Código fonte em PDF

Collections

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

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

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

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

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

```
264     private static <T>
265     int indexedBinarySearch(List<? extends Comparable<? super T>> list, T key)
266     {
267         int low = 0;
268         int high = list.size()-1;
269 
270         while (low <= high) {
271             int mid = (low + high) >>> 1;
272             Comparable<? super T> midVal = list.get(mid);
273             int cmp = midVal.compareTo(key);
274 
275             if (cmp < 0)
276                 low = mid + 1;
277             else if (cmp > 0)
278                 high = mid - 1;
279             else
280                 return mid; // key found
```

```
281             }
282             return -(low + 1);  // key not found
283         }
```

LinkedList

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

```
1   /*
2    * Copyright (c) 1997, 2012, Oracle and/or its affiliates. All rights reserved.
3    * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
4    *
5    * This code is free software; you can redistribute it and/or modify it
6    * under the terms of the GNU General Public License version 2 only, as
7    * published by the Free Software Foundation.  Oracle designates this
8    * particular file as subject to the "Classpath" exception as provided
9    * by Oracle in the LICENSE file that accompanied this code.
10   *
11   * This code is distributed in the hope that it will be useful, but WITHOUT
12   * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
13   * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
14   * version 2 for more details (a copy is included in the LICENSE file that
15   * accompanied this code).
16   *
17   * You should have received a copy of the GNU General Public License version
18   * 2 along with this work; if not, write to the Free Software Foundation,
19   * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
20   *
21   * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
22   * or visit www.oracle.com if you need additional information or have any
23   * questions.
24   */
25
26  package java.util;
27  import java.io.Serializable;
28  import java.io.ObjectOutputStream;
29  import java.io.IOException;
30  import java.lang.reflect.Array;
31
32  /**
33   * This class consists exclusively of static methods that operate on or return
34   * collections.  It contains polymorphic algorithms that operate on
35   * collections, "wrappers", which return a new collection backed by a
36   * specified collection, and a few other odds and ends.
37   *
```

```
38   * <p>The methods of this class all throw a <tt>NullPointerException</tt>
39   * if the collections or class objects provided to them are null.
40   *
41   * <p>The documentation for the polymorphic algorithms contained in this class
42   * generally includes a brief description of the <i>implementation</i>.  Such
43   * descriptions should be regarded as <i>implementation notes</i>, rather than
44   * parts of the <i>specification</i>.  Implementors should feel free to
45   * substitute other algorithms, so long as the specification itself is adhered
46   * to.  (For example, the algorithm used by <tt>sort</tt> does not have to be
47   * a mergesort, but it does have to be <i>stable</i>.)
48   *
49   * <p>The "destructive" algorithms contained in this class, that is, the
50   * algorithms that modify the collection on which they operate, are specified
51   * to throw <tt>UnsupportedOperationException</tt> if the collection does not
52   * support the appropriate mutation primitive(s), such as the <tt>set</tt>
53   * method.  These algorithms may, but are not required to, throw this
54   * exception if an invocation would have no effect on the collection.  For
55   * example, invoking the <tt>sort</tt> method on an unmodifiable list that is
56   * already sorted may or may not throw <tt>UnsupportedOperationException</tt>.
57   *
58   * <p>This class is a member of the
59   * <a href="{@docRoot}/../technotes/guides/collections/index.html">
60   * Java Collections Framework</a>.
61   *
62   * @author  Josh Bloch
63   * @author  Neal Gafter
64   * @see     Collection
65   * @see     Set
66   * @see     List
67   * @see     Map
68   * @since   1.2
69   */
70
71  public class Collections {
72      // Suppresses default constructor, ensuring non-instantiability.
73      private Collections() {
74      }
75
76      // Algorithms
77
78      /*
79       * Tuning parameters for algorithms - Many of the List algorithms have
80       * two implementations, one of which is appropriate for RandomAccess
81       * lists, the other for "sequential."  Often, the random access variant
82       * yields better performance on small sequential access lists.  The
83       * 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.
