asin**(double a) Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.
static double	**atan**(double a) Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.
static double	**atan2**(double y, double x) Returns the angle *theta* from the conversion of rectangular coordinates (x, y) to polar coordinates (r, *theta*).
static double	**cbrt**(double a) Returns the cube root of a double value.
static double	**ceil**(double a) Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.
static double	**copySign**(double magnitude, double sign) Returns the first floating-point argument with the sign of the second floating-point argument.
static float	**copySign**(float magnitude, float sign) Returns the first floating-point argument with the sign of the second floating-point argument.
static double	**cos**(double a) Returns the trigonometric cosine of an angle.
static double	**cosh**(double x) Returns the hyperbolic cosine of a double value.
static int	**decrementExact**(int a) Returns the argument decremented by one, throwing an exception if the result overflows an int.
static long	**decrementExact**(long a) Returns the argument decremented by one, throwing an exception if the result overflows a long.
static double	**exp**(double a) Returns Euler's number *e* raised to the power of a double value.
static double	**expm1**(double x) Returns e^x -1.
static double	**floor**(double a) Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
static int	**floorDiv**(int x, int y) Returns the largest (closest to positive infinity) int value that is less than or equal to the algebraic quotient.

static long	**floorDiv**(long x, long y) Returns the largest (closest to positive infinity) long value that is less than or equal to the algebraic quotient.
static int	**floorMod**(int x, int y) Returns the floor modulus of the int arguments.
static long	**floorMod**(long x, long y) Returns the floor modulus of the long arguments.
static int	**getExponent**(double d) Returns the unbiased exponent used in the representation of a double.
static int	**getExponent**(float f) Returns the unbiased exponent used in the representation of a float.
static double	**hypot**(double x, double y) Returns sqrt(x^2 +y^2) without intermediate overflow or underflow.
static double	**IEEEremainder**(double f1, double f2) Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
static int	**incrementExact**(int a) Returns the argument incremented by one, throwing an exception if the result overflows an int.
static long	**incrementExact**(long a) Returns the argument incremented by one, throwing an exception if the result overflows a long.
static double	**log**(double a) Returns the natural logarithm (base e) of a double value.
static double	**log10**(double a) Returns the base 10 logarithm of a double value.
static double	**log1p**(double x) Returns the natural logarithm of the sum of the argument and 1.
static double	**max**(double a, double b) Returns the greater of two double values.
static float	**max**(float a, float b) Returns the greater of two float values.
static int	**max**(int a, int b) Returns the greater of two int values.
static long	**max**(long a, long b) Returns the greater of two long values.
static double	**min**(double a, double b) Returns the smaller of two double values.

static float	**min**(float a, float b) Returns the smaller of two float values.
static int	**min**(int a, int b) Returns the smaller of two int values.
static long	**min**(long a, long b) Returns the smaller of two long values.
static int	**multiplyExact**(int x, int y) Returns the product of the arguments, throwing an exception if the result overflows an int.
static long	**multiplyExact**(long x, long y) Returns the product of the arguments, throwing an exception if the result overflows a long.
static int	**negateExact**(int a) Returns the negation of the argument, throwing an exception if the result overflows an int.
static long	**negateExact**(long a) Returns the negation of the argument, throwing an exception if the result overflows a long.
static double	**nextAfter**(double start, double direction) Returns the floating-point number adjacent to the first argument in the direction of the second argument.
static float	**nextAfter**(float start, double direction) Returns the floating-point number adjacent to the first argument in the direction of the second argument.
static double	**nextDown**(double d) Returns the floating-point value adjacent to d in the direction of negative infinity.
static float	**nextDown**(float f) Returns the floating-point value adjacent to f in the direction of negative infinity.
static double	**nextUp**(double d) Returns the floating-point value adjacent to d in the direction of positive infinity.
static float	**nextUp**(float f) Returns the floating-point value adjacent to f in the direction of positive infinity.
static double	**pow**(double a, double b) Returns the value of the first argument raised to the power of the second argument.
static double	**random**() Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

static double	**rint**(double a) Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
static long	**round**(double a) Returns the closest long to the argument, with ties rounding to positive infinity.
static int	**round**(float a) Returns the closest int to the argument, with ties rounding to positive infinity.
static double	**scalb**(double d, int scaleFactor) Returns $d \times 2^{scaleFactor}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the double value set.
static float	**scalb**(float f, int scaleFactor) Returns $f \times 2^{scaleFactor}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the float value set.
static double	**signum**(double d) Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.
static float	**signum**(float f) Returns the signum function of the argument; zero if the argument is zero, 1.0f if the argument is greater than zero, -1.0f if the argument is less than zero.
static double	**sin**(double a) Returns the trigonometric sine of an angle.
static double	**sinh**(double x) Returns the hyperbolic sine of a double value.
static double	**sqrt**(double a) Returns the correctly rounded positive square root of a double value.
static int	**subtractExact**(int x, int y) Returns the difference of the arguments, throwing an exception if the result overflows an int.
static long	**subtractExact**(long x, long y) Returns the difference of the arguments, throwing an exception if the result overflows a long.
static double	**tan**(double a) Returns the trigonometric tangent of an angle.
static double	**tanh**(double x) Returns the hyperbolic tangent of a double value.

static double	**toDegrees**(double angrad) Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
static int	**toIntExact**(long value) Returns the value of the long argument; throwing an exception if the value overflows an int.
static double	**toRadians**(double angdeg) Converts an angle measured in degrees to an approximately equivalent angle measured in radians.
static double	**ulp**(double d) Returns the size of an ulp of the argument.
static float	**ulp**(float f) Returns the size of an ulp of the argument.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

E

```
public static final double E
```

The double value that is closer than any other to *e*, the base of the natural logarithms.

See Also:

Constant Field Values

PI

```
public static final double PI
```

The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

See Also:

Constant Field Values

Method Detail

sin

```
public static double sin(double a)
```