Documentation de quelques classes conteneur de la librairie standard JAVA (Java Platform SE 8)

Département d'Informatique
Institut Galilée
Université Paris 13

3 octobre 2019

Table des matières

Chapitre 1

Interface Collection

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

Chapitre 2

Classe AbstractCollection

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()
```

Description copied from interface: `Collection`

Returns the number of elements in this collection. If this collection 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 collection`

isEmpty

```
public boolean isEmpty()
```

Returns `true` if this collection contains no elements.

This implementation returns `size() == 0`.

Specified by:

`isEmpty in interface Collection<E>`

Returns:

`true if this collection contains no elements`

contains

```
public 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))`.

This implementation iterates over the elements in the collection, checking each element in turn for equality with the specified element.

Specified by:

`contains in interface Collection<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)`

toArray

`public 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.

This implementation returns an array containing all the elements returned by this collection's iterator, in the same order, stored in consecutive elements of the array, starting with index `0`. The length of the returned array is equal to the number of elements returned by the iterator, even if the size of this collection changes during iteration, as might happen if the collection permits concurrent modification during iteration. The `size` method is called only as an optimization hint; the correct result is returned even if the iterator returns a different number of elements.

This method is equivalent to:

```
 List<E> list = new ArrayList<E>(size());
 for (E e : this)
     list.add(e);
 return list.toArray();
```

Specified by:

`toArray in interface Collection<E>`

Returns:

`an array containing all of the elements in this collection`

toArray

```
public <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 `Collection.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()`.

This implementation returns an array containing all the elements returned by this collection's iterator in the same order, stored in consecutive elements of the array, starting with index `0`. If the number of elements returned by the iterator is too large to fit into the specified array, then the elements are returned in a newly allocated array with length equal to the number of elements returned by the iterator, even if the size of this collection changes during iteration, as might happen if the collection permits concurrent modification during iteration. The `size` method is called only as an optimization hint; the correct result is returned even if the iterator returns a different number of elements.

This method is equivalent to:

```
 List<E> list = new ArrayList<E>(size());
 for (E e : this)
     list.add(e);
 return list.toArray(a);
```

Specified by:

`toArray` in interface `Collection<E>`

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

```
public 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.

This implementation always throws an `UnsupportedOperationException`.

Specified by:

`add` in interface `Collection<E>`

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

`public 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).

This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's remove method.

Note that this implementation throws an `UnsupportedOperationException` if the iterator returned by this collection's iterator method does not implement the `remove` method and this collection contains the specified object.

Specified by:

`remove in interface Collection<E>`

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:

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

`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)`

containsAll

`public boolean containsAll(Collection<?> c)`

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

This implementation iterates over the specified collection, checking each element returned by the iterator in turn to see if it's contained in this collection. If all elements are so contained `true` is returned, otherwise `false`.

Specified by:

`containsAll in interface Collection<E>`

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

```
public 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.)

This implementation iterates over the specified collection, and adds each object returned by the iterator to this collection, in turn.

Note that this implementation will throw an `UnsupportedOperationException` unless `add` is overridden (assuming the specified collection is non-empty).

Specified by:

`addAll in interface Collection<E>`

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

```
public 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.

This implementation iterates over this collection, checking each element returned by the iterator in turn to see if it's contained in the specified collection. If it's so contained, it's removed from this collection with the iterator's `remove` method.

Note that this implementation will throw an `UnsupportedOperationException` if the iterator returned by the `iterator` method does not implement the `remove` method and this collection contains one or more elements in common with the specified collection.

