```java
/*
 * Copyright (c) 1994, 2013, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.  Oracle designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Oracle in the LICENSE file that accompanied this code.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 */

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><a name="fail-fast">
 * The iterators returned by this class's {@link #iterator() iterator} and
 * {@link #listIterator(int) listIterator} methods are <em>fail-fast</em></a>:
 * 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.
 *
 * <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>
```

```
68   *
69   * <p>As of the Java 2 platform v1.2, this class was retrofitted to
70   * implement the {@link List} interface, making it a member of the
71   * <a href="{@docRoot}/../technotes/guides/collections/index.html">
72   * Java Collections Framework</a>.  Unlike the new collection
73   * implementations, {@code Vector} is synchronized.  If a thread-safe
74   * implementation is not needed, it is recommended to use {@link
75   * ArrayList} in place of {@code Vector}.
76   *
77   * @author  Lee Boynton
78   * @author  Jonathan Payne
79   * @see Collection
80   * @see LinkedList
81   * @since   JDK1.0
82   */
83  public class Vector<E>
84      extends AbstractList<E>
85      implements List<E>, RandomAccess, Cloneable, java.io.Serializable
86  {
87      /**
88       * The array buffer into which the components of the vector are
89       * stored. The capacity of the vector is the length of this array buffer,
90       * and is at least large enough to contain all the vector's elements.
91       *
92       * <p>Any array elements following the last element in the Vector are null.
93       *
94       * @serial
95       */
96      protected Object[] elementData;
97
98      /**
99       * The number of valid components in this {@code Vector} object.
100      * Components {@code elementData[0]} through
101      * {@code elementData[elementCount-1]} are the actual items.
102      *
103      * @serial
104      */
105     protected int elementCount;
106
107     /**
108      * The amount by which the capacity of the vector is automatically
109      * incremented when its size becomes greater than its capacity.  If
110      * the capacity increment is less than or equal to zero, the capacity
111      * of the vector is doubled each time it needs to grow.
112      *
113      * @serial
114      */
115     protected int capacityIncrement;
116
117     /** use serialVersionUID from JDK 1.0.2 for interoperability */
118     private static final long serialVersionUID = -2767605614048989439L;
119
120     /**
121      * Constructs an empty vector with the specified initial capacity and
122      * capacity increment.
123      *
124      * @param   initialCapacity     the initial capacity of the vector
125      * @param   capacityIncrement   the amount by which the capacity is
126      *                              increased when the vector overflows
127      * @throws IllegalArgumentException if the specified initial capacity
128      *         is negative
129      */
130     public Vector(int initialCapacity, int capacityIncrement) {
131         super();
132         if (initialCapacity < 0)
133             throw new IllegalArgumentException("Illegal Capacity: "+
134                                                initialCapacity);
```

```
135         this.elementData = new Object[initialCapacity];
136         this.capacityIncrement = capacityIncrement;
137     }
138 
139     /**
140      * Constructs an empty vector with the specified initial capacity and
141      * with its capacity increment equal to zero.
142      *
143      * @param   initialCapacity   the initial capacity of the vector
144      * @throws IllegalArgumentException if the specified initial capacity
145      *         is negative
146      */
147     public Vector(int initialCapacity) {
148         this(initialCapacity, 0);
149     }
150 
151     /**
152      * Constructs an empty vector so that its internal data array
153      * has size {@code 10} and its standard capacity increment is
154      * zero.
155      */
156     public Vector() {
157         this(10);
158     }
159 
160     /**
161      * Constructs a vector containing the elements of the specified
162      * collection, in the order they are returned by the collection's
163      * iterator.
164      *
165      * @param c the collection whose elements are to be placed into this
166      *        vector
167      * @throws NullPointerException if the specified collection is null
168      * @since   1.2
169      */
170     public Vector(Collection<? extends E> c) {
171         elementData = c.toArray();
172         elementCount = elementData.length;
173         // c.toArray might (incorrectly) not return Object[] (see 6260652)
174         if (elementData.getClass() != Object[].class)
175             elementData = Arrays.copyOf(elementData, elementCount, Object[].class);
176     }
177 
178     /**
179      * Copies the components of this vector into the specified array.
180      * The item at index {@code k} in this vector is copied into
181      * component {@code k} of {@code anArray}.
182      *
183      * @param  anArray the array into which the components get copied
184      * @throws NullPointerException if the given array is null
185      * @throws IndexOutOfBoundsException if the specified array is not
186      *         large enough to hold all the components of this vector
187      * @throws ArrayStoreException if a component of this vector is not of
188      *         a runtime type that can be stored in the specified array
189      * @see #toArray(Object[])
190      */
191     public synchronized void copyInto(Object[] anArray) {
192         System.arraycopy(elementData, 0, anArray, 0, elementCount);
193     }
194 
195     /**
196      * Trims the capacity of this vector to be the vector's current
197      * size. If the capacity of this vector is larger than its current
198      * size, then the capacity is changed to equal the size by replacing
199      * its internal data array, kept in the field {@code elementData},
200      * with a smaller one. An application can use this operation to
201      * minimize the storage of a vector.
