```
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.
 *
 * <p>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.
 *
 * <p id="fail-fast">
 * 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> fail-fast; if the
 * Vector is structurally modified at any time after the enumeration is
 * created then the results of enumerating are undefined.
 *
 * <p>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>
 *
 * <p>As of the Java 2 platform v1.2, this class was retrofitted to
 * implement the {@link List} interface, making it a member of the
 * <a href="{@docRoot}/java/util/package-summary.html#CollectionsFramework">
 * Java Collections Framework</a>.  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}.
 *
```

```
 * @param <E> Type of component elements
 *
 * @author  Lee Boynton
 * @author  Jonathan Payne
 * @see Collection
 * @see LinkedList
 * @since   1.0
 */
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, 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.
     *
     * <p>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 < 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;
        // defend against c.toArray (incorrectly) not returning Object[]
        // (see e.g. https://bugs.openjdk.java.net/browse/JDK-6260652)
        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
     * @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++;
        if (minCapacity > elementData.length)
            grow(minCapacity);
    }
}

/**
 * The maximum size of array to allocate (unless necessary).
 * 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;

/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 * @throws OutOfMemoryError if minCapacity is less than zero
 */
private Object[] grow(int minCapacity) {
    return elementData = Arrays.copyOf(elementData,
                                       newCapacity(minCapacity));
}

private Object[] grow() {
    return grow(elementCount + 1);
}

/**
 * Returns a capacity at least as large as the given minimum capacity.
 * Will not return a capacity greater than MAX_ARRAY_SIZE unless
 * the given minimum capacity is greater than MAX_ARRAY_SIZE.
 *
 * @param minCapacity the desired minimum capacity
 * @throws OutOfMemoryError if minCapacity is less than zero
 */
private int newCapacity(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                     capacityIncrement : oldCapacity);
    if (newCapacity - minCapacity <= 0) {
        if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
        return minCapacity;
    }
    return (newCapacity - MAX_ARRAY_SIZE <= 0)
        ? newCapacity
        : hugeCapacity(minCapacity);
}

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 > elementData.length)
        grow(newSize);
    final Object[] es = elementData;
    for (int to = elementCount, i = newSize; i < to; i++)
        es[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. If the vector is
 * structurally modified while enumerating over the elements then the
 * results of enumerating are undefined.
 *
 * @return  an enumeration of the components of this vector
 * @see     Iterator
 */
public Enumeration<E> elements() {
    return new Enumeration<E>() {
        int count = 0;

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

        public 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
 * {@code Objects.equals(o, e)}.
 *
 * @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
     * {@code Objects.equals(o, get(i))},
     * 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
     * {@code (i >= index && Objects.equals(o, get(i)))},
     * 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 {@code i} such that
     * {@code Objects.equals(o, 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 {@code index}, or returns -1 if
     * the element is not found.
     * More formally, returns the highest index {@code i} such that
     * {@code (i <= index && Objects.equals(o, 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 {@code 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
     */
    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
 * @return     the component at the specified index
 * @throws ArrayIndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= size()})
 */
public synchronized E elementAt(int index) {
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
    }

    return elementData(index);
}

/**
 * Returns the first component (the item at index {@code 0}) of
 * this vector.
 *
 * @return     the first component of this vector
 * @throws NoSuchElementException if this vector has no components
 */
public synchronized E firstElement() {
    if (elementCount == 0) {
        throw new NoSuchElementException();
    }
    return elementData(0);
}

/**
 * Returns the last component of the vector.
 *
 * @return  the last component of the vector, i.e., the component at index
 *          {@code size() - 1}
 * @throws NoSuchElementException if this vector is empty
 */
public synchronized E lastElement() {
    if (elementCount == 0) {
        throw new NoSuchElementException();
    }
    return elementData(elementCount - 1);
}

/**
 * Sets the component at the specified {@code index} of this
 * vector to be the specified object. The previous component at that
 * position is discarded.
 *
 * <p>The index must be a value greater than or equal to {@code 0}
 * and less than the current size of the vector.