Specified by:

`removeAll` in interface `Collection<E>`

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)`

retainAll

```
public 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.

This implementation iterates over this collection, checking each element returned by the iterator in turn to see if it's contained in the specified collection. If it's not so contained, it's removed from this collection with the iterator's `remove` method.

Note that this implementation will throw an `UnsupportedOperationException` if the iterator returned by the `iterator` method does not implement the `remove` method and this collection contains one or more elements not present in the specified collection.

Specified by:

`retainAll` in interface `Collection<E>`

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

```
public void clear()
```

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

This implementation iterates over this collection, removing each element using the `Iterator.remove` operation. Most implementations will probably choose to override this method for efficiency.

Note that this implementation will throw an `UnsupportedOperationException` if the iterator returned by this collection's `iterator` method does not implement the `remove` method and this collection is non-empty.

Specified by:

`clear` in interface `Collection<E>`

Throws:

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

toString

```
public String toString()
```

Returns a string representation of this collection. The string representation consists of a list of the collection's elements in the order they are returned by its iterator, enclosed in square brackets (`"[]"`). Adjacent elements are separated by the characters `", "` (comma and space). Elements are converted to strings as by `String.valueOf(Object)`.

Overrides:

`toString` in class `Object`

Returns:

a string representation of this collection

Chapitre 3

Interface List

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()`

Returns `true` if this list contains no elements.

Specified by:

`isEmpty` in interface `Collection<E>`

Returns:

`true` if this list contains no elements

contains

`boolean contains(Object o)`

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

Specified by:

`contains` in interface `Collection<E>`

Parameters:

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

Returns:

`true` if this list contains the specified element

Throws:

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

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

iterator

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

Returns an iterator over the elements in this list in proper sequence.

Specified by:

`iterator in interface Collection<E>`

Specified by:

`iterator in interface Iterable<E>`

Returns:

`an iterator over the elements in this list in proper sequence`

toArray

```
Object[] toArray()
```

Returns an array containing all of the elements in this list in proper sequence (from first to last element).

The returned array will be "safe" in that no references to it are maintained by this list. (In other words, this method must allocate a new array even if this list 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.

Specified by:

`toArray in interface Collection<E>`

Returns:

`an array containing all of the elements in this list in proper sequence`

See Also:

`Arrays.asList(Object[])`

toArray

```
<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. If the list 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 list.

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

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 list known to contain only strings. The following code can be used to dump the list 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()`.

Specified by:

`toArray in interface Collection<E>`

Type Parameters:

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

Parameters:

`a - the array into which the elements of this list 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 this list`

Throws:

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

`NullPointerException - if the specified array is null`

add

`boolean add(E e)`

Appends the specified element to the end of this list (optional operation).

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

Specified by:

`add in interface Collection<E>`

Parameters:

`e - element to be appended to this list`

Returns:

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

Throws:

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

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

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

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

remove

```
boolean remove(Object o)
```

Removes the first occurrence of the specified element from this list, if it is present (optional operation). If this list 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). Returns `true` if this list contained the specified element (or equivalently, if this list changed as a result of the call).

Specified by:

`remove in interface Collection<E>`

Parameters:

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

Returns:

`true if this list contained the specified element`

Throws:

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

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

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

containsAll

```
boolean containsAll(Collection<?> c)
```

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

Specified by:

`containsAll in interface Collection<E>`

Parameters:

`c - collection to be checked for containment in this list`

Returns:

`true if this list contains all of the elements of the specified collection`

Throws:

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

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

See Also:

`contains(Object)`

addAll

```
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). The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (Note that this will occur if the specified collection is this list, and it's nonempty.)

Specified by:

`addAll in interface Collection<E>`

Parameters:

`c - collection containing elements to be added to this list`

Returns:

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

Throws:

`UnsupportedOperationException - if the addAll operation is not supported by this list`

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

`NullPointerException - if the specified collection contains one or more null elements and this list 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 list

See Also:

`add(Object)`

addAll

```
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). 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 this list 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. (Note that this will occur if the specified collection is this list, and it's nonempty.)

Parameters:

`index` - index at which to insert the first element from the specified collection

`c` - collection containing elements to be added to this list

Returns:

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

Throws:

`UnsupportedOperationException` - if the `addAll` operation is not supported by this list

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

`NullPointerException` - if the specified collection contains one or more null elements and this list 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 list

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index > size()`)

removeAll

```
boolean removeAll(Collection<?> c)
```

Removes from this list all of its elements that are contained in the specified collection (optional operation).

