Código fonte em PDF

Collections

Aqui como incluir apenas um trecho do código fonte, especificando linha de início e de fim.

```
    private static <T>
    int indexedBinarySearch(List<? extends Comparable<? super T>> list, T key)
    {
        int low = 0;
        int high = list.size()-1;

        while (low <= high) {
            int mid = (low + high) >>> 1;
            Comparable<? super T> midVal = list.get(mid);
            int cmp = midVal.compareTo(key);

            if (cmp < 0)
                low = mid + 1;
            else if (cmp > 0)
                high = mid - 1;
            else
                return mid; // key found
        }
        return -(low + 1);  // key not found
    }
```

Desta vez, inclui o mesmo trecho, e adiciona a numeração de linhas.

```
264     private static <T>
265     int indexedBinarySearch(List<? extends Comparable<? super T>> list, T key)
266     {
267         int low = 0;
268         int high = list.size()-1;
269 
270         while (low <= high) {
271             int mid = (low + high) >>> 1;
272             Comparable<? super T> midVal = list.get(mid);
273             int cmp = midVal.compareTo(key);
274 
275             if (cmp < 0)
276                 low = mid + 1;
277             else if (cmp > 0)
278                 high = mid - 1;
279             else
280                 return mid; // key found
281         }
282         return -(low + 1);  // key not found
283     }
```

LinkedList

Por fim, colocar o arquivo completo é o mais simples. Contudo, é importante olhar como resolver o caso de linhas muito longas.

```
 1 /*
 2  * Copyright (c) 1997, 2012, Oracle and/or its affiliates. All rights reserved.
 3  * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 4  *
 5  * This code is free software; you can redistribute it and/or modify it
 6  * under the terms of the GNU General Public License version 2 only, as
 7  * published by the Free Software Foundation.  Oracle designates this
 8  * particular file as subject to the "Classpath" exception as provided
 9  * by Oracle in the LICENSE file that accompanied this code.
10  *
11  * This code is distributed in the hope that it will be useful, but WITHOUT
12  * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
13  * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
14  * 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.io.Serializable;
import java.io.ObjectOutputStream;
import java.io.IOException;
import java.lang.reflect.Array;

/**
 * This class consists exclusively of static methods that operate on or return
 * collections.  It contains polymorphic algorithms that operate on
 * collections, "wrappers", which return a new collection backed by a
 * specified collection, and a few other odds and ends.
 *
 * <p>The methods of this class all throw a <tt>NullPointerException</tt>
 * if the collections or class objects provided to them are null.
 *
 * <p>The documentation for the polymorphic algorithms contained in this class
 * generally includes a brief description of the <i>implementation</i>.  Such
 * descriptions should be regarded as <i>implementation notes</i>, rather than
 * parts of the <i>specification</i>.  Implementors should feel free to
 * substitute other algorithms, so long as the specification itself is adhered
 * to.  (For example, the algorithm used by <tt>sort</tt> does not have to be
 * a mergesort, but it does have to be <i>stable</i>.)
 *
 * <p>The "destructive" algorithms contained in this class, that is, the
 * algorithms that modify the collection on which they operate, are specified
 * to throw <tt>UnsupportedOperationException</tt> if the collection does not
 * support the appropriate mutation primitive(s), such as the <tt>set</tt>
 * method.  These algorithms may, but are not required to, throw this
 * exception if an invocation would have no effect on the collection.  For
 * example, invoking the <tt>sort</tt> method on an unmodifiable list that is
 * already sorted may or may not throw <tt>UnsupportedOperationException</tt>.
 *
 * <p>This class is a member of the
 * <a href="{@docRoot}/../technotes/guides/collections/index.html">
 * Java Collections Framework</a>.
 *
 * @author  Josh Bloch
 * @author  Neal Gafter
 * @see     Collection
 * @see     Set
 * @see     List
 * @see     Map
 * @since   1.2
 */

public class Collections {
    // Suppresses default constructor, ensuring non-instantiability.
    private Collections() {
    }

    // Algorithms

    /*
     * Tuning parameters for algorithms - Many of the List algorithms have
     * two implementations, one of which is appropriate for RandomAccess
     * lists, the other for "sequential."  Often, the random access variant
     * yields better performance on small sequential access lists.  The
     * tuning parameters below determine the cutoff point for what constitutes
     * a "small" sequential access list for each algorithm.  The values below
     * were empirically determined to work well for LinkedList. Hopefully
     * they should be reasonable for other sequential access List
     * implementations.  Those doing performance work on this code would
     * do well to validate the values of these parameters from time to time.
```

```
     * (The first word of each tuning parameter name is the algorithm to which
     * it applies.)
     */
    private static final int BINARYSEARCH_THRESHOLD   = 5000;
    private static final int REVERSE_THRESHOLD        =   18;
    private static final int SHUFFLE_THRESHOLD        =    5;
    private static final int FILL_THRESHOLD           =   25;
    private static final int ROTATE_THRESHOLD         =  100;
    private static final int COPY_THRESHOLD           =   10;
    private static final int REPLACEALL_THRESHOLD     =   11;
    private static final int INDEXOFSUBLIST_THRESHOLD =   35;

    /**
     * Sorts the specified list into ascending order, according to the
     * {@linkplain Comparable natural ordering} of its elements.
     * All elements in the list must implement the {@link Comparable}
     * interface.  Furthermore, all elements in the list must be
     * <i>mutually comparable</i> (that is, {@code e1.compareTo(e2)}
     * must not throw a {@code ClassCastException} for any elements
     * {@code e1} and {@code e2} in the list).
     *
     * <p>This sort is guaranteed to be <i>stable</i>:  equal elements will
     * not be reordered as a result of the sort.
     *
     * <p>The specified list must be modifiable, but need not be resizable.
     *
     * <p>Implementation note: This implementation is a stable, adaptive,
     * iterative mergesort that requires far fewer than n lg(n) comparisons
     * when the input array is partially sorted, while offering the
     * performance of a traditional mergesort when the input array is
     * randomly ordered.  If the input array is nearly sorted, the
     * implementation requires approximately n comparisons.  Temporary
     * storage requirements vary from a small constant for nearly sorted
     * input arrays to n/2 object references for randomly ordered input
     * arrays.
     *
     * <p>The implementation takes equal advantage of ascending and
     * descending order in its input array, and can take advantage of
     * ascending and descending order in different parts of the same
     * input array.  It is well-suited to merging two or more sorted arrays:
     * simply concatenate the arrays and sort the resulting array.
     *
     * <p>The implementation was adapted from Tim Peters's list sort for Python
     * (<a href="http://svn.python.org/projects/python/trunk/Objects/listsort.txt">
     * TimSort</a>).  It uses techiques from Peter McIlroy's "Optimistic
     * Sorting and Information Theoretic Complexity", in Proceedings of the
     * Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474,
     * January 1993.
     *
     * <p>This implementation dumps the specified list into an array, sorts
     * the array, and iterates over the list resetting each element
     * from the corresponding position in the array.  This avoids the
     * n<sup>2</sup> log(n) performance that would result from attempting
     * to sort a linked list in place.
     *
     * @param  list the list to be sorted.
     * @throws ClassCastException if the list contains elements that are not
     *         <i>mutually comparable</i> (for example, strings and integers).
     * @throws UnsupportedOperationException if the specified list's
     *         list-iterator does not support the {@code set} operation.
     * @throws IllegalArgumentException (optional) if the implementation
     *         detects that the natural ordering of the list elements is
     *         found to violate the {@link Comparable} contract
     */
    public static <T extends Comparable<? super T>> void sort(List<T> list) {
        Object[] a = list.toArray();
        Arrays.sort(a);
        ListIterator<T> i = list.listIterator();
        for (int j=0; j<a.length; j++) {
            i.next();
            i.set((T)a[j]);
        }
    }

```

```
    /**
     * Sorts the specified list according to the order induced by the
     * specified comparator.  All elements in the list must be <i>mutually
     * comparable</i> using the specified comparator (that is,
     * {@code c.compare(e1, e2)} must not throw a {@code ClassCastException}
     * for any elements {@code e1} and {@code e2} in the list).
     *
     * <p>This sort is guaranteed to be <i>stable</i>:  equal elements will
     * not be reordered as a result of the sort.
     *
     * <p>The specified list must be modifiable, but need not be resizable.
     *
     * <p>Implementation note: This implementation is a stable, adaptive,
     * iterative mergesort that requires far fewer than n lg(n) comparisons
     * when the input array is partially sorted, while offering the
     * performance of a traditional mergesort when the input array is
     * randomly ordered.  If the input array is nearly sorted, the
     * implementation requires approximately n comparisons.  Temporary
     * storage requirements vary from a small constant for nearly sorted
     * input arrays to n/2 object references for randomly ordered input
     * arrays.
     *
     * <p>The implementation takes equal advantage of ascending and
     * descending order in its input array, and can take advantage of
     * ascending and descending order in different parts of the same
     * input array.  It is well-suited to merging two or more sorted arrays:
     * simply concatenate the arrays and sort the resulting array.
     *
     * <p>The implementation was adapted from Tim Peters's list sort for Python
     * (<a href="http://svn.python.org/projects/python/trunk/Objects/listsort.txt">
     * TimSort</a>).  It uses techiques from Peter McIlroy's "Optimistic
     * Sorting and Information Theoretic Complexity", in Proceedings of the
     * Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474,
     * January 1993.
     *
     * <p>This implementation dumps the specified list into an array, sorts
     * the array, and iterates over the list resetting each element
     * from the corresponding position in the array.  This avoids the
     * n<sup>2</sup> log(n) performance that would result from attempting
     * to sort a linked list in place.
     *
     * @param  list the list to be sorted.
     * @param  c the comparator to determine the order of the list.  A
     *        {@code null} value indicates that the elements' <i>natural
     *        ordering</i> should be used.
     * @throws ClassCastException if the list contains elements that are not
     *         <i>mutually comparable</i> using the specified comparator.
     * @throws UnsupportedOperationException if the specified list's
     *         list-iterator does not support the {@code set} operation.
     * @throws IllegalArgumentException (optional) if the comparator is
     *         found to violate the {@link Comparator} contract
     */
    public static <T> void sort(List<T> list, Comparator<? super T> c) {
        Object[] a = list.toArray();
        Arrays.sort(a, (Comparator)c);
        ListIterator i = list.listIterator();
        for (int j=0; j<a.length; j++) {
            i.next();
            i.set(a[j]);
        }
    }


    /**
     * Searches the specified list for the specified object using the binary
     * search algorithm.  The list must be sorted into ascending order
     * according to the {@linkplain Comparable natural ordering} of its
     * elements (as by the {@link #sort(List)} method) prior to making this
     * call.  If it is not sorted, the results are undefined.  If the list
     * contains multiple elements equal to the specified object, there is no
     * guarantee which one will be found.
     *
     * <p>This method runs in log(n) time for a "random access" list (which
     * provides near-constant-time positional access).  If the specified list
```

```
     * does not implement the {@link RandomAccess} interface and is large,
     * this method will do an iterator-based binary search that performs
     * O(n) link traversals and O(log n) element comparisons.
     *
     * @param  list the list to be searched.
     * @param  key the key to be searched for.
     * @return the index of the search key, if it is contained in the list;
     *         otherwise, <tt>(-(<i>insertion point</i>) - 1)</tt>.  The
     *         <i>insertion point</i> is defined as the point at which the
     *         key would be inserted into the list: the index of the first
     *         element greater than the key, or <tt>list.size()</tt> if all
     *         elements in the list are less than the specified key.  Note
     *         that this guarantees that the return value will be &gt;= 0 if
     *         and only if the key is found.
     * @throws ClassCastException if the list contains elements that are not
     *         <i>mutually comparable</i> (for example, strings and
     *         integers), or the search key is not mutually comparable
     *         with the elements of the list.
     */
    public static <T>
    int binarySearch(List<? extends Comparable<? super T>> list, T key) {
        if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
            return Collections.indexedBinarySearch(list, key);
        else
            return Collections.iteratorBinarySearch(list, key);
    }

    private static <T>
    int indexedBinarySearch(List<? extends Comparable<? super T>> list, T key)
    {
        int low = 0;
        int high = list.size()-1;

        while (low <= high) {
            int mid = (low + high) >>> 1;
            Comparable<? super T> midVal = list.get(mid);
            int cmp = midVal.compareTo(key);

            if (cmp < 0)
                low = mid + 1;
            else if (cmp > 0)
                high = mid - 1;
            else
                return mid; // key found
        }
        return -(low + 1);  // key not found
    }

    private static <T>
    int iteratorBinarySearch(List<? extends Comparable<? super T>> list, T key)
    {
        int low = 0;
        int high = list.size()-1;
        ListIterator<? extends Comparable<? super T>> i = list.listIterator();

        while (low <= high) {
            int mid = (low + high) >>> 1;
            Comparable<? super T> midVal = get(i, mid);
            int cmp = midVal.compareTo(key);

            if (cmp < 0)
                low = mid + 1;
            else if (cmp > 0)
                high = mid - 1;
            else
                return mid; // key found
        }
        return -(low + 1);  // key not found
    }

    /**
     * Gets the ith element from the given list by repositioning the specified
     * list listIterator.
     */
```

```
311     private static <T> T get(ListIterator<? extends T> i, int index) {
312         T obj = null;
313         int pos = i.nextIndex();
314         if (pos <= index) {
315             do {
316                 obj = i.next();
317             } while (pos++ < index);
318         } else {
319             do {
320                 obj = i.previous();
321             } while (--pos > index);
322         }
323         return obj;
324     }
325
326     /**
327      * Searches the specified list for the specified object using the binary
328      * search algorithm.  The list must be sorted into ascending order
329      * according to the specified comparator (as by the
330      * {@link #sort(List, Comparator) sort(List, Comparator)}
331      * method), prior to making this call.  If it is
332      * not sorted, the results are undefined.  If the list contains multiple
333      * elements equal to the specified object, there is no guarantee which one
334      * will be found.
335      *
336      * <p>This method runs in log(n) time for a "random access" list (which
337      * provides near-constant-time positional access).  If the specified list
338      * does not implement the {@link RandomAccess} interface and is large,
339      * this method will do an iterator-based binary search that performs
340      * O(n) link traversals and O(log n) element comparisons.
341      *
342      * @param  list the list to be searched.
343      * @param  key the key to be searched for.
344      * @param  c the comparator by which the list is ordered.
345      *         A <tt>null</tt> value indicates that the elements'
346      *         {@linkplain Comparable natural ordering} should be used.
347      * @return the index of the search key, if it is contained in the list;
348      *         otherwise, <tt>(-(<i>insertion point</i>) - 1)</tt>.  The
349      *         <i>insertion point</i> is defined as the point at which the
350      *         key would be inserted into the list: the index of the first
351      *         element greater than the key, or <tt>list.size()</tt> if all
352      *         elements in the list are less than the specified key.  Note
353      *         that this guarantees that the return value will be &gt;= 0 if
354      *         and only if the key is found.
355      * @throws ClassCastException if the list contains elements that are not
356      *         <i>mutually comparable</i> using the specified comparator,
357      *         or the search key is not mutually comparable with the
358      *         elements of the list using this comparator.
359      */
360     public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T>
    c) {
361         if (c==null)
362             return binarySearch((List) list, key);
363
364         if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
365             return Collections.indexedBinarySearch(list, key, c);
366         else
367             return Collections.iteratorBinarySearch(list, key, c);
368     }
369
370     private static <T> int indexedBinarySearch(List<? extends T> l, T key, Comparator<?
    super T> c) {
371         int low = 0;
372         int high = l.size()-1;
373
374         while (low <= high) {
375             int mid = (low + high) >>> 1;
376             T midVal = l.get(mid);
377             int cmp = c.compare(midVal, key);
378
379             if (cmp < 0)
380                 low = mid + 1;
381             else if (cmp > 0)
382                 high = mid - 1;
```

```
            else
                return mid; // key found
        }
        return -(low + 1);  // key not found
    }

    private static <T> int iteratorBinarySearch(List<? extends T> l, T key, Comparator<?
super T> c) {
        int low = 0;
        int high = l.size()-1;
        ListIterator<? extends T> i = l.listIterator();

        while (low <= high) {
            int mid = (low + high) >>> 1;
            T midVal = get(i, mid);
            int cmp = c.compare(midVal, key);

            if (cmp < 0)
                low = mid + 1;
            else if (cmp > 0)
                high = mid - 1;
            else
                return mid; // key found
        }
        return -(low + 1);  // key not found
    }

    private interface SelfComparable extends Comparable<SelfComparable> {}


    /**
     * Reverses the order of the elements in the specified list.<p>
     *
     * This method runs in linear time.
     *
     * @param  list the list whose elements are to be reversed.
     * @throws UnsupportedOperationException if the specified list or
     *         its list-iterator does not support the <tt>set</tt> operation.