```

```
202      */
203     public synchronized void trimToSize() {
204         modCount++;
205         int oldCapacity = elementData.length;
206         if (elementCount < oldCapacity) {
207             elementData = Arrays.copyOf(elementData, elementCount);
208         }
209     }
210 
211     /**
212      * Increases the capacity of this vector, if necessary, to ensure
213      * that it can hold at least the number of components specified by
214      * the minimum capacity argument.
215      *
216      * <p>If the current capacity of this vector is less than
217      * {@code minCapacity}, then its capacity is increased by replacing its
218      * internal data array, kept in the field {@code elementData}, with a
219      * larger one.  The size of the new data array will be the old size plus
220      * {@code capacityIncrement}, unless the value of
221      * {@code capacityIncrement} is less than or equal to zero, in which case
222      * the new capacity will be twice the old capacity; but if this new size
223      * is still smaller than {@code minCapacity}, then the new capacity will
224      * be {@code minCapacity}.
225      *
226      * @param minCapacity the desired minimum capacity
227      */
228     public synchronized void ensureCapacity(int minCapacity) {
229         if (minCapacity > 0) {
230             modCount++;
231             ensureCapacityHelper(minCapacity);
232         }
233     }
234 
235     /**
236      * This implements the unsynchronized semantics of ensureCapacity.
237      * Synchronized methods in this class can internally call this
238      * method for ensuring capacity without incurring the cost of an
239      * extra synchronization.
240      *
241      * @see #ensureCapacity(int)
242      */
243     private void ensureCapacityHelper(int minCapacity) {
244         // overflow-conscious code
245         if (minCapacity - elementData.length > 0)
246             grow(minCapacity);
247     }
248 
249     /**
250      * The maximum size of array to allocate.
251      * Some VMs reserve some header words in an array.
252      * Attempts to allocate larger arrays may result in
253      * OutOfMemoryError: Requested array size exceeds VM limit
254      */
255     private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
256 
257     private void grow(int minCapacity) {
258         // overflow-conscious code
259         int oldCapacity = elementData.length;
260         int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
261                                          capacityIncrement : oldCapacity);
262         if (newCapacity - minCapacity < 0)
263             newCapacity = minCapacity;
264         if (newCapacity - MAX_ARRAY_SIZE > 0)
265             newCapacity = hugeCapacity(minCapacity);
266         elementData = Arrays.copyOf(elementData, newCapacity);
267     }
268 
```

```
269     private static int hugeCapacity(int minCapacity) {
270         if (minCapacity < 0) // overflow
271             throw new OutOfMemoryError();
272         return (minCapacity > MAX_ARRAY_SIZE) ?
273             Integer.MAX_VALUE :
274             MAX_ARRAY_SIZE;
275     }
276
277     /**
278      * Sets the size of this vector. If the new size is greater than the
279      * current size, new {@code null} items are added to the end of
280      * the vector. If the new size is less than the current size, all
281      * components at index {@code newSize} and greater are discarded.
282      *
283      * @param  newSize   the new size of this vector
284      * @throws ArrayIndexOutOfBoundsException if the new size is negative
285      */
286     public synchronized void setSize(int newSize) {
287         modCount++;
288         if (newSize > elementCount) {
289             ensureCapacityHelper(newSize);
290         } else {
291             for (int i = newSize ; i < elementCount ; i++) {
292                 elementData[i] = null;
293             }
294         }
295         elementCount = newSize;
296     }
297
298     /**
299      * Returns the current capacity of this vector.
300      *
301      * @return  the current capacity (the length of its internal
302      *          data array, kept in the field {@code elementData}
303      *          of this vector)
304      */
305     public synchronized int capacity() {
306         return elementData.length;
307     }
308
309     /**
310      * Returns the number of components in this vector.
311      *
312      * @return  the number of components in this vector
313      */
314     public synchronized int size() {
315         return elementCount;
316     }
317
318     /**
319      * Tests if this vector has no components.
320      *
321      * @return  {@code true} if and only if this vector has
322      *          no components, that is, its size is zero;
323      *          {@code false} otherwise.
324      */
325     public synchronized boolean isEmpty() {
326         return elementCount == 0;
327     }
328
329     /**
330      * Returns an enumeration of the components of this vector. The
331      * returned {@code Enumeration} object will generate all items in
332      * this vector. The first item generated is the item at index {@code 0},
333      * then the item at index {@code 1}, and so on.
