Vector class

28 March 2025 09:04

- Underlying data structure is resizable array or growable array.
- Insertion order is preserved.
- Duplicates are allowed.
- Hetrogenous objects are allowed.
- 'null' insertion is possible.
- It implements Serializable, Cloneable, RandomAccess Interfaces.
- Every method present in the Vector is synchronized and hence Vector object is thread safe.

Constructors:

i. Vector v = new Vector(); => Creates an empty vector object with default initial capacity 10. Once vector reaches its maximum capacity, then a new vector object will be created with new capacity = current_capacity*2; (Double capacity.)

ii. Vector v = new Vector(int initialCapacity); => Creates an empty Vector object with specified initial capacity.

iii. Vector v = new Vector(int initialCapacity, int incrementalCapacity);

iv. Vector v = new Vector(Collection c); => Create an equivalent object for the given collection. This constructor is meant for inter conversion between collection object.

Official Documentation:

```
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

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.

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 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.
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 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.

`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 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.
`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 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)`

`<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.