```

```
130         *
131         * <p>The implementation was adapted from Tim Peters's list sort for Python
132         * (<a href="http://svn.python.org/projects/python/trunk/Objects/listsort.txt">
133         * TimSort</a>).  It uses techiques from Peter McIlroy's "Optimistic
134         * Sorting and Information Theoretic Complexity", in Proceedings of the
135         * Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474,
136         * January 1993.
137         *
138         * <p>This implementation dumps the specified list into an array, sorts
139         * the array, and iterates over the list resetting each element
140         * from the corresponding position in the array.  This avoids the
141         * n<sup>2</sup> log(n) performance that would result from attempting
142         * to sort a linked list in place.
143         *
144         * @param  list the list to be sorted.
145         * @throws ClassCastException if the list contains elements that are not
146         *         <i>mutually comparable</i> (for example, strings and integers).
147         * @throws UnsupportedOperationException if the specified list's
148         *         list-iterator does not support the {@code set} operation.
149         * @throws IllegalArgumentException (optional) if the implementation
150         *         detects that the natural ordering of the list elements is
151         *         found to violate the {@link Comparable} contract
152         */
153        public static <T extends Comparable<? super T>> void sort(List<T> list) {
154            Object[] a = list.toArray();
155            Arrays.sort(a);
156            ListIterator<T> i = list.listIterator();
157            for (int j=0; j<a.length; j++) {
158                i.next();
159                i.set((T)a[j]);
160            }
161        }
162
163        /**
164         * Sorts the specified list according to the order induced by the
165         * specified comparator.  All elements in the list must be <i>mutually
166         * comparable</i> using the specified comparator (that is,
167         * {@code c.compare(e1, e2)} must not throw a {@code ClassCastException}
168         * for any elements {@code e1} and {@code e2} in the list).
169         *
170         * <p>This sort is guaranteed to be <i>stable</i>:  equal elements will
171         * not be reordered as a result of the sort.
172         *
173         * <p>The specified list must be modifiable, but need not be resizable.
174         *
175         * <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]);
```

```
222             }
223         }
224
225
226         /**
227          * Searches the specified list for the specified object using the binary
228          * search algorithm.  The list must be sorted into ascending order
229          * according to the {@linkplain Comparable natural ordering} of its
230          * elements (as by the {@link #sort(List)} method) prior to making this
231          * call.  If it is not sorted, the results are undefined.  If the list
232          * contains multiple elements equal to the specified object, there is no
233          * guarantee which one will be found.
234          *
235          * <p>This method runs in log(n) time for a "random access" list (which
236          * provides near-constant-time positional access).  If the specified list
237          * does not implement the {@link RandomAccess} interface and is large,
238          * this method will do an iterator-based binary search that performs
239          * O(n) link traversals and O(log n) element comparisons.
240          *
241          * @param  list the list to be searched.
242          * @param  key the key to be searched for.
243          * @return the index of the search key, if it is contained in the list;
244          *         otherwise, <tt>(-(<i>insertion point</i>) - 1)</tt>.  The
245          *         <i>insertion point</i> is defined as the point at which the
246          *         key would be inserted into the list: the index of the first
247          *         element greater than the key, or <tt>list.size()</tt> if all
248          *         elements in the list are less than the specified key.  Note
249          *         that this guarantees that the return value will be &gt;= 0 if
250          *         and only if the key is found.
251          * @throws ClassCastException if the list contains elements that are not
252          *         <i>mutually comparable</i> (for example, strings and
253          *         integers), or the search key is not mutually comparable
254          *         with the elements of the list.