Specified by:

`removeAll` in interface `Collection<E>`

Parameters:

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

Returns:

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

Throws:

`UnsupportedOperationException` - if the `removeAll` operation is not supported by this list

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

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

See Also:

`remove(Object)`, `contains(Object)`

retainAll

```
boolean retainAll(Collection<?> c)
```

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

Specified by:

`retainAll` in interface `Collection<E>`

Parameters:

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

Returns:

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

Throws:

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

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

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

See Also:

`remove(Object)`, `contains(Object)`

replaceAll

`default void replaceAll(UnaryOperator<E> operator)`

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.

Implementation Requirements:

The default implementation is equivalent to, for this list:

```
    final ListIterator<E> li = list.listIterator();
    while (li.hasNext()) {
        li.set(operator.apply(li.next()));
    }
```

If the list's list-iterator does not support the set operation then an UnsupportedOperationException will be thrown when replacing the first element.

Parameters:

operator - the operator to apply to each element

Throws:

UnsupportedOperationException - if this list is unmodifiable. Implementations may throw this exception if an element cannot be replaced or if, in general, modification is not supported

NullPointerException - if the specified operator is null or if the operator result is a null value and this list does not permit null elements (optional)

Since:

1.8

sort

`default void sort(Comparator<? super E> c)`

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.

Implementation Requirements:

The default implementation obtains an array containing all elements in this list, sorts the array, and iterates over this list resetting each element from the corresponding position in the array. (This avoids the n^2 log(n) performance that would result from attempting to sort a linked list in place.)

Implementation Note:

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

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

The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Parameters:

c - the Comparator used to compare list elements. A null value indicates that the elements' natural ordering should be used

Throws:

ClassCastException - if the list contains elements that are not *mutually comparable* using the specified comparator

UnsupportedOperationException - if the list's list-iterator does not support the set operation

IllegalArgumentException - (optional) if the comparator is found to violate the Comparator contract

Since:

1.8

clear

```
void clear()
```

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

Specified by:

`clear in interface Collection<E>`

Throws:

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

equals

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

Compares the specified object with this list 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. This definition ensures that the equals method works properly across different implementations of the `List` interface.

Specified by:

`equals in interface Collection<E>`

Overrides:

`equals in class Object`

Parameters:

`o - the object to be compared for equality with this list`

Returns:

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

See Also:

`Object.hashCode(), HashMap`

hashCode

```
int hashCode()
```

Returns the hash code value for this list. The hash code of a list is defined to be the result of the following calculation:

```
     int hashCode = 1;
     for (E e : list)
         hashCode = 31*hashCode + (e==null ? 0 : e.hashCode());
```

This ensures that `list1.equals(list2)` implies that `list1.hashCode()==list2.hashCode()` for any two lists, `list1` and `list2`, as required by the general contract of `Object.hashCode()`.

Specified by:

`hashCode in interface Collection<E>`

Overrides:

`hashCode in class Object`

Returns:

`the hash code value for this list`

See Also:

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

get

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

Returns the element at the specified position in this list.

Parameters:

`index - index of the element to return`

Returns:

`the element at the specified position in this list`

Throws:

`IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

set

```
E set(int index,
      E element)
```

Replaces the element at the specified position in this list with the specified element (optional operation).

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:

`UnsupportedOperationException - if the set operation is not supported by this list`

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

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

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

`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index >= size())

add

```
void add(int index,
         E element)
```

Inserts the specified element at the specified position in this list (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

Parameters:

`index` - index at which the specified element is to be inserted

`element` - element to be inserted

Throws:

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

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

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

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

`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index > size())

remove

```
E remove(int index)
```

Removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.

Parameters:

`index` - the index of the element to be removed

Returns:

the element previously at the specified position

Throws:

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

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

indexOf

```
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. 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.

Parameters:

`o` - element to search for

Returns:

the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element

Throws:

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

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

lastIndexOf

```
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. 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.

Parameters:

`o` - element to search for

Returns:

the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element

Throws:

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

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

listIterator

```
ListIterator<E> listIterator()
```

Returns a list iterator over the elements in this list (in proper sequence).

