java.util

Class Vector<E>

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

All Implemented Interfaces:

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

Direct Known Subclasses:

Stack

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

The `Vector` class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a `Vector` can grow or shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Each vector tries to optimize storage management by maintaining a `capacity` and a `capacityIncrement`. The `capacity` is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of `capacityIncrement`. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the vector is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. The `Enumerations` returned by the `elements` method are *not* fail-fast.

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

As of the Java 2 platform v1.2, this class was retrofitted to implement the `List` interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, `Vector` is synchronized. If a thread-safe implementation is not needed, it is recommended to use `ArrayList` in place of `Vector`.

Since:

JDK1.0

See Also:

Collection, LinkedList, Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
protected int	**capacityIncrement**

	The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.
protected int	**elementCount** The number of valid components in this `Vector` object.
protected **Object**[]	**elementData** The array buffer into which the components of the vector are stored.

Fields inherited from class java.util.**AbstractList**

modCount

Constructor Summary

Constructors

Constructor and Description
Vector() Constructs an empty vector so that its internal data array has size `10` and its standard capacity increment is zero.
Vector(**Collection**<? extends **E**> c) Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
Vector(int initialCapacity) Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
Vector(int initialCapacity, int capacityIncrement) Constructs an empty vector with the specified initial capacity and capacity increment.

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) Appends the specified element to the end of this Vector.
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this Vector.
boolean	**addAll**(**Collection**<? extends **E**> c) Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified Collection into this Vector at the specified position.
void	**addElement**(**E** obj) Adds the specified component to the end of this vector, increasing its size by one.
int	**capacity**() Returns the current capacity of this vector.
void	**clear**() Removes all of the elements from this Vector.
Object	**clone**() Returns a clone of this vector.
boolean	**contains**(**Object** o)

	Returns true if this vector contains the specified element.
boolean	**containsAll(Collection<?> c)** Returns true if this Vector contains all of the elements in the specified Collection.
void	**copyInto(Object[] anArray)** Copies the components of this vector into the specified array.
E	**elementAt(int index)** Returns the component at the specified index.
Enumeration<E>	**elements()** Returns an enumeration of the components of this vector.
void	**ensureCapacity(int minCapacity)** Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
boolean	**equals(Object o)** Compares the specified Object with this Vector for equality.
E	**firstElement()** Returns the first component (the item at index 0) of this vector.
E	**get(int index)** Returns the element at the specified position in this Vector.
int	**hashCode()** Returns the hash code value for this Vector.
int	**indexOf(Object o)** Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
int	**indexOf(Object o, int index)** Returns the index of the first occurrence of the specified element in this vector, searching forwards from index, or returns -1 if the element is not found.
void	**insertElementAt(E obj, int index)** Inserts the specified object as a component in this vector at the specified index.
boolean	**isEmpty()** Tests if this vector has no components.
Iterator<E>	**iterator()** Returns an iterator over the elements in this list in proper sequence.
E	**lastElement()** Returns the last component of the vector.
int	**lastIndexOf(Object o)** Returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
int	**lastIndexOf(Object o, int index)** Returns the index of the last occurrence of the specified element in this vector, searching backwards from index, or returns -1 if the element is not found.
ListIterator<E>	**listIterator()** Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator(int index)** Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove(int index)** Removes the element at the specified position in this Vector.
boolean	**remove(Object o)** Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
boolean	**removeAll(Collection<?> c)** Removes from this Vector all of its elements that are contained in the specified Collection.
void	**removeAllElements()** Removes all components from this vector and sets its size to zero.

