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.

```
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
}
```

LinkedList

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

```
/*
 * Copyright (c) 1997, 2012, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.  Oracle designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Oracle in the LICENSE file that accompanied this code.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 */

package java.util;
import java.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.
     */
    private static <T> T get(ListIterator<? extends T> i, int index) {
        T obj = null;
        int pos = i.nextIndex();
        if (pos <= index) {
            do {
                obj = i.next();
            } while (pos++ < index);
        } else {
            do {
                obj = i.previous();
            } while (--pos > index);
        }
        return obj;
    }

    /**
     * Searches the specified list for the specified object using the binary
     * search algorithm.  The list must be sorted into ascending order
     * according to the specified comparator (as by the
     * {@link #sort(List, Comparator) sort(List, Comparator)}
     * 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.
     * @param  c the comparator by which the list is ordered.
     *         A <tt>null</tt> value indicates that the elements'
     *         {@linkplain Comparable natural ordering} should be used.
     * @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> using the specified comparator,
     *         or the search key is not mutually comparable with the
     *         elements of the list using this comparator.
     */
```

```
public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T>
    if (c==null)
        return binarySearch((List) list, key);

    if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
        return Collections.indexedBinarySearch(list, key, c);
    else
        return Collections.iteratorBinarySearch(list, key, c);
}

private static <T> int indexedBinarySearch(List<? extends T> l, T key, Comparator<? supe
    int low = 0;
    int high = l.size()-1;

    while (low <= high) {
        int mid = (low + high) >>> 1;
        T midVal = l.get(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 static <T> int iteratorBinarySearch(List<? extends T> l, T key, Comparator<? sup
    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());
        }
    }
}

/**
 * Returns the minimum 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 minimum element is to be determined.
 * @return the minimum 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 min(Collection<? extends T> c
    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 minimum 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 minimum element is to be determined.
 * @param  comp the comparator with which to determine the minimum element.
 *         A <tt>null</tt> value indicates that the elements' <i>natural
 *         ordering</i> should be used.
 * @return the minimum 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 min(Collection<? extends T> coll, Comparator<? super T> comp) {
    if (comp==null)
        return (T)min((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;
}

/**
 * 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> c
    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;

    final Collection<? extends E> c;

    UnmodifiableCollection(Collection<? extends E> c) {
```

```
    if (c==null)
        throw new NullPointerException();
    this.c = c;
}

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 Iterator<E> iterator() {
    return new Iterator<E>() {
        private final Iterator<? extends E> i = c.iterator();

        public boolean hasNext() {return i.hasNext();}
        public E next()          {return i.next();}
        public void remove() {
            throw new UnsupportedOperationException();
        }
    };
}

public boolean add(E e) {
    throw new UnsupportedOperationException();
}
public boolean remove(Object o) {
    throw new UnsupportedOperationException();
}

public boolean containsAll(Collection<?> coll) {
    return c.containsAll(coll);
}
public boolean addAll(Collection<? extends E> coll) {
    throw new UnsupportedOperationException();
}
public boolean removeAll(Collection<?> coll) {
    throw new UnsupportedOperationException();
}
public boolean retainAll(Collection<?> coll) {
    throw new UnsupportedOperationException();
}
public void clear() {
    throw new UnsupportedOperationException();
}
```

```
}

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

/**
 * @serial include
 */
static class UnmodifiableSet<E> extends UnmodifiableCollection<E>
                             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));
}

/**
 * @serial include
 */
static class UnmodifiableList<E> extends UnmodifiableCollection<E>
                              implements List<E> {
    private static final long serialVersionUID = -283967356065247728L;
    final List<? extends E> list;

    UnmodifiableList(List<? extends E> list) {
        super(list);
        this.list = list;
    }

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

    public E get(int index) {return list.get(index);}
    public E set(int index, E element) {
        throw new UnsupportedOperationException();
    }
    public void add(int index, E element) {
        throw new UnsupportedOperationException();
    }
    public E remove(int index) {
        throw new UnsupportedOperationException();
    }
    public int indexOf(Object o)            {return list.indexOf(o);}
    public int lastIndexOf(Object o)        {return list.lastIndexOf(o);}
    public boolean addAll(int index, Collection<? extends E> c) {
        throw new UnsupportedOperationException();
    }
    public ListIterator<E> listIterator()   {return listIterator(0);}

    public ListIterator<E> listIterator(final int index) {
        return new ListIterator<E>() {
            private final ListIterator<? extends E> i
                = list.listIterator(index);

            public boolean hasNext()     {return i.hasNext();}
            public E next()              {return i.next();}
            public boolean hasPrevious() {return i.hasPrevious();}
            public E previous()          {return i.previous();}
            public int nextIndex()       {return i.nextIndex();}
            public int previousIndex()   {return i.previousIndex();}

            public void remove() {
                throw new UnsupportedOperationException();
            }
            public void set(E e) {
                throw new UnsupportedOperationException();
            }
            public void add(E e) {
                throw new UnsupportedOperationException();
            }
        };
    }

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

    /**
     * UnmodifiableRandomAccessList instances are serialized as
     * UnmodifiableList instances to allow them to be deserialized
     * in pre-1.4 JREs (which do not have UnmodifiableRandomAccessList).
     * This method inverts the transformation.  As a beneficial
     * side-effect, it also grafts the RandomAccess marker onto
     * UnmodifiableList instances that were serialized in pre-1.4 JREs.
     *
     * Note: Unfortunately, UnmodifiableRandomAccessList instances
     * serialized in 1.4.1 and deserialized in 1.4 will become
     * UnmodifiableList instances, as this method was missing in 1.4.