 *
 * <p>This method is identical in functionality to the
 * {@link #set(int, Object) set(int, E)}
 * method (which is part of the {@link List} interface). Note that the
 * {@code set} method reverses the order of the parameters, to more closely
 * match array usage.  Note also that the {@code set} method returns the
 * old value that was stored at the specified position.
 *
 * @param      obj     what the component is to be set to
 * @param      index   the specified index
 * @throws ArrayIndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= size()})
 */
```

```
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}.
 *
 * <p>The index must be a value greater than or equal to {@code 0}
 * and less than the current size of the vector.
 *
 * <p>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) {
    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);
    }
    modCount++;
    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.
 *
 * <p>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.)
 *
 * <p>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) {
    if (index > elementCount) {
        throw new ArrayIndexOutOfBoundsException(index
                                                 + " > " + elementCount);
    }
    modCount++;
    final int s = elementCount;
    Object[] elementData = this.elementData;
    if (s == elementData.length)
        elementData = grow();
    System.arraycopy(elementData, index,
```

```
                     elementData, index + 1,
                     s - index);
    elementData[index] = obj;
    elementCount = s + 1;
}

/**
 * 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 #add(Object) 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++;
    add(obj, elementData, elementCount);
}

/**
 * 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 #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 #clear}
 * method (which is part of the {@link List} interface).
 */
public synchronized void removeAllElements() {
    final Object[] es = elementData;
    for (int to = elementCount, i = elementCount = 0; i < to; i++)
        es[i] = null;
    modCount++;
}

/**
 * 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.
 *
 * @return  a clone of this vector
 */
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);
    }
```

```
}

/**
 * Returns an array containing all of the elements in this Vector
 * in the correct order.
 *
 * @since 1.2
 */
public synchronized Object[] toArray() {
    return Arrays.copyOf(elementData, elementCount);
}

/**
 * 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.
 *
 * <p>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 <em>only</em> if the caller knows that the Vector
 * does not contain any null elements.)
 *
 * @param <T> type of array elements. The same type as {@code <E>} or a
 * supertype of {@code <E>}.
 * @param 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.
 * @return an array containing the elements of the Vector
 * @throws ArrayStoreException if the runtime type of a, {@code <T>}, is not
 * a supertype of the runtime type, {@code <E>}, of every element in this
 * Vector
 * @throws NullPointerException if the given array is null
 * @since 1.2
 */
@SuppressWarnings("unchecked")
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];
}

@SuppressWarnings("unchecked")
static <E> E elementAt(Object[] es, int index) {
    return (E) es[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);
}
```

```java
/**
 * 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;
}

/**
 * This helper method split out from add(E) to keep method
 * bytecode size under 35 (the -XX:MaxInlineSize default value),
 * which helps when add(E) is called in a C1-compiled loop.
 */
private void add(E e, Object[] elementData, int s) {
    if (s == elementData.length)
        elementData = grow();
    elementData[s] = e;
    elementCount = s + 1;
}

/**
 * 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++;
    add(e, elementData, elementCount);
    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 Objects.equals(o, 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.
     *
     * @param index the index of the element to be removed
     * @return element that was removed
     * @throws ArrayIndexOutOfBoundsException if the index is out of range
     *         ({@code index < 0 || index >= size()})
     * @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<?> c) {
        return super.containsAll(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.)
     *
     * @param c elements to be inserted into this Vector
     * @return {@code true} if this Vector changed as a result of the call
     * @throws NullPointerException if the specified collection is null
     * @since 1.2
     */
    public boolean addAll(Collection<? extends E> c) {
        Object[] a = c.toArray();
        modCount++;
        int numNew = a.length;
        if (numNew == 0)
            return false;
        synchronized (this) {
            Object[] elementData = this.elementData;
            final int s = elementCount;
            if (numNew > elementData.length - s)
                elementData = grow(s + numNew);
            System.arraycopy(a, 0, elementData, s, numNew);
            elementCount = s + numNew;
            return true;
        }
```

```
}

/**
 * Removes from this Vector all of its elements that are contained in the
 * specified Collection.