boolean	**removeElement**(**Object** obj)
	Removes the first (lowest-indexed) occurrence of the argument from this vector.
void	**removeElementAt**(int index)
	Deletes the component at the specified index.
protected void	**removeRange**(int fromIndex, int toIndex)
	Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive.
boolean	**retainAll**(**Collection**<?> c)
	Retains only the elements in this Vector that are contained in the specified Collection.
E	**set**(int index, **E** element)
	Replaces the element at the specified position in this Vector with the specified element.
void	**setElementAt**(**E** obj, int index)
	Sets the component at the specified `index` of this vector to be the specified object.
void	**setSize**(int newSize)
	Sets the size of this vector.
int	**size**()
	Returns the number of components in this vector.
List<E>	**subList**(int fromIndex, int toIndex)
	Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**()
	Returns an array containing all of the elements in this Vector in the correct order.
<T> T[]	**toArray**(T[] a)
	Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
String	**toString**()
	Returns a string representation of this Vector, containing the String representation of each element.
void	**trimToSize**()
	Trims the capacity of this vector to be the vector's current size.

Methods inherited from class java.lang.Object

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

Field Detail

elementData

protected Object[] elementData

The array buffer into which the components of the vector are stored. The capacity of the vector is the length of this array buffer, and is at least large enough to contain all the vector's elements.

Any array elements following the last element in the Vector are null.

elementCount

protected int elementCount

The number of valid components in this `Vector` object. Components `elementData[0]` through `elementData[elementCount-1]` are the actual items.

capacityIncrement

```
protected int capacityIncrement
```

The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity. If the capacity increment is less than or equal to zero, the capacity of the vector is doubled each time it needs to grow.

Constructor Detail

Vector

```
public Vector(int initialCapacity,
        int capacityIncrement)
```

Constructs an empty vector with the specified initial capacity and capacity increment.

Parameters:

   initialCapacity - the initial capacity of the vector

   capacityIncrement - the amount by which the capacity is increased when the vector overflows

Throws:

   IllegalArgumentException - if the specified initial capacity is negative

Vector

```
public Vector(int initialCapacity)
```

Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.

Parameters:

   initialCapacity - the initial capacity of the vector

Throws:

   IllegalArgumentException - if the specified initial capacity is negative

Vector

```
public Vector()
```

Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.

Vector

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

Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Parameters:

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

Throws:

>   NullPointerException - if the specified collection is null

Since:

>   1.2

Method Detail

copyInto

```
public void copyInto(Object[] anArray)
```

Copies the components of this vector into the specified array. The item at index k in this vector is copied into component k of anArray.

Parameters:

>   anArray - the array into which the components get copied

Throws:

>   NullPointerException - if the given array is null

>   IndexOutOfBoundsException - if the specified array is not large enough to hold all the components of this vector

>   ArrayStoreException - if a component of this vector is not of a runtime type that can be stored in the specified array

See Also:

>   toArray(Object[])

---

trimToSize

```
public void trimToSize()
```

Trims the capacity of this vector to be the vector's current size. If the capacity of this vector is larger than its current size, then the capacity is changed to equal the size by replacing its internal data array, kept in the field elementData, with a smaller one. An application can use this operation to minimize the storage of a vector.

---

ensureCapacity

```
public void ensureCapacity(int minCapacity)
```

Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.

If the current capacity of this vector is less than minCapacity, then its capacity is increased by replacing its internal data array, kept in the field elementData, with a larger one. The size of the new data array will be the old size plus capacityIncrement, unless the value of capacityIncrement is less than or equal to zero, in which case the new capacity will be twice the old capacity; but if this new size is still smaller than minCapacity, then the new capacity will be minCapacity.

Parameters:

>   minCapacity - the desired minimum capacity

---

setSize

```
public void setSize(int newSize)
```

Sets the size of this vector. If the new size is greater than the current size, new `null` items are added to the end of the vector. If the new size is less than the current size, all components at index `newSize` and greater are discarded.

Parameters:

>   `newSize` - the new size of this vector

Throws:

>   `ArrayIndexOutOfBoundsException` - if the new size is negative

capacity

```
public int capacity()
```

Returns the current capacity of this vector.

Returns:

>   the current capacity (the length of its internal data array, kept in the field `elementData` of this vector)

size

```
public int size()
```

Returns the number of components in this vector.