     */
    private Object readResolve() {
        return (list instanceof RandomAccess
                ? new UnmodifiableRandomAccessList<>(list)
                : this);
    }
}

/**
 * @serial include
 */
static class UnmodifiableRandomAccessList<E> extends UnmodifiableList<E>
                                          implements RandomAccess
```

```
{
    UnmodifiableRandomAccessList(List<? extends E> list) {
        super(list);
    }

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

    private static final long serialVersionUID = -2542308836966382001L;

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

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

/**
 * @serial include
 */
private static class UnmodifiableMap<K,V> implements Map<K,V>, Serializable {
    private static final long serialVersionUID = -1034234728574286014L;
    private final Map<? extends K, ? extends V> m;

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

    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 val) {return m.containsValue(val);}
    public V get(Object key)                 {return m.get(key);}

    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

            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);
    }

    public boolean equals(Object o) {
        if (this == o)
            return true;
        synchronized (mutex) {return c.equals(o);}
    }
    public int hashCode() {
        synchronized (mutex) {return c.hashCode();}
    }
}

/**
 * Returns a synchronized (thread-safe) sorted set backed by the specified
 * sorted set.  In order to guarantee serial access, it is critical that
 * <strong>all</strong> access to the backing sorted set is accomplished
 * through the returned sorted set (or its views).<p>
 *
 * It is imperative that the user manually synchronize on the returned
 * sorted set when iterating over it or any of its <tt>subSet</tt>,
 * <tt>headSet</tt>, or <tt>tailSet</tt> views.
 * <pre>
 *  SortedSet s = Collections.synchronizedSortedSet(new TreeSet());
 *      ...
 *  synchronized (s) {
 *      Iterator i = s.iterator(); // Must be in the synchronized block
 *      while (i.hasNext())
 *          foo(i.next());
 *  }
 * </pre>
 * or:
 * <pre>
 *  SortedSet s = Collections.synchronizedSortedSet(new TreeSet());
 *  SortedSet s2 = s.headSet(foo);
 *      ...
 *  synchronized (s) {  // Note: s, not s2!!!
 *      Iterator i = s2.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 sorted set will be serializable if the specified
 * sorted set is serializable.
 *
 * @param  s the sorted set to be "wrapped" in a synchronized sorted set.
 * @return a synchronized view of the specified sorted set.
 */
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s) {
    return new SynchronizedSortedSet<>(s);
}

/**
 * @serial include
 */
static class SynchronizedSortedSet<E>
    extends SynchronizedSet<E>
    implements SortedSet<E>
{
    private static final long serialVersionUID = 8695801310862127406L;

    private final SortedSet<E> ss;

    SynchronizedSortedSet(SortedSet<E> s) {
        super(s);
        ss = s;
    }
    SynchronizedSortedSet(SortedSet<E> s, Object mutex) {
        super(s, mutex);
        ss = s;
    }

    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;

        SynchronizedList(List<E> list) {
            super(list);
            this.list = list;
        }
        SynchronizedList(List<E> list, Object mutex) {
            super(list, mutex);
            this.list = list;
        }

        public boolean equals(Object o) {
            if (this == o)
                return true;
            synchronized (mutex) {return list.equals(o);}
        }
        public int hashCode() {
            synchronized (mutex) {return list.hashCode();}
        }
```

```
public E get(int index) {
    synchronized (mutex) {return list.get(index);}
}
public E set(int index, E element) {
    synchronized (mutex) {return list.set(index, element);}
}
public void add(int index, E element) {
    synchronized (mutex) {list.add(index, element);}
}
public E remove(int index) {
    synchronized (mutex) {return list.remove(index);}
}

public int indexOf(Object o) {
    synchronized (mutex) {return list.indexOf(o);}
}
public int lastIndexOf(Object o) {
    synchronized (mutex) {return list.lastIndexOf(o);}
}

public boolean addAll(int index, Collection<? extends E> c) {
    synchronized (mutex) {return list.addAll(index, c);}
}

public ListIterator<E> listIterator() {
    return list.listIterator(); // Must be manually synched by user
}

public ListIterator<E> listIterator(int index) {
    return list.listIterator(index); // Must be manually synched by user
}

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

/**
 * SynchronizedRandomAccessList instances are serialized as
 * SynchronizedList instances to allow them to be deserialized
 * in pre-1.4 JREs (which do not have SynchronizedRandomAccessList).
 * This method inverts the transformation.  As a beneficial
 * side-effect, it also grafts the RandomAccess marker onto
 * SynchronizedList instances that were serialized in pre-1.4 JREs.
```

```
     *
     * Note: Unfortunately, SynchronizedRandomAccessList instances
     * serialized in 1.4.1 and deserialized in 1.4 will become
     * SynchronizedList instances, as this method was missing in 1.4.
     */
    private Object readResolve() {
        return (list instanceof RandomAccess
                ? new SynchronizedRandomAccessList<>(list)
                : 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;
    SynchronizedSortedMap(SortedMap<K,V> m) {
        super(m);
        sm = m;
    }
    SynchronizedSortedMap(SortedMap<K,V> m, Object mutex) {
        super(m, mutex);
        sm = m;
    }

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

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

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

// Dynamically typesafe collection wrappers

/**
 * Returns a dynamically typesafe view of the specified collection.
 * Any attempt to insert an element of the wrong type will result in an
 * immediate {@link ClassCastException}.  Assuming a collection
 * contains no incorrectly typed elements prior to the time a
 * dynamically typesafe view is generated, and that all subsequent
 * access to the collection takes place through the view, it is
 * <i>guaranteed</i> that the collection cannot contain an incorrectly
 * typed element.