Returns:

a list iterator over the elements in this list (in proper sequence)

listIterator

```
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.

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()`)

subList

```
List<E> subList(int fromIndex,
                int toIndex)
```

Returns a view of the portion of this list between the specified `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 non-structural 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 passing 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 this list, or otherwise perturb it in such a fashion that iterations in progress may yield incorrect results.)

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 - for an illegal endpoint index value (fromIndex < 0 || toIndex > size || fromIndex > toIndex)`

spliterator

`default Spliterator<E> spliterator()`

Creates a `Spliterator` over the elements in this list.

The `Spliterator` reports `Spliterator.SIZED` and `Spliterator.ORDERED`. Implementations should document the reporting of additional characteristic values.

Specified by:

`spliterator in interface Collection<E>`

Specified by:

`spliterator in interface Iterable<E>`

Implementation Requirements:

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

Implementation Note:

The created Spliterator additionally reports Spliterator.SUBSIZED.

Returns:

a Spliterator over the elements in this list

Since:

1.8

Chapitre 4

Classe ArrayList

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.

Parameters:

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

Throws:

`NullPointerException` - if the specified collection is null

Method Detail

trimToSize

```
public void trimToSize()
```

Trims the capacity of this `ArrayList` instance to be the list's current size. An application can use this operation to minimize the storage of an `ArrayList` instance.

ensureCapacity

```
public 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.

Parameters:

`minCapacity - the desired minimum capacity`

size

```
public int size()
```

Returns the number of elements in this list.

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 elements in this list`

isEmpty

```
public boolean isEmpty()
```

Returns `true` if this list contains no elements.

Specified by:

`isEmpty in interface Collection<E>`

Specified by:

`isEmpty in interface List<E>`

Overrides:

`isEmpty in class AbstractCollection<E>`

Returns:

`true if this list contains no elements`

contains

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

Returns `true` if this list contains the specified element. More formally, returns `true` if and only if this list 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 list is to be tested

Returns:

`true` if this list contains the specified element

indexOf

```
public 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. 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 list, or -1 if this list does not contain the element

lastIndexOf

```
public 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. 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 `list, or -1 if this list does not contain the element`

clone

```
public Object clone()
```

Returns a shallow copy of this `ArrayList` instance. (The elements themselves are not copied.)

Overrides:

`clone in class Object`

Returns:

`a clone of this ArrayList instance`

See Also:

`Cloneable`

toArray

```
public Object[] toArray()
```

Returns an array containing all of the elements in this list in proper sequence (from first to last element).

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

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

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 list in proper sequence`

See Also:

`Arrays.asList(Object[])`

toArray

```
public <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. If the list 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 list.

If the list fits in the specified array with room to spare (i.e., the array has more elements than the list), the element in the array immediately following the end of the collection is set to `null`. (This is useful in determining the length of the list *only* if the caller knows that the list 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 list 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 list`

Throws:

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

`NullPointerException - if the specified array is null`

get

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

Returns the element at the specified position in this list.

Specified by:

`get in interface List<E>`

Specified by:

`get in class AbstractList<E>`

Parameters:

`index - index of the element to return`

Returns:

`the element at the specified position in this list`

Throws:

`IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

set

```
public E set(int index,
             E element)
```

Replaces the element at the specified position in this list 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:

`IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

add

```
public boolean add(E e)
```

Appends the specified element to the end of this list.

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 list`

Returns:

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

add

```
public void add(int index,
                E element)
```

Inserts the specified element at the specified position in this list. 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:

`IndexOutOfBoundsException - if the index is out of range (index < 0 || index > size())`

remove

```
public E remove(int index)
```

Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices).

Specified by:

`remove in interface List<E>`

Overrides:

`remove in class AbstractList<E>`

Parameters:

`index - the index of the element to be removed`

Returns:

`the element that was removed from the list`

Throws:

`IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

remove

```
public boolean remove(Object o)
```

Removes the first occurrence of the specified element from this list, if it is present. If the list 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). Returns `true` if this list contained the specified element (or equivalently, if this list changed as a result of the call).

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 list, if present`

Returns:

`true if this list contained the specified element`

clear

```
public void clear()
```

Removes all of the elements from this list. The list will be empty after this call returns.

Specified by:

`clear in interface Collection<E>`

Specified by:

`clear` in interface `List<E>`

Overrides:

`clear` in class `AbstractList<E>`

addAll

```
public 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. 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 list, and this list is nonempty.)