Specified by:

>   `size` in interface `Collection<E>`

Specified by:

>   `size` in interface `List<E>`

Specified by:

>   `size` in class `AbstractCollection<E>`

Returns:

>   the number of components in this vector

isEmpty

```
public boolean isEmpty()
```

Tests if this vector has no components.

Specified by:

>   `isEmpty` in interface `Collection<E>`

Specified by:

>   `isEmpty` in interface `List<E>`

Overrides:

>   `isEmpty` in class `AbstractCollection<E>`

Returns:

>   `true` if and only if this vector has no components, that is, its size is zero; `false` otherwise.

elements

```
public Enumeration<E> elements()
```

Returns an enumeration of the components of this vector. The returned `Enumeration` object will generate all items in this vector. The first item generated is the item at index 0, then the item at index 1, and so on.

Returns:

　　an enumeration of the components of this vector

See Also:

　　Iterator

contains

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

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

Specified by:

　　contains in interface Collection<E>

Specified by:

　　contains in interface List<E>

Overrides:

　　contains in class AbstractCollection<E>

Parameters:

　　o - element whose presence in this vector is to be tested

Returns:

　　true if this vector contains the specified element

indexOf

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

Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element. More formally, returns the lowest index i such that (o==null ? get(i)==null : o.equals(get(i))), or -1 if there is no such index.

Specified by:

　　indexOf in interface List<E>

Overrides:

　　indexOf in class AbstractList<E>

Parameters:

　　o - element to search for

Returns:

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

indexOf

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

Returns the index of the first occurrence of the specified element in this vector, searching forwards from `index`, or returns -1 if the element is not found. More formally, returns the lowest index i such that
`(i >= index && (o==null ? get(i)==null : o.equals(get(i))))`, or -1 if there is no such index.

Parameters:

    o - element to search for

    `index` - index to start searching from

Returns:

    the index of the first occurrence of the element in this vector at position `index` or later in the vector; -1 if the element is not found.

Throws:

    `IndexOutOfBoundsException` - if the specified index is negative

See Also:

    `Object.equals(Object)`

lastIndexOf

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

Returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element. More formally, returns the highest index i such that `(o==null ? get(i)==null : o.equals(get(i)))`, or -1 if there is no such index.

Specified by:

    `lastIndexOf` in interface `List<E>`

Overrides:

    `lastIndexOf` in class `AbstractList<E>`

Parameters:

    o - element to search for

Returns:

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

lastIndexOf

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

Returns the index of the last occurrence of the specified element in this vector, searching backwards from `index`, or returns -1 if the element is not found. More formally, returns the highest index i such that
`(i <= index && (o==null ? get(i)==null : o.equals(get(i))))`, or -1 if there is no such index.

Parameters:

    o - element to search for

    `index` - index to start searching backwards from

Returns:

the index of the last occurrence of the element at position less than or equal to `index` in this vector; -1 if the element is not found.

Throws:

   `IndexOutOfBoundsException` - if the specified index is greater than or equal to the current size of this vector

---

elementAt

`public E elementAt(int index)`

Returns the component at the specified index.

This method is identical in functionality to the `get(int)` method (which is part of the `List` interface).

Parameters:

   `index` - an index into this vector

Returns:

   the component at the specified index

Throws:

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

---

firstElement

`public E firstElement()`

Returns the first component (the item at index `0`) of this vector.

Returns:

   the first component of this vector

Throws:

   `NoSuchElementException` - if this vector has no components

---

lastElement

`public E lastElement()`

Returns the last component of the vector.

Returns:

   the last component of the vector, i.e., the component at index `size() - 1`.

Throws:

   `NoSuchElementException` - if this vector is empty

---

setElementAt

`public void setElementAt(E obj,`
`                int index)`

Sets the component at the specified `index` of this vector to be the specified object. The previous component at that position is discarded.

The index must be a value greater than or equal to 0 and less than the current size of the vector.

This method is identical in functionality to the `set(int, E)` method (which is part of the `List` interface). Note that the `set` method reverses the order of the parameters, to more closely match array usage. Note also that the `set` method returns the old value that was stored at the specified position.