334      *
335      * @return  an enumeration of the components of this vector
```

```
336         * @see     Iterator
337         */
338        public Enumeration<E> elements() {
339            return new Enumeration<E>() {
340                int count = 0;
341
342                public boolean hasMoreElements() {
343                    return count < elementCount;
344                }
345
346                public E nextElement() {
347                    synchronized (Vector.this) {
348                        if (count < elementCount) {
349                            return elementData(count++);
350                        }
351                    }
352                    throw new NoSuchElementException("Vector Enumeration");
353                }
354            };
355        }
356
357        /**
358         * Returns {@code true} if this vector contains the specified element.
359         * More formally, returns {@code true} if and only if this vector
360         * contains at least one element {@code e} such that
361         * <tt>(o==null ? e==null : o.equals(e))</tt>.
362         *
363         * @param o element whose presence in this vector is to be tested
364         * @return {@code true} if this vector contains the specified element
365         */
366        public boolean contains(Object o) {
367            return indexOf(o, 0) >= 0;
368        }
369
370        /**
371         * Returns the index of the first occurrence of the specified element
372         * in this vector, or -1 if this vector does not contain the element.
373         * More formally, returns the lowest index {@code i} such that
374         * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>,
375         * or -1 if there is no such index.
376         *
377         * @param o element to search for
378         * @return the index of the first occurrence of the specified element in
379         *         this vector, or -1 if this vector does not contain the element
380         */
381        public int indexOf(Object o) {
382            return indexOf(o, 0);
383        }
384
385        /**
386         * Returns the index of the first occurrence of the specified element in
387         * this vector, searching forwards from {@code index}, or returns -1 if
388         * the element is not found.
389         * More formally, returns the lowest index {@code i} such that
390         *
         <tt>(i &gt;= index &amp;&amp; (o==null ? get(i)==null 
391         * or -1 if there is no such index.
392         *
393         * @param o element to search for
394         * @param index index to start searching from
395         * @return the index of the first occurrence of the element in
396         *         this vector at position {@code index} or later in the vector;
397         *         {@code -1} if the element is not found.
398         * @throws IndexOutOfBoundsException if the specified index is negative
399         * @see     Object#equals(Object)
400         */
401        public synchronized int indexOf(Object o, int index) {
```

```
402         if (o == null) {
403             for (int i = index ; i < elementCount ; i++)
404                 if (elementData[i]==null)
405                     return i;
406         } else {
407             for (int i = index ; i < elementCount ; i++)
408                 if (o.equals(elementData[i]))
409                     return i;
410         }
411         return -1;
412     }
413 
414     /**
415      * Returns the index of the last occurrence of the specified element
416      * in this vector, or -1 if this vector does not contain the element.
417      * More formally, returns the highest index {@code i} such that
418      * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>,
419      * or -1 if there is no such index.
420      *
421      * @param o element to search for
422      * @return the index of the last occurrence of the specified element in
423      *         this vector, or -1 if this vector does not contain the element
424      */
425     public synchronized int lastIndexOf(Object o) {
426         return lastIndexOf(o, elementCount-1);
427     }
428 
429     /**
430      * Returns the index of the last occurrence of the specified element in
431      * this vector, searching backwards from {@code index}, or returns -1 if
432      * the element is not found.
433      * More formally, returns the highest index {@code i} such that
434      *
      <tt>(i &lt;= index &amp;&amp; (o==null ? get(i)==null 
435      * or -1 if there is no such index.
436      *
437      * @param o element to search for
438      * @param index index to start searching backwards from
439      * @return the index of the last occurrence of the element at position
440      *         less than or equal to {@code index} in this vector;
441      *         -1 if the element is not found.
442      * @throws IndexOutOfBoundsException if the specified index is greater
443      *         than or equal to the current size of this vector
444      */
445     public synchronized int lastIndexOf(Object o, int index) {
446         if (index >= elementCount)
447             throw new IndexOutOfBoundsException(index + " >= "+ elementCount);
448 
449         if (o == null) {
450             for (int i = index; i >= 0; i--)
451                 if (elementData[i]==null)
452                     return i;
453         } else {
454             for (int i = index; i >= 0; i--)
455                 if (o.equals(elementData[i]))
456                     return i;
457         }
458         return -1;
459     }
460 
461     /**
462      * Returns the component at the specified index.
463      *
464      * <p>This method is identical in functionality to the {@link #get(int)}
465      * method (which is part of the {@link List} interface).
466      *
467      * @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</code>.
     * @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;
    }
```

```
535
536     /**
537      * Deletes the component at the specified index. Each component in
538      * this vector with an index greater or equal to the specified
539      * {@code index} is shifted downward to have an index one
540      * smaller than the value it had previously. The size of this vector
541      * is decreased by {@code 1}.
542      *
543      * <p>The index must be a value greater than or equal to {@code 0}
544      * and less than the current size of the vector.
545      *
546      * <p>This method is identical in functionality to the {@link #remove(int)}
547      * method (which is part of the {@link List} interface).  Note that the
548      * {@code remove} method returns the old value that was stored at the
549      * specified position.