 *
 * <p>The generics mechanism in the language provides compile-time
 * (static) type checking, but it is possible to defeat this mechanism
 * with unchecked casts.  Usually this is not a problem, as the compiler
 * issues warnings on all such unchecked operations.  There are, however,
 * times when static type checking alone is not sufficient.  For example,
 * suppose a collection is passed to a third-party library and it is
 * imperative that the library code not corrupt the collection by
 * inserting an element of the wrong type.
 *
 * <p>Another use of dynamically typesafe views is debugging.  Suppose a
 * program fails with a {@code ClassCastException}, indicating that an
 * incorrectly typed element was put into a parameterized collection.
 * Unfortunately, the exception can occur at any time after the erroneous
 * element is inserted, so it typically provides little or no information
 * as to the real source of the problem.  If the problem is reproducible,
 * one can quickly determine its source by temporarily modifying the
 * program to wrap the collection with a dynamically typesafe view.
 * For example, this declaration:
 *  <pre> {@code
 *     Collection<String> c = new HashSet<String>();
 * }</pre>
 * may be replaced temporarily by this one:
 *  <pre> {@code
 *     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);
    }
    public boolean removeAll(Collection<?> coll) {
        return c.removeAll(coll);
    }
    public boolean retainAll(Collection<?> coll) {
        return c.retainAll(coll);
    }

    public Iterator<E> iterator() {
        final Iterator<E> it = c.iterator();
        return new Iterator<E>() {
            public boolean hasNext() { return it.hasNext(); }
            public E next()          { return it.next(); }
            public void remove()     {        it.remove(); }};
    }

    public boolean add(E e) {
        typeCheck(e);
        return c.add(e);
    }

    private E[] zeroLengthElementArray = null; // Lazily initialized

    private E[] zeroLengthElementArray() {
        return zeroLengthElementArray != null ? zeroLengthElementArray :
            (zeroLengthElementArray = zeroLengthArray(type));
    }

    @SuppressWarnings("unchecked")
    Collection<E> checkedCopyOf(Collection<? extends E> coll) {
        Object[] a = null;
        try {
            E[] z = zeroLengthElementArray();
            a = coll.toArray(z);
            // Defend against coll violating the toArray contract
            if (a.getClass() != z.getClass())
```

```
                a = Arrays.copyOf(a, a.length, z.getClass());
        } catch (ArrayStoreException ignore) {
            // To get better and consistent diagnostics,
            // we call typeCheck explicitly on each element.
            // We call clone() to defend against coll retaining a
            // reference to the returned array and storing a bad
            // element into it after it has been type checked.
            a = coll.toArray().clone();
            for (Object o : a)
                typeCheck(o);
        }
        // A slight abuse of the type system, but safe here.
        return (Collection<E>) Arrays.asList(a);
    }

    public boolean addAll(Collection<? extends E> coll) {
        // Doing things this way insulates us from concurrent changes
        // in the contents of coll and provides all-or-nothing
        // semantics (which we wouldn't get if we type-checked each
        // element as we added it)
        return c.addAll(checkedCopyOf(coll));
    }
}

/**
 * Returns a dynamically typesafe view of the specified set.
 * Any attempt to insert an element of the wrong type will result in
 * an immediate {@link ClassCastException}.  Assuming a set contains
 * no incorrectly typed elements prior to the time a dynamically typesafe
 * view is generated, and that all subsequent access to the set
 * takes place through the view, it is <i>guaranteed</i> that the
 * 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 set will be serializable if the specified set is
 * 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;
    }

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

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

/**
 * Returns a dynamically typesafe view of the specified list.
 * Any attempt to insert an element of the wrong type will result in
 * an immediate {@link ClassCastException}.  Assuming a list contains
 * no incorrectly typed elements prior to the time a dynamically typesafe
 * view is generated, and that all subsequent access to the list
 * takes place through the view, it is <i>guaranteed</i> that the
 * list 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 list will be serializable if the specified list
 * is serializable.
 *
 * <p>Since {@code null} is considered to be a value of any reference
 * type, the returned list permits insertion of null elements whenever
 * the backing list does.
 *
 * @param list the list for which a dynamically typesafe view is to be
 *             returned
 * @param type the type of element that {@code list} is permitted to hold
 * @return a dynamically typesafe view of the specified list
 * @since 1.5
 */
public static <E> List<E> checkedList(List<E> list, Class<E> type) {
    return (list instanceof RandomAccess ?
