package java.util;

import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.function.UnaryOperator;

/**
 * The {code 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 {code Vector} can grow or shrink as needed to accommodate adding and removing items after the {code Vector} has been created.
 *
 * Each vector tries to optimize storage management by maintaining a {code capacity} and a {code capacityIncrement}. The {code 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 {code 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 {link #iterator() iterator} and {link #listIterator(int) listIterator} methods are <em>fail-fast</em>:
 * if the vector is structurally modified at any time after the iterator is created, in any way except through the iterator’s own {link ListIterator#remove() remove} or {link ListIterator#add(Object) add} methods, the iterator will throw a {link 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 {link Enumeration Enumerations} returned by the {link #elements() elements} method are <em>not</em> <em>fail-fast</em>.
 *
 * 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 {code ConcurrentModificationException} on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: <i>the fail-fast behavior of iterators should be used only to detect bugs.</i>
As of the Java 2 platform v1.2, this class was retrofitted to implement the {@link List} interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, {@code Vector} is synchronized. If a thread-safe implementation is not needed, it is recommended to use {@link ArrayList} in place of {@code Vector}.

@see Collection
@see LinkedList
@since JDK1.0

public class Vector&lt;E&gt;
extends AbstractList&lt;E&gt;
implements List&lt;E&gt;, RandomAccess, Cloneable, java.io.Serializable
{

/**
 * 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.
 * 
 * &lt;p&gt;Any array elements following the last element in the Vector are null.
 * 
 * @serial
 */
protected Object[] elementData;

/**
 * The number of valid components in this {@code Vector} object.
 * Components {@code elementData[0]} through {@code elementData[elementCount-1]} are the actual items.
 * 
 * @serial
 */
protected int elementCount;

/**
 * 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.
 * 
 * @serial
 */
protected int capacityIncrement;

/** use serialVersionUID from JDK 1.0.2 for interoperability */
private static final long serialVersionUID = -2767605614048989439L;

/**
 * Constructs an empty vector with the specified initial capacity and capacity increment.
 * 
 * @param initialCapacity the initial capacity of the vector
 * @param capacityIncrement the amount by which the capacity is increased when the vector overflows
 * @throws IllegalArgumentException if the specified initial capacity is negative
 */
public Vector(int initialCapacity, int capacityIncrement) {
    super();
    if (initialCapacity &lt; 0)
        throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
this.elementData = new Object[initialCapacity];
this.capacityIncrement = capacityIncrement;

/**
* Constructs an empty vector with the specified initial capacity and
* with its capacity increment equal to zero.
* @param initialCapacity the initial capacity of the vector
* @throws IllegalArgumentException if the specified initial capacity
*         is negative
*/
public Vector(int initialCapacity) {
  this(initialCapacity, 0);
}

/**
* Constructs an empty vector so that its internal data array
* has size (code 10) and its standard capacity increment is
* zero.
*/
public Vector() {
  this(10);
}

/**
* Constructs a vector containing the elements of the specified
* collection, in the order they are returned by the collection’s
* iterator.
* @param c the collection whose elements are to be placed into this
* vector
* @throws NullPointerException if the specified collection is null
* @since 1.2
*/
public Vector(Collection<? extends E> c) {
  elementData = c.toArray();
  elementCount = elementData.length;
  if (elementData.getClass() != Object[].class)
    elementData = Arrays.copyOf(elementData, elementCount, Object[].class);
}

/**
* Copies the components of this vector into the specified array.
* The item at index (code k) in this vector is copied into
* component (code k) of (code anArray).
* @param anArray the array into which the components get copied
* @param anArray the array into which the components get copied
* @param anArray the array into which the components get copied
* @param anArray the array into which the components get copied
* @param anArray the array into which the components get copied
* @throws NullPointerException if the given array is null
* @throws IndexOutOfBoundsException if the specified array is not
*         large enough to hold all the components of this vector
* @throws ArrayStoreException if a component of this vector is not of
*         a runtime type that can be stored in the specified array
* @see #toArray(Object[])
*/
public synchronized void copyInto(Object[] anArray) {
  System.arraycopy(elementData, 0, anArray, 0, elementCount);
}