     */
    public static void reverse(List<?> list) {
        int size = list.size();
        if (size < REVERSE_THRESHOLD || list instanceof RandomAccess) {
            for (int i=0, mid=size>>1, j=size-1; i<mid; i++, j--)
                swap(list, i, j);
        } else {
            ListIterator fwd = list.listIterator();
            ListIterator rev = list.listIterator(size);
            for (int i=0, mid=list.size()>>1; i<mid; i++) {
                Object tmp = fwd.next();
                fwd.set(rev.previous());
                rev.set(tmp);
            }
        }
    }

    /**
     * Randomly permutes the specified list using a default source of
     * randomness.  All permutations occur with approximately equal
     * likelihood.<p>
     *
     * The hedge "approximately" is used in the foregoing description because
     * default source of randomness is only approximately an unbiased source
     * of independently chosen bits. If it were a perfect source of randomly
     * chosen bits, then the algorithm would choose permutations with perfect
     * uniformity.<p>
     *
     * This implementation traverses the list backwards, from the last element
     * up to the second, repeatedly swapping a randomly selected element into
     * the "current position".  Elements are randomly selected from the
     * portion of the list that runs from the first element to the current
     * position, inclusive.<p>
     *
     * This method runs in linear time.  If the specified list does not
     * implement the {@link RandomAccess} interface and is large, this
```

```
     * implementation dumps the specified list into an array before shuffling
     * it, and dumps the shuffled array back into the list.  This avoids the
     * quadratic behavior that would result from shuffling a "sequential
     * access" list in place.
     *
     * @param  list the list to be shuffled.
     * @throws UnsupportedOperationException if the specified list or
     *         its list-iterator does not support the <tt>set</tt> operation.
     */
    public static void shuffle(List<?> list) {
        Random rnd = r;
        if (rnd == null)
            r = rnd = new Random();
        shuffle(list, rnd);
    }
    private static Random r;

    /**
     * Randomly permute the specified list using the specified source of
     * randomness.  All permutations occur with equal likelihood
     * assuming that the source of randomness is fair.<p>
     *
     * This implementation traverses the list backwards, from the last element
     * up to the second, repeatedly swapping a randomly selected element into
     * the "current position".  Elements are randomly selected from the
     * portion of the list that runs from the first element to the current
     * position, inclusive.<p>
     *
     * This method runs in linear time.  If the specified list does not
     * implement the {@link RandomAccess} interface and is large, this
     * implementation dumps the specified list into an array before shuffling
     * it, and dumps the shuffled array back into the list.  This avoids the
     * quadratic behavior that would result from shuffling a "sequential
     * access" list in place.
     *
     * @param  list the list to be shuffled.
     * @param  rnd the source of randomness to use to shuffle the list.
     * @throws UnsupportedOperationException if the specified list or its
     *         list-iterator does not support the <tt>set</tt> operation.
     */
    public static void shuffle(List<?> list, Random rnd) {
        int size = list.size();
        if (size < SHUFFLE_THRESHOLD || list instanceof RandomAccess) {
            for (int i=size; i>1; i--)
                swap(list, i-1, rnd.nextInt(i));
        } else {
            Object arr[] = list.toArray();

            // Shuffle array
            for (int i=size; i>1; i--)
                swap(arr, i-1, rnd.nextInt(i));

            // Dump array back into list
            ListIterator it = list.listIterator();
            for (int i=0; i<arr.length; i++) {
                it.next();
                it.set(arr[i]);
            }
        }
    }

    /**
     * Swaps the elements at the specified positions in the specified list.
     * (If the specified positions are equal, invoking this method leaves
     * the list unchanged.)
     *
     * @param list The list in which to swap elements.
     * @param i the index of one element to be swapped.
     * @param j the index of the other element to be swapped.
     * @throws IndexOutOfBoundsException if either <tt>i</tt> or <tt>j</tt>
     *         is out of range (i &lt; 0 || i &gt;= list.size()
     *         || j &lt; 0 || j &gt;= list.size()).
     * @since 1.4
     */
```

```
    public static void swap(List<?> list, int i, int j) {
        final List l = list;
        l.set(i, l.set(j, l.get(i)));
    }

    /**
     * Swaps the two specified elements in the specified array.
     */
    private static void swap(Object[] arr, int i, int j) {
        Object tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }

    /**
     * Replaces all of the elements of the specified list with the specified
     * element. <p>
     *
     * This method runs in linear time.
     *
     * @param  list the list to be filled with the specified element.
     * @param  obj The element with which to fill the specified list.
     * @throws UnsupportedOperationException if the specified list or its
     *         list-iterator does not support the <tt>set</tt> operation.
     */
    public static <T> void fill(List<? super T> list, T obj) {
        int size = list.size();

        if (size < FILL_THRESHOLD || list instanceof RandomAccess) {
            for (int i=0; i<size; i++)
                list.set(i, obj);
        } else {
            ListIterator<? super T> itr = list.listIterator();
            for (int i=0; i<size; i++) {
                itr.next();
                itr.set(obj);
            }
        }
    }

    /**
     * Copies all of the elements from one list into another.  After the
     * operation, the index of each copied element in the destination list
     * will be identical to its index in the source list.  The destination
     * list must be at least as long as the source list.  If it is longer, the
     * remaining elements in the destination list are unaffected. <p>
     *
     * This method runs in linear time.
     *
     * @param  dest The destination list.
     * @param  src The source list.
     * @throws IndexOutOfBoundsException if the destination list is too small
     *         to contain the entire source List.
     * @throws UnsupportedOperationException if the destination list's
     *         list-iterator does not support the <tt>set</tt> operation.
     */
    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        int srcSize = src.size();
        if (srcSize > dest.size())
            throw new IndexOutOfBoundsException("Source does not fit in dest");

        if (srcSize < COPY_THRESHOLD ||
            (src instanceof RandomAccess && dest instanceof RandomAccess)) {
            for (int i=0; i<srcSize; i++)
                dest.set(i, src.get(i));
        } else {
            ListIterator<? super T> di=dest.listIterator();
            ListIterator<? extends T> si=src.listIterator();
            for (int i=0; i<srcSize; i++) {
                di.next();
                di.set(si.next());
            }
        }
    }
```

```
604
605     /**
606      * Returns the minimum element of the given collection, according to the
607      * <i>natural ordering</i> of its elements.  All elements in the
608      * collection must implement the <tt>Comparable</tt> interface.
609      * Furthermore, all elements in the collection must be <i>mutually
610      * comparable</i> (that is, <tt>e1.compareTo(e2)</tt> must not throw a
611      * <tt>ClassCastException</tt> for any elements <tt>e1</tt> and
612      * <tt>e2</tt> in the collection).<p>
613      *
614      * This method iterates over the entire collection, hence it requires
615      * time proportional to the size of the collection.
616      *
617      * @param  coll the collection whose minimum element is to be determined.
618      * @return the minimum element of the given collection, according
619      *         to the <i>natural ordering</i> of its elements.
620      * @throws ClassCastException if the collection contains elements that are
621      *         not <i>mutually comparable</i> (for example, strings and
622      *         integers).
623      * @throws NoSuchElementException if the collection is empty.
624      * @see Comparable
625      */
626     public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T>
    coll) {
627         Iterator<? extends T> i = coll.iterator();
628         T candidate = i.next();
629
630         while (i.hasNext()) {
631             T next = i.next();
632             if (next.compareTo(candidate) < 0)
633                 candidate = next;
634         }
635         return candidate;
636     }
637
638     /**
639      * Returns the minimum element of the given collection, according to the
640      * order induced by the specified comparator.  All elements in the
641      * collection must be <i>mutually comparable</i> by the specified
642      * comparator (that is, <tt>comp.compare(e1, e2)</tt> must not throw a
643      * <tt>ClassCastException</tt> for any elements <tt>e1</tt> and
644      * <tt>e2</tt> in the collection).<p>
645      *
646      * This method iterates over the entire collection, hence it requires
647      * time proportional to the size of the collection.
648      *
649      * @param  coll the collection whose minimum element is to be determined.
650      * @param  comp the comparator with which to determine the minimum element.
651      *         A <tt>null</tt> value indicates that the elements' <i>natural
652      *         ordering</i> should be used.
653      * @return the minimum element of the given collection, according
654      *         to the specified comparator.
655      * @throws ClassCastException if the collection contains elements that are
656      *         not <i>mutually comparable</i> using the specified comparator.
657      * @throws NoSuchElementException if the collection is empty.
658      * @see Comparable
659      */
660     public static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp) {
661         if (comp==null)
662             return (T)min((Collection<SelfComparable>) (Collection) coll);
663
664         Iterator<? extends T> i = coll.iterator();
665         T candidate = i.next();
666
667         while (i.hasNext()) {
668             T next = i.next();
669             if (comp.compare(next, candidate) < 0)
670                 candidate = next;
671         }
672         return candidate;
673     }
674
675     /**
676      * Returns the maximum element of the given collection, according to the
```

```
     * <i>natural ordering</i> of its elements.  All elements in the
     * collection must implement the <tt>Comparable</tt> interface.
     * Furthermore, all elements in the collection must be <i>mutually
     * comparable</i> (that is, <tt>e1.compareTo(e2)</tt> must not throw a
     * <tt>ClassCastException</tt> for any elements <tt>e1</tt> and
     * <tt>e2</tt> in the collection).<p>
     *
     * This method iterates over the entire collection, hence it requires
     * time proportional to the size of the collection.
     *
     * @param  coll the collection whose maximum element is to be determined.
     * @return the maximum element of the given collection, according
     *         to the <i>natural ordering</i> of its elements.
     * @throws ClassCastException if the collection contains elements that are
     *         not <i>mutually comparable</i> (for example, strings and
     *         integers).
     * @throws NoSuchElementException if the collection is empty.
     * @see Comparable
     */
    public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T>
coll) {
        Iterator<? extends T> i = coll.iterator();
        T candidate = i.next();

        while (i.hasNext()) {
            T next = i.next();
            if (next.compareTo(candidate) > 0)
                candidate = next;
        }
        return candidate;
    }

    /**
     * Returns the maximum element of the given collection, according to the
     * order induced by the specified comparator.  All elements in the
     * collection must be <i>mutually comparable</i> by the specified
     * comparator (that is, <tt>comp.compare(e1, e2)</tt> must not throw a
     * <tt>ClassCastException</tt> for any elements <tt>e1</tt> and
     * <tt>e2</tt> in the collection).<p>
     *
     * This method iterates over the entire collection, hence it requires
     * time proportional to the size of the collection.
     *
     * @param  coll the collection whose maximum element is to be determined.
     * @param  comp the comparator with which to determine the maximum element.
     *         A <tt>null</tt> value indicates that the elements' <i>natural
     *        ordering</i> should be used.
     * @return the maximum element of the given collection, according
     *         to the specified comparator.
     * @throws ClassCastException if the collection contains elements that are
     *         not <i>mutually comparable</i> using the specified comparator.
     * @throws NoSuchElementException if the collection is empty.
     * @see Comparable
     */
    public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp) {
        if (comp==null)
            return (T)max((Collection<SelfComparable>) (Collection) coll);

        Iterator<? extends T> i = coll.iterator();
        T candidate = i.next();

        while (i.hasNext()) {
            T next = i.next();
            if (comp.compare(next, candidate) > 0)
                candidate = next;
        }
        return candidate;
    }

    /**
     * Rotates the elements in the specified list by the specified distance.
     * After calling this method, the element at index <tt>i</tt> will be
     * the element previously at index <tt>(i - distance)</tt> mod
     * <tt>list.size()</tt>, for all values of <tt>i</tt> between <tt>0</tt>
```

```
     * and <tt>list.size()-1</tt>, inclusive.  (This method has no effect on
     * the size of the list.)
     *
     * <p>For example, suppose <tt>list</tt> comprises<tt> [t, a, n, k, s]</tt>.
     * After invoking <tt>Collections.rotate(list, 1)</tt> (or
     * <tt>Collections.rotate(list, -4)</tt>), <tt>list</tt> will comprise
     * <tt>[s, t, a, n, k]</tt>.
     *
     * <p>Note that this method can usefully be applied to sublists to
     * move one or more elements within a list while preserving the
     * order of the remaining elements.  For example, the following idiom
     * moves the element at index <tt>j</tt> forward to position
     * <tt>k</tt> (which must be greater than or equal to <tt>j</tt>):
     * <pre>
     *     Collections.rotate(list.subList(j, k+1), -1);
     * </pre>
     * To make this concrete, suppose <tt>list</tt> comprises
     * <tt>[a, b, c, d, e]</tt>.  To move the element at index <tt>1</tt>
     * (<tt>b</tt>) forward two positions, perform the following invocation:
     * <pre>
     *     Collections.rotate(l.subList(1, 4), -1);
     * </pre>
     * The resulting list is <tt>[a, c, d, b, e]</tt>.
     *
     * <p>To move more than one element forward, increase the absolute value
     * of the rotation distance.  To move elements backward, use a positive
     * shift distance.
     *
     * <p>If the specified list is small or implements the {@link
     * RandomAccess} interface, this implementation exchanges the first
     * element into the location it should go, and then repeatedly exchanges
     * the displaced element into the location it should go until a displaced
     * element is swapped into the first element.  If necessary, the process
     * is repeated on the second and successive elements, until the rotation
     * is complete.  If the specified list is large and doesn't implement the
     * <tt>RandomAccess</tt> interface, this implementation breaks the
     * list into two sublist views around index <tt>-distance mod size</tt>.
     * Then the {@link #reverse(List)} method is invoked on each sublist view,
     * and finally it is invoked on the entire list.  For a more complete
     * description of both algorithms, see Section 2.3 of Jon Bentley's
     * <i>Programming Pearls</i> (Addison-Wesley, 1986).
     *
     * @param list the list to be rotated.
     * @param distance the distance to rotate the list.  There are no
     *        constraints on this value; it may be zero, negative, or
     *        greater than <tt>list.size()</tt>.
     * @throws UnsupportedOperationException if the specified list or
     *         its list-iterator does not support the <tt>set</tt> operation.
     * @since 1.4
     */
    public static void rotate(List<?> list, int distance) {
        if (list instanceof RandomAccess || list.size() < ROTATE_THRESHOLD)
            rotate1(list, distance);
        else
            rotate2(list, distance);
    }

    private static <T> void rotate1(List<T> list, int distance) {
        int size = list.size();
        if (size == 0)
            return;
        distance = distance % size;
        if (distance < 0)
            distance += size;
        if (distance == 0)
            return;

        for (int cycleStart = 0, nMoved = 0; nMoved != size; cycleStart++) {
            T displaced = list.get(cycleStart);
            int i = cycleStart;
            do {
                i += distance;
                if (i >= size)
                    i -= size;
```

```
                displaced = list.set(i, displaced);
                nMoved ++;
            } while (i != cycleStart);
        }
    }

    private static void rotate2(List<?> list, int distance) {
        int size = list.size();
        if (size == 0)
            return;
        int mid =  -distance % size;
        if (mid < 0)
            mid += size;
        if (mid == 0)
            return;

        reverse(list.subList(0, mid));
        reverse(list.subList(mid, size));
        reverse(list);
    }

    /**
     * Replaces all occurrences of one specified value in a list with another.
     * More formally, replaces with <tt>newVal</tt> each element <tt>e</tt>
     * in <tt>list</tt> such that
     * <tt>(oldVal==null ? e==null : oldVal.equals(e))</tt>.
     * (This method has no effect on the size of the list.)
     *
     * @param list the list in which replacement is to occur.
     * @param oldVal the old value to be replaced.
     * @param newVal the new value with which <tt>oldVal</tt> is to be
     *        replaced.
     * @return <tt>true</tt> if <tt>list</tt> contained one or more elements
     *         <tt>e</tt> such that
     *         <tt>(oldVal==null ?  e==null : oldVal.equals(e))</tt>.
     * @throws UnsupportedOperationException if the specified list or
     *         its list-iterator does not support the <tt>set</tt> operation.
     * @since  1.4
     */
    public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal) {
        boolean result = false;
        int size = list.size();
        if (size < REPLACEALL_THRESHOLD || list instanceof RandomAccess) {
            if (oldVal==null) {
                for (int i=0; i<size; i++) {
                    if (list.get(i)==null) {
                        list.set(i, newVal);
                        result = true;
                    }
                }
            } else {
                for (int i=0; i<size; i++) {
                    if (oldVal.equals(list.get(i))) {
                        list.set(i, newVal);
                        result = true;
                    }
                }
            }
        } else {
            ListIterator<T> itr=list.listIterator();
            if (oldVal==null) {
                for (int i=0; i<size; i++) {
                    if (itr.next()==null) {
                        itr.set(newVal);
                        result = true;
                    }
                }
            } else {
                for (int i=0; i<size; i++) {
                    if (oldVal.equals(itr.next())) {
                        itr.set(newVal);
                        result = true;
                    }
                }
```

```
            }
        }
        return result;
    }

    /**
     * Returns the starting position of the first occurrence of the specified
     * target list within the specified source list, or -1 if there is no
     * such occurrence.  More formally, returns the lowest index <tt>i</tt>
     * such that <tt>source.subList(i, i+target.size()).equals(target)</tt>,
     * or -1 if there is no such index.  (Returns -1 if
     * <tt>target.size() > source.size()</tt>.)
     *
     * <p>This implementation uses the "brute force" technique of scanning
     * over the source list, looking for a match with the target at each
     * location in turn.
     *
     * @param source the list in which to search for the first occurrence
     *        of <tt>target</tt>.
     * @param target the list to search for as a subList of <tt>source</tt>.
     * @return the starting position of the first occurrence of the specified
     *         target list within the specified source list, or -1 if there
     *         is no such occurrence.