            new CheckedRandomAccessList<>(list, type) :
            new CheckedList<>(list, type));
}

/**
 * @serial include
 */
static class CheckedList<E>
    extends CheckedCollection<E>
    implements List<E>
{
    private static final long serialVersionUID = 65247728283967356L;
    final List<E> list;

    CheckedList(List<E> list, Class<E> type) {
        super(list, type);
        this.list = list;
    }

    public boolean equals(Object o)  { return o == this || list.equals(o); }
    public int hashCode()            { return list.hashCode(); }
    public E get(int index)          { return list.get(index); }
    public E remove(int index)       { return list.remove(index); }
    public int indexOf(Object o)     { return list.indexOf(o); }
    public int lastIndexOf(Object o) { return list.lastIndexOf(o); }

    public E set(int index, E element) {
        typeCheck(element);
        return list.set(index, element);
    }

    public void add(int index, E element) {
        typeCheck(element);
        list.add(index, element);
    }

    public boolean addAll(int index, Collection<? extends E> c) {
        return list.addAll(index, checkedCopyOf(c));
    }
    public ListIterator<E> listIterator()   { return listIterator(0); }

    public ListIterator<E> listIterator(final int index) {
        final ListIterator<E> i = list.listIterator(index);

        return new ListIterator<E>() {
            public boolean hasNext()     { return i.hasNext(); }
            public E next()              { return i.next(); }
            public boolean hasPrevious() { return i.hasPrevious(); }
            public E previous()          { return i.previous(); }
            public int nextIndex()       { return i.nextIndex(); }
            public int previousIndex()   { return i.previousIndex(); }
            public void remove()         {        i.remove(); }

            public void set(E e) {
                typeCheck(e);
                i.set(e);
            }

            public void add(E e) {
                typeCheck(e);
                i.add(e);
            }
        };
    }

    public List<E> subList(int fromIndex, int toIndex) {
        return new CheckedList<>(list.subList(fromIndex, toIndex), type);
    }
}

/**
 * @serial include
 */
```

```
static class CheckedRandomAccessList<E> extends CheckedList<E>
                                        implements RandomAccess
{
    private static final long serialVersionUID = 1638200125423088369L;

    CheckedRandomAccessList(List<E> list, Class<E> type) {
        super(list, type);
    }

    public List<E> subList(int fromIndex, int toIndex) {
        return new CheckedRandomAccessList<>(
            list.subList(fromIndex, toIndex), type);
    }
}

/**
 * Returns a dynamically typesafe view of the specified 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> 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(); }
public String toString()               { return m.toString(); }

public V put(K key, V value) {
    typeCheck(key, value);
    return m.put(key, value);
}

@SuppressWarnings("unchecked")
public void putAll(Map<? extends K, ? extends V> t) {
    // Satisfy the following goals:
    // - good diagnostics in case of type mismatch
    // - all-or-nothing semantics
    // - protection from malicious t
    // - correct behavior if t is a concurrent map
    Object[] entries = t.entrySet().toArray();
    List<Map.Entry<K,V>> checked = new ArrayList<>(entries.length);
    for (Object o : entries) {
        Map.Entry<?,?> e = (Map.Entry<?,?>) o;
        Object k = e.getKey();
        Object v = e.getValue();
        typeCheck(k, v);
        checked.add(
            new AbstractMap.SimpleImmutableEntry<>((K) k, (V) v));
    }
    for (Map.Entry<K,V> e : checked)
        m.put(e.getKey(), e.getValue());
}

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

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

/**
 * We need this class in addition to CheckedSet as Map.Entry permits
 * modification of the backing Map via the setValue operation.  This
 * class is subtle: there are many possible attacks that must be
 * thwarted.
 *
 * @serial exclude
 */
static class CheckedEntrySet<K,V> implements Set<Map.Entry<K,V>> {
    private final Set<Map.Entry<K,V>> s;
    private final Class<V> valueType;

    CheckedEntrySet(Set<Map.Entry<K, V>> s, Class<V> valueType) {
        this.s = s;
        this.valueType = valueType;
    }

    public int size()        { return s.size(); }
    public boolean isEmpty() { return s.isEmpty(); }
    public String toString() { return s.toString(); }
    public int hashCode()    { return s.hashCode(); }
    public void clear()      {        s.clear(); }

    public boolean add(Map.Entry<K, V> e) {
        throw new UnsupportedOperationException();
    }
    public boolean addAll(Collection<? extends Map.Entry<K, V>> coll) {
        throw new UnsupportedOperationException();
    }

    public Iterator<Map.Entry<K,V>> iterator() {
        final Iterator<Map.Entry<K, V>> i = s.iterator();
        final Class<V> valueType = this.valueType;

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

            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));
}

public boolean removeAll(Collection<?> c) {
    return batchRemove(c, false);
}
public boolean retainAll(Collection<?> c) {
    return batchRemove(c, true);
}
private boolean batchRemove(Collection<?> c, boolean complement) {
    boolean modified = false;
    Iterator<Map.Entry<K,V>> it = iterator();
    while (it.hasNext()) {
        if (c.contains(it.next()) != complement) {
            it.remove();
            modified = true;
        }
    }
    return modified;
}

public boolean equals(Object o) {
    if (o == this)
        return true;
    if (!(o instanceof Set))
        return false;
    Set<?> that = (Set<?>) o;
    return that.size() == s.size()
        && containsAll(that); // Invokes safe containsAll() above
}

static <K,V,T> CheckedEntry<K,V,T> checkedEntry(Map.Entry<K,V> e,
                                                Class<T> valueType) {
    return new CheckedEntry<>(e, valueType);
}

/**
 * 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 CheckedEntry<K,V,T> implements Map.Entry<K,V> {
    private final Map.Entry<K, V> e;
    private final Class<T> valueType;

    CheckedEntry(Map.Entry<K, V> e, Class<T> valueType) {
        this.e = e;
        this.valueType = valueType;
    }

    public K getKey()        { return e.getKey(); }
    public V getValue()      { return e.getValue(); }
    public int hashCode()    { return e.hashCode(); }
    public String toString() { return e.toString(); }

    public V setValue(V value) {
        if (value != null && !valueType.isInstance(value))
            throw new ClassCastException(badValueMsg(value));
        return e.setValue(value);
    }

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

    public boolean equals(Object o) {
        if (o == this)
```

```
                    return true;
                if (!(o instanceof Map.Entry))
                    return false;
                return e.equals(new AbstractMap.SimpleImmutableEntry
                                <>((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);
    }
}

// Empty collections

/**
 * Returns an iterator that has no elements.  More precisely,
 *
 * <ul compact>
 *
 * <li>{@link Iterator#hasNext hasNext} always returns {@code
 * false}.