550      *
551      * @param      index   the index of the object to remove
552      * @throws ArrayIndexOutOfBoundsException if the index is out of range
553      *         ({@code index < 0 || index >= size()})
554      */
555     public synchronized void removeElementAt(int index) {
556         modCount++;
557         if (index >= elementCount) {
558             throw new ArrayIndexOutOfBoundsException(index + " >= " +
559                                                      elementCount);
560         }
561         else if (index < 0) {
562             throw new ArrayIndexOutOfBoundsException(index);
563         }
564         int j = elementCount - index - 1;
565         if (j > 0) {
566             System.arraycopy(elementData, index + 1, elementData, index, j);
567         }
568         elementCount--;
569         elementData[elementCount] = null; /* to let gc do its work */
570     }
571
572     /**
573      * Inserts the specified object as a component in this vector at the
574      * specified {@code index}. Each component in this vector with
575      * an index greater or equal to the specified {@code index} is
576      * shifted upward to have an index one greater than the value it had
577      * previously.
578      *
579      * <p>The index must be a value greater than or equal to {@code 0}
580      * and less than or equal to the current size of the vector. (If the
581      * index is equal to the current size of the vector, the new element
582      * is appended to the Vector.)
583      *
584      * <p>This method is identical in functionality to the
585      * {@link #add(int, Object) add(int, E)}
586      * method (which is part of the {@link List} interface).  Note that the
587      * {@code add} method reverses the order of the parameters, to more closely
588      * match array usage.
589      *
590      * @param      obj     the component to insert
591      * @param      index   where to insert the new component
592      * @throws ArrayIndexOutOfBoundsException if the index is out of range
593      *         ({@code index < 0 || index > size()})
594      */
595     public synchronized void insertElementAt(E obj, int index) {
596         modCount++;
597         if (index > elementCount) {
598             throw new ArrayIndexOutOfBoundsException(index
599                                                      + " > " + elementCount);
600         }
601         ensureCapacityHelper(elementCount + 1);
```

```
602             System.arraycopy(elementData, index, elementData, index + 1, elementCount - index);
603             elementData[index] = obj;
604             elementCount++;
605         }
606 
607         /**
608          * Adds the specified component to the end of this vector,
609          * increasing its size by one. The capacity of this vector is
610          * increased if its size becomes greater than its capacity.
611          *
612          * <p>This method is identical in functionality to the
613          * {@link #add(Object) add(E)}
614          * method (which is part of the {@link List} interface).
615          *
616          * @param   obj   the component to be added
617          */
618         public synchronized void addElement(E obj) {
619             modCount++;
620             ensureCapacityHelper(elementCount + 1);
621             elementData[elementCount++] = obj;
622         }
623 
624         /**
625          * Removes the first (lowest-indexed) occurrence of the argument
626          * from this vector. If the object is found in this vector, each
627          * component in the vector with an index greater or equal to the
628          * object's index is shifted downward to have an index one smaller
629          * than the value it had previously.
630          *
631          * <p>This method is identical in functionality to the
632          * {@link #remove(Object)} method (which is part of the
633          * {@link List} interface).
634          *
635          * @param   obj   the component to be removed
636          * @return  {@code true} if the argument was a component of this
637          *          vector; {@code false} otherwise.
638          */
639         public synchronized boolean removeElement(Object obj) {
640             modCount++;
641             int i = indexOf(obj);
642             if (i >= 0) {
643                 removeElementAt(i);
644                 return true;
645             }
646             return false;
647         }
648 
649         /**
650          * Removes all components from this vector and sets its size to zero.
651          *
652          * <p>This method is identical in functionality to the {@link #clear}
653          * method (which is part of the {@link List} interface).
654          */
655         public synchronized void removeAllElements() {
656             modCount++;
657             // Let gc do its work
658             for (int i = 0; i < elementCount; i++)
659                 elementData[i] = null;
660 
661             elementCount = 0;
662         }
663 
664         /**
665          * Returns a clone of this vector. The copy will contain a
666          * reference to a clone of the internal data array, not a reference
667          * to the original internal data array of this {@code Vector} object.
668          *
```

```
     * @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 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 is not a supertype
     * of the runtime type 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];
    }
```

```
736
737     /**
738      * Returns the element at the specified position in this Vector.
739      *
740      * @param index index of the element to return
741      * @return object at the specified index
742      * @throws ArrayIndexOutOfBoundsException if the index is out of range
743      *            ({@code index < 0 || index >= size()})
744      * @since 1.2
745      */
746     public synchronized E get(int index) {
747         if (index >= elementCount)
748             throw new ArrayIndexOutOfBoundsException(index);
749
750         return elementData(index);
751     }
752
753     /**
754      * Replaces the element at the specified position in this Vector with the
755      * specified element.