Specified by:

`addAll` in interface `Collection<E>`

Specified by:

`addAll` in interface `List<E>`

Overrides:

`addAll` in class `AbstractCollection<E>`

Parameters:

`c` - collection containing elements to be added to this list

Returns:

true if this list changed as a result of the call

Throws:

`NullPointerException` - if the specified collection is null

See Also:

`AbstractCollection.add(Object)`

addAll

```
public 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. 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 list 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` - collection containing elements to be added to this list

Returns:

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

Throws:

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index > size()`)

`NullPointerException` - if the specified collection is null

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

Throws:

`IndexOutOfBoundsException` - if `fromIndex` or `toIndex` is out of range (`fromIndex < 0 || fromIndex >= size() || toIndex > size() || toIndex < fromIndex`)

removeAll

```
public boolean removeAll(Collection<?> c)
```

Removes from this list 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` - collection containing elements to be removed from this list

Returns:

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

Throws:

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

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

See Also:

`Collection.contains(Object)`

retainAll

```
public boolean retainAll(Collection<?> c)
```

Retains only the elements in this list that are contained in the specified collection. In other words, removes from this list 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` - collection containing elements to be retained in this list

Returns:

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

Throws:

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

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

See Also:

`Collection.contains(Object)`

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`

subList

```
public List<E> subList(int fromIndex,
                       int toIndex)
```

Returns a view of the portion of this list between the specified `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 non-structural changes in the returned list are reflected in this list, and vice-versa. The returned list supports all of the optional list operations.

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 passing 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(Object)` and `lastIndexOf(Object)`, 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 this 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)`

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`

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`

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

Chapitre 5

Interface Set

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).
`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

```
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>`

Parameters:

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

Returns:

`true if this set contains the specified element`

Throws:

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

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

iterator

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

Returns an iterator over the elements in this set. The elements are returned in no particular order (unless this set is an instance of some class that provides a guarantee).

Specified by:

`iterator in interface Collection<E>`

Specified by:

`iterator in interface Iterable<E>`

Returns:

`an iterator over the elements in this set`

toArray

```
Object[] toArray()
```

Returns an array containing all of the elements in this set. If this set 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 set. (In other words, this method must allocate a new array even if this set 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.

Specified by:

`toArray` in interface `Collection<E>`

Returns:

an array containing all the elements in this set

toArray

```
<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. If the set 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 set.

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

If this set 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 set known to contain only strings. The following code can be used to dump the set 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()`.

Specified by:

`toArray` in interface `Collection<E>`

Type Parameters:

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

Parameters:

`a` - the array into which the elements of this set 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 the elements in this set

Throws:

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

`NullPointerException` - if the specified array is null

add

```
boolean add(E e)
```

Adds the specified element to this set if it is not already present (optional operation). 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`. In combination with the restriction on constructors, this ensures that sets never contain duplicate elements.

The stipulation above does not imply that sets must accept all elements; sets may refuse to add any particular element, including `null`, and throw an exception, as described in the specification for `Collection.add`. Individual set implementations should clearly document any restrictions on the elements that they may contain.

Specified by:

`add` in interface `Collection<E>`

Parameters:

`e` - element to be added to this set

Returns:

true if this set did not already contain the specified element

Throws:

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

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

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

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

remove

```
boolean remove(Object o)
```

Removes the specified element from this set if it is present (optional operation). 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>`

Parameters:

`o` - object to be removed from this set, if present

Returns:

true if this set contained the specified element

Throws:

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

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

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

containsAll

