java.util

Interface List

All Superinterfaces:
Collection

All Known Implementing Classes:
AbstractList, ArrayList, LinkedList, Vector

---

public interface **List**
extends Collection

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 a 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

void	**add**(int index, Object element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**add**(Object o) Appends the specified element to the end of this list (optional operation).
boolean	**addAll**(Collection 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 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.
Object	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if

	this list does not contain this element.
boolean	**isEmpty**() Returns `true` if this list contains no elements.
Iterator	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element.
ListIterator	**listIterator**() Returns a list iterator of the elements in this list (in proper sequence).
ListIterator	**listIterator**(int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.
Object	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence in this list of the specified element (optional operation).
boolean	**removeAll**(Collection c) Removes from this list all the elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection c) Retains only the elements in this list that are contained in the specified collection (optional operation).
Object	**set**(int index, Object 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	**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.
Object[]	**toArray**(Object[] a) Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

Method Detail

size

```
public 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`

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`

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`

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 support null elements (optional).

---

iterator

```
public Iterator iterator()
```

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

Specified by:

iterator in interface Collection

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

---

toArray

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

Returns an array containing all of the elements in this list in proper sequence. Obeys the general contract of the `Collection.toArray` method.

Specified by:
toArray in interface Collection

Returns:
an array containing all of the elements in this list in proper sequence.

See Also:
Arrays.asList(Object[])

---

toArray

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

Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array. Obeys the general contract of the `Collection.toArray(Object[])` method.

Specified by:
toArray in interface 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

```
public boolean add(Object o)
```

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`

Parameters:
`o` - element to be appended to this list.
Returns:
`true` (as per the general contract of the `Collection.add` method).
Throws:
`UnsupportedOperationException` - if the `add` method 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 support null elements.
`IllegalArgumentException` - if some aspect of this element prevents it from being added to this list.

---

remove

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

Removes the first occurrence in this list of the specified element (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).

Specified by:
`remove` in interface `Collection`

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 support null elements (optional).
`UnsupportedOperationException` - if the `remove` method is not supported by this list.

---

containsAll

```
public boolean containsAll(Collection c)
```

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

Specified by:
`containsAll` in interface `Collection`

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 support null elements (optional).

`NullPointerException` - if the specified collection is `null`.

See Also:

`contains(Object)`

---

addAll

```
public boolean addAll(Collection 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 unspecified 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`

Parameters:

`c` - collection whose elements are to be added to this list.

Returns:

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

Throws:

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

`ClassCastException` - if the class of an element in 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 support null elements, or if the specified collection is `null`.

`IllegalArgumentException` - if some aspect of an element in the specified collection prevents it from being added to this list.

See Also:

`add(Object)`

---

addAll

```
public boolean addAll(int index,
                      Collection 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 unspecified 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 first element from the specified collection.
`c` - elements to be inserted into this list.

Returns:

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

Throws:

`UnsupportedOperationException` - if the `addAll` method is not supported by this list.
`ClassCastException` - if the class of one of elements 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 support null elements, or if the specified collection is `null`.
`IllegalArgumentException` - if some aspect of one of elements 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

```
public boolean removeAll(Collection c)
```

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

Specified by:

`removeAll` in interface `Collection`

Parameters:

`c` - collection that defines which elements will be removed from this list.

Returns:

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

Throws:

`UnsupportedOperationException` - if the `removeAll` method is not supported by this list.
`ClassCastException` - if the types of one or more elements in this list are incompatible with the specified collection (optional).
`NullPointerException` - if this list contains one or more null elements and the specified collection does not support null elements (optional).
`NullPointerException` - if the specified collection is `null`.

See Also:

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

---

retainAll

```
public 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 the elements that are not contained in the specified collection.

Specified by:
`retainAll` in interface `Collection`

Parameters:
`c` - collection that defines which elements this set will retain.

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

Throws:
`UnsupportedOperationException` - if the `retainAll` method is not supported by this list.
`ClassCastException` - if the types of one or more elements in this list are incompatible with the specified collection (optional).
`NullPointerException` - if this list contains one or more null elements and the specified collection does not support null elements (optional).
`NullPointerException` - if the specified collection is `null`.

See Also:
`remove(Object)`, `contains(Object)`

---

clear

```
public void clear()
```

Removes all of the elements from this list (optional operation). This list will be empty after this call returns (unless it throws an exception).

Specified by:
`clear` in interface `Collection`

Throws:
`UnsupportedOperationException` - if the `clear` method is not supported by this list.

---

equals

```
public 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`

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()`, `Hashtable`

---

hashCode

```
public 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:

```
hashCode = 1;
Iterator i = list.iterator();
while (i.hasNext()) {
    Object obj = i.next();
    hashCode = 31*hashCode + (obj==null ? 0 : obj.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`

Overrides:

`hashCode` in class `Object`

Returns:

the hash code value for this list.

See Also:

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

---

get

```
public Object get(int index)
```

Returns the element at the specified position in this list.

Parameters:

`index` - index of 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 Object set(int index,
                  Object element)
```

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

Parameters:

`index` - index of 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` method 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 support null elements.

`IllegalArgumentException` - if some aspect 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

```
public void add(int index,
                Object 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` method 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 support null elements.

`IllegalArgumentException` - if some aspect 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

```
public Object 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 removed.

Returns:

the element previously at the specified position.

Throws:

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

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

---

indexOf

```
public int indexOf(Object o)
```

Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element. 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 in this list of the first occurrence of the specified element, or -1 if this list does not contain this 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 support null elements (optional).

---

lastIndexOf

```
public int lastIndexOf(Object o)
```

Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element. 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 in this list of the last occurrence of the specified element, or -1 if this list does not contain this 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 support null elements (optional).

---

listIterator

```
public ListIterator listIterator()
```

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

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

---

listIterator

```
public ListIterator listIterator(int index)
```

Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list. The specified index indicates the first element that would be returned by an initial call to the `next` method. An initial call to the `previous` method would return the element with the specified index minus one.

Parameters:
`index` - index of first element to be returned from the list iterator (by a call to the `next` method).
Returns:
a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.
Throws:
`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index > size()).

---

subList

```
public List 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).