Parameters:

    `obj` - what the component is to be set to

    `index` - the specified index

Throws:

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

removeElementAt

`public void removeElementAt(int index)`

Deletes the component at the specified index. Each component in this vector with an index greater or equal to the specified `index` is shifted downward to have an index one smaller than the value it had previously. The size of this vector is decreased by 1.

The index must be a value greater than or equal to 0 and less than the current size of the vector.

This method is identical in functionality to the `remove(int)` method (which is part of the `List` interface). Note that the `remove` method returns the old value that was stored at the specified position.

Parameters:

    `index` - the index of the object to remove

Throws:

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

insertElementAt

```
public void insertElementAt(E obj,
                            int index)
```

Inserts the specified object as a component in this vector at the specified `index`. Each component in this vector with an index greater or equal to the specified `index` is shifted upward to have an index one greater than the value it had previously.

The index must be a value greater than or equal to 0 and less than or equal to the current size of the vector. (If the index is equal to the current size of the vector, the new element is appended to the Vector.)

This method is identical in functionality to the `add(int, E)` method (which is part of the `List` interface). Note that the `add` method reverses the order of the parameters, to more closely match array usage.

Parameters:

    `obj` - the component to insert

    `index` - where to insert the new component

Throws:

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

addElement

`public void addElement(E obj)`

Adds the specified component to the end of this vector, increasing its size by one. The capacity of this vector is increased if its size becomes greater than its capacity.

This method is identical in functionality to the `add(E)` method (which is part of the `List` interface).

Parameters:

    `obj` - the component to be added

removeElement

`public boolean removeElement(Object obj)`

Removes the first (lowest-indexed) occurrence of the argument from this vector. If the object is found in this vector, each component in the vector with an index greater or equal to the object's index is shifted downward to have an index one smaller than the value it had previously.

This method is identical in functionality to the `remove(Object)` method (which is part of the `List` interface).

Parameters:

    `obj` - the component to be removed

Returns:

    `true` if the argument was a component of this vector; `false` otherwise.

removeAllElements

`public void removeAllElements()`

Removes all components from this vector and sets its size to zero.

This method is identical in functionality to the `clear()` method (which is part of the `List` interface).

clone

`public Object clone()`

Returns a clone of this vector. The copy will contain a reference to a clone of the internal data array, not a reference to the original internal data array of this `Vector` object.

Overrides:

    `clone` in class `Object`

Returns:

    a clone of this vector

See Also:

    `Cloneable`

toArray

`public Object[] toArray()`

Returns an array containing all of the elements in this Vector in the correct order.

Specified by:

toArray in interface `Collection<E>`

Specified by:

toArray in interface `List<E>`

Overrides:

toArray in class `AbstractCollection<E>`

Returns:

an array containing all of the elements in this collection

Since:

1.2

See Also:

`Arrays.asList(Object[])`

---

toArray

```
public <T> T[] toArray(T[] a)
```

Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array. If the Vector fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this Vector.

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

Specified by:

toArray in interface `Collection<E>`

Specified by:

toArray in interface `List<E>`

Overrides:

toArray in class `AbstractCollection<E>`

Parameters:

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

Returns:

an array containing the elements of the Vector

Throws:

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

`NullPointerException` - if the given array is null

Since:

1.2

---

get

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

Returns the element at the specified position in this Vector.

Specified by:

get in interface `List<E>`

Specified by:

get in class `AbstractList<E>`

Parameters:

`index` - index of the element to return

Returns:

object at the specified index

Throws:

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

Since:

1.2

---

set

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

Replaces the element at the specified position in this Vector with the specified element.

Specified by:

set in interface `List<E>`

Overrides:

set in class `AbstractList<E>`

Parameters:

`index` - index of the element to replace

`element` - element to be stored at the specified position

Returns:

the element previously at the specified position

Throws:

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

Since:

1.2

---

add

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

Appends the specified element to the end of this Vector.