255          */
256         public static <T>
257         int binarySearch(List<? extends Comparable<? super T>> list, T key) {
258             if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
259                 return Collections.indexedBinarySearch(list, key);
260             else
261                 return Collections.iteratorBinarySearch(list, key);
262         }
263
264         private static <T>
265         int indexedBinarySearch(List<? extends Comparable<? super T>> list, T key)
266         {
267             int low = 0;
```

```
268         int high = list.size()-1;
269 
270         while (low <= high) {
271             int mid = (low + high) >>> 1;
272             Comparable<? super T> midVal = list.get(mid);
273             int cmp = midVal.compareTo(key);
274 
275             if (cmp < 0)
276                 low = mid + 1;
277             else if (cmp > 0)
278                 high = mid - 1;
279             else
280                 return mid; // key found
281         }
282         return -(low + 1);  // key not found
283     }
284 
285     private static <T>
286     int iteratorBinarySearch(List<? extends Comparable<? super T>> list, T key)
287     {
288         int low = 0;
289         int high = list.size()-1;
290         ListIterator<? extends Comparable<? super T>> i = list.listIterator();
291 
292         while (low <= high) {
293             int mid = (low + high) >>> 1;
294             Comparable<? super T> midVal = get(i, mid);
295             int cmp = midVal.compareTo(key);
296 
297             if (cmp < 0)
298                 low = mid + 1;
299             else if (cmp > 0)
300                 high = mid - 1;
301             else
302                 return mid; // key found
303         }
304         return -(low + 1);  // key not found
305     }
306 
307     /**
308      * Gets the ith element from the given list by repositioning the specified
309      * list listIterator.
310      */
311     private static <T> T get(ListIterator<? extends T> i, int index) {
312         T obj = null;
313         int pos = i.nextIndex();
```

```
314             if (pos <= index) {
315                 do {
316                     obj = i.next();
317                 } while (pos++ < index);
318             } else {
319                 do {
320                     obj = i.previous();
321                 } while (--pos > index);
322             }
323             return obj;
324         }
325 
326         /**
327          * Searches the specified list for the specified object using the binary
328          * search algorithm.  The list must be sorted into ascending order
329          * according to the specified comparator (as by the
330          * {@link #sort(List, Comparator) sort(List, Comparator)}
331          * method), prior to making this call.  If it is
332          * not sorted, the results are undefined.  If the list contains multiple
333          * elements equal to the specified object, there is no guarantee which one
334          * will be found.
335          *
336          * <p>This method runs in log(n) time for a "random access" list (which
337          * provides near-constant-time positional access).  If the specified list
338          * does not implement the {@link RandomAccess} interface and is large,
339          * this method will do an iterator-based binary search that performs
340          * O(n) link traversals and O(log n) element comparisons.
341          *
342          * @param  list the list to be searched.
343          * @param  key the key to be searched for.
344          * @param  c the comparator by which the list is ordered.
345          *         A <tt>null</tt> value indicates that the elements'
346          *         {@linkplain Comparable natural ordering} should be used.
347          * @return the index of the search key, if it is contained in the list;
348          *         otherwise, <tt>(-(<i>insertion point</i>) - 1)</tt>.  The
349          *         <i>insertion point</i> is defined as the point at which the
350          *         key would be inserted into the list: the index of the first
351          *         element greater than the key, or <tt>list.size()</tt> if all
352          *         elements in the list are less than the specified key.  Note
353          *         that this guarantees that the return value will be &gt;= 0 if
354          *         and only if the key is found.
355          * @throws ClassCastException if the list contains elements that are not
356          *         <i>mutually comparable</i> using the specified comparator,
357          *         or the search key is not mutually comparable with the
358          *         elements of the list using this comparator.