/**
* 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 (code elementData),
* with a smaller one. An application can use this operation to
* minimize the storage of a vector.
*/
public synchronized void trimToSize() {
    modCount++;
    int oldCapacity = elementData.length;
    if (elementCount < oldCapacity) {
        elementData = Arrays.copyOf(elementData, elementCount);
    }
}

/**
 * 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.
 *
 * <p>If the current capacity of this vector is less than
 * {code minCapacity}, then its capacity is increased by replacing its
 * internal data array, kept in the field {code elementData}, with a
 * larger one. The size of the new data array will be the old size plus
 * {code capacityIncrement}, unless the value of
 * {code 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 {code minCapacity}, then the new capacity will
 * be {code minCapacity}.
 *
 * @param minCapacity the desired minimum capacity
 */
public synchronized void ensureCapacity(int minCapacity) {
    if (minCapacity > 0) {
        modCount++;
        ensureCapacityHelper(minCapacity);
    }
}

/**
 * This implements the unsynchronized semantics of ensureCapacity.
 * Synchronized methods in this class can internally call this
 * method for ensuring capacity without incurring the cost of an
 * extra synchronization.
 *
 * @see #ensureCapacity(int)
 */
private void ensureCapacityHelper(int minCapacity) {
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}

/**
 * The maximum size of array to allocate.
 * Some VMs reserve some header words in an array.
 * Attempts to allocate larger arrays may result in
 * OutOfMemoryError: Requested array size exceeds VM limit
 */
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                        capacityIncrement : oldCapacity);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity);
private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
        throw new OutOfMemoryError();
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
}

/**
 * Sets the size of this vector. If the new size is greater than the
 * current size, new {code null} items are added to the end of
 * the vector. If the new size is less than the current size, all
 * components at index {code newSize} and greater are discarded.
 * @param newSize the new size of this vector
 * @throws ArrayIndexOutOfBoundsException if the new size is negative
 */
public synchronized void setSize(int newSize) {
    modCount++;
    if (newSize > elementCount) {
        ensureCapacityHelper(newSize);
    } else {
        for (int i = newSize ; i < elementCount ; i++) {
            elementData[i] = null;
        }
    }
    elementCount = newSize;
}

/**
 * Returns the current capacity of this vector.
 * @return the current capacity (the length of its internal
 * data array, kept in the field {code elementData}
 * of this vector)
 */
public synchronized int capacity() {
    return elementData.length;
}

/**
 * Returns the number of components in this vector.
 * @return the number of components in this vector
 */
public synchronized int size() {
    return elementCount;
}

/**
 * Tests if this vector has no components.
 * @return {code true} if and only if this vector has
 * no components, that is, its size is zero;
 * {code false} otherwise.
 */
public synchronized boolean isEmpty() {
    return elementCount == 0;
}

/**
 * Returns an enumeration of the components of this vector. The
 * returned {code Enumeration} object will generate all items in
 * this vector. The first item generated is the item at index {code 0},
 * then the item at index {code 1}, and so on.
 * @return an enumeration of the components of this vector
 */
    */
  
  public Enumeration<E> elements() {
    return new Enumeration<E>() {
      int count = 0;

      public boolean hasMoreElements() {
        return count < elementCount;
      }

      public synchronized E nextElement() {
        synchronized (Vector.this) {
          if (count < elementCount) {
            return elementData(count++);
          }
        }
        throw new NoSuchElementException("Vector Enumeration");
      }
    };
  }

  /**
   * Returns \{code true\} if this vector contains the specified element.
   * More formally, returns \{code true\} if and only if this vector
   * contains at least one element \{code e\} such that
   * \(<\tt{o==null&nbsp;&nbsp;?&nbsp;&nbsp;e==null&nbsp;&nbsp;:&nbsp;&nbsp;o.equals(e)}\&nbsp;</tt>.
   * @param o element whose presence in this vector is to be tested
   * @return \{code true\} if this vector contains the specified element
   */
  public boolean contains(Object o) {
    return indexOf(o, 0) >= 0;
  }

  /**
   * 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 \{code i\} such that
   * \(<\tt{o==null&nbsp;&nbsp;?&nbsp;&nbsp;get(i)==null&nbsp;&nbsp;:&nbsp;&nbsp;o.equals(get(i))}\&nbsp;</tt>,
   * or -1 if there is no such index.
   * @param o element to search for
   * @return the index of the first occurrence of the specified element in
   *         this vector, or -1 if this vector does not contain the element
   */
  public int indexOf(Object o) {
    return indexOf(o, 0);
  }