756      *
757      * @param index index of the element to replace
758      * @param element element to be stored at the specified position
759      * @return the element previously at the specified position
760      * @throws ArrayIndexOutOfBoundsException if the index is out of range
761      *         ({@code index < 0 || index >= size()})
762      * @since 1.2
763      */
764     public synchronized E set(int index, E element) {
765         if (index >= elementCount)
766             throw new ArrayIndexOutOfBoundsException(index);
767
768         E oldValue = elementData(index);
769         elementData[index] = element;
770         return oldValue;
771     }
772
773     /**
774      * Appends the specified element to the end of this Vector.
775      *
776      * @param e element to be appended to this Vector
777      * @return {@code true} (as specified by {@link Collection#add})
778      * @since 1.2
779      */
780     public synchronized boolean add(E e) {
781         modCount++;
782         ensureCapacityHelper(elementCount + 1);
783         elementData[elementCount++] = e;
784         return true;
785     }
786
787     /**
788      * Removes the first occurrence of the specified element in this Vector
789      * If the Vector does not contain the element, it is unchanged.  More
790      * formally, removes the element with the lowest index i such that
791      * {@code (o==null ? get(i)==null : o.equals(get(i)))} (if such
792      * an element exists).
793      *
794      * @param o element to be removed from this Vector, if present
795      * @return true if the Vector contained the specified element
796      * @since 1.2
797      */
798     public boolean remove(Object o) {
799         return removeElement(o);
800     }
801
802     /**
```

```
803         * Inserts the specified element at the specified position in this Vector.
804         * Shifts the element currently at that position (if any) and any
805         * subsequent elements to the right (adds one to their indices).
806         *
807         * @param index index at which the specified element is to be inserted
808         * @param element element to be inserted
809         * @throws ArrayIndexOutOfBoundsException if the index is out of range
810         *         ({@code index < 0 || index > size()})
811         * @since 1.2
812         */
813        public void add(int index, E element) {
814            insertElementAt(element, index);
815        }
816
817        /**
818         * Removes the element at the specified position in this Vector.
819         * Shifts any subsequent elements to the left (subtracts one from their
820         * indices).  Returns the element that was removed from the Vector.
821         *
822         * @throws ArrayIndexOutOfBoundsException if the index is out of range
823         *         ({@code index < 0 || index >= size()})
824         * @param index the index of the element to be removed
825         * @return element that was removed
826         * @since 1.2
827         */
828        public synchronized E remove(int index) {
829            modCount++;
830            if (index >= elementCount)
831                throw new ArrayIndexOutOfBoundsException(index);
832            E oldValue = elementData(index);
833
834            int numMoved = elementCount - index - 1;
835            if (numMoved > 0)
836                System.arraycopy(elementData, index+1, elementData, index,
837                                 numMoved);
838            elementData[--elementCount] = null; // Let gc do its work
839
840            return oldValue;
841        }
842
843        /**
844         * Removes all of the elements from this Vector.  The Vector will
845         * be empty after this call returns (unless it throws an exception).
846         *
847         * @since 1.2
848         */
849        public void clear() {
850            removeAllElements();
851        }
852
853        // Bulk Operations
854
855        /**
856         * Returns true if this Vector contains all of the elements in the
857         * specified Collection.
858         *
859         * @param   c a collection whose elements will be tested for containment
860         *          in this Vector
861         * @return true if this Vector contains all of the elements in the
862         *         specified collection
863         * @throws NullPointerException if the specified collection is null
864         */
865        public synchronized boolean containsAll(Collection<?> c) {
866            return super.containsAll(c);
867        }
868
869        /**
```

```
870         * Appends all of the elements in the specified Collection to the end of
871         * this Vector, in the order that they are returned by the specified
872         * Collection's Iterator.  The behavior of this operation is undefined if
873         * the specified Collection is modified while the operation is in progress.
874         * (This implies that the behavior of this call is undefined if the
875         * specified Collection is this Vector, and this Vector is nonempty.)
876         *
877         * @param c elements to be inserted into this Vector
878         * @return {@code true} if this Vector changed as a result of the call
879         * @throws NullPointerException if the specified collection is null
880         * @since 1.2
881         */
882        public synchronized boolean addAll(Collection<? extends E> c) {
883            modCount++;
884            Object[] a = c.toArray();
885            int numNew = a.length;
886            ensureCapacityHelper(elementCount + numNew);
887            System.arraycopy(a, 0, elementData, elementCount, numNew);
888            elementCount += numNew;
889            return numNew != 0;
890        }
891
892        /**
893         * Removes from this Vector all of its elements that are contained in the
894         * specified Collection.