     * @since  1.4
     */
    public static int indexOfSubList(List<?> source, List<?> target) {
        int sourceSize = source.size();
        int targetSize = target.size();
        int maxCandidate = sourceSize - targetSize;

        if (sourceSize < INDEXOFSUBLIST_THRESHOLD ||
            (source instanceof RandomAccess&&target instanceof RandomAccess)) {
        nextCand:
            for (int candidate = 0; candidate <= maxCandidate; candidate++) {
                for (int i=0, j=candidate; i<targetSize; i++, j++)
                    if (!eq(target.get(i), source.get(j)))
                        continue nextCand;  // Element mismatch, try next cand
                return candidate;  // All elements of candidate matched target
            }
        } else {  // Iterator version of above algorithm
            ListIterator<?> si = source.listIterator();
        nextCand:
            for (int candidate = 0; candidate <= maxCandidate; candidate++) {
                ListIterator<?> ti = target.listIterator();
                for (int i=0; i<targetSize; i++) {
                    if (!eq(ti.next(), si.next())) {
                        // Back up source iterator to next candidate
                        for (int j=0; j<i; j++)
                            si.previous();
                        continue nextCand;
                    }
                }
                return candidate;
            }
        }
        return -1;  // No candidate matched the target
    }

    /**
     * Returns the starting position of the last occurrence of the specified
     * target list within the specified source list, or -1 if there is no such
     * occurrence.  More formally, returns the highest index <tt>i</tt>
     * such that <tt>source.subList(i, i+target.size()).equals(target)</tt>,
     * or -1 if there is no such index.  (Returns -1 if
     * <tt>target.size() > source.size()</tt>.)
     *
     * <p>This implementation uses the "brute force" technique of iterating
     * over the source list, looking for a match with the target at each
     * location in turn.
     *
     * @param source the list in which to search for the last occurrence
     *        of <tt>target</tt>.
     * @param target the list to search for as a subList of <tt>source</tt>.
     * @return the starting position of the last occurrence of the specified
```

```
     *         target list within the specified source list, or -1 if there
     *         is no such occurrence.
     * @since  1.4
     */
    public static int lastIndexOfSubList(List<?> source, List<?> target) {
        int sourceSize = source.size();
        int targetSize = target.size();
        int maxCandidate = sourceSize - targetSize;

        if (sourceSize < INDEXOFSUBLIST_THRESHOLD ||
            source instanceof RandomAccess) {   // Index access version
        nextCand:
            for (int candidate = maxCandidate; candidate >= 0; candidate--) {
                for (int i=0, j=candidate; i<targetSize; i++, j++)
                    if (!eq(target.get(i), source.get(j)))
                        continue nextCand;  // Element mismatch, try next cand
                return candidate;  // All elements of candidate matched target
            }
        } else {  // Iterator version of above algorithm
            if (maxCandidate < 0)
                return -1;
            ListIterator<?> si = source.listIterator(maxCandidate);
        nextCand:
            for (int candidate = maxCandidate; candidate >= 0; candidate--) {
                ListIterator<?> ti = target.listIterator();
                for (int i=0; i<targetSize; i++) {
                    if (!eq(ti.next(), si.next())) {
                        if (candidate != 0) {
                            // Back up source iterator to next candidate
                            for (int j=0; j<=i+1; j++)
                                si.previous();
                        }
                        continue nextCand;
                    }
                }
                return candidate;
            }
        }
        return -1;  // No candidate matched the target
    }


    // Unmodifiable Wrappers

    /**
     * Returns an unmodifiable view of the specified collection.  This method
     * allows modules to provide users with "read-only" access to internal
     * collections.  Query operations on the returned collection "read through"
     * to the specified collection, and attempts to modify the returned
     * collection, whether direct or via its iterator, result in an
     * <tt>UnsupportedOperationException</tt>.<p>
     *
     * The returned collection does <i>not</i> pass the hashCode and equals
     * operations through to the backing collection, but relies on
     * <tt>Object</tt>'s <tt>equals</tt> and <tt>hashCode</tt> methods.  This
     * is necessary to preserve the contracts of these operations in the case
     * that the backing collection is a set or a list.<p>
     *
     * The returned collection will be serializable if the specified collection
     * is serializable.
     *
     * @param  c the collection for which an unmodifiable view is to be
     *         returned.
     * @return an unmodifiable view of the specified collection.
     */
    public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c) {
        return new UnmodifiableCollection<>(c);
    }

    /**
     * @serial include
     */
    static class UnmodifiableCollection<E> implements Collection<E>, Serializable {
        private static final long serialVersionUID = 1820017752578914078L;
```

```
1046
1047            final Collection<? extends E> c;
1048
1049            UnmodifiableCollection(Collection<? extends E> c) {
1050                if (c==null)
1051                    throw new NullPointerException();
1052                this.c = c;
1053            }
1054
1055            public int size()                   {return c.size();}
1056            public boolean isEmpty()            {return c.isEmpty();}
1057            public boolean contains(Object o)   {return c.contains(o);}
1058            public Object[] toArray()           {return c.toArray();}
1059            public <T> T[] toArray(T[] a)       {return c.toArray(a);}
1060            public String toString()            {return c.toString();}
1061
1062            public Iterator<E> iterator() {
1063                return new Iterator<E>() {
1064                    private final Iterator<? extends E> i = c.iterator();
1065
1066                    public boolean hasNext() {return i.hasNext();}
1067                    public E next()          {return i.next();}
1068                    public void remove() {
1069                        throw new UnsupportedOperationException();
1070                    }
1071                };
1072            }
1073
1074            public boolean add(E e) {
1075                throw new UnsupportedOperationException();
1076            }
1077            public boolean remove(Object o) {
1078                throw new UnsupportedOperationException();
1079            }
1080
1081            public boolean containsAll(Collection<?> coll) {
1082                return c.containsAll(coll);
1083            }
1084            public boolean addAll(Collection<? extends E> coll) {
1085                throw new UnsupportedOperationException();
1086            }
1087            public boolean removeAll(Collection<?> coll) {
1088                throw new UnsupportedOperationException();
1089            }
1090            public boolean retainAll(Collection<?> coll) {
1091                throw new UnsupportedOperationException();
1092            }
1093            public void clear() {
1094                throw new UnsupportedOperationException();
1095            }
1096        }
1097
1098        /**
1099         * Returns an unmodifiable view of the specified set.  This method allows
1100         * modules to provide users with "read-only" access to internal sets.
1101         * Query operations on the returned set "read through" to the specified
1102         * set, and attempts to modify the returned set, whether direct or via its
1103         * iterator, result in an <tt>UnsupportedOperationException</tt>.<p>
1104         *
1105         * The returned set will be serializable if the specified set
1106         * is serializable.
1107         *
1108         * @param  s the set for which an unmodifiable view is to be returned.
1109         * @return an unmodifiable view of the specified set.
1110         */
1111        public static <T> Set<T> unmodifiableSet(Set<? extends T> s) {
1112            return new UnmodifiableSet<>(s);
1113        }
1114
1115        /**
1116         * @serial include
1117         */
1118        static class UnmodifiableSet<E> extends UnmodifiableCollection<E>
1119                                     implements Set<E>, Serializable {
```

```
        private static final long serialVersionUID = -9215047833775013803L;

        UnmodifiableSet(Set<? extends E> s)     {super(s);}
        public boolean equals(Object o) {return o == this || c.equals(o);}
        public int hashCode()           {return c.hashCode();}
    }

    /**
     * Returns an unmodifiable view of the specified sorted set.  This method
     * allows modules to provide users with "read-only" access to internal
     * sorted sets.  Query operations on the returned sorted set "read
     * through" to the specified sorted set.  Attempts to modify the returned
     * sorted set, whether direct, via its iterator, or via its
     * <tt>subSet</tt>, <tt>headSet</tt>, or <tt>tailSet</tt> views, result in
     * an <tt>UnsupportedOperationException</tt>.<p>
     *
     * The returned sorted set will be serializable if the specified sorted set
     * is serializable.
     *
     * @param s the sorted set for which an unmodifiable view is to be
     *        returned.
     * @return an unmodifiable view of the specified sorted set.
     */
    public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s) {
        return new UnmodifiableSortedSet<>(s);
    }

    /**
     * @serial include
     */
    static class UnmodifiableSortedSet<E>
                             extends UnmodifiableSet<E>
                             implements SortedSet<E>, Serializable {
        private static final long serialVersionUID = -4929149591599911165L;
        private final SortedSet<E> ss;

        UnmodifiableSortedSet(SortedSet<E> s) {super(s); ss = s;}

        public Comparator<? super E> comparator() {return ss.comparator();}

        public SortedSet<E> subSet(E fromElement, E toElement) {
            return new UnmodifiableSortedSet<>(ss.subSet(fromElement,toElement));
        }
        public SortedSet<E> headSet(E toElement) {
            return new UnmodifiableSortedSet<>(ss.headSet(toElement));
        }
        public SortedSet<E> tailSet(E fromElement) {
            return new UnmodifiableSortedSet<>(ss.tailSet(fromElement));
        }

        public E first()                    {return ss.first();}
        public E last()                     {return ss.last();}
    }

    /**
     * Returns an unmodifiable view of the specified list.  This method allows
     * modules to provide users with "read-only" access to internal
     * lists.  Query operations on the returned list "read through" to the
     * specified list, and attempts to modify the returned list, whether
     * direct or via its iterator, result in an
     * <tt>UnsupportedOperationException</tt>.<p>
     *
     * The returned list will be serializable if the specified list
     * is serializable. Similarly, the returned list will implement
     * {@link RandomAccess} if the specified list does.
     *
     * @param  list the list for which an unmodifiable view is to be returned.
     * @return an unmodifiable view of the specified list.
     */
    public static <T> List<T> unmodifiableList(List<? extends T> list) {
        return (list instanceof RandomAccess ?
                new UnmodifiableRandomAccessList<>(list) :
                new UnmodifiableList<>(list));
    }
```

```
1194
1195        /**
1196         * @serial include
1197         */
1198        static class UnmodifiableList<E> extends UnmodifiableCollection<E>
1199                                         implements List<E> {
1200            private static final long serialVersionUID = -283967356065247728L;
1201            final List<? extends E> list;
1202
1203            UnmodifiableList(List<? extends E> list) {
1204                super(list);
1205                this.list = list;
1206            }
1207
1208            public boolean equals(Object o) {return o == this || list.equals(o);}
1209            public int hashCode()           {return list.hashCode();}
1210
1211            public E get(int index) {return list.get(index);}
1212            public E set(int index, E element) {
1213                throw new UnsupportedOperationException();
1214            }
1215            public void add(int index, E element) {
1216                throw new UnsupportedOperationException();
1217            }
1218            public E remove(int index) {
1219                throw new UnsupportedOperationException();
1220            }
1221            public int indexOf(Object o)            {return list.indexOf(o);}
1222            public int lastIndexOf(Object o)        {return list.lastIndexOf(o);}
1223            public boolean addAll(int index, Collection<? extends E> c) {
1224                throw new UnsupportedOperationException();
1225            }
1226            public ListIterator<E> listIterator()   {return listIterator(0);}
1227
1228            public ListIterator<E> listIterator(final int index) {
1229                return new ListIterator<E>() {
1230                    private final ListIterator<? extends E> i
1231                        = list.listIterator(index);
1232
1233                    public boolean hasNext()     {return i.hasNext();}
1234                    public E next()              {return i.next();}
1235                    public boolean hasPrevious() {return i.hasPrevious();}
1236                    public E previous()          {return i.previous();}
1237                    public int nextIndex()       {return i.nextIndex();}
1238                    public int previousIndex()   {return i.previousIndex();}
1239
1240                    public void remove() {
1241                        throw new UnsupportedOperationException();
1242                    }
1243                    public void set(E e) {
1244                        throw new UnsupportedOperationException();
1245                    }
1246                    public void add(E e) {
1247                        throw new UnsupportedOperationException();
1248                    }
1249                };
1250            }
1251
1252            public List<E> subList(int fromIndex, int toIndex) {
1253                return new UnmodifiableList<>(list.subList(fromIndex, toIndex));
1254            }
1255
1256            /**
1257             * UnmodifiableRandomAccessList instances are serialized as
1258             * UnmodifiableList instances to allow them to be deserialized
1259             * in pre-1.4 JREs (which do not have UnmodifiableRandomAccessList).
1260             * This method inverts the transformation.  As a beneficial
1261             * side-effect, it also grafts the RandomAccess marker onto
1262             * UnmodifiableList instances that were serialized in pre-1.4 JREs.
1263             *
1264             * Note: Unfortunately, UnmodifiableRandomAccessList instances
1265             * serialized in 1.4.1 and deserialized in 1.4 will become
1266             * UnmodifiableList instances, as this method was missing in 1.4.
1267             */
```

```
1268            private Object readResolve() {
1269                return (list instanceof RandomAccess
1270                        ? new UnmodifiableRandomAccessList<>(list)
1271                        : this);
1272            }
1273        }
1274
1275        /**
1276         * @serial include
1277         */
1278        static class UnmodifiableRandomAccessList<E> extends UnmodifiableList<E>
1279                                                  implements RandomAccess
1280        {
1281            UnmodifiableRandomAccessList(List<? extends E> list) {
1282                super(list);
1283            }
1284
1285            public List<E> subList(int fromIndex, int toIndex) {
1286                return new UnmodifiableRandomAccessList<>(
1287                    list.subList(fromIndex, toIndex));
1288            }
1289
1290            private static final long serialVersionUID = -2542308836966382001L;
1291
1292            /**
1293             * Allows instances to be deserialized in pre-1.4 JREs (which do
1294             * not have UnmodifiableRandomAccessList).  UnmodifiableList has
1295             * a readResolve method that inverts this transformation upon
1296             * deserialization.
1297             */
1298            private Object writeReplace() {
1299                return new UnmodifiableList<>(list);
1300            }
1301        }
1302
1303        /**
1304         * Returns an unmodifiable view of the specified map.  This method
1305         * allows modules to provide users with "read-only" access to internal
1306         * maps.  Query operations on the returned map "read through"
1307         * to the specified map, and attempts to modify the returned
1308         * map, whether direct or via its collection views, result in an
1309         * <tt>UnsupportedOperationException</tt>.<p>
1310         *
1311         * The returned map will be serializable if the specified map
1312         * is serializable.
1313         *
1314         * @param  m the map for which an unmodifiable view is to be returned.
1315         * @return an unmodifiable view of the specified map.
1316         */
1317        public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m) {
1318            return new UnmodifiableMap<>(m);
1319        }
1320
1321        /**
1322         * @serial include
1323         */
1324        private static class UnmodifiableMap<K,V> implements Map<K,V>, Serializable {
1325            private static final long serialVersionUID = -1034234728574286014L;
1326
1327            private final Map<? extends K, ? extends V> m;
1328
1329            UnmodifiableMap(Map<? extends K, ? extends V> m) {
1330                if (m==null)
1331                    throw new NullPointerException();
1332                this.m = m;
1333            }
1334
1335            public int size()                        {return m.size();}
1336            public boolean isEmpty()                 {return m.isEmpty();}
1337            public boolean containsKey(Object key)   {return m.containsKey(key);}
1338            public boolean containsValue(Object val) {return m.containsValue(val);}
1339            public V get(Object key)                 {return m.get(key);}
1340
1341            public V put(K key, V value) {
```

```
            throw new UnsupportedOperationException();
        }
        public V remove(Object key) {
            throw new UnsupportedOperationException();
        }
        public void putAll(Map<? extends K, ? extends V> m) {
            throw new UnsupportedOperationException();
        }
        public void clear() {
            throw new UnsupportedOperationException();
        }

        private transient Set<K> keySet = null;
        private transient Set<Map.Entry<K,V>> entrySet = null;
        private transient Collection<V> values = null;

        public Set<K> keySet() {
            if (keySet==null)
                keySet = unmodifiableSet(m.keySet());
            return keySet;
        }

        public Set<Map.Entry<K,V>> entrySet() {
            if (entrySet==null)
                entrySet = new UnmodifiableEntrySet<>(m.entrySet());
            return entrySet;
        }

        public Collection<V> values() {
            if (values==null)
                values = unmodifiableCollection(m.values());
            return values;
        }

        public boolean equals(Object o) {return o == this || m.equals(o);}
        public int hashCode()           {return m.hashCode();}
        public String toString()        {return m.toString();}

        /**
         * We need this class in addition to UnmodifiableSet as
         * Map.Entries themselves permit modification of the backing Map
         * via their setValue operation.  This class is subtle: there are
         * many possible attacks that must be thwarted.
         *
         * @serial include
         */
        static class UnmodifiableEntrySet<K,V>
            extends UnmodifiableSet<Map.Entry<K,V>> {
            private static final long serialVersionUID = 7854390611657943733L;

            UnmodifiableEntrySet(Set<? extends Map.Entry<? extends K, ? extends V>> s) {
                super((Set)s);
            }
            public Iterator<Map.Entry<K,V>> iterator() {
                return new Iterator<Map.Entry<K,V>>() {
                    private final Iterator<? extends Map.Entry<? extends K, ? extends V>> i
= c.iterator();

                    public boolean hasNext() {
                        return i.hasNext();
                    }
                    public Map.Entry<K,V> next() {
                        return new UnmodifiableEntry<>(i.next());
                    }
                    public void remove() {
                        throw new UnsupportedOperationException();
                    }
                };
            }

            public Object[] toArray() {
                Object[] a = c.toArray();
                for (int i=0; i<a.length; i++)
                    a[i] = new UnmodifiableEntry<>((Map.Entry<K,V>)a[i]);
```

```
            return a;
        }

        public <T> T[] toArray(T[] a) {
            // We don't pass a to c.toArray, to avoid window of
            // vulnerability wherein an unscrupulous multithreaded client
            // could get his hands on raw (unwrapped) Entries from c.