  /**
   * Returns the index of the first occurrence of the specified element in
   * this vector, searching forwards from \{code index\}, or returns -1 if
   * the element is not found.
   * More formally, returns the lowest index \{code i\} such that
   * \(<\tt{i&amp;&nbsp;gte&amp;&nbsp;index&amp;&nbsp;&nbsp;?&nbsp;&nbsp;o==null&nbsp;&nbsp;?&nbsp;&nbsp;get(i)==null&nbsp;&nbsp;:&nbsp;&nbsp;o.equals(get(i))}\&nbsp;</tt>,
   * or -1 if there is no such index.
   * @param o element to search for
   * @param index index to start searching from
   * @return the index of the first occurrence of the element in
   *         this vector at position \{code index\} or later in the vector;
   *         \{code -1\} if the element is not found.
   * @throws IndexOutOfBoundsException if the specified index is negative
   * @see Object#equals(Object)
   */
  public synchronized int indexOf(Object o, int index) {
if (o == null) {
    for (int i = index; i < elementCount; i++)
        if (elementData[i] == null)
            return i;
} else {
    for (int i = index; i < elementCount; i++)
        if (o.equals(elementData[i]))
            return i;
}
return -1;

/**
 * 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.
 *
 * @param o element to search for
 * @return the index of the last occurrence of the specified element in
 * this vector, or -1 if this vector does not contain the element
 */
public synchronized int lastIndexOf(Object o) {
    return lastIndexOf(o, elementCount - 1);
}

/**
 * Returns the index of the last occurrence of the specified element in
 * this vector, searching backwards from \( i \), or returns -1 if
 * the element is not found.
 * More formally, returns the highest index \( i \) such that
 * \( (i \leq index \land (o==null ? get(i)==null : o.equals(get(i)))) \),
 * or -1 if there is no such index.
 *
 * @param o element to search for
 * @param index index to start searching backwards from
 * @return the index of the last occurrence of the element at position
 * less than or equal to \( i \) 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
 */
public synchronized int lastIndexOf(Object o, int index) {
    if (index >= elementCount)
        throw new IndexOutOfBoundsException(index + " >= "+ elementCount);
    if (o == null) {
        for (int i = index; i >= 0; i--)
            if (elementData[i] == null)
                return i;
    } else {
        for (int i = index; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

/**
 * Returns the component at the specified index.
 * <p>This method is identical in functionality to the \{@link #get(int)\}
 * method (which is part of the \{@link List\} interface).
 *
 * @param index an index into this vector
 */
public synchronized E elementAt(int index) {
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
    }
    return elementData(index);
}

public synchronized E firstElement() {
    if (elementCount == 0) {
        throw new NoSuchElementException();
    }
    return elementData(0);
}

public synchronized E lastElement() {
    if (elementCount == 0) {
        throw new NoSuchElementException();
    }
    return elementData(elementCount - 1);
}

public synchronized void setElementAt(E obj, int index) {
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
    }
    elementData[index] = obj;
}
/**
 * Deletes the component at the specified index. Each component in
 * this vector with an index greater or equal to the specified
 * {code index} is shifted downward to have an index one
 * smaller than the value it had previously. The size of this vector
 * is decreased by {code 1}.
 * 
 * The index must be a value greater than or equal to {code 0}
 * and less than the current size of the vector.
 * 
 * This method is identical in functionality to the 
 * {link #remove(int)} method (which is part of the 
 * {link List} interface). Note that the 
 * {code remove} method returns the old value that was stored at the
 * specified position.
 * 
 * @param index the index of the object to remove
 * @throws ArrayIndexOutOfBoundsException if the index is out of range
 *       ({code index < 0 || index >= size()})
 */

public synchronized void removeElementAt(int index) {
    modCount++;
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " +
                elementCount);
    } else if (index < 0) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    int j = elementCount - index - 1;
    if (j > 0) {
        System.arraycopy(elementData, index + 1, elementData, index, j);
    }
    elementCount--;
    elementData[elementCount] = null; /* to let gc do its work */
}