895         *
896         * @param c a collection of elements to be removed from the Vector
897         * @return true if this Vector changed as a result of the call
898         * @throws ClassCastException if the types of one or more elements
899         *         in this vector are incompatible with the specified
900         *         collection
901         * (<a href="Collection.html#optional-restrictions">optional</a>)
902         * @throws NullPointerException if this vector contains one or more null
903         *         elements and the specified collection does not support null
904         *         elements
905         * (<a href="Collection.html#optional-restrictions">optional</a>),
906         *         or if the specified collection is null
907         * @since 1.2
908         */
909        public synchronized boolean removeAll(Collection<?> c) {
910            return super.removeAll(c);
911        }
912
913        /**
914         * Retains only the elements in this Vector that are contained in the
915         * specified Collection.  In other words, removes from this Vector all
916         * of its elements that are not contained in the specified Collection.
917         *
918         * @param c a collection of elements to be retained in this Vector
919         *          (all other elements are removed)
920         * @return true if this Vector changed as a result of the call
921         * @throws ClassCastException if the types of one or more elements
922         *         in this vector are incompatible with the specified
923         *         collection
924         * (<a href="Collection.html#optional-restrictions">optional</a>)
925         * @throws NullPointerException if this vector contains one or more null
926         *         elements and the specified collection does not support null
927         *         elements
928         *         (<a href="Collection.html#optional-restrictions">optional</a>),
929         *         or if the specified collection is null
930         * @since 1.2
931         */
932        public synchronized boolean retainAll(Collection<?> c) {
933            return super.retainAll(c);
934        }
935
936        /**
```

```
937         * Inserts all of the elements in the specified Collection into this
938         * Vector at the specified position.  Shifts the element currently at
939         * that position (if any) and any subsequent elements to the right
940         * (increases their indices).  The new elements will appear in the Vector
941         * in the order that they are returned by the specified Collection's
942         * iterator.
943         *
944         * @param index index at which to insert the first element from the
945         *              specified collection
946         * @param c elements to be inserted into this Vector
947         * @return {@code true} if this Vector changed as a result of the call
948         * @throws ArrayIndexOutOfBoundsException if the index is out of range
949         *         ({@code index < 0 || index > size()})
950         * @throws NullPointerException if the specified collection is null
951         * @since 1.2
952         */
953        public synchronized boolean addAll(int index, Collection<? extends E> c) {
954            modCount++;
955            if (index < 0 || index > elementCount)
956                throw new ArrayIndexOutOfBoundsException(index);
957
958            Object[] a = c.toArray();
959            int numNew = a.length;
960            ensureCapacityHelper(elementCount + numNew);
961
962            int numMoved = elementCount - index;
963            if (numMoved > 0)
964                System.arraycopy(elementData, index, elementData, index + numNew,
965                                 numMoved);
966
967            System.arraycopy(a, 0, elementData, index, numNew);
968            elementCount += numNew;
969            return numNew != 0;
970        }
971
972        /**
973         * Compares the specified Object with this Vector for equality.  Returns
974         * true if and only if the specified Object is also a List, both Lists
975         * have the same size, and all corresponding pairs of elements in the two
976         * Lists are <em>equal</em>.  (Two elements {@code e1} and
977         * {@code e2} are <em>equal</em> if {@code (e1==null ? e2==null :
978         * e1.equals(e2))}.)  In other words, two Lists are defined to be
979         * equal if they contain the same elements in the same order.
980         *
981         * @param o the Object to be compared for equality with this Vector
982         * @return true if the specified Object is equal to this Vector
983         */
984        public synchronized boolean equals(Object o) {
985            return super.equals(o);
986        }
987
988        /**
989         * Returns the hash code value for this Vector.
990         */
991        public synchronized int hashCode() {
992            return super.hashCode();
993        }
994
995        /**
996         * Returns a string representation of this Vector, containing
997         * the String representation of each element.
998         */
999        public synchronized String toString() {
1000           return super.toString();
1001       }
1002
1003       /**
```

```
1004     * Returns a view of the portion of this List between fromIndex,
1005     * inclusive, and toIndex, exclusive.  (If fromIndex and toIndex are
1006     * equal, the returned List is empty.)  The returned List is backed by this
1007     * List, so changes in the returned List are reflected in this List, and
1008     * vice-versa.  The returned List supports all of the optional List
1009     * operations supported by this List.
1010     *
1011     * <p>This method eliminates the need for explicit range operations (of
1012     * the sort that commonly exist for arrays).  Any operation that expects
1013     * a List can be used as a range operation by operating on a subList view
1014     * instead of a whole List.  For example, the following idiom
1015     * removes a range of elements from a List:
1016     * <pre>
1017     *      list.subList(from, to).clear();
1018     * </pre>
1019     * Similar idioms may be constructed for indexOf and lastIndexOf,
1020     * and all of the algorithms in the Collections class can be applied to
1021     * a subList.
1022     *
1023     * <p>The semantics of the List returned by this method become undefined if
1024     * the backing list (i.e., this List) is <i>structurally modified</i> in
1025     * any way other than via the returned List.  (Structural modifications are
1026     * those that change the size of the List, or otherwise perturb it in such
1027     * a fashion that iterations in progress may yield incorrect results.)