 *
 * <li>{@link Iterator#next next} always throws {@link
 * NoSuchElementException}.
 *
 * <li>{@link Iterator#remove remove} always throws {@link
 * IllegalStateException}.
 *
 * </ul>
 *
 * <p>Implementations of this method are permitted, but not
 * required, to return the same object from multiple invocations.
 *
 * @return an empty iterator
 * @since 1.7
 */
@SuppressWarnings("unchecked")
public static <T> Iterator<T> emptyIterator() {
    return (Iterator<T>) EmptyIterator.EMPTY_ITERATOR;
}

private static class EmptyIterator<E> implements Iterator<E> {
    static final EmptyIterator<Object> EMPTY_ITERATOR
        = new EmptyIterator<>();

    public boolean hasNext() { return false; }
    public E next() { throw new NoSuchElementException(); }
    public void remove() { throw new IllegalStateException(); }
}

/**
 * Returns a list iterator that has no elements.  More precisely,
 *
 * <ul compact>
 *
 * <li>{@link Iterator#hasNext hasNext} and {@link
 * ListIterator#hasPrevious hasPrevious} always return {@code
 * false}.
 *
 * <li>{@link Iterator#next next} and {@link ListIterator#previous
 * previous} always throw {@link NoSuchElementException}.
 *
 * <li>{@link Iterator#remove remove} and {@link ListIterator#set
 * set} always throw {@link IllegalStateException}.
 *
 * <li>{@link ListIterator#add add} always throws {@link
 * UnsupportedOperationException}.
 *
 * <li>{@link ListIterator#nextIndex nextIndex} always returns
 * {@code 0} .
 *
 * <li>{@link ListIterator#previousIndex previousIndex} always
 * returns {@code -1}.
 *
 * </ul>
 *
 * <p>Implementations of this method are permitted, but not
 * required, to return the same object from multiple invocations.
 *
 * @return an empty list iterator
 * @since 1.7
 */
@SuppressWarnings("unchecked")
public static <T> ListIterator<T> emptyListIterator() {
    return (ListIterator<T>) EmptyListIterator.EMPTY_ITERATOR;
}

private static class EmptyListIterator<E>
    extends EmptyIterator<E>
    implements ListIterator<E>
{
    static final EmptyListIterator<Object> EMPTY_ITERATOR
        = new EmptyListIterator<>();

    public boolean hasPrevious() { return false; }
    public E previous() { throw new NoSuchElementException(); }
    public int nextIndex()     { return 0; }
    public int previousIndex() { return -1; }
    public void set(E e) { throw new IllegalStateException(); }
    public void add(E e) { throw new UnsupportedOperationException(); }
}

/**
 * Returns an enumeration that has no elements.  More precisely,
 *
 * <ul compact>
 *
 * <li>{@link Enumeration#hasMoreElements hasMoreElements} always
 * returns {@code false}.
 *
 * <li> {@link Enumeration#nextElement nextElement} always throws
 * {@link NoSuchElementException}.
 *
 * </ul>
 *
 * <p>Implementations of this method are permitted, but not
 * required, to return the same object from multiple invocations.
 *
 * @return an empty enumeration
 * @since 1.7
 */
@SuppressWarnings("unchecked")
public static <T> Enumeration<T> emptyEnumeration() {
    return (Enumeration<T>) EmptyEnumeration.EMPTY_ENUMERATION;
}

private static class EmptyEnumeration<E> implements Enumeration<E> {
    static final EmptyEnumeration<Object> EMPTY_ENUMERATION
        = new EmptyEnumeration<>();

    public boolean hasMoreElements() { return false; }
    public E nextElement() { throw new NoSuchElementException(); }
}

/**
 * The empty set (immutable).  This set is serializable.
 *
 * @see #emptySet()
 */
@SuppressWarnings("unchecked")
public static final Set EMPTY_SET = new EmptySet<>();

/**
 * Returns the empty set (immutable).  This set is serializable.
 * Unlike the like-named field, this method is parameterized.
 *
 * <p>This example illustrates the type-safe way to obtain an empty set:
 * <pre>
 *     Set&lt;String&gt; s = Collections.emptySet();
 * </pre>
 * Implementation note:  Implementations of this method need not
 * create a separate <tt>Set</tt> object for each call.   Using this
 * method is likely to have comparable cost to using the like-named
 * field.  (Unlike this method, the field does not provide type safety.)
 *
 * @see #EMPTY_SET
 * @since 1.5
 */
@SuppressWarnings("unchecked")
public static final <T> Set<T> emptySet() {
    return (Set<T>) EMPTY_SET;
}

/**
 * @serial include
 */
private static class EmptySet<E>
    extends AbstractSet<E>
    implements Serializable
{
    private static final long serialVersionUID = 1582296315990362920L;

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

    public int size() {return 0;}
    public boolean isEmpty() {return true;}

    public boolean contains(Object obj) {return false;}
    public boolean containsAll(Collection<?> c) { return c.isEmpty(); }

    public Object[] toArray() { return new Object[0]; }

    public <T> T[] toArray(T[] a) {
        if (a.length > 0)
            a[0] = null;
        return a;
    }

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

/**
 * The empty list (immutable).  This list is serializable.
 *
 * @see #emptyList()
 */
@SuppressWarnings("unchecked")
public static final List EMPTY_LIST = new EmptyList<>();

/**
 * Returns the empty list (immutable).  This list is serializable.
 *
 * <p>This example illustrates the type-safe way to obtain an empty list:
```

```
 * <pre>
 *     List&lt;String&gt; s = Collections.emptyList();
 * </pre>
 * Implementation note:  Implementations of this method need not
 * create a separate <tt>List</tt> object for each call.   Using this
 * method is likely to have comparable cost to using the like-named
 * field.  (Unlike this method, the field does not provide type safety.)