            Object[] arr = c.toArray(a.length==0 ? a : Arrays.copyOf(a, 0));

            for (int i=0; i<arr.length; i++)
                arr[i] = new UnmodifiableEntry<>((Map.Entry<K,V>)arr[i]);

            if (arr.length > a.length)
                return (T[])arr;

            System.arraycopy(arr, 0, a, 0, arr.length);
            if (a.length > arr.length)
                a[arr.length] = null;
            return a;
        }

        /**
         * This method is overridden to protect the backing set against
         * an object with a nefarious equals function that senses
         * that the equality-candidate is Map.Entry and calls its
         * setValue method.
         */
        public boolean contains(Object o) {
            if (!(o instanceof Map.Entry))
                return false;
            return c.contains(
                new UnmodifiableEntry<>((Map.Entry<?,?>) o));
        }

        /**
         * The next two methods are overridden to protect against
         * an unscrupulous List whose contains(Object o) method senses
         * when o is a Map.Entry, and calls o.setValue.
         */
        public boolean containsAll(Collection<?> coll) {
            for (Object e : coll) {
                if (!contains(e)) // Invokes safe contains() above
                    return false;
            }
            return true;
        }
        public boolean equals(Object o) {
            if (o == this)
                return true;

            if (!(o instanceof Set))
                return false;
            Set s = (Set) o;
            if (s.size() != c.size())
                return false;
            return containsAll(s); // Invokes safe containsAll() above
        }

        /**
         * This "wrapper class" serves two purposes: it prevents
         * the client from modifying the backing Map, by short-circuiting
         * the setValue method, and it protects the backing Map against
         * an ill-behaved Map.Entry that attempts to modify another
         * Map Entry when asked to perform an equality check.
         */
        private static class UnmodifiableEntry<K,V> implements Map.Entry<K,V> {
            private Map.Entry<? extends K, ? extends V> e;

            UnmodifiableEntry(Map.Entry<? extends K, ? extends V> e) {this.e = e;}

            public K getKey()        {return e.getKey();}
            public V getValue()      {return e.getValue();}
            public V setValue(V value) {
                throw new UnsupportedOperationException();
```

```
                }
                public int hashCode()    {return e.hashCode();}
                public boolean equals(Object o) {
                    if (this == o)
                        return true;
                    if (!(o instanceof Map.Entry))
                        return false;
                    Map.Entry t = (Map.Entry)o;
                    return eq(e.getKey(),   t.getKey()) &&
                           eq(e.getValue(), t.getValue());
                }
                public String toString() {return e.toString();}
            }
        }
    }

    /**
     * Returns an unmodifiable view of the specified sorted map.  This method
     * allows modules to provide users with "read-only" access to internal
     * sorted maps.  Query operations on the returned sorted map "read through"
     * to the specified sorted map.  Attempts to modify the returned
     * sorted map, whether direct, via its collection views, or via its
     * <tt>subMap</tt>, <tt>headMap</tt>, or <tt>tailMap</tt> views, result in
     * an <tt>UnsupportedOperationException</tt>.<p>
     *
     * The returned sorted map will be serializable if the specified sorted map
     * is serializable.
     *
     * @param m the sorted map for which an unmodifiable view is to be
     *        returned.
     * @return an unmodifiable view of the specified sorted map.
     */
    public static <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K, ? extends V> m) {
        return new UnmodifiableSortedMap<>(m);
    }

    /**
     * @serial include
     */
    static class UnmodifiableSortedMap<K,V>
          extends UnmodifiableMap<K,V>
          implements SortedMap<K,V>, Serializable {
        private static final long serialVersionUID = -8806743815996713206L;

        private final SortedMap<K, ? extends V> sm;

        UnmodifiableSortedMap(SortedMap<K, ? extends V> m) {super(m); sm = m;}

        public Comparator<? super K> comparator() {return sm.comparator();}

        public SortedMap<K,V> subMap(K fromKey, K toKey) {
            return new UnmodifiableSortedMap<>(sm.subMap(fromKey, toKey));
        }
        public SortedMap<K,V> headMap(K toKey) {
            return new UnmodifiableSortedMap<>(sm.headMap(toKey));
        }
        public SortedMap<K,V> tailMap(K fromKey) {
            return new UnmodifiableSortedMap<>(sm.tailMap(fromKey));
        }

        public K firstKey()           {return sm.firstKey();}
        public K lastKey()            {return sm.lastKey();}
    }


    // Synch Wrappers

    /**
     * Returns a synchronized (thread-safe) collection backed by the specified
     * collection.  In order to guarantee serial access, it is critical that
     * <strong>all</strong> access to the backing collection is accomplished
     * through the returned collection.<p>
     *
     * It is imperative that the user manually synchronize on the returned
```

```
     * collection when iterating over it:
     * <pre>
     *  Collection c = Collections.synchronizedCollection(myCollection);
     *     ...
     *  synchronized (c) {
     *      Iterator i = c.iterator(); // Must be in the synchronized block
     *      while (i.hasNext())
     *         foo(i.next());
     *  }
     * </pre>
     * Failure to follow this advice may result in non-deterministic behavior.
     *
     * <p>The returned collection does <i>not</i> pass the <tt>hashCode</tt>
     * and <tt>equals</tt> operations through to the backing collection, but
     * relies on <tt>Object</tt>'s equals and hashCode methods.  This is
     * necessary to preserve the contracts of these operations in the case
     * that the backing collection is a set or a list.<p>
     *
     * The returned collection will be serializable if the specified collection
     * is serializable.
     *
     * @param  c the collection to be "wrapped" in a synchronized collection.
     * @return a synchronized view of the specified collection.
     */
    public static <T> Collection<T> synchronizedCollection(Collection<T> c) {
        return new SynchronizedCollection<>(c);
    }

    static <T> Collection<T> synchronizedCollection(Collection<T> c, Object mutex) {
        return new SynchronizedCollection<>(c, mutex);
    }

    /**
     * @serial include
     */
    static class SynchronizedCollection<E> implements Collection<E>, Serializable {
        private static final long serialVersionUID = 3053995032091335093L;

        final Collection<E> c;  // Backing Collection
        final Object mutex;     // Object on which to synchronize

        SynchronizedCollection(Collection<E> c) {
            if (c==null)
                throw new NullPointerException();
            this.c = c;
            mutex = this;
        }
        SynchronizedCollection(Collection<E> c, Object mutex) {
            this.c = c;
            this.mutex = mutex;
        }

        public int size() {
            synchronized (mutex) {return c.size();}
        }
        public boolean isEmpty() {
            synchronized (mutex) {return c.isEmpty();}
        }
        public boolean contains(Object o) {
            synchronized (mutex) {return c.contains(o);}
        }
        public Object[] toArray() {
            synchronized (mutex) {return c.toArray();}
        }
        public <T> T[] toArray(T[] a) {
            synchronized (mutex) {return c.toArray(a);}
        }

        public Iterator<E> iterator() {
            return c.iterator(); // Must be manually synched by user!
        }

        public boolean add(E e) {
            synchronized (mutex) {return c.add(e);}
```

```
        }
        public boolean remove(Object o) {
            synchronized (mutex) {return c.remove(o);}
        }

        public boolean containsAll(Collection<?> coll) {
            synchronized (mutex) {return c.containsAll(coll);}
        }
        public boolean addAll(Collection<? extends E> coll) {
            synchronized (mutex) {return c.addAll(coll);}
        }
        public boolean removeAll(Collection<?> coll) {
            synchronized (mutex) {return c.removeAll(coll);}
        }
        public boolean retainAll(Collection<?> coll) {
            synchronized (mutex) {return c.retainAll(coll);}
        }
        public void clear() {
            synchronized (mutex) {c.clear();}
        }
        public String toString() {
            synchronized (mutex) {return c.toString();}
        }
        private void writeObject(ObjectOutputStream s) throws IOException {
            synchronized (mutex) {s.defaultWriteObject();}
        }
    }

    /**
     * Returns a synchronized (thread-safe) set backed by the specified
     * set.  In order to guarantee serial access, it is critical that
     * <strong>all</strong> access to the backing set is accomplished
     * through the returned set.<p>
     *
     * It is imperative that the user manually synchronize on the returned
     * set when iterating over it:
     * <pre>
     *  Set s = Collections.synchronizedSet(new HashSet());
     *      ...
     *  synchronized (s) {
     *      Iterator i = s.iterator(); // Must be in the synchronized block
     *      while (i.hasNext())
     *          foo(i.next());
     *  }
     * </pre>
     * Failure to follow this advice may result in non-deterministic behavior.
     *
     * <p>The returned set will be serializable if the specified set is
     * serializable.
     *
     * @param  s the set to be "wrapped" in a synchronized set.
     * @return a synchronized view of the specified set.
     */
    public static <T> Set<T> synchronizedSet(Set<T> s) {
        return new SynchronizedSet<>(s);
    }

    static <T> Set<T> synchronizedSet(Set<T> s, Object mutex) {
        return new SynchronizedSet<>(s, mutex);
    }

    /**
     * @serial include
     */
    static class SynchronizedSet<E>
          extends SynchronizedCollection<E>
          implements Set<E> {
        private static final long serialVersionUID = 487447009682186044L;

        SynchronizedSet(Set<E> s) {
            super(s);
        }
        SynchronizedSet(Set<E> s, Object mutex) {
            super(s, mutex);
```

```
1711          }
1712
1713          public boolean equals(Object o) {
1714              if (this == o)
1715                  return true;
1716              synchronized (mutex) {return c.equals(o);}
1717          }
1718          public int hashCode() {
1719              synchronized (mutex) {return c.hashCode();}
1720          }
1721      }
1722
1723      /**
1724       * Returns a synchronized (thread-safe) sorted set backed by the specified
1725       * sorted set.  In order to guarantee serial access, it is critical that
1726       * <strong>all</strong> access to the backing sorted set is accomplished
1727       * through the returned sorted set (or its views).<p>
1728       *
1729       * It is imperative that the user manually synchronize on the returned
1730       * sorted set when iterating over it or any of its <tt>subSet</tt>,
1731       * <tt>headSet</tt>, or <tt>tailSet</tt> views.
1732       * <pre>
1733       *  SortedSet s = Collections.synchronizedSortedSet(new TreeSet());
1734       *      ...
1735       *  synchronized (s) {
1736       *      Iterator i = s.iterator(); // Must be in the synchronized block
1737       *      while (i.hasNext())
1738       *          foo(i.next());
1739       *  }
1740       * </pre>
1741       * or:
1742       * <pre>
1743       *  SortedSet s = Collections.synchronizedSortedSet(new TreeSet());
1744       *  SortedSet s2 = s.headSet(foo);
1745       *      ...
1746       *  synchronized (s) {  // Note: s, not s2!!!
1747       *      Iterator i = s2.iterator(); // Must be in the synchronized block
1748       *      while (i.hasNext())
1749       *          foo(i.next());
1750       *  }
1751       * </pre>
1752       * Failure to follow this advice may result in non-deterministic behavior.
1753       *
1754       * <p>The returned sorted set will be serializable if the specified
1755       * sorted set is serializable.
1756       *
1757       * @param  s the sorted set to be "wrapped" in a synchronized sorted set.
1758       * @return a synchronized view of the specified sorted set.
1759       */
1760      public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s) {
1761          return new SynchronizedSortedSet<>(s);
1762      }
1763
1764      /**
1765       * @serial include
1766       */
1767      static class SynchronizedSortedSet<E>
1768          extends SynchronizedSet<E>
1769          implements SortedSet<E>
1770      {
1771          private static final long serialVersionUID = 8695801310862127406L;
1772
1773          private final SortedSet<E> ss;
1774
1775          SynchronizedSortedSet(SortedSet<E> s) {
1776              super(s);
1777              ss = s;
1778          }
1779          SynchronizedSortedSet(SortedSet<E> s, Object mutex) {
1780              super(s, mutex);
1781              ss = s;
1782          }
1783
1784          public Comparator<? super E> comparator() {
```

```
            synchronized (mutex) {return ss.comparator();}
        }

        public SortedSet<E> subSet(E fromElement, E toElement) {
            synchronized (mutex) {
                return new SynchronizedSortedSet<>(
                    ss.subSet(fromElement, toElement), mutex);
            }
        }
        public SortedSet<E> headSet(E toElement) {
            synchronized (mutex) {
                return new SynchronizedSortedSet<>(ss.headSet(toElement), mutex);
            }
        }
        public SortedSet<E> tailSet(E fromElement) {
            synchronized (mutex) {
               return new SynchronizedSortedSet<>(ss.tailSet(fromElement),mutex);
            }
        }

        public E first() {
            synchronized (mutex) {return ss.first();}
        }
        public E last() {
            synchronized (mutex) {return ss.last();}
        }
    }

    /**
     * Returns a synchronized (thread-safe) list backed by the specified
     * list.  In order to guarantee serial access, it is critical that
     * <strong>all</strong> access to the backing list is accomplished
     * through the returned list.<p>
     *
     * It is imperative that the user manually synchronize on the returned
     * list when iterating over it:
     * <pre>
     *  List list = Collections.synchronizedList(new ArrayList());
     *      ...
     *  synchronized (list) {
     *      Iterator i = list.iterator(); // Must be in synchronized block
     *      while (i.hasNext())
     *          foo(i.next());
     *  }
     * </pre>
     * Failure to follow this advice may result in non-deterministic behavior.
     *
     * <p>The returned list will be serializable if the specified list is
     * serializable.
     *
     * @param  list the list to be "wrapped" in a synchronized list.
     * @return a synchronized view of the specified list.
     */
    public static <T> List<T> synchronizedList(List<T> list) {
        return (list instanceof RandomAccess ?
                new SynchronizedRandomAccessList<>(list) :
                new SynchronizedList<>(list));
    }

    static <T> List<T> synchronizedList(List<T> list, Object mutex) {
        return (list instanceof RandomAccess ?
                new SynchronizedRandomAccessList<>(list, mutex) :
                new SynchronizedList<>(list, mutex));
    }

    /**
     * @serial include
     */
    static class SynchronizedList<E>
        extends SynchronizedCollection<E>
        implements List<E> {
        private static final long serialVersionUID = -7754090372962971524L;

        final List<E> list;
```

```
1859
1860        SynchronizedList(List<E> list) {
1861            super(list);
1862            this.list = list;
1863        }
1864        SynchronizedList(List<E> list, Object mutex) {
1865            super(list, mutex);
1866            this.list = list;
1867        }
1868
1869        public boolean equals(Object o) {
1870            if (this == o)
1871                return true;
1872            synchronized (mutex) {return list.equals(o);}
1873        }
1874        public int hashCode() {
1875            synchronized (mutex) {return list.hashCode();}
1876        }
1877
1878        public E get(int index) {
1879            synchronized (mutex) {return list.get(index);}
1880        }
1881        public E set(int index, E element) {
1882            synchronized (mutex) {return list.set(index, element);}
1883        }
1884        public void add(int index, E element) {
1885            synchronized (mutex) {list.add(index, element);}
1886        }
1887        public E remove(int index) {
1888            synchronized (mutex) {return list.remove(index);}
1889        }
1890
1891        public int indexOf(Object o) {
1892            synchronized (mutex) {return list.indexOf(o);}
1893        }
1894        public int lastIndexOf(Object o) {
1895            synchronized (mutex) {return list.lastIndexOf(o);}
1896        }
1897
1898        public boolean addAll(int index, Collection<? extends E> c) {
1899            synchronized (mutex) {return list.addAll(index, c);}
1900        }
1901
1902        public ListIterator<E> listIterator() {
1903            return list.listIterator(); // Must be manually synched by user
1904        }
1905
1906        public ListIterator<E> listIterator(int index) {
1907            return list.listIterator(index); // Must be manually synched by user
1908        }
1909
1910        public List<E> subList(int fromIndex, int toIndex) {
1911            synchronized (mutex) {
1912                return new SynchronizedList<>(list.subList(fromIndex, toIndex),
1913                                              mutex);
1914            }
1915        }
1916
1917        /**
1918         * SynchronizedRandomAccessList instances are serialized as
1919         * SynchronizedList instances to allow them to be deserialized
1920         * in pre-1.4 JREs (which do not have SynchronizedRandomAccessList).
1921         * This method inverts the transformation.  As a beneficial
1922         * side-effect, it also grafts the RandomAccess marker onto
1923         * SynchronizedList instances that were serialized in pre-1.4 JREs.
1924         *
1925         * Note: Unfortunately, SynchronizedRandomAccessList instances
1926         * serialized in 1.4.1 and deserialized in 1.4 will become
1927         * SynchronizedList instances, as this method was missing in 1.4.
1928         */
1929        private Object readResolve() {
1930            return (list instanceof RandomAccess
1931                    ? new SynchronizedRandomAccessList<>(list)
1932                    : this);
```

```
        }
    }

    /**
     * @serial include
     */
    static class SynchronizedRandomAccessList<E>
        extends SynchronizedList<E>
        implements RandomAccess {

        SynchronizedRandomAccessList(List<E> list) {
            super(list);
        }

        SynchronizedRandomAccessList(List<E> list, Object mutex) {
            super(list, mutex);
        }

        public List<E> subList(int fromIndex, int toIndex) {
            synchronized (mutex) {
                return new SynchronizedRandomAccessList<>(
                    list.subList(fromIndex, toIndex), mutex);
            }
        }

        private static final long serialVersionUID = 1530674583602358482L;

        /**
         * Allows instances to be deserialized in pre-1.4 JREs (which do
         * not have SynchronizedRandomAccessList).  SynchronizedList has
         * a readResolve method that inverts this transformation upon
         * deserialization.