1028     *
1029     * @param fromIndex low endpoint (inclusive) of the subList
1030     * @param toIndex high endpoint (exclusive) of the subList
1031     * @return a view of the specified range within this List
1032     * @throws IndexOutOfBoundsException if an endpoint index value is out of range
1033     *         {@code (fromIndex < 0 || toIndex > size)}
1034     * @throws IllegalArgumentException if the endpoint indices are out of order
1035     *         {@code (fromIndex > toIndex)}
1036     */
1037    public synchronized List<E> subList(int fromIndex, int toIndex) {
1038        return Collections.synchronizedList(super.subList(fromIndex, toIndex),
1039                                            this);
1040    }
1041
1042    /**
1043     * Removes from this list all of the elements whose index is between
1044     * {@code fromIndex}, inclusive, and {@code toIndex}, exclusive.
1045     * Shifts any succeeding elements to the left (reduces their index).
1046     * This call shortens the list by {@code (toIndex - fromIndex)} elements.
1047     * (If {@code toIndex==fromIndex}, this operation has no effect.)
1048     */
1049    protected synchronized void removeRange(int fromIndex, int toIndex) {
1050        modCount++;
1051        int numMoved = elementCount - toIndex;
1052        System.arraycopy(elementData, toIndex, elementData, fromIndex,
1053                         numMoved);
1054
1055        // Let gc do its work
1056        int newElementCount = elementCount - (toIndex-fromIndex);
1057        while (elementCount != newElementCount)
1058            elementData[--elementCount] = null;
1059    }
1060
1061    /**
1062     * Save the state of the {@code Vector} instance to a stream (that
1063     * is, serialize it).
1064     * This method performs synchronization to ensure the consistency
1065     * of the serialized data.
1066     */
1067    private void writeObject(java.io.ObjectOutputStream s)
1068            throws java.io.IOException {
1069        final java.io.ObjectOutputStream.PutField fields = s.putFields();
1070        final Object[] data;
```

```
1071            synchronized (this) {
1072                fields.put("capacityIncrement", capacityIncrement);
1073                fields.put("elementCount", elementCount);
1074                data = elementData.clone();
1075            }
1076            fields.put("elementData", data);
1077            s.writeFields();
1078        }
1079
1080        /**
1081         * Returns a list iterator over the elements in this list (in proper
1082         * sequence), starting at the specified position in the list.
1083         * The specified index indicates the first element that would be
1084         * returned by an initial call to {@link ListIterator#next next}.
1085         * An initial call to {@link ListIterator#previous previous} would
1086         * return the element with the specified index minus one.
1087         *
1088         * <p>The returned list iterator is <a href="#fail-fast"><i>fail-fast</i></a>.
1089         *
1090         * @throws IndexOutOfBoundsException {@inheritDoc}
1091         */
1092        public synchronized ListIterator<E> listIterator(int index) {
1093            if (index < 0 || index > elementCount)
1094                throw new IndexOutOfBoundsException("Index: "+index);
1095            return new ListItr(index);
1096        }
1097
1098        /**
1099         * Returns a list iterator over the elements in this list (in proper
1100         * sequence).
1101         *
1102         * <p>The returned list iterator is <a href="#fail-fast"><i>fail-fast</i></a>.
1103         *
1104         * @see #listIterator(int)
1105         */
1106        public synchronized ListIterator<E> listIterator() {
1107            return new ListItr(0);
1108        }
1109
1110        /**
1111         * Returns an iterator over the elements in this list in proper sequence.
1112         *
1113         * <p>The returned iterator is <a href="#fail-fast"><i>fail-fast</i></a>.