Specified by:

add in interface `Collection<E>`

Specified by:

add in interface `List<E>`

Overrides:

   add in class `AbstractList<E>`

Parameters:

   e - element to be appended to this Vector

Returns:

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

Since:

   1.2

---

remove

`public boolean remove(Object o)`

Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged. More formally, removes the element with the lowest index i such that (`o==null ? get(i)==null : o.equals(get(i))`) (if such an element exists).

Specified by:

   remove in interface `Collection<E>`

Specified by:

   remove in interface `List<E>`

Overrides:

   remove in class `AbstractCollection<E>`

Parameters:

   o - element to be removed from this Vector, if present

Returns:

   true if the Vector contained the specified element

Since:

   1.2

---

add

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

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

Specified by:

   add in interface `List<E>`

Overrides:

   add in class `AbstractList<E>`

Parameters:

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

   element - element to be inserted

Throws:

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

Since:

> 1.2

remove

`public E remove(int index)`

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

Specified by:

> `remove` in interface `List<E>`

Overrides:

> `remove` in class `AbstractList<E>`

Parameters:

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

Returns:

> element that was removed

Throws:

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

Since:

> 1.2

clear

`public void clear()`

Removes all of the elements from this Vector. The Vector will be empty after this call returns (unless it throws an exception).

Specified by:

> `clear` in interface `Collection<E>`

Specified by:

> `clear` in interface `List<E>`

Overrides:

> `clear` in class `AbstractList<E>`

Since:

> 1.2

containsAll

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

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

Specified by:

　containsAll in interface Collection<E>

Specified by:

　containsAll in interface List<E>

Overrides:

　containsAll in class AbstractCollection<E>

Parameters:

　c - a collection whose elements will be tested for containment in this Vector

Returns:

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

Throws:

　NullPointerException - if the specified collection is null

See Also:

　AbstractCollection.contains(Object)

---

addAll

```
public boolean addAll(Collection<? extends E> c)
```

Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator. The behavior of this operation is undefined if the specified Collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified Collection is this Vector, and this Vector is nonempty.)

Specified by:

　addAll in interface Collection<E>

Specified by:

　addAll in interface List<E>

Overrides:

　addAll in class AbstractCollection<E>

Parameters:

　c - elements to be inserted into this Vector

Returns:

　true if this Vector changed as a result of the call

Throws:

　NullPointerException - if the specified collection is null

Since:

　1.2

See Also:

　AbstractCollection.add(Object)

---

removeAll

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

Removes from this Vector all of its elements that are contained in the specified Collection.

Specified by:

removeAll in interface Collection<E>

Specified by:

removeAll in interface List<E>

Overrides:

removeAll in class AbstractCollection<E>

Parameters:

c - a collection of elements to be removed from the Vector

Returns:

true if this Vector changed as a result of the call

Throws:

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

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

Since:

1.2

See Also:

AbstractCollection.remove(Object), AbstractCollection.contains(Object)

retainAll

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

Retains only the elements in this Vector that are contained in the specified Collection. In other words, removes from this Vector all of its elements that are not contained in the specified Collection.

Specified by:

retainAll in interface Collection<E>

Specified by:

retainAll in interface List<E>

Overrides:

retainAll in class AbstractCollection<E>

Parameters:

c - a collection of elements to be retained in this Vector (all other elements are removed)

Returns:

true if this Vector changed as a result of the call

Throws:

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

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

Since:

> 1.2

See Also:

> `AbstractCollection.remove(Object)`, `AbstractCollection.contains(Object)`

---

addAll

```
public boolean addAll(int index,
                Collection<? extends E> c)
```

Inserts all of the elements in the specified Collection into this Vector at the specified position. Shifts the element currently at that position (if any) and any subsequent elements to the right (increases their indices). The new elements will appear in the Vector in the order that they are returned by the specified Collection's iterator.