359          */
```

```
360     public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T>
361         if (c==null)
362             return binarySearch((List) list, key);
363 
364         if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
365             return Collections.indexedBinarySearch(list, key, c);
366         else
367             return Collections.iteratorBinarySearch(list, key, c);
368     }
369 
370     private static <T> int indexedBinarySearch(List<? extends T> l, T key, Comparator<? supe
371         int low = 0;
372         int high = l.size()-1;
373 
374         while (low <= high) {
375             int mid = (low + high) >>> 1;
376             T midVal = l.get(mid);
377             int cmp = c.compare(midVal, key);
378 
379             if (cmp < 0)
380                 low = mid + 1;
381             else if (cmp > 0)
382                 high = mid - 1;
383             else
384                 return mid; // key found
385         }
386         return -(low + 1);  // key not found
387     }
388 
389     private static <T> int iteratorBinarySearch(List<? extends T> l, T key, Comparator<? sup
390         int low = 0;
391         int high = l.size()-1;
392         ListIterator<? extends T> i = l.listIterator();
393 
394         while (low <= high) {
395             int mid = (low + high) >>> 1;
396             T midVal = get(i, mid);
397             int cmp = c.compare(midVal, key);
398 
399             if (cmp < 0)
400                 low = mid + 1;
401             else if (cmp > 0)
402                 high = mid - 1;
403             else
404                 return mid; // key found
405         }
```

```
406             return -(low + 1);  // key not found
407         }
408
409         private interface SelfComparable extends Comparable<SelfComparable> {}
410
411
412         /**
413          * Reverses the order of the elements in the specified list.<p>
414          *
415          * This method runs in linear time.
416          *
417          * @param  list the list whose elements are to be reversed.
418          * @throws UnsupportedOperationException if the specified list or
419          *         its list-iterator does not support the <tt>set</tt> operation.
420          */
421         public static void reverse(List<?> list) {
422             int size = list.size();
423             if (size < REVERSE_THRESHOLD || list instanceof RandomAccess) {
424                 for (int i=0, mid=size>>1, j=size-1; i<mid; i++, j--)
425                     swap(list, i, j);
426             } else {
427                 ListIterator fwd = list.listIterator();
428                 ListIterator rev = list.listIterator(size);
429                 for (int i=0, mid=list.size()>>1; i<mid; i++) {
430                     Object tmp = fwd.next();
431                     fwd.set(rev.previous());
432                     rev.set(tmp);
433                 }
434             }
435         }
436
437         /**
438          * Randomly permutes the specified list using a default source of
439          * randomness.  All permutations occur with approximately equal
440          * likelihood.<p>
441          *
442          * The hedge "approximately" is used in the foregoing description because
443          * default source of randomness is only approximately an unbiased source
444          * of independently chosen bits.  If it were a perfect source of randomly
445          * chosen bits, then the algorithm would choose permutations with perfect
446          * uniformity.<p>
447          *
448          * This implementation traverses the list backwards, from the last element
449          * up to the second, repeatedly swapping a randomly selected element into
450          * the "current position".  Elements are randomly selected from the
451          * portion of the list that runs from the first element to the current
```

```
452         * position, inclusive. <p>
453         *
454         * This method runs in linear time.  If the specified list does not
455         * implement the {@link RandomAccess} interface and is large, this
456         * implementation dumps the specified list into an array before shuffling
457         * it, and dumps the shuffled array back into the list.  This avoids the
458         * quadratic behavior that would result from shuffling a "sequential
459         * access" list in place.
460         *
461         * @param  list the list to be shuffled.
462         * @throws UnsupportedOperationException if the specified list or
463         *         its list-iterator does not support the <tt>set</tt> operation.
464         */
465        public static void shuffle(List<?> list) {
466            Random rnd = r;
467            if (rnd == null)
468                r = rnd = new Random();
469            shuffle(list, rnd);
470        }
471        private static Random r;
472
473        /**
474         * Randomly permute the specified list using the specified source of
475         * randomness.  All permutations occur with equal likelihood
476         * assuming that the source of randomness is fair. <p>
477         *
478         * This implementation traverses the list backwards, from the last element
479         * up to the second, repeatedly swapping a randomly selected element into
480         * the "current position".  Elements are randomly selected from the
481         * portion of the list that runs from the first element to the current
482         * position, inclusive. <p>
483         *
484         * This method runs in linear time.  If the specified list does not
485         * implement the {@link RandomAccess} interface and is large, this
486         * implementation dumps the specified list into an array before shuffling
487         * it, and dumps the shuffled array back into the list.  This avoids the
488         * quadratic behavior that would result from shuffling a "sequential
489         * access" list in place.
490         *
491         * @param  list the list to be shuffled.
492         * @param  rnd the source of randomness to use to shuffle the list.