 *
 * @see #EMPTY_LIST
 * @since 1.5
 */
@SuppressWarnings("unchecked")
public static final <T> List<T> emptyList() {
    return (List<T>) EMPTY_LIST;
}

/**
 * @serial include
 */
private static class EmptyList<E>
    extends AbstractList<E>
    implements RandomAccess, Serializable {
    private static final long serialVersionUID = 8842843931221139166L;

    public Iterator<E> iterator() {
        return emptyIterator();
    }
    public ListIterator<E> listIterator() {
        return emptyListIterator();
    }

    public int size() {return 0;}
    public boolean isEmpty() {return true;}

    public boolean contains(Object obj) {return false;}
    public boolean containsAll(Collection<?> c) { return c.isEmpty(); }

    public Object[] toArray() { return new Object[0]; }

    public <T> T[] toArray(T[] a) {
        if (a.length > 0)
            a[0] = null;
        return a;
    }

    public E get(int index) {
        throw new IndexOutOfBoundsException("Index: "+index);
    }

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

    public int hashCode() { return 1; }

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

/**
 * The empty map (immutable).  This map is serializable.
 *
 * @see #emptyMap()
 * @since 1.3
 */
@SuppressWarnings("unchecked")
public static final Map EMPTY_MAP = new EmptyMap<>();

/**
 * Returns the empty map (immutable).  This map is serializable.
 *
 * <p>This example illustrates the type-safe way to obtain an empty set:
 * <pre>
 *     Map&lt;String, Date&gt; s = Collections.emptyMap();
 * </pre>
 * Implementation note:  Implementations of this method need not
 * create a separate <tt>Map</tt> object for each call.   Using this
 * method is likely to have comparable cost to using the like-named
 * field.  (Unlike this method, the field does not provide type safety.)
 *
 * @see #EMPTY_MAP
 * @since 1.5
 */
@SuppressWarnings("unchecked")
public static final <K,V> Map<K,V> emptyMap() {
    return (Map<K,V>) EMPTY_MAP;
}

/**
 * @serial include
```

```
 */
private static class EmptyMap<K,V>
    extends AbstractMap<K,V>
    implements Serializable
{
    private static final long serialVersionUID = 6428348081105594320L;

    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);}
}

/**
 * Returns an immutable list containing only the specified object.
 * The returned list is serializable.
 *
 * @param o the sole object to be stored in the returned list.
 * @return an immutable list containing only the specified object.
 * @since 1.3
 */
public static <T> List<T> singletonList(T o) {
    return new SingletonList<>(o);
```

```
}

/**
 * @serial include
 */
private static class SingletonList<E>
    extends AbstractList<E>
    implements RandomAccess, Serializable {

    private static final long serialVersionUID = 3093736618740652951L;

    private final E element;

    SingletonList(E obj)                {element = obj;}

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

    public int size()                   {return 1;}

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

    public E get(int index) {
        if (index != 0)
          throw new IndexOutOfBoundsException("Index: "+index+", Size: 1");
        return element;
    }
}

/**
 * Returns an immutable map, mapping only the specified key to the
 * specified value.  The returned map is serializable.
 *
 * @param key the sole key to be stored in the returned map.
 * @param value the value to which the returned map maps <tt>key</tt>.
 * @return an immutable map containing only the specified key-value
 *         mapping.
 * @since 1.3
 */
public static <K,V> Map<K,V> singletonMap(K key, V value) {
    return new SingletonMap<>(key, value);
}

/**
 * @serial include
 */
private static class SingletonMap<K,V>
      extends AbstractMap<K,V>
      implements Serializable {
    private static final long serialVersionUID = -6979724477215052911L;

    private final K k;
    private final V v;

    SingletonMap(K key, V value) {
        k = key;
        v = value;
    }

    public int size()                          {return 1;}

    public boolean isEmpty()                   {return false;}

    public boolean containsKey(Object key)     {return eq(key, k);}

    public boolean containsValue(Object value) {return eq(value, v);}

    public V get(Object key)                   {return (eq(key, k) ? v : null);}

    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 = singleton(k);
        return keySet;
    }

    public Set<Map.Entry<K,V>> entrySet() {
        if (entrySet==null)
            entrySet = Collections.<Map.Entry<K,V>>singleton(
                new SimpleImmutableEntry<>(k, v));
        return entrySet;
    }

    public Collection<V> values() {
        if (values==null)
            values = singleton(v);
        return values;
    }
```

```
}

// Miscellaneous

/**
 * Returns an immutable list consisting of <tt>n</tt> copies of the
 * specified object.  The newly allocated data object is tiny (it contains
 * a single reference to the data object).  This method is useful in
 * combination with the <tt>List.addAll</tt> method to grow lists.
 * The returned list is serializable.
 *
 * @param  n the number of elements in the returned list.
 * @param  o the element to appear repeatedly in the returned list.
 * @return an immutable list consisting of <tt>n</tt> copies of the
 *         specified object.