Specified by:

> `addAll` in interface `List<E>`

Overrides:

> `addAll` in class `AbstractList<E>`

Parameters:

> `index` - index at which to insert the first element from the specified collection
>
> `c` - elements to be inserted into this Vector

Returns:

> `true` if this Vector changed as a result of the call

Throws:

> `ArrayIndexOutOfBoundsException` - if the index is out of range (index < 0 || index > size())
>
> `NullPointerException` - if the specified collection is null

Since:

> 1.2

---

equals

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

Compares the specified Object with this Vector for equality. Returns true if and only if the specified Object is also a List, both Lists have the same size, and all corresponding pairs of elements in the two Lists are *equal*. (Two elements e1 and e2 are *equal* if (e1==null ? e2==null : e1.equals(e2)).) In other words, two Lists are defined to be equal if they contain the same elements in the same order.

Specified by:

> `equals` in interface `Collection<E>`

Specified by:

> `equals` in interface `List<E>`

Overrides:

> `equals` in class `AbstractList<E>`

Parameters:

> `o` - the Object to be compared for equality with this Vector

Returns:

  true if the specified Object is equal to this Vector

See Also:

  Object.hashCode(), HashMap

hashCode

public int hashCode()

Returns the hash code value for this Vector.

Specified by:

  hashCode in interface Collection<E>

Specified by:

  hashCode in interface List<E>

Overrides:

  hashCode in class AbstractList<E>

Returns:

  the hash code value for this list

See Also:

  Object.equals(java.lang.Object), System.identityHashCode(java.lang.Object)

toString

public String toString()

Returns a string representation of this Vector, containing the String representation of each element.

Overrides:

  toString in class AbstractCollection<E>

Returns:

  a string representation of this collection

subList

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

Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive. (If fromIndex and toIndex are equal, the returned List is empty.) The returned List is backed by this List, so changes in the returned List are reflected in this List, and vice-versa. The returned List supports all of the optional List operations supported by this List.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a List can be used as a range operation by operating on a subList view instead of a whole List. For example, the following idiom removes a range of elements from a List:

```
list.subList(from, to).clear();
```

Similar idioms may be constructed for indexOf and lastIndexOf, and all of the algorithms in the Collections class can be applied to a subList.

The semantics of the List returned by this method become undefined if the backing list (i.e., this List) is *structurally modified* in any way other than via the returned List. (Structural modifications are those that change the size of the List, or otherwise perturb it in such a fashion that iterations in progress may yield incorrect results.)

Specified by:

   subList in interface List<E>

Overrides:

   subList in class AbstractList<E>

Parameters:

   fromIndex - low endpoint (inclusive) of the subList

   toIndex - high endpoint (exclusive) of the subList

Returns:

   a view of the specified range within this List

Throws:

   IndexOutOfBoundsException - if an endpoint index value is out of range (fromIndex < 0 || toIndex > size)

   IllegalArgumentException - if the endpoint indices are out of order (fromIndex > toIndex)

---

removeRange

```
protected void removeRange(int fromIndex,
                int toIndex)
```

Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the list by (toIndex - fromIndex) elements. (If toIndex==fromIndex, this operation has no effect.)

Overrides:

   removeRange in class AbstractList<E>

Parameters:

   fromIndex - index of first element to be removed

   toIndex - index after last element to be removed

---

listIterator

```
public ListIterator<E> listIterator(int index)
```

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

The returned list iterator is *fail-fast*.

Specified by:

   listIterator in interface List<E>

Overrides:

   listIterator in class AbstractList<E>

Parameters:

index - index of the first element to be returned from the list iterator (by a call to next)

Returns:

a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list

Throws:

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

---

listIterator

public ListIterator<E> listIterator()

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

The returned list iterator is *fail-fast*.

Specified by:

listIterator in interface List<E>

Overrides:

listIterator in class AbstractList<E>

Returns:

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

See Also:

listIterator(int)

---

iterator

public Iterator<E> iterator()

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

The returned iterator is *fail-fast*.

Specified by:

iterator in interface Iterable<E>

Specified by:

iterator in interface Collection<E>

Specified by:

iterator in interface List<E>

Overrides:

iterator in class AbstractList<E>

Returns:

an iterator over the elements in this list in proper sequence

Submit a bug or feature

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