/**
 * Inserts the specified object as a component in this vector at the
 * specified {code index}. Each component in this vector with
 * an index greater or equal to the specified {code 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 {code 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
 * {link #add(int, Object) add(int, E)} method (which is part of the 
 * {link List} interface). Note that the 
 * {code add} method reverses the order of the parameters, to more closely
 * match array usage.
 * 
 * @param obj the component to insert
 * @param index where to insert the new component
 * @throws ArrayIndexOutOfBoundsException if the index is out of range
 *       ({code index < 0 || index > size()})
 */

public synchronized void insertElementAt(E obj, int index) {
    modCount++;
    if (index > elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " > " + elementCount);
    }
    ensureCapacityHelper(elementCount + 1);
System.arraycopy(elementData, index, elementData, index + 1, elementCount - index);

elementData[index] = obj;
elementCount++;

/**
 * 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.
 *<p>This method is identical in functionality to the
 * {@link List#add(E) add(E)} method (which is part of the
 * {@link List} interface).
 * @param obj   the component to be added
 */
 public synchronized void addElement(E obj) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = 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.
 *<p>This method is identical in functionality to the
 * {@link List#remove(Object) remove(Object)} method (which is part of the
 * {@link List} interface).
 * @param obj   the component to be removed
 * @return {code true} if the argument was a component of this
 * vector; {code false} otherwise.
 */
 public synchronized boolean removeElement(Object obj) {
    modCount++;
    int i = indexOf(obj);
    if (i >= 0) {
        removeElementAt(i);
        return true;
    }
    return false;
 }

/**
 * Removes all components from this vector and sets its size to zero.
 *<p>This method is identical in functionality to the
 * {@link List#clear} method (which is part of the
 * {@link List} interface).
 */
 public synchronized void removeAllElements() {
    modCount++;
    // Let gc do its work
    for (int i = 0; i < elementCount; i++)
        elementData[i] = null;
    elementCount = 0;
 }

/**
 * 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 {code Vector} object.
 */
public synchronized Object clone() {
    try {
        @SuppressWarnings("unchecked")
        Vector<E> v = (Vector<E>) super.clone();
        v.elementData = Arrays.copyOf(elementData, elementCount);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError(e);
    }
}

public synchronized Object[] toArray() {
    return Arrays.copyOf(elementData, elementCount);
}

/*
 * Returns an array containing all of the elements in this Vector
 * in the correct order.
 * @since 1.2
 * @return an array containing the elements of the Vector
 */
public synchronized <T> T[] toArray(T[] a) {
    if (a.length < elementCount)
        return (T[]) Arrays.copyOf(elementData, elementCount, a.getClass());
    System.arraycopy(elementData, 0, a, 0, elementCount);
    if (a.length > elementCount)
        a[elementCount] = null;
    return a;
}

// Positional Access Operations

@SuppressWarnings("unchecked")
E elementData(int index) {
    return (E) elementData[index];
}
/**
 * Returns the element at the specified position in this Vector.
 * @param index index of the element to return
 * @return object at the specified index
 * @throws ArrayIndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= size()})
 * @since 1.2
 */
public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    return elementData[index];
}

/**
 * Replaces the element at the specified position in this Vector with the
 * specified element.
 * @param index index of the element to replace
 * @param element element to be stored at the specified position
 * @return the element previously at the specified position
 * @throws ArrayIndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= size()})
 * @since 1.2
 */
public synchronized E set(int index, E element) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    E oldValue = elementData[index];
    elementData[index] = element;
    return oldValue;
}

/**
 * Appends the specified element to the end of this Vector.
 * @param e element to be appended to this Vector
 * @return {code true} (as specified by {@link Collection#add})
 * @since 1.2
 */
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}