493         * @throws UnsupportedOperationException if the specified list or its
494         *         list-iterator does not support the <tt>set</tt> operation.
495         */
496        public static void shuffle(List<?> list, Random rnd) {
497            int size = list.size();
```

```
498             if (size < SHUFFLE_THRESHOLD || list instanceof RandomAccess) {
499                 for (int i=size; i>1; i--)
500                     swap(list, i-1, rnd.nextInt(i));
501             } else {
502                 Object arr[] = list.toArray();
503 
504                 // Shuffle array
505                 for (int i=size; i>1; i--)
506                     swap(arr, i-1, rnd.nextInt(i));
507 
508                 // Dump array back into list
509                 ListIterator it = list.listIterator();
510                 for (int i=0; i<arr.length; i++) {
511                     it.next();
512                     it.set(arr[i]);
513                 }
514             }
515         }
516 
517         /**
518          * Swaps the elements at the specified positions in the specified list.
519          * (If the specified positions are equal, invoking this method leaves
520          * the list unchanged.)
521          *
522          * @param list The list in which to swap elements.
523          * @param i the index of one element to be swapped.
524          * @param j the index of the other element to be swapped.
525          * @throws IndexOutOfBoundsException if either <tt>i</tt> or <tt>j</tt>
526          *         is out of range (i &lt; 0 || i &gt;= list.size()
527          *         || j &lt; 0 || j &gt;= list.size()).
528          * @since 1.4
529          */
530         public static void swap(List<?> list, int i, int j) {
531             final List l = list;
532             l.set(i, l.set(j, l.get(i)));
533         }
534 
535         /**
536          * Swaps the two specified elements in the specified array.
537          */
538         private static void swap(Object[] arr, int i, int j) {
539             Object tmp = arr[i];
540             arr[i] = arr[j];
541             arr[j] = tmp;
542         }
543 
```

```
544         /**
545          * Replaces all of the elements of the specified list with the specified
546          * element. <p>
547          *
548          * This method runs in linear time.
549          *
550          * @param  list the list to be filled with the specified element.
551          * @param  obj The element with which to fill the specified list.
552          * @throws UnsupportedOperationException if the specified list or its
553          *         list-iterator does not support the <tt>set</tt> operation.
554          */
555         public static <T> void fill(List<? super T> list, T obj) {
556             int size = list.size();
557 
558             if (size < FILL_THRESHOLD || list instanceof RandomAccess) {
559                 for (int i=0; i<size; i++)
560                     list.set(i, obj);
561             } else {
562                 ListIterator<? super T> itr = list.listIterator();
563                 for (int i=0; i<size; i++) {
564                     itr.next();
565                     itr.set(obj);
566                 }
567             }
568         }
569 
570         /**
571          * Copies all of the elements from one list into another.  After the
572          * operation, the index of each copied element in the destination list
573          * will be identical to its index in the source list.  The destination
574          * list must be at least as long as the source list.  If it is longer, the
575          * remaining elements in the destination list are unaffected. <p>
576          *
577          * This method runs in linear time.
578          *
579          * @param  dest The destination list.
580          * @param  src The source list.
581          * @throws IndexOutOfBoundsException if the destination list is too small
582          *         to contain the entire source List.
583          * @throws UnsupportedOperationException if the destination list's
584          *         list-iterator does not support the <tt>set</tt> operation.
585          */
586         public static <T> void copy(List<? super T> dest, List<? extends T> src) {
587             int srcSize = src.size();
588             if (srcSize > dest.size())
589                 throw new IndexOutOfBoundsException("Source does not fit in dest");
```

```
590
591         if (srcSize < COPY_THRESHOLD ||
592             (src instanceof RandomAccess && dest instanceof RandomAccess)) {
593             for (int i=0; i<srcSize; i++)
594                 dest.set(i, src.get(i));
595         } else {
596             ListIterator<? super T> di=dest.listIterator();
597             ListIterator<? extends T> si=src.listIterator();
598             for (int i=0; i<srcSize; i++) {
599                 di.next();
600                 di.set(si.next());
601             }
602         }
603     }
604
605     /**
606      * Returns the minimum element of the given collection, according to the
607      * <i>natural ordering</i> of its elements.  All elements in the
608      * collection must implement the <tt>Comparable</tt> interface.