 *
 * @param c a collection of elements to be removed from the Vector
 * @return 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
 * (<a href="Collection.html#optional-restrictions">optional</a>)
 * @throws NullPointerException if this vector contains one or more null
 *         elements and the specified collection does not support null
 *         elements
 * (<a href="Collection.html#optional-restrictions">optional</a>),
 *         or if the specified collection is null
 * @since 1.2
 */
public boolean removeAll(Collection<?> c) {
    Objects.requireNonNull(c);
    return bulkRemove(e -> c.contains(e));
}

/**
 * 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.
 *
 * @param c a collection of elements to be retained in this Vector
 *          (all other elements are removed)
 * @return 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
 * (<a href="Collection.html#optional-restrictions">optional</a>)
 * @throws NullPointerException if this vector contains one or more null
 *         elements and the specified collection does not support null
 *         elements
 *         (<a href="Collection.html#optional-restrictions">optional</a>),
 *         or if the specified collection is null
 * @since 1.2
 */
public boolean retainAll(Collection<?> c) {
    Objects.requireNonNull(c);
    return bulkRemove(e -> !c.contains(e));
}

/**
 * @throws NullPointerException {@inheritDoc}
 */
@Override
public boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    return bulkRemove(filter);
}

// A tiny bit set implementation

private static long[] nBits(int n) {
    return new long[((n - 1) >> 6) + 1];
}
private static void setBit(long[] bits, int i) {
    bits[i >> 6] |= 1L << i;
}
private static boolean isClear(long[] bits, int i) {
    return (bits[i >> 6] & (1L << i)) == 0;
}

private synchronized boolean bulkRemove(Predicate<? super E> filter) {
    int expectedModCount = modCount;
    final Object[] es = elementData;
    final int end = elementCount;
    int i;
    // Optimize for initial run of survivors
    for (i = 0; i < end && !filter.test(elementAt(es, i)); i++)
        ;
    // Tolerate predicates that reentrantly access the collection for
```

```
    // read (but writers still get CME), so traverse once to find
    // elements to delete, a second pass to physically expunge.
    if (i < end) {
        final int beg = i;
        final long[] deathRow = nBits(end - beg);
        deathRow[0] = 1L;   // set bit 0
        for (i = beg + 1; i < end; i++)
            if (filter.test(elementAt(es, i)))
                setBit(deathRow, i - beg);
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        expectedModCount++;
        modCount++;
        int w = beg;
        for (i = beg; i < end; i++)
            if (isClear(deathRow, i - beg))
                es[w++] = es[i];
        for (i = elementCount = w; i < end; i++)
            es[i] = null;
        return true;
    } else {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        return false;
    }
}

/**
 * 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) {
    if (index < 0 || index > elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    Object[] a = c.toArray();
    modCount++;
    int numNew = a.length;
    if (numNew == 0)
        return false;
    Object[] elementData = this.elementData;
    final int s = elementCount;
    if (numNew > elementData.length - s)
        elementData = grow(s + numNew);

    int numMoved = s - index;
    if (numMoved > 0)
        System.arraycopy(elementData, index,
                         elementData, index + numNew,
                         numMoved);
    System.arraycopy(a, 0, elementData, index, numNew);
    elementCount = s + numNew;
    return true;
}

/**
 * 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 Objects.equals(e1, 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();
    }

    /**
     * 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.
     *
     * <p>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:
     * <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.
     *
     * <p>The semantics of the List returned by this method become undefined if
     * the backing list (i.e., this List) is <i>structurally modified</i> 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.)
     *
     * @param fromIndex low endpoint (inclusive) of the subList
     * @param toIndex high endpoint (exclusive) of the subList
     * @return a view of the specified range within this List
     * @throws IndexOutOfBoundsException if an endpoint index value is out of range
     *         {@code (fromIndex < 0 || toIndex > size)}
     * @throws IllegalArgumentException if the endpoint indices are out of order
     *         {@code (fromIndex > toIndex)}
     */
    public synchronized List<E> subList(int fromIndex, int toIndex) {
        return Collections.synchronizedList(super.subList(fromIndex, toIndex),
                                            this);
    }

    /**
     * Removes from this list all of the elements whose index is between
     * {@code fromIndex}, inclusive, and {@code toIndex}, exclusive.
