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).
`boolean`	`retainAll(Collection<?> c)` Retains only the elements in this list that are contained in the specified collection (optional operation).

	specified collection (optional operation).
E	**set**(int index, **E** element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.
List<**E**>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (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