         */
        private Object writeReplace() {
            return new SynchronizedList<>(list);
        }
    }

    /**
     * Returns a synchronized (thread-safe) map backed by the specified
     * map.  In order to guarantee serial access, it is critical that
     * <strong>all</strong> access to the backing map is accomplished
     * through the returned map.<p>
     *
     * It is imperative that the user manually synchronize on the returned
     * map when iterating over any of its collection views:
     * <pre>
     *  Map m = Collections.synchronizedMap(new HashMap());
     *      ...
     *  Set s = m.keySet();  // Needn't be in synchronized block
     *      ...
     *  synchronized (m) {  // Synchronizing on m, not s!
     *      Iterator i = s.iterator(); // Must be in synchronized block
     *      while (i.hasNext())
     *          foo(i.next());
     *  }
     * </pre>
     * Failure to follow this advice may result in non-deterministic behavior.
     *
     * <p>The returned map will be serializable if the specified map is
     * serializable.
     *
     * @param  m the map to be "wrapped" in a synchronized map.
     * @return a synchronized view of the specified map.
     */
    public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {
        return new SynchronizedMap<>(m);
    }

    /**
     * @serial include
     */
    private static class SynchronizedMap<K,V>
        implements Map<K,V>, Serializable {
```

```
        private static final long serialVersionUID = 1978198479659022715L;

        private final Map<K,V> m;     // Backing Map
        final Object      mutex;        // Object on which to synchronize

        SynchronizedMap(Map<K,V> m) {
            if (m==null)
                throw new NullPointerException();
            this.m = m;
            mutex = this;
        }

        SynchronizedMap(Map<K,V> m, Object mutex) {
            this.m = m;
            this.mutex = mutex;
        }

        public int size() {
            synchronized (mutex) {return m.size();}
        }
        public boolean isEmpty() {
            synchronized (mutex) {return m.isEmpty();}
        }
        public boolean containsKey(Object key) {
            synchronized (mutex) {return m.containsKey(key);}
        }
        public boolean containsValue(Object value) {
            synchronized (mutex) {return m.containsValue(value);}
        }
        public V get(Object key) {
            synchronized (mutex) {return m.get(key);}
        }

        public V put(K key, V value) {
            synchronized (mutex) {return m.put(key, value);}
        }
        public V remove(Object key) {
            synchronized (mutex) {return m.remove(key);}
        }
        public void putAll(Map<? extends K, ? extends V> map) {
            synchronized (mutex) {m.putAll(map);}
        }
        public void clear() {
            synchronized (mutex) {m.clear();}
        }

        private transient Set<K> keySet = null;
        private transient Set<Map.Entry<K,V>> entrySet = null;
        private transient Collection<V> values = null;

        public Set<K> keySet() {
            synchronized (mutex) {
                if (keySet==null)
                    keySet = new SynchronizedSet<>(m.keySet(), mutex);
                return keySet;
            }
        }

        public Set<Map.Entry<K,V>> entrySet() {
            synchronized (mutex) {
                if (entrySet==null)
                    entrySet = new SynchronizedSet<>(m.entrySet(), mutex);
                return entrySet;
            }
        }

        public Collection<V> values() {
            synchronized (mutex) {
                if (values==null)
                    values = new SynchronizedCollection<>(m.values(), mutex);
                return values;
            }
        }

```

```
        public boolean equals(Object o) {
            if (this == o)
                return true;
            synchronized (mutex) {return m.equals(o);}
        }
        public int hashCode() {
            synchronized (mutex) {return m.hashCode();}
        }
        public String toString() {
            synchronized (mutex) {return m.toString();}
        }
        private void writeObject(ObjectOutputStream s) throws IOException {
            synchronized (mutex) {s.defaultWriteObject();}
        }
    }

    /**
     * Returns a synchronized (thread-safe) sorted map backed by the specified
     * sorted map.  In order to guarantee serial access, it is critical that
     * <strong>all</strong> access to the backing sorted map is accomplished
     * through the returned sorted map (or its views).<p>
     *
     * It is imperative that the user manually synchronize on the returned
     * sorted map when iterating over any of its collection views, or the
     * collections views of any of its <tt>subMap</tt>, <tt>headMap</tt> or
     * <tt>tailMap</tt> views.
     * <pre>
     *  SortedMap m = Collections.synchronizedSortedMap(new TreeMap());
     *      ...
     *  Set s = m.keySet();  // Needn't be in synchronized block
     *      ...
     *  synchronized (m) {  // Synchronizing on m, not s!
     *      Iterator i = s.iterator(); // Must be in synchronized block
     *      while (i.hasNext())
     *          foo(i.next());
     *  }
     * </pre>
     * or:
     * <pre>
     *  SortedMap m = Collections.synchronizedSortedMap(new TreeMap());
     *  SortedMap m2 = m.subMap(foo, bar);
     *      ...
     *  Set s2 = m2.keySet();  // Needn't be in synchronized block
     *      ...
     *  synchronized (m) {  // Synchronizing on m, not m2 or s2!
     *      Iterator i = s.iterator(); // Must be in synchronized block
     *      while (i.hasNext())
     *          foo(i.next());
     *  }
     * </pre>
     * Failure to follow this advice may result in non-deterministic behavior.
     *
     * <p>The returned sorted map will be serializable if the specified
     * sorted map is serializable.
     *
     * @param  m the sorted map to be "wrapped" in a synchronized sorted map.
     * @return a synchronized view of the specified sorted map.
     */
    public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m) {
        return new SynchronizedSortedMap<>(m);
    }


    /**
     * @serial include
     */
    static class SynchronizedSortedMap<K,V>
        extends SynchronizedMap<K,V>
        implements SortedMap<K,V>
    {
        private static final long serialVersionUID = -8798146769416483793L;

        private final SortedMap<K,V> sm;

```

```
2155          SynchronizedSortedMap(SortedMap<K,V> m) {
2156              super(m);
2157              sm = m;
2158          }
2159          SynchronizedSortedMap(SortedMap<K,V> m, Object mutex) {
2160              super(m, mutex);
2161              sm = m;
2162          }
2163
2164          public Comparator<? super K> comparator() {
2165              synchronized (mutex) {return sm.comparator();}
2166          }
2167
2168          public SortedMap<K,V> subMap(K fromKey, K toKey) {
2169              synchronized (mutex) {
2170                  return new SynchronizedSortedMap<>(
2171                      sm.subMap(fromKey, toKey), mutex);
2172              }
2173          }
2174          public SortedMap<K,V> headMap(K toKey) {
2175              synchronized (mutex) {
2176                  return new SynchronizedSortedMap<>(sm.headMap(toKey), mutex);
2177              }
2178          }
2179          public SortedMap<K,V> tailMap(K fromKey) {
2180              synchronized (mutex) {
2181                 return new SynchronizedSortedMap<>(sm.tailMap(fromKey),mutex);
2182              }
2183          }
2184
2185          public K firstKey() {
2186              synchronized (mutex) {return sm.firstKey();}
2187          }
2188          public K lastKey() {
2189              synchronized (mutex) {return sm.lastKey();}
2190          }
2191      }
2192
2193      // Dynamically typesafe collection wrappers
2194
2195      /**
2196       * Returns a dynamically typesafe view of the specified collection.
2197       * Any attempt to insert an element of the wrong type will result in an
2198       * immediate {@link ClassCastException}.  Assuming a collection
2199       * contains no incorrectly typed elements prior to the time a
2200       * dynamically typesafe view is generated, and that all subsequent
2201       * access to the collection takes place through the view, it is
2202       * <i>guaranteed</i> that the collection cannot contain an incorrectly
2203       * typed element.
2204       *
2205       * <p>The generics mechanism in the language provides compile-time
2206       * (static) type checking, but it is possible to defeat this mechanism
2207       * with unchecked casts.  Usually this is not a problem, as the compiler
2208       * issues warnings on all such unchecked operations.  There are, however,
2209       * times when static type checking alone is not sufficient.  For example,
2210       * suppose a collection is passed to a third-party library and it is
2211       * imperative that the library code not corrupt the collection by
2212       * inserting an element of the wrong type.
2213       *
2214       * <p>Another use of dynamically typesafe views is debugging.  Suppose a
2215       * program fails with a {@code ClassCastException}, indicating that an
2216       * incorrectly typed element was put into a parameterized collection.
2217       * Unfortunately, the exception can occur at any time after the erroneous
2218       * element is inserted, so it typically provides little or no information
2219       * as to the real source of the problem.  If the problem is reproducible,
2220       * one can quickly determine its source by temporarily modifying the
2221       * program to wrap the collection with a dynamically typesafe view.
2222       * For example, this declaration:
2223       *  <pre> {@code
2224       *     Collection<String> c = new HashSet<String>();
2225       * }</pre>
2226       * may be replaced temporarily by this one:
2227       *  <pre> {@code
2228       *     Collection<String> c = Collections.checkedCollection(
```

```
     *         new HashSet<String>(), String.class);
     * }</pre>
     * Running the program again will cause it to fail at the point where
     * an incorrectly typed element is inserted into the collection, clearly
     * identifying the source of the problem.  Once the problem is fixed, the
     * modified declaration may be reverted back to the original.
     *
     * <p>The returned collection does <i>not</i> pass the hashCode and equals
     * operations through to the backing collection, but relies on
     * {@code Object}'s {@code equals} and {@code hashCode} methods.  This
     * is necessary to preserve the contracts of these operations in the case
     * that the backing collection is a set or a list.
     *
     * <p>The returned collection will be serializable if the specified
     * collection is serializable.
     *
     * <p>Since {@code null} is considered to be a value of any reference
     * type, the returned collection permits insertion of null elements
     * whenever the backing collection does.
     *
     * @param c the collection for which a dynamically typesafe view is to be
     *          returned
     * @param type the type of element that {@code c} is permitted to hold
     * @return a dynamically typesafe view of the specified collection
     * @since 1.5
     */
    public static <E> Collection<E> checkedCollection(Collection<E> c,
                                                      Class<E> type) {
        return new CheckedCollection<>(c, type);
    }

    @SuppressWarnings("unchecked")
    static <T> T[] zeroLengthArray(Class<T> type) {
        return (T[]) Array.newInstance(type, 0);
    }

    /**
     * @serial include
     */
    static class CheckedCollection<E> implements Collection<E>, Serializable {
        private static final long serialVersionUID = 1578914078182001775L;

        final Collection<E> c;
        final Class<E> type;

        void typeCheck(Object o) {
            if (o != null && !type.isInstance(o))
                throw new ClassCastException(badElementMsg(o));
        }

        private String badElementMsg(Object o) {
            return "Attempt to insert " + o.getClass() +
                " element into collection with element type " + type;
        }

        CheckedCollection(Collection<E> c, Class<E> type) {
            if (c==null || type == null)
                throw new NullPointerException();
            this.c = c;
            this.type = type;
        }

        public int size()                { return c.size(); }
        public boolean isEmpty()         { return c.isEmpty(); }
        public boolean contains(Object o) { return c.contains(o); }
        public Object[] toArray()        { return c.toArray(); }
        public <T> T[] toArray(T[] a)    { return c.toArray(a); }
        public String toString()         { return c.toString(); }
        public boolean remove(Object o)  { return c.remove(o); }
        public void clear()              {        c.clear(); }

        public boolean containsAll(Collection<?> coll) {
            return c.containsAll(coll);
        }
```

```
2303          public boolean removeAll(Collection<?> coll) {
2304              return c.removeAll(coll);
2305          }
2306          public boolean retainAll(Collection<?> coll) {
2307              return c.retainAll(coll);
2308          }
2309
2310          public Iterator<E> iterator() {
2311              final Iterator<E> it = c.iterator();
2312              return new Iterator<E>() {
2313                  public boolean hasNext() { return it.hasNext(); }
2314                  public E next()          { return it.next(); }
2315                  public void remove()     {        it.remove(); }};
2316          }
2317
2318          public boolean add(E e) {
2319              typeCheck(e);
2320              return c.add(e);
2321          }
2322
2323          private E[] zeroLengthElementArray = null; // Lazily initialized
2324
2325          private E[] zeroLengthElementArray() {
2326              return zeroLengthElementArray != null ? zeroLengthElementArray :
2327                  (zeroLengthElementArray = zeroLengthArray(type));
2328          }
2329
2330          @SuppressWarnings("unchecked")
2331          Collection<E> checkedCopyOf(Collection<? extends E> coll) {
2332              Object[] a = null;
2333              try {
2334                  E[] z = zeroLengthElementArray();
2335                  a = coll.toArray(z);
2336                  // Defend against coll violating the toArray contract
2337                  if (a.getClass() != z.getClass())
2338                      a = Arrays.copyOf(a, a.length, z.getClass());
2339              } catch (ArrayStoreException ignore) {
2340                  // To get better and consistent diagnostics,
2341                  // we call typeCheck explicitly on each element.
2342                  // We call clone() to defend against coll retaining a
2343                  // reference to the returned array and storing a bad
2344                  // element into it after it has been type checked.
2345                  a = coll.toArray().clone();
2346                  for (Object o : a)
2347                      typeCheck(o);
2348              }
2349              // A slight abuse of the type system, but safe here.
2350              return (Collection<E>) Arrays.asList(a);
2351          }
2352
2353          public boolean addAll(Collection<? extends E> coll) {
2354              // Doing things this way insulates us from concurrent changes
2355              // in the contents of coll and provides all-or-nothing
2356              // semantics (which we wouldn't get if we type-checked each
2357              // element as we added it)
2358              return c.addAll(checkedCopyOf(coll));
2359          }
2360      }
2361
2362      /**
2363       * Returns a dynamically typesafe view of the specified set.
2364       * Any attempt to insert an element of the wrong type will result in
2365       * an immediate {@link ClassCastException}.  Assuming a set contains
2366       * no incorrectly typed elements prior to the time a dynamically typesafe
2367       * view is generated, and that all subsequent access to the set
2368       * takes place through the view, it is <i>guaranteed</i> that the
2369       * set cannot contain an incorrectly typed element.
2370       *
2371       * <p>A discussion of the use of dynamically typesafe views may be
2372       * found in the documentation for the {@link #checkedCollection
2373       * checkedCollection} method.
2374       *
2375       * <p>The returned set will be serializable if the specified set is
2376       * serializable.
```

```
     *
     * <p>Since {@code null} is considered to be a value of any reference
     * type, the returned set permits insertion of null elements whenever
     * the backing set does.
     *
     * @param s the set for which a dynamically typesafe view is to be
     *          returned
     * @param type the type of element that {@code s} is permitted to hold
     * @return a dynamically typesafe view of the specified set
     * @since 1.5
     */
    public static <E> Set<E> checkedSet(Set<E> s, Class<E> type) {
        return new CheckedSet<>(s, type);
    }

    /**
     * @serial include
     */
    static class CheckedSet<E> extends CheckedCollection<E>
                                implements Set<E>, Serializable
    {
        private static final long serialVersionUID = 4694047833775013803L;

        CheckedSet(Set<E> s, Class<E> elementType) { super(s, elementType); }

        public boolean equals(Object o) { return o == this || c.equals(o); }
        public int hashCode()           { return c.hashCode(); }
    }

    /**
     * Returns a dynamically typesafe view of the specified sorted set.
     * Any attempt to insert an element of the wrong type will result in an
     * immediate {@link ClassCastException}.  Assuming a sorted set
     * contains no incorrectly typed elements prior to the time a
     * dynamically typesafe view is generated, and that all subsequent
     * access to the sorted set takes place through the view, it is
     * <i>guaranteed</i> that the sorted set cannot contain an incorrectly
     * typed element.
     *
     * <p>A discussion of the use of dynamically typesafe views may be
     * found in the documentation for the {@link #checkedCollection
     * checkedCollection} method.
     *
     * <p>The returned sorted set will be serializable if the specified sorted
     * set is serializable.
     *
     * <p>Since {@code null} is considered to be a value of any reference
     * type, the returned sorted set permits insertion of null elements
     * whenever the backing sorted set does.
     *
     * @param s the sorted set for which a dynamically typesafe view is to be
     *          returned
     * @param type the type of element that {@code s} is permitted to hold
     * @return a dynamically typesafe view of the specified sorted set
     * @since 1.5
     */
    public static <E> SortedSet<E> checkedSortedSet(SortedSet<E> s,
                                                    Class<E> type) {
        return new CheckedSortedSet<>(s, type);
    }

    /**
     * @serial include
     */
    static class CheckedSortedSet<E> extends CheckedSet<E>
        implements SortedSet<E>, Serializable
    {
        private static final long serialVersionUID = 1599911165492914959L;
        private final SortedSet<E> ss;

        CheckedSortedSet(SortedSet<E> s, Class<E> type) {
            super(s, type);
            ss = s;
        }
```

```
2451
2452            public Comparator<? super E> comparator() { return ss.comparator(); }
2453            public E first()                  { return ss.first(); }
2454            public E last()                   { return ss.last(); }
2455
2456            public SortedSet<E> subSet(E fromElement, E toElement) {
2457                return checkedSortedSet(ss.subSet(fromElement,toElement), type);
2458            }
2459            public SortedSet<E> headSet(E toElement) {
2460                return checkedSortedSet(ss.headSet(toElement), type);
2461            }
2462            public SortedSet<E> tailSet(E fromElement) {
2463                return checkedSortedSet(ss.tailSet(fromElement), type);
2464            }
2465        }
2466
2467        /**
2468         * Returns a dynamically typesafe view of the specified list.