     * Shifts any succeeding elements to the left (reduces their index).
     * This call shortens the list by {@code (toIndex - fromIndex)} elements.
     * (If {@code toIndex==fromIndex}, this operation has no effect.)
     */
    protected synchronized void removeRange(int fromIndex, int toIndex) {
        modCount++;
        shiftTailOverGap(elementData, fromIndex, toIndex);
    }

    /** Erases the gap from lo to hi, by sliding down following elements. */
    private void shiftTailOverGap(Object[] es, int lo, int hi) {
        System.arraycopy(es, hi, es, lo, elementCount - hi);
        for (int to = elementCount, i = (elementCount -= hi - lo); i < to; i++)
            es[i] = null;
    }
```

```
/**
 * Saves the state of the {@code Vector} instance to a stream
 * (that is, serializes it).
 * This method performs synchronization to ensure the consistency
 * of the serialized data.
 *
 * @param s the stream
 * @throws java.io.IOException if an I/O error occurs
 */
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.
 *
 * <p>The returned list iterator is <a href="#fail-fast"><i>fail-fast</i></a>.
 *
 * @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).
 *
 * <p>The returned list iterator is <a href="#fail-fast"><i>fail-fast</i></a>.
 *
 * @see #listIterator(int)
 */
public synchronized ListIterator<E> listIterator() {
    return new ListItr(0);
}

/**
 * Returns an iterator over the elements in this list in proper sequence.
 *
 * <p>The returned iterator is <a href="#fail-fast"><i>fail-fast</i></a>.
 *
 * @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();
            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;
            }
            final Object[] es = elementData;
            if (i >= es.length)
                throw new ConcurrentModificationException();
            while (i < size && modCount == expectedModCount)
                action.accept(elementAt(es, 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) {
            checkForComodification();
            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;
    }
}

/**
 * @throws NullPointerException {@inheritDoc}
 */
@Override
public synchronized void forEach(Consumer<? super E> action) {
    Objects.requireNonNull(action);
    final int expectedModCount = modCount;
    final Object[] es = elementData;
    final int size = elementCount;
    for (int i = 0; modCount == expectedModCount && i < size; i++)
        action.accept(elementAt(es, i));
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}

/**
 * @throws NullPointerException {@inheritDoc}
 */
@Override
public synchronized void replaceAll(UnaryOperator<E> operator) {
    Objects.requireNonNull(operator);
    final int expectedModCount = modCount;
    final Object[] es = elementData;
    final int size = elementCount;
    for (int i = 0; modCount == expectedModCount && i < size; i++)
        es[i] = operator.apply(elementAt(es, i));
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
    modCount++;
}

@SuppressWarnings("unchecked")
@Override
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 <em><a href="Spliterator.html#binding">late-binding</a></em>
 * and <em>fail-fast</em> {@link Spliterator} over the elements in this
 * list.
 *
 * <p>The {@code Spliterator} reports {@link Spliterator#SIZED},
 * {@link Spliterator#SUBSIZED}, and {@link Spliterator#ORDERED}.
 * Overriding implementations should document the reporting of additional
 * characteristic values.
 *
 * @return a {@code Spliterator} over the elements in this list
 * @since 1.8
```

```
 */
@Override
public Spliterator<E> spliterator() {
    return new VectorSpliterator(null, 0, -1, 0);
}

/** Similar to ArrayList Spliterator */
final class VectorSpliterator implements Spliterator<E> {
    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

    /** Creates new spliterator covering the given range. */
    VectorSpliterator(Object[] array, int origin, int fence,
                      int expectedModCount) {
        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 (Vector.this) {
                array = elementData;
                expectedModCount = modCount;
                hi = fence = elementCount;
            }
        }
        return hi;
    }

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

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

    @SuppressWarnings("unchecked")
    public void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        final int hi = getFence();
        final Object[] a = array;
        int i;
        for (i = index, index = hi; i < hi; i++)
            action.accept((E) a[i]);
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }

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

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

void checkInvariants() {
    // assert elementCount >= 0;
    // assert elementCount == elementData.length || elementData[elementCount] == null;
```

```
    }
}
```