 * @throws IllegalArgumentException if {@code n < 0}
 * @see    List#addAll(Collection)
 * @see    List#addAll(int, Collection)
 */
public static <T> List<T> nCopies(int n, T o) {
    if (n < 0)
        throw new IllegalArgumentException("List length = " + n);
    return new CopiesList<>(n, o);
}

/**
 * @serial include
 */
private static class CopiesList<E>
    extends AbstractList<E>
    implements RandomAccess, Serializable
{
    private static final long serialVersionUID = 2739099268398711800L;

    final int n;
    final E element;

    CopiesList(int n, E e) {
        assert n >= 0;
        this.n = n;
        element = e;
    }

    public int size() {
        return n;
```

```
}

public boolean contains(Object obj) {
    return n != 0 && eq(obj, element);
}

public int indexOf(Object o) {
    return contains(o) ? 0 : -1;
}

public int lastIndexOf(Object o) {
    return contains(o) ? n - 1 : -1;
}

public E get(int index) {
    if (index < 0 || index >= n)
        throw new IndexOutOfBoundsException("Index: "+index+
                                            ", Size: "+n);
    return element;
}

public Object[] toArray() {
    final Object[] a = new Object[n];
    if (element != null)
        Arrays.fill(a, 0, n, element);
    return a;
}

public <T> T[] toArray(T[] a) {
    final int n = this.n;
    if (a.length < n) {
        a = (T[])java.lang.reflect.Array
            .newInstance(a.getClass().getComponentType(), n);
        if (element != null)
            Arrays.fill(a, 0, n, element);
    } else {
        Arrays.fill(a, 0, n, element);
        if (a.length > n)
            a[n] = null;
    }
    return a;
}

public List<E> subList(int fromIndex, int toIndex) {
    if (fromIndex < 0)
        throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);
        if (toIndex > n)
            throw new IndexOutOfBoundsException("toIndex = " + toIndex);
        if (fromIndex > toIndex)
            throw new IllegalArgumentException("fromIndex(" + fromIndex +
                                               ") > toIndex(" + toIndex + ")");
        return new CopiesList<>(toIndex - fromIndex, element);
    }
}

/**
 * Returns a comparator that imposes the reverse of the <em>natural
 * ordering</em> on a collection of objects that implement the
 * {@code Comparable} interface.  (The natural ordering is the ordering
 * imposed by the objects' own {@code compareTo} method.)  This enables a
 * simple idiom for sorting (or maintaining) collections (or arrays) of
 * objects that implement the {@code Comparable} interface in
 * reverse-natural-order.  For example, suppose {@code a} is an array of
 * strings. Then: <pre>
 *          Arrays.sort(a, Collections.reverseOrder());
 * </pre> sorts the array in reverse-lexicographic (alphabetical) order.<p>
 *
 * The returned comparator is serializable.
 *
 * @return A comparator that imposes the reverse of the <i>natural
 *         ordering</i> on a collection of objects that implement
 *         the <tt>Comparable</tt> interface.
 * @see Comparable
 */
public static <T> Comparator<T> reverseOrder() {
    return (Comparator<T>) ReverseComparator.REVERSE_ORDER;
}

/**
 * @serial include
 */
private static class ReverseComparator
    implements Comparator<Comparable<Object>>, Serializable {

    private static final long serialVersionUID = 7207038068494060240L;

    static final ReverseComparator REVERSE_ORDER
        = new ReverseComparator();

    public int compare(Comparable<Object> c1, Comparable<Object> c2) {
        return c2.compareTo(c1);
    }
    private Object readResolve() { return reverseOrder(); }
}

/**
 * Returns a comparator that imposes the reverse ordering of the specified
 * comparator.  If the specified comparator is {@code null}, this method is
 * equivalent to {@link #reverseOrder()} (in other words, it returns a
 * comparator that imposes the reverse of the <em>natural ordering</em> on
 * a collection of objects that implement the Comparable interface).
 *
 * <p>The returned comparator is serializable (assuming the specified
 * comparator is also serializable or {@code null}).
 *
 * @param cmp a comparator who's ordering is to be reversed by the returned
 * comparator or {@code null}
 * @return A comparator that imposes the reverse ordering of the
 *         specified comparator.
 * @since 1.5
 */
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp) {
    if (cmp == null)
        return reverseOrder();

    if (cmp instanceof ReverseComparator2)
        return ((ReverseComparator2<T>)cmp).cmp;

    return new ReverseComparator2<>(cmp);
}

/**
 * @serial include
 */
private static class ReverseComparator2<T> implements Comparator<T>,
    Serializable
{
    private static final long serialVersionUID = 4374092139857L;

    /**
     * The comparator specified in the static factory.  This will never
     * be null, as the static factory returns a ReverseComparator
     * instance if its argument is null.
     *
     * @serial
     */
    final Comparator<T> cmp;
    ReverseComparator2(Comparator<T> cmp) {
        assert cmp != null;
        this.cmp = cmp;
    }

    public int compare(T t1, T t2) {
        return cmp.compare(t2, t1);
    }

    public boolean equals(Object o) {
        return (o == this) ||
            (o instanceof ReverseComparator2 &&
             cmp.equals(((ReverseComparator2)o).cmp));
    }

    public int hashCode() {
        return cmp.hashCode() ^ Integer.MIN_VALUE;
    }
}

/**
 * Returns an enumeration over the specified collection.  This provides
 * interoperability with legacy APIs that require an enumeration
 * as input.
 *
 * @param c the collection for which an enumeration is to be returned.
 * @return an enumeration over the specified collection.
 * @see Enumeration
 */
public static <T> Enumeration<T> enumeration(final Collection<T> c) {
    return new Enumeration<T>() {
        private final Iterator<T> i = c.iterator();

        public boolean hasMoreElements() {
            return i.hasNext();
        }

        public T nextElement() {
            return i.next();
        }
    };
}

/**
 * Returns an array list containing the elements returned by the
 * specified enumeration in the order they are returned by the
 * enumeration.  This method provides interoperability between
 * legacy APIs that return enumerations and new APIs that require
 * collections.