1114         *
1115         * @return an iterator over the elements in this list in proper sequence
1116         */
1117        public synchronized Iterator<E> iterator() {
1118            return new Itr();
1119        }
1120
1121        /**
1122         * An optimized version of AbstractList.Itr
1123         */
1124        private class Itr implements Iterator<E> {
1125            int cursor;       // index of next element to return
1126            int lastRet = -1; // index of last element returned; -1 if no such
1127            int expectedModCount = modCount;
1128
1129            public boolean hasNext() {
1130                // Racy but within spec, since modifications are checked
1131                // within or after synchronization in next/previous
1132                return cursor != elementCount;
1133            }
1134
1135            public E next() {
1136                synchronized (Vector.this) {
1137                    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;
                }
        @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;
        }
```

```
1205        public int previousIndex() {
1206            return cursor - 1;
1207        }
1208
1209        public E previous() {
1210            synchronized (Vector.this) {
1211                checkForComodification();
1212                int i = cursor - 1;
1213                if (i < 0)
1214                    throw new NoSuchElementException();
1215                cursor = i;
1216                return elementData(lastRet = i);
1217            }
1218        }
1219
1220        public void set(E e) {
1221            if (lastRet == -1)
1222                throw new IllegalStateException();
1223            synchronized (Vector.this) {
1224                checkForComodification();
1225                Vector.this.set(lastRet, e);
1226            }
1227        }
1228
1229        public void add(E e) {
1230            int i = cursor;
1231            synchronized (Vector.this) {
1232                checkForComodification();
1233                Vector.this.add(i, e);
1234                expectedModCount = modCount;
1235            }
1236            cursor = i + 1;
1237            lastRet = -1;
1238        }
1239    }
1240
1241    @Override
1242    public synchronized void forEach(Consumer<? super E> action) {
1243        Objects.requireNonNull(action);
1244        final int expectedModCount = modCount;
1245        @SuppressWarnings("unchecked")
1246        final E[] elementData = (E[]) this.elementData;
1247        final int elementCount = this.elementCount;
1248        for (int i=0; modCount == expectedModCount && i < elementCount; i++) {
1249            action.accept(elementData[i]);
1250        }
1251        if (modCount != expectedModCount) {
1252            throw new ConcurrentModificationException();
1253        }
1254    }
1255
1256    @Override
1257    @SuppressWarnings("unchecked")
1258    public synchronized boolean removeIf(Predicate<? super E> filter) {
1259        Objects.requireNonNull(filter);
1260        // figure out which elements are to be removed
1261        // any exception thrown from the filter predicate at this stage
1262        // will leave the collection unmodified
1263        int removeCount = 0;
1264        final int size = elementCount;
1265        final BitSet removeSet = new BitSet(size);
1266        final int expectedModCount = modCount;
1267        for (int i=0; modCount == expectedModCount && i < size; i++) {
1268            @SuppressWarnings("unchecked")
1269            final E element = (E) elementData[i];
1270            if (filter.test(element)) {
1271                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
    @SuppressWarnings("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++;
    }

    @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
     */
```

```
1339    @Override
1340    public Spliterator<E> spliterator() {
1341        return new VectorSpliterator<>(this, null, 0, -1, 0);
1342    }
1343
1344    /** Similar to ArrayList Spliterator */
1345    static final class VectorSpliterator<E> implements Spliterator<E> {
1346        private final Vector<E> list;
1347        private Object[] array;
1348        private int index; // current index, modified on advance/split
1349        private int fence; // -1 until used; then one past last index
1350        private int expectedModCount; // initialized when fence set
1351
1352        /** Create new spliterator covering the given  range */
1353        VectorSpliterator(Vector<E> list, Object[] array, int origin, int fence,
1354                          int expectedModCount) {
1355            this.list = list;
1356            this.array = array;
1357            this.index = origin;
1358            this.fence = fence;
1359            this.expectedModCount = expectedModCount;
1360        }
1361
1362        private int getFence() { // initialize on first use
1363            int hi;
1364            if ((hi = fence) < 0) {
1365                synchronized(list) {
1366                    array = list.elementData;
1367                    expectedModCount = list.modCount;
1368                    hi = fence = list.elementCount;
1369                }
1370            }
1371            return hi;
1372        }
1373
1374        public Spliterator<E> trySplit() {
1375            int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
1376            return (lo >= mid) ? null :
1377                new VectorSpliterator<E>(list, array, lo, index = mid,
1378                                         expectedModCount);
1379        }
1380
1381        @SuppressWarnings("unchecked")
1382        public boolean tryAdvance(Consumer<? super E> action) {
1383            int i;
1384            if (action == null)
1385                throw new NullPointerException();
1386            if (getFence() > (i = index)) {
1387                index = i + 1;
1388                action.accept((E)array[i]);
1389                if (list.modCount != expectedModCount)
1390                    throw new ConcurrentModificationException();
1391                return true;
1392            }
1393            return false;
1394        }
1395
1396        @SuppressWarnings("unchecked")
1397        public void forEachRemaining(Consumer<? super E> action) {
1398            int i, hi; // hoist accesses and checks from loop
1399            Vector<E> lst; Object[] a;
1400            if (action == null)
1401                throw new NullPointerException();
1402            if ((lst = list) != null) {
1403                if ((hi = fence) < 0) {
1404                    synchronized(lst) {
1405                        expectedModCount = lst.modCount;
```

```
1406                        a = array = lst.elementData;
1407                        hi = fence = lst.elementCount;
1408                    }
1409                }
1410                else
1411                    a = array;
1412                if (a != null && (i = index) >= 0 && (index = hi) <= a.length) {
1413                    while (i < hi)
1414                        action.accept((E) a[i++]);
1415                    if (lst.modCount == expectedModCount)
1416                        return;
1417                }
1418            }
1419            throw new ConcurrentModificationException();
1420        }
1421
1422        public long estimateSize() {
1423            return (long) (getFence() - index);
1424        }
1425
1426        public int characteristics() {
1427            return Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;
1428        }
1429    }
1430 }
1431
```