609      * Furthermore, all elements in the collection must be <i>mutually
610      * comparable</i> (that is, <tt>e1.compareTo(e2)</tt> must not throw a
611      * <tt>ClassCastException</tt> for any elements <tt>e1</tt> and
612      * <tt>e2</tt> in the collection).<p>
613      *
614      * This method iterates over the entire collection, hence it requires
615      * time proportional to the size of the collection.
616      *
617      * @param  coll the collection whose minimum element is to be determined.
618      * @return the minimum element of the given collection, according
619      *         to the <i>natural ordering</i> of its elements.
620      * @throws ClassCastException if the collection contains elements that are
621      *         not <i>mutually comparable</i> (for example, strings and
622      *         integers).
623      * @throws NoSuchElementException if the collection is empty.
624      * @see Comparable
625      */
626     public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> c
627         Iterator<? extends T> i = coll.iterator();
628         T candidate = i.next();
629
630         while (i.hasNext()) {
631             T next = i.next();
632             if (next.compareTo(candidate) < 0)
633                 candidate = next;
634         }
635         return candidate;
```

```
636     }
637
638     /**
639      * Returns the minimum element of the given collection, according to the
640      * order induced by the specified comparator.  All elements in the
641      * collection must be <i>mutually comparable</i> by the specified
642      * comparator (that is, <tt>comp.compare(e1, e2)</tt> must not throw a
643      * <tt>ClassCastException</tt> for any elements <tt>e1</tt> and
644      * <tt>e2</tt> in the collection).<p>
645      *
646      * This method iterates over the entire collection, hence it requires
647      * time proportional to the size of the collection.
648      *
649      * @param  coll the collection whose minimum element is to be determined.
650      * @param  comp the comparator with which to determine the minimum element.
651      *         A <tt>null</tt> value indicates that the elements' <i>natural
652      *         ordering</i> should be used.
653      * @return the minimum element of the given collection, according
654      *         to the specified comparator.
655      * @throws ClassCastException if the collection contains elements that are
656      *         not <i>mutually comparable</i> using the specified comparator.
657      * @throws NoSuchElementException if the collection is empty.
658      * @see Comparable
659      */
660     public static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp) {
661         if (comp==null)
662             return (T)min((Collection<SelfComparable>) (Collection) coll);
663
664         Iterator<? extends T> i = coll.iterator();
665         T candidate = i.next();
666
667         while (i.hasNext()) {
668             T next = i.next();
669             if (comp.compare(next, candidate) < 0)
670                 candidate = next;
671         }
672         return candidate;
673     }
674
675     /**
676      * Returns the maximum element of the given collection, according to the
677      * <i>natural ordering</i> of its elements.  All elements in the
678      * collection must implement the <tt>Comparable</tt> interface.
679      * Furthermore, all elements in the collection must be <i>mutually
680      * comparable</i> (that is, <tt>e1.compareTo(e2)</tt> must not throw a
681      * <tt>ClassCastException</tt> for any elements <tt>e1</tt> and
```

```
682         * <tt>e2</tt> in the collection).<p>
683         *
684         * This method iterates over the entire collection, hence it requires
685         * time proportional to the size of the collection.
686         *
687         * @param  coll the collection whose maximum element is to be determined.
688         * @return the maximum element of the given collection, according
689         *         to the <i>natural ordering</i> of its elements.
690         * @throws ClassCastException if the collection contains elements that are
691         *         not <i>mutually comparable</i> (for example, strings and
692         *         integers).
693         * @throws NoSuchElementException if the collection is empty.
