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