```
boolean containsAll(Collection<?> c)
```

Returns `true` if this set contains all of the elements of the specified collection. If the specified collection is also a set, this method returns `true` if it is a *subset* of this set.

Specified by:

`containsAll` in interface `Collection<E>`

Parameters:

`c` - collection to be checked for containment in this set

Returns:

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

Throws:

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

`NullPointerException` - if the specified collection contains one or more null elements and this set 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 set if they're not already present (optional operation). If the specified collection is also a set, the `addAll` operation effectively modifies this set so that its value is the *union* of the two sets. The behavior of this operation is undefined if the specified collection is modified while the operation is in progress.

Specified by:

`addAll in interface Collection<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:

`UnsupportedOperationException - if the addAll operation is not supported by this set`

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

`NullPointerException - if the specified collection contains one or more null elements and this set 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 set`

See Also:

`add(Object)`

retainAll

`boolean retainAll(Collection<?> c)`

Retains only the elements in this set that are contained in the specified collection (optional operation). In other words, removes from this set all of its elements that are not contained in the specified collection. If the specified collection is also a set, this operation effectively modifies this set so that its value is the *intersection* of the two sets.

Specified by:

`retainAll in interface Collection<E>`

Parameters:

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

Returns:

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

Throws:

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

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

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

See Also:

`remove(Object)`

removeAll

```
boolean removeAll(Collection<?> c)
```

Removes from this set all of its elements that are contained in the specified collection (optional operation). If the specified collection is also a set, this operation effectively modifies this set so that its value is the *asymmetric set difference* of the two sets.

Specified by:

`removeAll in interface Collection<E>`

Parameters:

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

Returns:

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

Throws:

`UnsupportedOperationException - if the removeAll operation is not supported by this set`

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

`NullPointerException - if this set contains a null element 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 set (optional operation). The set will be empty after this call returns.

Specified by:

`clear` in interface `Collection<E>`

Throws:

`UnsupportedOperationException` - if the `clear` method is not supported by this set

equals

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

Compares the specified object with this set for equality. Returns `true` if the specified object is also a set, the two sets have the same size, and every member of the specified set is contained in this set (or equivalently, every member of this set is contained in the specified set). This definition ensures that the equals method works properly across different implementations of the set interface.

Specified by:

`equals` in interface `Collection<E>`

Overrides:

`equals` in class `Object`

Parameters:

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

Returns:

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

See Also:

`Object.hashCode()`, `HashMap`

hashCode

```
int hashCode()
```

Returns the hash code value for this set. The hash code of a set is defined to be the sum of the hash codes of the elements in the set, where the hash code of a `null` element is defined to be zero. This ensures that `s1.equals(s2)` implies that `s1.hashCode()==s2.hashCode()` for any two sets `s1` and `s2`, as required by the general contract of `Object.hashCode()`.

Specified by:

`hashCode` in interface `Collection<E>`

Overrides:

`hashCode` in class `Object`

Returns:

the hash code value for this set

See Also:

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

spliterator

```
default Spliterator<E> spliterator()
```

Creates a `Spliterator` over the elements in this set.

The `Spliterator` reports `Spliterator.DISTINCT`. Implementations should document the reporting of additional characteristic values.

Specified by:

`spliterator` in interface `Collection<E>`

Specified by:

`spliterator` in interface `Iterable<E>`

Implementation Requirements:

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

The created Spliterator additionally reports `Spliterator.SIZED`.

Implementation Note:

The created Spliterator additionally reports `Spliterator.SUBSIZED`.

Returns:

a Spliterator over the elements in this set

Since:

1.8

Chapitre 6

Classe HashSet

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()
```

Returns an iterator over the elements in this set. The elements are returned in no particular order.

Specified by:

`iterator in interface Iterable<E>`

Specified by:

`iterator in interface Collection<E>`

Specified by:

`iterator in interface Set<E>`

Specified by:

`iterator in class AbstractCollection<E>`

Returns:

`an Iterator over the elements in this set`

See Also:

`ConcurrentModificationException`

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 - element whose presence in this set is to be tested`

Returns:

`true if this set contains the specified element`

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 this 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`

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 the set contained the specified element`

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>`

clone

```
public Object clone()
```

Returns a shallow copy of this `HashSet` 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` and `Spliterator.DISTINCT`. 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 Set<E>`

Returns:

`a Spliterator over the elements in this set`

Since:

`1.8`