2469         * Any attempt to insert an element of the wrong type will result in
2470         * an immediate {@link ClassCastException}.  Assuming a list contains
2471         * no incorrectly typed elements prior to the time a dynamically typesafe
2472         * view is generated, and that all subsequent access to the list
2473         * takes place through the view, it is <i>guaranteed</i> that the
2474         * list cannot contain an incorrectly typed element.
2475         *
2476         * <p>A discussion of the use of dynamically typesafe views may be
2477         * found in the documentation for the {@link #checkedCollection
2478         * checkedCollection} method.
2479         *
2480         * <p>The returned list will be serializable if the specified list
2481         * is serializable.
2482         *
2483         * <p>Since {@code null} is considered to be a value of any reference
2484         * type, the returned list permits insertion of null elements whenever
2485         * the backing list does.
2486         *
2487         * @param list the list for which a dynamically typesafe view is to be
2488         *             returned
2489         * @param type the type of element that {@code list} is permitted to hold
2490         * @return a dynamically typesafe view of the specified list
2491         * @since 1.5
2492         */
2493        public static <E> List<E> checkedList(List<E> list, Class<E> type) {
2494            return (list instanceof RandomAccess ?
2495                    new CheckedRandomAccessList<>(list, type) :
2496                    new CheckedList<>(list, type));
2497        }
2498
2499        /**
2500         * @serial include
2501         */
2502        static class CheckedList<E>
2503            extends CheckedCollection<E>
2504            implements List<E>
2505        {
2506            private static final long serialVersionUID = 65247728283967356L;
2507            final List<E> list;
2508
2509            CheckedList(List<E> list, Class<E> type) {
2510                super(list, type);
2511                this.list = list;
2512            }
2513
2514            public boolean equals(Object o)  { return o == this || list.equals(o); }
2515            public int hashCode()            { return list.hashCode(); }
2516            public E get(int index)          { return list.get(index); }
2517            public E remove(int index)       { return list.remove(index); }
2518            public int indexOf(Object o)     { return list.indexOf(o); }
2519            public int lastIndexOf(Object o) { return list.lastIndexOf(o); }
2520
2521            public E set(int index, E element) {
2522                typeCheck(element);
2523                return list.set(index, element);
2524            }
```

```
2525
2526            public void add(int index, E element) {
2527                typeCheck(element);
2528                list.add(index, element);
2529            }
2530
2531            public boolean addAll(int index, Collection<? extends E> c) {
2532                return list.addAll(index, checkedCopyOf(c));
2533            }
2534            public ListIterator<E> listIterator()   { return listIterator(0); }
2535
2536            public ListIterator<E> listIterator(final int index) {
2537                final ListIterator<E> i = list.listIterator(index);
2538
2539                return new ListIterator<E>() {
2540                    public boolean hasNext()     { return i.hasNext(); }
2541                    public E next()              { return i.next(); }
2542                    public boolean hasPrevious() { return i.hasPrevious(); }
2543                    public E previous()          { return i.previous(); }
2544                    public int nextIndex()       { return i.nextIndex(); }
2545                    public int previousIndex()   { return i.previousIndex(); }
2546                    public void remove()         {        i.remove(); }
2547
2548                    public void set(E e) {
2549                        typeCheck(e);
2550                        i.set(e);
2551                    }
2552
2553                    public void add(E e) {
2554                        typeCheck(e);
2555                        i.add(e);
2556                    }
2557                };
2558            }
2559
2560            public List<E> subList(int fromIndex, int toIndex) {
2561                return new CheckedList<>(list.subList(fromIndex, toIndex), type);
2562            }
2563        }
2564
2565        /**
2566         * @serial include
2567         */
2568        static class CheckedRandomAccessList<E> extends CheckedList<E>
2569                                                implements RandomAccess
2570        {
2571            private static final long serialVersionUID = 1638200125423088369L;
2572
2573            CheckedRandomAccessList(List<E> list, Class<E> type) {
2574                super(list, type);
2575            }
2576
2577            public List<E> subList(int fromIndex, int toIndex) {
2578                return new CheckedRandomAccessList<>(
2579                    list.subList(fromIndex, toIndex), type);
2580            }
2581        }
2582
2583        /**
2584         * Returns a dynamically typesafe view of the specified map.
2585         * Any attempt to insert a mapping whose key or value have the wrong
2586         * type will result in an immediate {@link ClassCastException}.
2587         * Similarly, any attempt to modify the value currently associated with
2588         * a key will result in an immediate {@link ClassCastException},
2589         * whether the modification is attempted directly through the map
2590         * itself, or through a {@link Map.Entry} instance obtained from the
2591         * map's {@link Map#entrySet() entry set} view.
2592         *
2593         * <p>Assuming a map contains no incorrectly typed keys or values
2594         * prior to the time a dynamically typesafe view is generated, and
2595         * that all subsequent access to the map takes place through the view
2596         * (or one of its collection views), it is <i>guaranteed</i> that the
2597         * map cannot contain an incorrectly typed key or value.
2598         *
```

```
     * <p>A discussion of the use of dynamically typesafe views may be
     * found in the documentation for the {@link #checkedCollection
     * checkedCollection} method.
     *
     * <p>The returned map will be serializable if the specified map is
     * serializable.
     *
     * <p>Since {@code null} is considered to be a value of any reference
     * type, the returned map permits insertion of null keys or values
     * whenever the backing map does.
     *
     * @param m the map for which a dynamically typesafe view is to be
     *          returned
     * @param keyType the type of key that {@code m} is permitted to hold
     * @param valueType the type of value that {@code m} is permitted to hold
     * @return a dynamically typesafe view of the specified map
     * @since 1.5
     */
    public static <K, V> Map<K, V> checkedMap(Map<K, V> m,
                                              Class<K> keyType,
                                              Class<V> valueType) {
        return new CheckedMap<>(m, keyType, valueType);
    }


    /**
     * @serial include
     */
    private static class CheckedMap<K,V>
        implements Map<K,V>, Serializable
    {
        private static final long serialVersionUID = 5742860141034234728L;

        private final Map<K, V> m;
        final Class<K> keyType;
        final Class<V> valueType;

        private void typeCheck(Object key, Object value) {
            if (key != null && !keyType.isInstance(key))
                throw new ClassCastException(badKeyMsg(key));

            if (value != null && !valueType.isInstance(value))
                throw new ClassCastException(badValueMsg(value));
        }

        private String badKeyMsg(Object key) {
            return "Attempt to insert " + key.getClass() +
                " key into map with key type " + keyType;
        }

        private String badValueMsg(Object value) {
            return "Attempt to insert " + value.getClass() +
                " value into map with value type " + valueType;
        }

        CheckedMap(Map<K, V> m, Class<K> keyType, Class<V> valueType) {
            if (m == null || keyType == null || valueType == null)
                throw new NullPointerException();
            this.m = m;
            this.keyType = keyType;
            this.valueType = valueType;
        }

        public int size()                       { return m.size(); }
        public boolean isEmpty()                { return m.isEmpty(); }
        public boolean containsKey(Object key) { return m.containsKey(key); }
        public boolean containsValue(Object v) { return m.containsValue(v); }
        public V get(Object key)                { return m.get(key); }
        public V remove(Object key)             { return m.remove(key); }
        public void clear()                     { m.clear(); }
        public Set<K> keySet()                  { return m.keySet(); }
        public Collection<V> values()           { return m.values(); }
        public boolean equals(Object o)         { return o == this || m.equals(o); }
        public int hashCode()                   { return m.hashCode(); }
```

```
2673        public String toString()              { return m.toString(); }
2674
2675        public V put(K key, V value) {
2676            typeCheck(key, value);
2677            return m.put(key, value);
2678        }
2679
2680        @SuppressWarnings("unchecked")
2681        public void putAll(Map<? extends K, ? extends V> t) {
2682            // Satisfy the following goals:
2683            // - good diagnostics in case of type mismatch
2684            // - all-or-nothing semantics
2685            // - protection from malicious t
2686            // - correct behavior if t is a concurrent map
2687            Object[] entries = t.entrySet().toArray();
2688            List<Map.Entry<K,V>> checked = new ArrayList<>(entries.length);
2689            for (Object o : entries) {
2690                Map.Entry<?,?> e = (Map.Entry<?,?>) o;
2691                Object k = e.getKey();
2692                Object v = e.getValue();
2693                typeCheck(k, v);
2694                checked.add(
2695                    new AbstractMap.SimpleImmutableEntry<>((K) k, (V) v));
2696            }
2697            for (Map.Entry<K,V> e : checked)
2698                m.put(e.getKey(), e.getValue());
2699        }
2700
2701        private transient Set<Map.Entry<K,V>> entrySet = null;
2702
2703        public Set<Map.Entry<K,V>> entrySet() {
2704            if (entrySet==null)
2705                entrySet = new CheckedEntrySet<>(m.entrySet(), valueType);
2706            return entrySet;
2707        }
2708
2709        /**
2710         * We need this class in addition to CheckedSet as Map.Entry permits
2711         * modification of the backing Map via the setValue operation.  This
2712         * class is subtle: there are many possible attacks that must be
2713         * thwarted.
2714         *
2715         * @serial exclude
2716         */
2717        static class CheckedEntrySet<K,V> implements Set<Map.Entry<K,V>> {
2718            private final Set<Map.Entry<K,V>> s;
2719            private final Class<V> valueType;
2720
2721            CheckedEntrySet(Set<Map.Entry<K, V>> s, Class<V> valueType) {
2722                this.s = s;
2723                this.valueType = valueType;
2724            }
2725
2726            public int size()        { return s.size(); }
2727            public boolean isEmpty() { return s.isEmpty(); }
2728            public String toString() { return s.toString(); }
2729            public int hashCode()    { return s.hashCode(); }
2730            public void clear()      {        s.clear(); }
2731
2732            public boolean add(Map.Entry<K, V> e) {
2733                throw new UnsupportedOperationException();
2734            }
2735            public boolean addAll(Collection<? extends Map.Entry<K, V>> coll) {
2736                throw new UnsupportedOperationException();
2737            }
2738
2739            public Iterator<Map.Entry<K,V>> iterator() {
2740                final Iterator<Map.Entry<K, V>> i = s.iterator();
2741                final Class<V> valueType = this.valueType;
2742
2743                return new Iterator<Map.Entry<K,V>>() {
2744                    public boolean hasNext() { return i.hasNext(); }
2745                    public void remove()     { i.remove(); }
2746
```

```
                public Map.Entry<K,V> next() {
                    return checkedEntry(i.next(), valueType);
                }
            };
        }

        @SuppressWarnings("unchecked")
        public Object[] toArray() {
            Object[] source = s.toArray();

            /*
             * Ensure that we don't get an ArrayStoreException even if
             * s.toArray returns an array of something other than Object
             */
            Object[] dest = (CheckedEntry.class.isInstance(
                source.getClass().getComponentType()) ? source :
                            new Object[source.length]);

            for (int i = 0; i < source.length; i++)
                dest[i] = checkedEntry((Map.Entry<K,V>)source[i],
                                       valueType);
            return dest;
        }

        @SuppressWarnings("unchecked")
        public <T> T[] toArray(T[] a) {
            // We don't pass a to s.toArray, to avoid window of
            // vulnerability wherein an unscrupulous multithreaded client
            // could get his hands on raw (unwrapped) Entries from s.
            T[] arr = s.toArray(a.length==0 ? a : Arrays.copyOf(a, 0));

            for (int i=0; i<arr.length; i++)
                arr[i] = (T) checkedEntry((Map.Entry<K,V>)arr[i],
                                          valueType);
            if (arr.length > a.length)
                return arr;

            System.arraycopy(arr, 0, a, 0, arr.length);
            if (a.length > arr.length)
                a[arr.length] = null;
            return a;
        }

        /**
         * This method is overridden to protect the backing set against
         * an object with a nefarious equals function that senses
         * that the equality-candidate is Map.Entry and calls its
         * setValue method.
         */
        public boolean contains(Object o) {
            if (!(o instanceof Map.Entry))
                return false;
            Map.Entry<?,?> e = (Map.Entry<?,?>) o;
            return s.contains(
                (e instanceof CheckedEntry) ? e : checkedEntry(e, valueType));
        }

        /**
         * The bulk collection methods are overridden to protect
         * against an unscrupulous collection whose contains(Object o)
         * method senses when o is a Map.Entry, and calls o.setValue.
         */
        public boolean containsAll(Collection<?> c) {
            for (Object o : c)
                if (!contains(o)) // Invokes safe contains() above
                    return false;
            return true;
        }

        public boolean remove(Object o) {
            if (!(o instanceof Map.Entry))
                return false;
            return s.remove(new AbstractMap.SimpleImmutableEntry
                            <>((Map.Entry<?,?>)o));
```

```
2821            }
2822
2823            public boolean removeAll(Collection<?> c) {
2824                return batchRemove(c, false);
2825            }
2826            public boolean retainAll(Collection<?> c) {
2827                return batchRemove(c, true);
2828            }
2829            private boolean batchRemove(Collection<?> c, boolean complement) {
2830                boolean modified = false;
2831                Iterator<Map.Entry<K,V>> it = iterator();
2832                while (it.hasNext()) {
2833                    if (c.contains(it.next()) != complement) {
2834                        it.remove();
2835                        modified = true;
2836                    }
2837                }
2838                return modified;
2839            }
2840
2841            public boolean equals(Object o) {
2842                if (o == this)
2843                    return true;
2844                if (!(o instanceof Set))
2845                    return false;
2846                Set<?> that = (Set<?>) o;
2847                return that.size() == s.size()
2848                    && containsAll(that); // Invokes safe containsAll() above
2849            }
2850
2851            static <K,V,T> CheckedEntry<K,V,T> checkedEntry(Map.Entry<K,V> e,
2852                                                            Class<T> valueType) {
2853                return new CheckedEntry<>(e, valueType);
2854            }
2855
2856            /**
2857             * This "wrapper class" serves two purposes: it prevents
2858             * the client from modifying the backing Map, by short-circuiting
2859             * the setValue method, and it protects the backing Map against
2860             * an ill-behaved Map.Entry that attempts to modify another
2861             * Map.Entry when asked to perform an equality check.
2862             */
2863            private static class CheckedEntry<K,V,T> implements Map.Entry<K,V> {
2864                private final Map.Entry<K, V> e;
2865                private final Class<T> valueType;
2866
2867                CheckedEntry(Map.Entry<K, V> e, Class<T> valueType) {
2868                    this.e = e;
2869                    this.valueType = valueType;
2870                }
2871
2872                public K getKey()        { return e.getKey(); }
2873                public V getValue()      { return e.getValue(); }
2874                public int hashCode()    { return e.hashCode(); }
2875                public String toString() { return e.toString(); }
2876
2877                public V setValue(V value) {
2878                    if (value != null && !valueType.isInstance(value))
2879                        throw new ClassCastException(badValueMsg(value));
2880                    return e.setValue(value);
2881                }
2882
2883                private String badValueMsg(Object value) {
2884                    return "Attempt to insert " + value.getClass() +
2885                        " value into map with value type " + valueType;
2886                }
2887
2888                public boolean equals(Object o) {
2889                    if (o == this)
2890                        return true;
2891                    if (!(o instanceof Map.Entry))
2892                        return false;
2893                    return e.equals(new AbstractMap.SimpleImmutableEntry
2894                                    <>((Map.Entry<?,?>)o));
```

```
                }
            }
        }
    }

    /**
     * Returns a dynamically typesafe view of the specified sorted map.
     * Any attempt to insert a mapping whose key or value have the wrong
     * type will result in an immediate {@link ClassCastException}.
     * Similarly, any attempt to modify the value currently associated with
     * a key will result in an immediate {@link ClassCastException},
     * whether the modification is attempted directly through the map
     * itself, or through a {@link Map.Entry} instance obtained from the
     * map's {@link Map#entrySet() entry set} view.
     *
     * <p>Assuming a map contains no incorrectly typed keys or values
     * prior to the time a dynamically typesafe view is generated, and
     * that all subsequent access to the map takes place through the view
     * (or one of its collection views), it is <i>guaranteed</i> that the
     * map cannot contain an incorrectly typed key or value.
     *
     * <p>A discussion of the use of dynamically typesafe views may be
     * found in the documentation for the {@link #checkedCollection
     * checkedCollection} method.
     *
     * <p>The returned map will be serializable if the specified map is
     * serializable.
     *
     * <p>Since {@code null} is considered to be a value of any reference
     * type, the returned map permits insertion of null keys or values
     * whenever the backing map does.
     *
     * @param m the map for which a dynamically typesafe view is to be
     *          returned
     * @param keyType the type of key that {@code m} is permitted to hold
     * @param valueType the type of value that {@code m} is permitted to hold
     * @return a dynamically typesafe view of the specified map
     * @since 1.5
     */
    public static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K, V> m,
                                                        Class<K> keyType,
                                                        Class<V> valueType) {
        return new CheckedSortedMap<>(m, keyType, valueType);
    }

    /**
     * @serial include
     */
    static class CheckedSortedMap<K,V> extends CheckedMap<K,V>
        implements SortedMap<K,V>, Serializable
    {
        private static final long serialVersionUID = 1599671320688067438L;

        private final SortedMap<K, V> sm;

        CheckedSortedMap(SortedMap<K, V> m,
                         Class<K> keyType, Class<V> valueType) {
            super(m, keyType, valueType);
            sm = m;
        }

        public Comparator<? super K> comparator() { return sm.comparator(); }
        public K firstKey()                       { return sm.firstKey(); }
        public K lastKey()                        { return sm.lastKey(); }

        public SortedMap<K,V> subMap(K fromKey, K toKey) {
            return checkedSortedMap(sm.subMap(fromKey, toKey),
                                    keyType, valueType);
        }
        public SortedMap<K,V> headMap(K toKey) {
            return checkedSortedMap(sm.headMap(toKey), keyType, valueType);
        }
        public SortedMap<K,V> tailMap(K fromKey) {
            return checkedSortedMap(sm.tailMap(fromKey), keyType, valueType);
```

```
2969            }
2970        }
2971
2972        // Empty collections
2973
2974        /**
2975         * Returns an iterator that has no elements.  More precisely,
2976         *
2977         * <ul compact>
2978         *
2979         * <li>{@link Iterator#hasNext hasNext} always returns {@code
2980         * false}.