/**
 * 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
 * {code o==null ? get(i)==null : o.equals(get(i))} (if such
 * an element exists).
 * @param o element to be removed from this Vector, if present
 * @return true if the Vector contained the specified element
 * @since 1.2
 */
public boolean remove(Object o) {
    return removeElement(o);
}
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).
  * @param index index at which the specified element is to be inserted
  * @param element element to be inserted
  * @throws ArrayIndexOutOfBoundsException if the index is out of range
  *          (@code index < 0 || index > size())
  * @since 1.2
  */
public void add(int index, E element) {
    insertElementAt(element, 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.
 * @throws ArrayIndexOutOfBoundsException if the index is out of range
 *          (@code index < 0 || index >= size())
 * @param index the index of the element to be removed
 * @return element that was removed
 * @since 1.2
 */
public synchronized E remove(int index) {
    modCount++;
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    E oldValue = elementData(index);
    int numMoved = elementCount - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--elementCount] = null; // Let gc do its work
    return oldValue;
}
/**
 * Removes all of the elements from this Vector. The Vector will
 * be empty after this call returns (unless it throws an exception).
 * @since 1.2
 */
public void clear() {
    removeAllElements();
}
// Bulk Operations
/**
 * Returns true if this Vector contains all of the elements in the
 * specified Collection.
 * @param c a collection whose elements will be tested for containment
 * in this Vector
 * @return true if this Vector contains all of the elements in the
 * specified collection
 * @throws NullPointerException if the specified collection is null
 */
public synchronized boolean containsAll(Collection<? super E> c) {
    return super.containsAll(c);
}
/**
public synchronized boolean addAll(Collection<? extends E> c) {
    modCount++;
    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacityHelper(elementCount + numNew);
    System.arraycopy(a, 0, elementData, elementCount, numNew);
    elementCount += numNew;
    return numNew != 0;
}

public synchronized boolean removeAll(Collection<?> c) {
    return super.removeAll(c);
}

public synchronized boolean retainAll(Collection<?> c) {
    return super.retainAll(c);
}
public synchronized boolean addAll(int index, Collection<? extends E> c) {
    modCount++;
    if (index < 0 || index > elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacityHelper(elementCount + numNew);
    int numMoved = elementCount - index;
    if (numMoved > 0)
        System.arraycopy(elementData, index, elementData, index + numNew, numMoved);
    System.arraycopy(a, 0, elementData, index, numNew);
    elementCount += numNew;
    return numNew != 0;
}

/**
 * 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 <em>equal</em>. (Two elements {@code e1} and
 * {@code e2} are <em>equal</em> if {@code (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.
 * @param o the Object to be compared for equality with this Vector
 * @return true if the specified Object is equal to this Vector
 */
public synchronized boolean equals(Object o) {
    return super.equals(o);
}

/**
 * Returns the hash code value for this Vector.
 */
public synchronized int hashCode() {
    return super.hashCode();
}

/**
 * Returns a string representation of this Vector, containing
 * the String representation of each element.
 */
public synchronized String toString() {
    return super.toString();
}

/** 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.
 *
 * @param index index at which to insert the first element from the
 * specified collection
 * @param c elements to be inserted into this Vector
 * @return {@code true} if this Vector changed as a result of the call
 * @throws ArrayIndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index > size()})
 * @throws NullPointerException if the specified collection is null
 * @since 1.2
 */
public synchronized boolean addAll(int index, Collection<? extends E> c) {
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

<pre>
      list.subList(from, to).clear();
</pre>

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.)

@throws IndexOutOfBoundsException if an endpoint index value is out of range

@throws IllegalArgumentException if the endpoint indices are out of order

```
public synchronized List<E> subList(int fromIndex, int toIndex) {
    return Collections.synchronizedList(super.subList(fromIndex, toIndex), this);
}
```

```
protected synchronized void removeRange(int fromIndex, int toIndex) {
    modCount++;  
    int numMoved = elementCount - toIndex;  
    System.arraycopy(elementData, toIndex, elementData, fromIndex, numMoved);

    // Let gc do its work
    int newElementCount = elementCount - (toIndex-fromIndex);  
    while (elementCount != newElementCount)  
        elementData[--elementCount] = null;
}
```

```
private void writeObject(java.io.ObjectOutputStream s) throws java.io.IOException {
    final java.io.ObjectOutputStream.PutField fields = s.putFields();
    final Object[] data;
```
synchronized (this) {
    fields.put("capacityIncrement", capacityIncrement);
    fields.put("elementCount", elementCount);
    data = elementData.clone();
    }
fields.put("elementData", data);
s.writeFields();
}

/**
 * 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 {@link ListIterator#next next}.
 * An initial call to {@link ListIterator#previous previous} would
 * return the element with the specified index minus one.
 * <pre>The returned list iterator is &lt;a href="#fail-fast"&gt;i&gt;fail-fast&lt;/i&gt;.&lt;/a&gt;
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public synchronized ListIterator<E> listIterator(int index) {
    if (index < 0 || index > elementCount)
        throw new IndexOutOfBoundsException("Index: "+index);
    return new ListItr(index);
}