694         * @see Comparable
695         */
696        public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> c
697            Iterator<? extends T> i = coll.iterator();
698            T candidate = i.next();
699
700            while (i.hasNext()) {
701                T next = i.next();
702                if (next.compareTo(candidate) > 0)
703                    candidate = next;
704            }
705            return candidate;
706        }
707
708        /**
709         * Returns the maximum element of the given collection, according to the
710         * order induced by the specified comparator.  All elements in the
711         * collection must be <i>mutually comparable</i> by the specified
712         * comparator (that is, <tt>comp.compare(e1, e2)</tt> must not throw a
713         * <tt>ClassCastException</tt> for any elements <tt>e1</tt> and
714         * <tt>e2</tt> in the collection).<p>
715         *
716         * This method iterates over the entire collection, hence it requires
717         * time proportional to the size of the collection.
718         *
719         * @param  coll the collection whose maximum element is to be determined.
720         * @param  comp the comparator with which to determine the maximum element.
721         *         A <tt>null</tt> value indicates that the elements' <i>natural
722         *        ordering</i> should be used.
723         * @return the maximum element of the given collection, according
724         *         to the specified comparator.
725         * @throws ClassCastException if the collection contains elements that are
726         *         not <i>mutually comparable</i> using the specified comparator.
727         * @throws NoSuchElementException if the collection is empty.
```

```
728         * @see Comparable
729         */
730        public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp) {
731            if (comp==null)
732                return (T)max((Collection<SelfComparable>) (Collection) coll);
733
734            Iterator<? extends T> i = coll.iterator();
735            T candidate = i.next();
736
737            while (i.hasNext()) {
738                T next = i.next();
739                if (comp.compare(next, candidate) > 0)
740                    candidate = next;
741            }
742            return candidate;
743        }
744
745        /**
746         * Rotates the elements in the specified list by the specified distance.
747         * After calling this method, the element at index <tt>i</tt> will be
748         * the element previously at index <tt>(i - distance)</tt> mod
749         * <tt>list.size()</tt>, for all values of <tt>i</tt> between <tt>0</tt>
750         * and <tt>list.size()-1</tt>, inclusive.  (This method has no effect on
751         * the size of the list.)
752         *
753         * <p>For example, suppose <tt>list</tt> comprises<tt> [t, a, n, k, s]</tt>.
754         * After invoking <tt>Collections.rotate(list, 1)</tt> (or
755         * <tt>Collections.rotate(list, -4)</tt>), <tt>list</tt> will comprise
756         * <tt>[s, t, a, n, k]</tt>.
757         *
758         * <p>Note that this method can usefully be applied to sublists to
759         * move one or more elements within a list while preserving the
760         * order of the remaining elements.  For example, the following idiom
761         * moves the element at index <tt>j</tt> forward to position
762         * <tt>k</tt> (which must be greater than or equal to <tt>j</tt>):
763         * <pre>
764         *     Collections.rotate(list.subList(j, k+1), -1);
765         * </pre>
766         * To make this concrete, suppose <tt>list</tt> comprises
767         * <tt>[a, b, c, d, e]</tt>.  To move the element at index <tt>1</tt>
768         * (<tt>b</tt>) forward two positions, perform the following invocation:
769         * <pre>
770         *     Collections.rotate(l.subList(1, 4), -1);
771         * </pre>
772         * The resulting list is <tt>[a, c, d, b, e]</tt>.
773         *
```

```
774         * <p>To move more than one element forward, increase the absolute value
775         * of the rotation distance.  To move elements backward, use a positive
776         * shift distance.
777         *
778         * <p>If the specified list is small or implements the {@link
779         * RandomAccess} interface, this implementation exchanges the first
780         * element into the location it should go, and then repeatedly exchanges
781         * the displaced element into the location it should go until a displaced
782         * element is swapped into the first element.  If necessary, the process
783         * is repeated on the second and successive elements, until the rotation
784         * is complete.  If the specified list is large and doesn't implement the
785         * <tt>RandomAccess</tt> interface, this implementation breaks the
786         * list into two sublist views around index <tt>-distance mod size</tt>.
787         * Then the {@link #reverse(List)} method is invoked on each sublist view,
788         * and finally it is invoked on the entire list.  For a more complete
789         * description of both algorithms, see Section 2.3 of Jon Bentley's
790         * <i>Programming Pearls</i> (Addison-Wesley, 1986).