2981         *
2982         * <li>{@link Iterator#next next} always throws {@link
2983         * NoSuchElementException}.
2984         *
2985         * <li>{@link Iterator#remove remove} always throws {@link
2986         * IllegalStateException}.
2987         *
2988         * </ul>
2989         *
2990         * <p>Implementations of this method are permitted, but not
2991         * required, to return the same object from multiple invocations.
2992         *
2993         * @return an empty iterator
2994         * @since 1.7
2995         */
2996        @SuppressWarnings("unchecked")
2997        public static <T> Iterator<T> emptyIterator() {
2998            return (Iterator<T>) EmptyIterator.EMPTY_ITERATOR;
2999        }
3000
3001        private static class EmptyIterator<E> implements Iterator<E> {
3002            static final EmptyIterator<Object> EMPTY_ITERATOR
3003                = new EmptyIterator<>();
3004
3005            public boolean hasNext() { return false; }
3006            public E next() { throw new NoSuchElementException(); }
3007            public void remove() { throw new IllegalStateException(); }
3008        }
3009
3010        /**
3011         * Returns a list iterator that has no elements.  More precisely,
3012         *
3013         * <ul compact>
3014         *
3015         * <li>{@link Iterator#hasNext hasNext} and {@link
3016         * ListIterator#hasPrevious hasPrevious} always return {@code
3017         * false}.
3018         *
3019         * <li>{@link Iterator#next next} and {@link ListIterator#previous
3020         * previous} always throw {@link NoSuchElementException}.
3021         *
3022         * <li>{@link Iterator#remove remove} and {@link ListIterator#set
3023         * set} always throw {@link IllegalStateException}.
3024         *
3025         * <li>{@link ListIterator#add add} always throws {@link
3026         * UnsupportedOperationException}.
3027         *
3028         * <li>{@link ListIterator#nextIndex nextIndex} always returns
3029         * {@code 0} .
3030         *
3031         * <li>{@link ListIterator#previousIndex previousIndex} always
3032         * returns {@code -1}.
3033         *
3034         * </ul>
3035         *
3036         * <p>Implementations of this method are permitted, but not
3037         * required, to return the same object from multiple invocations.
3038         *
3039         * @return an empty list iterator
3040         * @since 1.7
3041         */
3042        @SuppressWarnings("unchecked")
```

```
3043      public static <T> ListIterator<T> emptyListIterator() {
3044          return (ListIterator<T>) EmptyListIterator.EMPTY_ITERATOR;
3045      }
3046
3047      private static class EmptyListIterator<E>
3048          extends EmptyIterator<E>
3049          implements ListIterator<E>
3050      {
3051          static final EmptyListIterator<Object> EMPTY_ITERATOR
3052              = new EmptyListIterator<>();
3053
3054          public boolean hasPrevious() { return false; }
3055          public E previous() { throw new NoSuchElementException(); }
3056          public int nextIndex()     { return 0; }
3057          public int previousIndex() { return -1; }
3058          public void set(E e) { throw new IllegalStateException(); }
3059          public void add(E e) { throw new UnsupportedOperationException(); }
3060      }
3061
3062      /**
3063       * Returns an enumeration that has no elements.  More precisely,
3064       *
3065       * <ul compact>
3066       *
3067       * <li>{@link Enumeration#hasMoreElements hasMoreElements} always
3068       * returns {@code false}.
3069       *
3070       * <li> {@link Enumeration#nextElement nextElement} always throws
3071       * {@link NoSuchElementException}.
3072       *
3073       * </ul>
3074       *
3075       * <p>Implementations of this method are permitted, but not
3076       * required, to return the same object from multiple invocations.
3077       *
3078       * @return an empty enumeration
3079       * @since 1.7
3080       */
3081      @SuppressWarnings("unchecked")
3082      public static <T> Enumeration<T> emptyEnumeration() {
3083          return (Enumeration<T>) EmptyEnumeration.EMPTY_ENUMERATION;
3084      }
3085
3086      private static class EmptyEnumeration<E> implements Enumeration<E> {
3087          static final EmptyEnumeration<Object> EMPTY_ENUMERATION
3088              = new EmptyEnumeration<>();
3089
3090          public boolean hasMoreElements() { return false; }
3091          public E nextElement() { throw new NoSuchElementException(); }
3092      }
3093
3094      /**
3095       * The empty set (immutable).  This set is serializable.
3096       *
3097       * @see #emptySet()
3098       */
3099      @SuppressWarnings("unchecked")
3100      public static final Set EMPTY_SET = new EmptySet<>();
3101
3102      /**
3103       * Returns the empty set (immutable).  This set is serializable.
3104       * Unlike the like-named field, this method is parameterized.
3105       *
3106       * <p>This example illustrates the type-safe way to obtain an empty set:
3107       * <pre>
3108       *     Set&lt;String&gt; s = Collections.emptySet();
3109       * </pre>
3110       * Implementation note:  Implementations of this method need not
3111       * create a separate <tt>Set</tt> object for each call.   Using this
3112       * method is likely to have comparable cost to using the like-named
3113       * field.  (Unlike this method, the field does not provide type safety.)
3114       *
3115       * @see #EMPTY_SET
3116       * @since 1.5
```

```
3117      */
3118     @SuppressWarnings("unchecked")
3119     public static final <T> Set<T> emptySet() {
3120         return (Set<T>) EMPTY_SET;
3121     }
3122
3123     /**
3124      * @serial include
3125      */
3126     private static class EmptySet<E>
3127         extends AbstractSet<E>
3128         implements Serializable
3129     {
3130         private static final long serialVersionUID = 1582296315990362920L;
3131
3132         public Iterator<E> iterator() { return emptyIterator(); }
3133
3134         public int size() {return 0;}
3135         public boolean isEmpty() {return true;}
3136
3137         public boolean contains(Object obj) {return false;}
3138         public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
3139
3140         public Object[] toArray() { return new Object[0]; }
3141
3142         public <T> T[] toArray(T[] a) {
3143             if (a.length > 0)
3144                 a[0] = null;
3145             return a;
3146         }
3147
3148         // Preserves singleton property
3149         private Object readResolve() {
3150             return EMPTY_SET;
3151         }
3152     }
3153
3154     /**
3155      * The empty list (immutable).  This list is serializable.
3156      *
3157      * @see #emptyList()
3158      */
3159     @SuppressWarnings("unchecked")
3160     public static final List EMPTY_LIST = new EmptyList<>();
3161
3162     /**
3163      * Returns the empty list (immutable).  This list is serializable.
3164      *
3165      * <p>This example illustrates the type-safe way to obtain an empty list:
3166      * <pre>
3167      *     List&lt;String&gt; s = Collections.emptyList();
3168      * </pre>
3169      * Implementation note:  Implementations of this method need not
3170      * create a separate <tt>List</tt> object for each call.   Using this
3171      * method is likely to have comparable cost to using the like-named
3172      * field.  (Unlike this method, the field does not provide type safety.)
3173      *
3174      * @see #EMPTY_LIST
3175      * @since 1.5
3176      */
3177     @SuppressWarnings("unchecked")
3178     public static final <T> List<T> emptyList() {
3179         return (List<T>) EMPTY_LIST;
3180     }
3181
3182     /**
3183      * @serial include
3184      */
3185     private static class EmptyList<E>
3186         extends AbstractList<E>
3187         implements RandomAccess, Serializable {
3188         private static final long serialVersionUID = 8842843931221139166L;
3189
3190         public Iterator<E> iterator() {
```

```
3191            return emptyIterator();
3192        }
3193        public ListIterator<E> listIterator() {
3194            return emptyListIterator();
3195        }
3196
3197        public int size() {return 0;}
3198        public boolean isEmpty() {return true;}
3199
3200        public boolean contains(Object obj) {return false;}
3201        public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
3202
3203        public Object[] toArray() { return new Object[0]; }
3204
3205        public <T> T[] toArray(T[] a) {
3206            if (a.length > 0)
3207                a[0] = null;
3208            return a;
3209        }
3210
3211        public E get(int index) {
3212            throw new IndexOutOfBoundsException("Index: "+index);
3213        }
3214
3215        public boolean equals(Object o) {
3216            return (o instanceof List) && ((List<?>)o).isEmpty();
3217        }
3218
3219        public int hashCode() { return 1; }
3220
3221        // Preserves singleton property
3222        private Object readResolve() {
3223            return EMPTY_LIST;
3224        }
3225    }
3226
3227    /**
3228     * The empty map (immutable).  This map is serializable.
3229     *
3230     * @see #emptyMap()
3231     * @since 1.3
3232     */
3233    @SuppressWarnings("unchecked")
3234    public static final Map EMPTY_MAP = new EmptyMap<>();
3235
3236    /**
3237     * Returns the empty map (immutable).  This map is serializable.
3238     *
3239     * <p>This example illustrates the type-safe way to obtain an empty set:
3240     * <pre>
3241     *     Map&lt;String, Date&gt; s = Collections.emptyMap();
3242     * </pre>
3243     * Implementation note:  Implementations of this method need not
3244     * create a separate <tt>Map</tt> object for each call.   Using this
3245     * method is likely to have comparable cost to using the like-named
3246     * field.  (Unlike this method, the field does not provide type safety.)
3247     *
3248     * @see #EMPTY_MAP
3249     * @since 1.5
3250     */
3251    @SuppressWarnings("unchecked")
3252    public static final <K,V> Map<K,V> emptyMap() {
3253        return (Map<K,V>) EMPTY_MAP;
3254    }
3255
3256    /**
3257     * @serial include
3258     */
3259    private static class EmptyMap<K,V>
3260        extends AbstractMap<K,V>
3261        implements Serializable
3262    {
3263        private static final long serialVersionUID = 6428348081105594320L;
3264
```

```
        public int size()                           {return 0;}
        public boolean isEmpty()                    {return true;}
        public boolean containsKey(Object key)     {return false;}
        public boolean containsValue(Object value) {return false;}
        public V get(Object key)                    {return null;}
        public Set<K> keySet()                      {return emptySet();}
        public Collection<V> values()               {return emptySet();}
        public Set<Map.Entry<K,V>> entrySet()       {return emptySet();}

        public boolean equals(Object o) {
            return (o instanceof Map) && ((Map<?,?>)o).isEmpty();
        }

        public int hashCode()                       {return 0;}

        // Preserves singleton property
        private Object readResolve() {
            return EMPTY_MAP;
        }
    }

    // Singleton collections

    /**
     * Returns an immutable set containing only the specified object.
     * The returned set is serializable.
     *
     * @param o the sole object to be stored in the returned set.
     * @return an immutable set containing only the specified object.
     */
    public static <T> Set<T> singleton(T o) {
        return new SingletonSet<>(o);
    }

    static <E> Iterator<E> singletonIterator(final E e) {
        return new Iterator<E>() {
            private boolean hasNext = true;
            public boolean hasNext() {
                return hasNext;
            }
            public E next() {
                if (hasNext) {
                    hasNext = false;
                    return e;
                }
                throw new NoSuchElementException();
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }

    /**
     * @serial include
     */
    private static class SingletonSet<E>
        extends AbstractSet<E>
        implements Serializable
    {
        private static final long serialVersionUID = 3193687207550431679L;

        private final E element;

        SingletonSet(E e) {element = e;}

        public Iterator<E> iterator() {
            return singletonIterator(element);
        }

        public int size() {return 1;}

        public boolean contains(Object o) {return eq(o, element);}
    }
```

```
3339
3340      /**
3341       * Returns an immutable list containing only the specified object.
3342       * The returned list is serializable.
3343       *
3344       * @param o the sole object to be stored in the returned list.
3345       * @return an immutable list containing only the specified object.
3346       * @since 1.3
3347       */
3348      public static <T> List<T> singletonList(T o) {
3349          return new SingletonList<>(o);
3350      }
3351
3352      /**
3353       * @serial include
3354       */
3355      private static class SingletonList<E>
3356          extends AbstractList<E>
3357          implements RandomAccess, Serializable {
3358
3359          private static final long serialVersionUID = 3093736618740652951L;
3360
3361          private final E element;
3362
3363          SingletonList(E obj)                {element = obj;}
3364
3365          public Iterator<E> iterator() {
3366              return singletonIterator(element);
3367          }
3368
3369          public int size()                   {return 1;}
3370
3371          public boolean contains(Object obj) {return eq(obj, element);}
3372
3373          public E get(int index) {
3374              if (index != 0)
3375                throw new IndexOutOfBoundsException("Index: "+index+", Size: 1");
3376              return element;
3377          }
3378      }
3379
3380      /**
3381       * Returns an immutable map, mapping only the specified key to the
3382       * specified value.  The returned map is serializable.
3383       *
3384       * @param key the sole key to be stored in the returned map.
3385       * @param value the value to which the returned map maps <tt>key</tt>.
3386       * @return an immutable map containing only the specified key-value
3387       *         mapping.
3388       * @since 1.3
3389       */
3390      public static <K,V> Map<K,V> singletonMap(K key, V value) {
3391          return new SingletonMap<>(key, value);
3392      }
3393
3394      /**
3395       * @serial include
3396       */
3397      private static class SingletonMap<K,V>
3398            extends AbstractMap<K,V>
3399            implements Serializable {
3400          private static final long serialVersionUID = -6979724477215052911L;
3401
3402          private final K k;
3403          private final V v;
3404
3405          SingletonMap(K key, V value) {
3406              k = key;
3407              v = value;
3408          }
3409
3410          public int size()                           {return 1;}
3411
3412          public boolean isEmpty()                    {return false;}
```

```
3413
3414            public boolean containsKey(Object key)     {return eq(key, k);}
3415
3416            public boolean containsValue(Object value) {return eq(value, v);}
3417
3418            public V get(Object key)                   {return (eq(key, k) ? v : null);}
3419
3420            private transient Set<K> keySet = null;
3421            private transient Set<Map.Entry<K,V>> entrySet = null;
3422            private transient Collection<V> values = null;
3423
3424            public Set<K> keySet() {
3425                if (keySet==null)
3426                    keySet = singleton(k);
3427                return keySet;
3428            }
3429
3430            public Set<Map.Entry<K,V>> entrySet() {
3431                if (entrySet==null)
3432                    entrySet = Collections.<Map.Entry<K,V>>singleton(
3433                        new SimpleImmutableEntry<>(k, v));
3434                return entrySet;
3435            }
3436
3437            public Collection<V> values() {
3438                if (values==null)
3439                    values = singleton(v);
3440                return values;
3441            }
3442
3443        }
3444
3445        // Miscellaneous
3446
3447        /**
3448         * Returns an immutable list consisting of <tt>n</tt> copies of the
3449         * specified object.  The newly allocated data object is tiny (it contains
3450         * a single reference to the data object).  This method is useful in
3451         * combination with the <tt>List.addAll</tt> method to grow lists.
3452         * The returned list is serializable.
3453         *
3454         * @param  n the number of elements in the returned list.
3455         * @param  o the element to appear repeatedly in the returned list.
3456         * @return an immutable list consisting of <tt>n</tt> copies of the
3457         *         specified object.
3458         * @throws IllegalArgumentException if {@code n < 0}
3459         * @see    List#addAll(Collection)
3460         * @see    List#addAll(int, Collection)
3461         */
3462        public static <T> List<T> nCopies(int n, T o) {
3463            if (n < 0)
3464                throw new IllegalArgumentException("List length = " + n);
3465            return new CopiesList<>(n, o);
3466        }
3467
3468        /**
3469         * @serial include
3470         */
3471        private static class CopiesList<E>
3472            extends AbstractList<E>
3473            implements RandomAccess, Serializable
3474        {
3475            private static final long serialVersionUID = 2739099268398711800L;
3476
3477            final int n;
3478            final E element;
3479
3480            CopiesList(int n, E e) {
3481                assert n >= 0;
3482                this.n = n;
3483                element = e;
3484            }
3485
3486            public int size() {
```

```
3487            return n;
3488        }
3489
3490        public boolean contains(Object obj) {
3491            return n != 0 && eq(obj, element);
3492        }
3493
3494        public int indexOf(Object o) {
3495            return contains(o) ? 0 : -1;
3496        }
3497
3498        public int lastIndexOf(Object o) {
3499            return contains(o) ? n - 1 : -1;
3500        }
3501
3502        public E get(int index) {
3503            if (index < 0 || index >= n)
3504                throw new IndexOutOfBoundsException("Index: "+index+
3505                                                    ", Size: "+n);
3506            return element;
3507        }
3508
3509        public Object[] toArray() {
3510            final Object[] a = new Object[n];
3511            if (element != null)
3512                Arrays.fill(a, 0, n, element);
3513            return a;
3514        }
3515
3516        public <T> T[] toArray(T[] a) {
3517            final int n = this.n;
3518            if (a.length < n) {
3519                a = (T[])java.lang.reflect.Array
3520                    .newInstance(a.getClass().getComponentType(), n);
3521                if (element != null)
3522                    Arrays.fill(a, 0, n, element);
3523            } else {
3524                Arrays.fill(a, 0, n, element);
3525                if (a.length > n)
3526                    a[n] = null;
3527            }
3528            return a;
3529        }
3530
3531        public List<E> subList(int fromIndex, int toIndex) {
3532            if (fromIndex < 0)
3533                throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);
3534            if (toIndex > n)
3535                throw new IndexOutOfBoundsException("toIndex = " + toIndex);
3536            if (fromIndex > toIndex)
3537                throw new IllegalArgumentException("fromIndex(" + fromIndex +
3538                                                   ") > toIndex(" + toIndex + ")");
3539            return new CopiesList<>(toIndex - fromIndex, element);
3540        }
3541    }
3542
3543    /**
3544     * Returns a comparator that imposes the reverse of the <em>natural
3545     * ordering</em> on a collection of objects that implement the
3546     * {@code Comparable} interface.  (The natural ordering is the ordering
3547     * imposed by the objects' own {@code compareTo} method.)  This enables a
3548     * simple idiom for sorting (or maintaining) collections (or arrays) of
3549     * objects that implement the {@code Comparable} interface in
3550     * reverse-natural-order.  For example, suppose {@code a} is an array of
3551     * strings. Then: <pre>
3552     *          Arrays.sort(a, Collections.reverseOrder());
3553     * </pre> sorts the array in reverse-lexicographic (alphabetical) order.<p>
3554     *
3555     * The returned comparator is serializable.