/**
 * Returns a list iterator over the elements in this list (in proper
 * sequence).
 * <pre>The returned list iterator is &lt;a href="#fail-fast"&gt;i&gt;fail-fast&lt;/i&gt;.&lt;/a&gt;
 * @see #listIterator(int)
 */
public synchronized ListIterator<E> listIterator() {
    return new ListItr(0);
}

/**
 * Returns an iterator over the elements in this list in proper sequence.
 * <pre>The returned iterator is &lt;a href="#fail-fast"&gt;i&gt;fail-fast&lt;/i&gt;.&lt;/a&gt;
 * @return an iterator over the elements in this list in proper sequence
 */
public synchronized Iterator<E> iterator() {
    return new Itr();
}

/**
 * An optimized version of AbstractList.Itr
 */
private class Itr implements Iterator<E> {
    int cursor; // index of next element to return
    int lastRet = -1; // index of last element returned; -1 if no such
    int expectedModCount = modCount;

    public boolean hasNext() {
        // Racy but within spec, since modifications are checked
        // within or after synchronization in next/previous
        return cursor != elementCount;
    }

    public E next() {
        synchronized (Vector.this) {
            checkForComodification();
            return cursor != elementCount;
        }
    }
int i = cursor;
if (i >= elementCount)
    throw new NoSuchElementException();
cursor = i + 1;
return elementData(lastRet = i);
}

public void remove() {
    if (lastRet == -1)
        throw new IllegalStateException();
synchronized (Vector.this) {
            checkForComodification();
            Vector.this.remove(lastRet);
            expectedModCount = modCount;
            cursor = lastRet;
lastRet = -1;
        }

@Override
public void forEachRemaining(Consumer<? super E> action) {
    Objects.requireNonNull(action);
synchronized (Vector.this) {
        final int size = elementCount;
        int i = cursor;
        if (i >= size) {
            return;
        }
        @SuppressWarnings("unchecked")
        final E[] elementData = (E[]) Vector.this.elementData;
        if (i >= elementData.length) {
            throw new ConcurrentModificationException();
        }
        while (i != size && modCount == expectedModCount) {
            action.accept(elementData[i++]);
        }
        // update once at end of iteration to reduce heap write traffic
        cursor = i;
lastRet = i - 1;
checkForComodification();
    }
}

final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}

/**
 * An optimized version of AbstractList.ListItr
 */
final class ListItr extends Itr implements ListIterator<E> {
    ListItr(int index) {
        super();
cursor = index;
    }

    public boolean hasPrevious() {
        return cursor != 0;
    }

    public int nextIndex() {
        return cursor;
    }
}
public int previousIndex() {
    return cursor - 1;
}

public E previous() {
    synchronized (Vector.this) {
        int i = cursor - 1;
        if (i < 0)
            throw new NoSuchElementException();
        cursor = i;
        return elementData(lastRet = i);
    }
}

public void set(E e) {
    if (lastRet == -1)
        throw new IllegalStateException();
    synchronized (Vector.this) {
        checkForComodification();
        Vector.this.set(lastRet, e);
    }
}

public void add(E e) {
    int i = cursor;
    synchronized (Vector.this) {
        checkForComodification();
        Vector.this.add(i, e);
        expectedModCount = modCount;
    }
    cursor = i + 1;
    lastRet = -1;
}

@Override
public synchronized void forEach(Consumer<? super E> action) {
    Objects.requireNonNull(action);
    final int expectedModCount = modCount;
    @SuppressWarnings("unchecked")
    final E[] elementData = (E[]) this.elementData;
    final int elementCount = this.elementCount;
    for (int i=0; modCount == expectedModCount && i < elementCount; i++) {
        action.accept(elementData[i]);
    }
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}

@Override
@SuppressWarnings("unchecked")
public synchronized boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    // figure out which elements are to be removed
    // any exception thrown from the filter predicate at this stage
    // will leave the collection unmodified
    int removeCount = 0;
    final int size = elementCount;
    final BitSet removeSet = new BitSet(size);
    final int expectedModCount = modCount;
    for (int i=0; modCount == expectedModCount && i < size; i++) {
        @SuppressWarnings("unchecked")
        final E element = (E) elementData[i];
        if (filter.test(element)) {
            removeSet.set(i);
        }
removeCount++;}