 *
 * @param e enumeration providing elements for the returned
 *          array list
 * @return an array list containing the elements returned
 *         by the specified enumeration.
 * @since 1.4
 * @see Enumeration
 * @see ArrayList
 */
public static <T> ArrayList<T> list(Enumeration<T> e) {
    ArrayList<T> l = new ArrayList<>();
    while (e.hasMoreElements())
        l.add(e.nextElement());
    return l;
}

/**
 * Returns true if the specified arguments are equal, or both null.
 */
static boolean eq(Object o1, Object o2) {
    return o1==null ? o2==null : o1.equals(o2);
}

/**
 * Returns the number of elements in the specified collection equal to the
 * specified object.  More formally, returns the number of elements
 * <tt>e</tt> in the collection such that
 * <tt>(o == null ? e == null : o.equals(e))</tt>.
 *
 * @param c the collection in which to determine the frequency
 *     of <tt>o</tt>
 * @param o the object whose frequency is to be determined
 * @throws NullPointerException if <tt>c</tt> is null
 * @since 1.5
 */
public static int frequency(Collection<?> c, Object o) {
    int result = 0;
    if (o == null) {
        for (Object e : c)
            if (e == null)
                result++;
    } else {
        for (Object e : c)
            if (o.equals(e))
                result++;
    }
    return result;
}

/**
 * Returns {@code true} if the two specified collections have no
 * elements in common.
 *
 * <p>Care must be exercised if this method is used on collections that
 * do not comply with the general contract for {@code Collection}.
 * Implementations may elect to iterate over either collection and test
 * for containment in the other collection (or to perform any equivalent
 * computation).  If either collection uses a nonstandard equality test
 * (as does a {@link SortedSet} whose ordering is not <em>compatible with
 * equals</em>, or the key set of an {@link IdentityHashMap}), both
 * collections must use the same nonstandard equality test, or the
 * result of this method is undefined.
 *
 * <p>Care must also be exercised when using collections that have
 * restrictions on the elements that they may contain. Collection
 * implementations are allowed to throw exceptions for any operation
 * involving elements they deem ineligible. For absolute safety the
 * specified collections should contain only elements which are
 * eligible elements for both collections.
 *
 * <p>Note that it is permissible to pass the same collection in both
 * parameters, in which case the method will return {@code true} if and
 * only if the collection is empty.
 *
 * @param c1 a collection
 * @param c2 a collection
 * @return {@code true} if the two specified collections have no
 * elements in common.
 * @throws NullPointerException if either collection is {@code null}.
 * @throws NullPointerException if one collection contains a {@code null}
 * element and {@code null} is not an eligible element for the other collection.
 * (<a href="Collection.html#optional-restrictions">optional</a>)
 * @throws ClassCastException if one collection contains an element that is
 * of a type which is ineligible for the other collection.
 * (<a href="Collection.html#optional-restrictions">optional</a>)
 * @since 1.5
 */
public static boolean disjoint(Collection<?> c1, Collection<?> c2) {
```

```
// The collection to be used for contains(). Preference is given to
// the collection who’s contains() has lower O() complexity.
Collection<?> contains = c2;
// The collection to be iterated. If the collections’ contains() impl
// are of different O() complexity, the collection with slower
// contains() will be used for iteration. For collections who’s
// contains() are of the same complexity then best performance is
// achieved by iterating the smaller collection.
Collection<?> iterate = c1;

// Performance optimization cases. The heuristics:
//   1. Generally iterate over c1.
//   2. If c1 is a Set then iterate over c2.
//   3. If either collection is empty then result is always true.
//   4. Iterate over the smaller Collection.
if (c1 instanceof Set) {
    // Use c1 for contains as a Set’s contains() is expected to perform
    // better than O(N/2)
    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.
 *
 * <p>Each method invocation on the queue returned by this method
 * results in exactly one method invocation on the backing deque, with
 * one exception.  The {@link Queue#addAll addAll} method is
 * implemented as a sequence of {@link Deque#addFirst addFirst}
 * invocations on the backing deque.
 *
 * @param deque the deque
 * @return the queue
 * @since  1.6
 */
public static <T> Queue<T> asLifoQueue(Deque<T> deque) {
    return new AsLIFOQueue<>(deque);
}

/**
 * @serial include
```

```
     */
    static class AsLIFOQueue<E> extends AbstractQueue<E>
        implements Queue<E>, Serializable {
        private static final long serialVersionUID = 1802017725587941708L;
        private final Deque<E> q;
        AsLIFOQueue(Deque<E> q)           { this.q = q; }
        public boolean add(E e)           { q.addFirst(e); return true; }
        public boolean offer(E e)         { return q.offerFirst(e); }
        public E poll()                   { return q.pollFirst(); }
        public E remove()                 { return q.removeFirst(); }
        public E peek()                   { return q.peekFirst(); }
        public E element()                { return q.getFirst(); }
        public void clear()               {        q.clear(); }
        public int size()                 { return q.size(); }
        public boolean isEmpty()          { return q.isEmpty(); }
        public boolean contains(Object o) { return q.contains(o); }
        public boolean remove(Object o)   { return q.remove(o); }
        public Iterator<E> iterator()     { return q.iterator(); }
        public Object[] toArray()         { return q.toArray(); }
        public <T> T[] toArray(T[] a)     { return q.toArray(a); }
        public String toString()          { return q.toString(); }
        public boolean containsAll(Collection<?> c) {return q.containsAll(c);}
        public boolean removeAll(Collection<?> c)   {return q.removeAll(c);}
        public boolean retainAll(Collection<?> c)   {return q.retainAll(c);}
        // We use inherited addAll; forwarding addAll would be wrong
    }
}
```