3556     *
3557     * @return A comparator that imposes the reverse of the <i>natural
3558     *         ordering</i> on a collection of objects that implement
3559     *         the <tt>Comparable</tt> interface.
3560     * @see Comparable
```

```
3561      */
3562     public static <T> Comparator<T> reverseOrder() {
3563         return (Comparator<T>) ReverseComparator.REVERSE_ORDER;
3564     }
3565
3566     /**
3567      * @serial include
3568      */
3569     private static class ReverseComparator
3570         implements Comparator<Comparable<Object>>, Serializable {
3571
3572         private static final long serialVersionUID = 7207038068494060240L;
3573
3574         static final ReverseComparator REVERSE_ORDER
3575             = new ReverseComparator();
3576
3577         public int compare(Comparable<Object> c1, Comparable<Object> c2) {
3578             return c2.compareTo(c1);
3579         }
3580
3581         private Object readResolve() { return reverseOrder(); }
3582     }
3583
3584     /**
3585      * Returns a comparator that imposes the reverse ordering of the specified
3586      * comparator.  If the specified comparator is {@code null}, this method is
3587      * equivalent to {@link #reverseOrder()} (in other words, it returns a
3588      * comparator that imposes the reverse of the <em>natural ordering</em> on
3589      * a collection of objects that implement the Comparable interface).
3590      *
3591      * <p>The returned comparator is serializable (assuming the specified
3592      * comparator is also serializable or {@code null}).
3593      *
3594      * @param cmp a comparator who's ordering is to be reversed by the returned
3595      * comparator or {@code null}
3596      * @return A comparator that imposes the reverse ordering of the
3597      *         specified comparator.
3598      * @since 1.5
3599      */
3600     public static <T> Comparator<T> reverseOrder(Comparator<T> cmp) {
3601         if (cmp == null)
3602             return reverseOrder();
3603
3604         if (cmp instanceof ReverseComparator2)
3605             return ((ReverseComparator2<T>)cmp).cmp;
3606
3607         return new ReverseComparator2<>(cmp);
3608     }
3609
3610     /**
3611      * @serial include
3612      */
3613     private static class ReverseComparator2<T> implements Comparator<T>,
3614         Serializable
3615     {
3616         private static final long serialVersionUID = 4374092139857L;
3617
3618         /**
3619          * The comparator specified in the static factory.  This will never
3620          * be null, as the static factory returns a ReverseComparator
3621          * instance if its argument is null.
3622          *
3623          * @serial
3624          */
3625         final Comparator<T> cmp;
3626
3627         ReverseComparator2(Comparator<T> cmp) {
3628             assert cmp != null;
3629             this.cmp = cmp;
3630         }
3631
3632         public int compare(T t1, T t2) {
3633             return cmp.compare(t2, t1);
3634         }
```

```
3635
3636            public boolean equals(Object o) {
3637                return (o == this) ||
3638                    (o instanceof ReverseComparator2 &&
3639                     cmp.equals(((ReverseComparator2)o).cmp));
3640            }
3641
3642            public int hashCode() {
3643                return cmp.hashCode() ^ Integer.MIN_VALUE;
3644            }
3645        }
3646
3647        /**
3648         * Returns an enumeration over the specified collection.  This provides
3649         * interoperability with legacy APIs that require an enumeration
3650         * as input.
3651         *
3652         * @param c the collection for which an enumeration is to be returned.
3653         * @return an enumeration over the specified collection.
3654         * @see Enumeration
3655         */
3656        public static <T> Enumeration<T> enumeration(final Collection<T> c) {
3657            return new Enumeration<T>() {
3658                private final Iterator<T> i = c.iterator();
3659
3660                public boolean hasMoreElements() {
3661                    return i.hasNext();
3662                }
3663
3664                public T nextElement() {
3665                    return i.next();
3666                }
3667            };
3668        }
3669
3670        /**
3671         * Returns an array list containing the elements returned by the
3672         * specified enumeration in the order they are returned by the
3673         * enumeration.  This method provides interoperability between
3674         * legacy APIs that return enumerations and new APIs that require
3675         * collections.
3676         *
3677         * @param e enumeration providing elements for the returned
3678         *          array list
3679         * @return an array list containing the elements returned
3680         *         by the specified enumeration.
3681         * @since 1.4
3682         * @see Enumeration
3683         * @see ArrayList
3684         */
3685        public static <T> ArrayList<T> list(Enumeration<T> e) {
3686            ArrayList<T> l = new ArrayList<>();
3687            while (e.hasMoreElements())
3688                l.add(e.nextElement());
3689            return l;
3690        }
3691
3692        /**
3693         * Returns true if the specified arguments are equal, or both null.
3694         */
3695        static boolean eq(Object o1, Object o2) {
3696            return o1==null ? o2==null : o1.equals(o2);
3697        }
3698
3699        /**
3700         * Returns the number of elements in the specified collection equal to the
3701         * specified object.  More formally, returns the number of elements
3702         * <tt>e</tt> in the collection such that
3703         * <tt>(o == null ? e == null : o.equals(e))</tt>.
3704         *
3705         * @param c the collection in which to determine the frequency
3706         *     of <tt>o</tt>
3707         * @param o the object whose frequency is to be determined
3708         * @throws NullPointerException if <tt>c</tt> is null
```

```
3709      * @since 1.5
3710      */
3711     public static int frequency(Collection<?> c, Object o) {
3712         int result = 0;
3713         if (o == null) {
3714             for (Object e : c)
3715                 if (e == null)
3716                     result++;
3717         } else {
3718             for (Object e : c)
3719                 if (o.equals(e))
3720                     result++;
3721         }
3722         return result;
3723     }
3724
3725     /**
3726      * Returns {@code true} if the two specified collections have no
3727      * elements in common.
3728      *
3729      * <p>Care must be exercised if this method is used on collections that
3730      * do not comply with the general contract for {@code Collection}.
3731      * Implementations may elect to iterate over either collection and test
3732      * for containment in the other collection (or to perform any equivalent
3733      * computation).  If either collection uses a nonstandard equality test
3734      * (as does a {@link SortedSet} whose ordering is not <em>compatible with
3735      * equals</em>, or the key set of an {@link IdentityHashMap}), both
3736      * collections must use the same nonstandard equality test, or the
3737      * result of this method is undefined.
3738      *
3739      * <p>Care must also be exercised when using collections that have
3740      * restrictions on the elements that they may contain. Collection
3741      * implementations are allowed to throw exceptions for any operation
3742      * involving elements they deem ineligible. For absolute safety the
3743      * specified collections should contain only elements which are
3744      * eligible elements for both collections.
3745      *
3746      * <p>Note that it is permissible to pass the same collection in both
3747      * parameters, in which case the method will return {@code true} if and
3748      * only if the collection is empty.
3749      *
3750      * @param c1 a collection
3751      * @param c2 a collection
3752      * @return {@code true} if the two specified collections have no
3753      * elements in common.
3754      * @throws NullPointerException if either collection is {@code null}.
3755      * @throws NullPointerException if one collection contains a {@code null}
3756      * element and {@code null} is not an eligible element for the other collection.
3757      * (<a href="Collection.html#optional-restrictions">optional</a>)
3758      * @throws ClassCastException if one collection contains an element that is
3759      * of a type which is ineligible for the other collection.
3760      * (<a href="Collection.html#optional-restrictions">optional</a>)
3761      * @since 1.5
3762      */
3763     public static boolean disjoint(Collection<?> c1, Collection<?> c2) {
3764         // The collection to be used for contains(). Preference is given to
3765         // the collection who's contains() has lower O() complexity.
3766         Collection<?> contains = c2;
3767         // The collection to be iterated. If the collections' contains() impl
3768         // are of different O() complexity, the collection with slower
3769         // contains() will be used for iteration. For collections who's
3770         // contains() are of the same complexity then best performance is
3771         // achieved by iterating the smaller collection.
3772         Collection<?> iterate = c1;
3773
3774         // Performance optimization cases. The heuristics:
3775         //   1. Generally iterate over c1.
3776         //   2. If c1 is a Set then iterate over c2.
3777         //   3. If either collection is empty then result is always true.
3778         //   4. Iterate over the smaller Collection.
3779         if (c1 instanceof Set) {
3780             // Use c1 for contains as a Set's contains() is expected to perform
3781             // better than O(N/2)
3782             iterate = c2;
```

```
            contains = c1;
        } else if (!(c2 instanceof Set)) {
            // Both are mere Collections. Iterate over smaller collection.
            // Example: If c1 contains 3 elements and c2 contains 50 elements and
            // assuming contains() requires ceiling(N/2) comparisons then
            // checking for all c1 elements in c2 would require 75 comparisons
            // (3 * ceiling(50/2)) vs. checking all c2 elements in c1 requiring
            // 100 comparisons (50 * ceiling(3/2)).
            int c1size = c1.size();
            int c2size = c2.size();
            if (c1size == 0 || c2size == 0) {
                // At least one collection is empty. Nothing will match.
                return true;
            }

            if (c1size > c2size) {
                iterate = c2;
                contains = c1;
            }
        }

        for (Object e : iterate) {
            if (contains.contains(e)) {
               // Found a common element. Collections are not disjoint.
                return false;
            }
        }

        // No common elements were found.
        return true;
    }

    /**
     * Adds all of the specified elements to the specified collection.
     * Elements to be added may be specified individually or as an array.
     * The behavior of this convenience method is identical to that of
     * <tt>c.addAll(Arrays.asList(elements))</tt>, but this method is likely
     * to run significantly faster under most implementations.
     *
     * <p>When elements are specified individually, this method provides a
     * convenient way to add a few elements to an existing collection:
     * <pre>
     *     Collections.addAll(flavors, "Peaches 'n Plutonium", "Rocky Racoon");
     * </pre>
     *
     * @param c the collection into which <tt>elements</tt> are to be inserted
     * @param elements the elements to insert into <tt>c</tt>
     * @return <tt>true</tt> if the collection changed as a result of the call
     * @throws UnsupportedOperationException if <tt>c</tt> does not support
     *         the <tt>add</tt> operation
     * @throws NullPointerException if <tt>elements</tt> contains one or more
     *         null values and <tt>c</tt> does not permit null elements, or
     *         if <tt>c</tt> or <tt>elements</tt> are <tt>null</tt>
     * @throws IllegalArgumentException if some property of a value in
     *         <tt>elements</tt> prevents it from being added to <tt>c</tt>
     * @see Collection#addAll(Collection)
     * @since 1.5
     */
    @SafeVarargs
    public static <T> boolean addAll(Collection<? super T> c, T... elements) {
        boolean result = false;
        for (T element : elements)
            result |= c.add(element);
        return result;
    }

    /**
     * Returns a set backed by the specified map.  The resulting set displays
     * the same ordering, concurrency, and performance characteristics as the
     * backing map.  In essence, this factory method provides a {@link Set}
     * implementation corresponding to any {@link Map} implementation.  There
     * is no need to use this method on a {@link Map} implementation that
     * already has a corresponding {@link Set} implementation (such as {@link
     * HashMap} or {@link TreeMap}).
```

```
     *
     * <p>Each method invocation on the set returned by this method results in
     * exactly one method invocation on the backing map or its <tt>keySet</tt>
     * view, with one exception.  The <tt>addAll</tt> method is implemented
     * as a sequence of <tt>put</tt> invocations on the backing map.
     *
     * <p>The specified map must be empty at the time this method is invoked,
     * and should not be accessed directly after this method returns.  These
     * conditions are ensured if the map is created empty, passed directly
     * to this method, and no reference to the map is retained, as illustrated
     * in the following code fragment:
     * <pre>
     *    Set&lt;Object&gt; weakHashSet = Collections.newSetFromMap(
     *        new WeakHashMap&lt;Object, Boolean&gt;());
     * </pre>
     *
     * @param map the backing map
     * @return the set backed by the map
     * @throws IllegalArgumentException if <tt>map</tt> is not empty
     * @since 1.6
     */
    public static <E> Set<E> newSetFromMap(Map<E, Boolean> map) {
        return new SetFromMap<>(map);
    }

    /**
     * @serial include
     */
    private static class SetFromMap<E> extends AbstractSet<E>
        implements Set<E>, Serializable
    {
        private final Map<E, Boolean> m;  // The backing map
        private transient Set<E> s;       // Its keySet

        SetFromMap(Map<E, Boolean> map) {
            if (!map.isEmpty())
                throw new IllegalArgumentException("Map is non-empty");
            m = map;
            s = map.keySet();
        }

        public void clear()               {        m.clear(); }
        public int size()                 { return m.size(); }
        public boolean isEmpty()          { return m.isEmpty(); }
        public boolean contains(Object o) { return m.containsKey(o); }
        public boolean remove(Object o)   { return m.remove(o) != null; }
        public boolean add(E e) { return m.put(e, Boolean.TRUE) == null; }
        public Iterator<E> iterator()     { return s.iterator(); }
        public Object[] toArray()         { return s.toArray(); }
        public <T> T[] toArray(T[] a)     { return s.toArray(a); }
        public String toString()          { return s.toString(); }
        public int hashCode()             { return s.hashCode(); }
        public boolean equals(Object o)   { return o == this || s.equals(o); }
        public boolean containsAll(Collection<?> c) {return s.containsAll(c);}
        public boolean removeAll(Collection<?> c)   {return s.removeAll(c);}
        public boolean retainAll(Collection<?> c)   {return s.retainAll(c);}
        // addAll is the only inherited implementation

        private static final long serialVersionUID = 2454657854757543876L;

        private void readObject(java.io.ObjectInputStream stream)
            throws IOException, ClassNotFoundException
        {
            stream.defaultReadObject();
            s = m.keySet();
        }
    }

    /**
     * Returns a view of a {@link Deque} as a Last-in-first-out (Lifo)
     * {@link Queue}. Method <tt>add</tt> is mapped to <tt>push</tt>,
     * <tt>remove</tt> is mapped to <tt>pop</tt> and so on. This
     * view can be useful when you would like to use a method
     * requiring a <tt>Queue</tt> but you need Lifo ordering.
```

```
3931        *
3932        * <p>Each method invocation on the queue returned by this method
3933        * results in exactly one method invocation on the backing deque, with
3934        * one exception.  The {@link Queue#addAll addAll} method is
3935        * implemented as a sequence of {@link Deque#addFirst addFirst}
3936        * invocations on the backing deque.
3937        *
3938        * @param deque the deque
3939        * @return the queue
3940        * @since  1.6
3941        */
3942       public static <T> Queue<T> asLifoQueue(Deque<T> deque) {
3943           return new AsLIFOQueue<>(deque);
3944       }
3945
3946       /**
3947        * @serial include
3948        */
3949       static class AsLIFOQueue<E> extends AbstractQueue<E>
3950           implements Queue<E>, Serializable {
3951           private static final long serialVersionUID = 1802017725587941708L;
3952           private final Deque<E> q;
3953           AsLIFOQueue(Deque<E> q)           { this.q = q; }
3954           public boolean add(E e)           { q.addFirst(e); return true; }
3955           public boolean offer(E e)         { return q.offerFirst(e); }
3956           public E poll()                   { return q.pollFirst(); }
3957           public E remove()                 { return q.removeFirst(); }
3958           public E peek()                   { return q.peekFirst(); }
3959           public E element()                { return q.getFirst(); }
3960           public void clear()               {        q.clear(); }
3961           public int size()                 { return q.size(); }
3962           public boolean isEmpty()          { return q.isEmpty(); }
3963           public boolean contains(Object o) { return q.contains(o); }
3964           public boolean remove(Object o)   { return q.remove(o); }
3965           public Iterator<E> iterator()     { return q.iterator(); }
3966           public Object[] toArray()         { return q.toArray(); }
3967           public <T> T[] toArray(T[] a)     { return q.toArray(a); }
3968           public String toString()          { return q.toString(); }
3969           public boolean containsAll(Collection<?> c) {return q.containsAll(c);}
3970           public boolean removeAll(Collection<?> c)   {return q.removeAll(c);}
3971           public boolean retainAll(Collection<?> c)   {return q.retainAll(c);}
3972           // We use inherited addAll; forwarding addAll would be wrong
3973       }
3974   }
```