} if (modCount != expectedModCount) {
    throw new ConcurrentModificationException();
}

// shift surviving elements left over the spaces left by removed elements
final boolean anyToRemove = removeCount > 0;
if (anyToRemove) {
    final int newSize = size - removeCount;
    for (int i = 0, j = 0; i < size && j < newSize; i++, j++) {
        i = removeSet.nextClearBit(i);
        elementData[j] = elementData[i];
    }
    for (int k = newSize; k < size; k++) {
        elementData[k] = null; // Let gc do its work
    }
    elementCount = newSize;
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
    modCount++;
}

return anyToRemove;
}

@Override
@SupressWarnings("unchecked")
public synchronized void replaceAll(UnaryOperator<E> operator) {
    Objects.requireNonNull(operator);
    final int expectedModCount = modCount;
    final int size = elementCount;
    for (int i = 0; modCount == expectedModCount && i < size; i++) {
        elementData[i] = operator.apply((E) elementData[i]);
    }
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
    modCount++;
}

@Override
@SupressWarnings("unchecked")
public synchronized void sort(Comparator<? super E> c) {
    final int expectedModCount = modCount;
    Arrays.sort((E[]) elementData, 0, elementCount, c);
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
    modCount++;
}

/**
 * Creates a $<a href="Spliterator.html#binding">late-binding</a>$ late-binding $<em>/a>$ over the elements in this
 * list.
 * <p>The $<code>Spliterator</code> reports $<a href="Spliterator$SIZED"$, $<a href="Spliterator$SUBSIZED", and $<a href="Spliterator$ORDERED"$).
 * Overriding implementations should document the reporting of additional
 * characteristic values.
 * 
 * @return a $<code>Spliterator</code> over the elements in this list
 * @since 1.8
 */
@Override
public Spliterator<E> spliterator() {
    return new VectorSpliterator<>(this, null, 0, -1, 0);
}

/** Similar to ArrayList Spliterator */
static final class VectorSpliterator<E> implements Spliterator<E> {
    private final Vector<E> list;
    private Object[] array;
    private int index; // current index, modified on advance/split
    private int fence; // -1 until used; then one past last index
    private int expectedModCount; // initialized when fence set

    /** Create new spliterator covering the given range */
    VectorSpliterator(Vector<E> list, Object[] array, int origin, int fence,
                      int expectedModCount) {
        this.list = list;
        this.array = array;
        this.index = origin;
        this.fence = fence;
        this.expectedModCount = expectedModCount;
    }

    private int getFence() { // initialize on first use
        int hi;
        if (((hi = fence) < 0) {
            synchronized(list) {
                array = list.elementData;
                expectedModCount = list.modCount;
                hi = fence = list.elementCount;
            }
        }
        return hi;
    }

    public Spliterator<E> trySplit() {
        int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
        return (lo >= mid) ? null :
               new VectorSpliterator<>(list, array, lo, index = mid,
                                      expectedModCount);
    }

    @SuppressWarnings("unchecked")
    public boolean tryAdvance(Consumer<? super E> action) {
        int i;
        if (action == null)
            throw new NullPointerException();
        if (getFence() > (i = index)) {
            index = i + 1;
            action.accept((E)array[i]);
            if (list.modCount != expectedModCount)
                throw new ConcurrentModificationException();
            return true;
        }
        return false;
    }

    @SuppressWarnings("unchecked")
    public void forEachRemaining(Consumer<? super E> action) {
        int i, hi; // hoist accesses and checks from loop
        Vector<E> lst; Object[] a;
        if (action == null)
            throw new NullPointerException();
        if ((lst = list) != null) {
            synchronized(lst) {
                expectedModCount = lst.modCount;
        }
a = array = lst.elementData;
hi = fence = lst.elementCount;
}
else
    a = array;
    if (a != null && (i = index) >= 0 && (index = hi) <= a.length) {
        while (i < hi)
            action.accept((E) a[i++]);
        if (lst.modCount == expectedModCount)
            return;
    }
    throw new ConcurrentModificationException();
}

public long estimateSize() {
    return (long) (getFence() - index);
}

public int characteristics() {
    return Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;
}