791         *
792         * @param list the list to be rotated.
793         * @param distance the distance to rotate the list.  There are no
794         *        constraints on this value; it may be zero, negative, or
795         *        greater than <tt>list.size()</tt>.
796         * @throws UnsupportedOperationException if the specified list or
797         *         its list-iterator does not support the <tt>set</tt> operation.
798         * @since 1.4
799         */
800        public static void rotate(List<?> list, int distance) {
801            if (list instanceof RandomAccess || list.size() < ROTATE_THRESHOLD)
802                rotate1(list, distance);
803            else
804                rotate2(list, distance);
805        }
806
807        private static <T> void rotate1(List<T> list, int distance) {
808            int size = list.size();
809            if (size == 0)
810                return;
811            distance = distance % size;
812            if (distance < 0)
813                distance += size;
814            if (distance == 0)
815                return;
816
817            for (int cycleStart = 0, nMoved = 0; nMoved != size; cycleStart++) {
818                T displaced = list.get(cycleStart);
819                int i = cycleStart;
```

```
820             do {
821                 i += distance;
822                 if (i >= size)
823                     i -= size;
824                 displaced = list.set(i, displaced);
825                 nMoved ++;
826             } while (i != cycleStart);
827         }
828     }
829 
830     private static void rotate2(List<?> list, int distance) {
831         int size = list.size();
832         if (size == 0)
833             return;
834         int mid =  -distance % size;
835         if (mid < 0)
836             mid += size;
837         if (mid == 0)
838             return;
839 
840         reverse(list.subList(0, mid));
841         reverse(list.subList(mid, size));
842         reverse(list);
843     }
844 
845     /**
846      * Replaces all occurrences of one specified value in a list with another.
847      * More formally, replaces with <tt>newVal</tt> each element <tt>e</tt>
848      * in <tt>list</tt> such that
849      * <tt>(oldVal==null ? e==null : oldVal.equals(e))</tt>.
850      * (This method has no effect on the size of the list.)
851      *
852      * @param list the list in which replacement is to occur.
853      * @param oldVal the old value to be replaced.
854      * @param newVal the new value with which <tt>oldVal</tt> is to be
855      *        replaced.
856      * @return <tt>true</tt> if <tt>list</tt> contained one or more elements
857      *         <tt>e</tt> such that
858      *         <tt>(oldVal==null ?  e==null : oldVal.equals(e))</tt>.
859      * @throws UnsupportedOperationException if the specified list or
860      *         its list-iterator does not support the <tt>set</tt> operation.
861      * @since  1.4
862      */
863     public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal) {
864         boolean result = false;
865         int size = list.size();
```

```
866             if (size < REPLACEALL_THRESHOLD || list instanceof RandomAccess) {
867                 if (oldVal==null) {
868                     for (int i=0; i<size; i++) {
869                         if (list.get(i)==null) {
870                             list.set(i, newVal);
871                             result = true;
872                         }
873                     }
874                 } else {
875                     for (int i=0; i<size; i++) {
876                         if (oldVal.equals(list.get(i))) {
877                             list.set(i, newVal);
878                             result = true;
879                         }
880                     }
881                 }
882             } else {
883                 ListIterator<T> itr=list.listIterator();
884                 if (oldVal==null) {
885                     for (int i=0; i<size; i++) {
886                         if (itr.next()==null) {
887                             itr.set(newVal);
888                             result = true;
889                         }
890                     }
891                 } else {
892                     for (int i=0; i<size; i++) {
893                         if (oldVal.equals(itr.next())) {
894                             itr.set(newVal);
895                             result = true;
896                         }
897                     }
898                 }
899             }
900             return result;
901         }
902 
903         /**
904          * Returns the starting position of the first occurrence of the specified
905          * target list within the specified source list, or -1 if there is no
906          * such occurrence.  More formally, returns the lowest index <tt>i</tt>
907          * such that <tt>source.subList(i, i+target.size()).equals(target)</tt>,
908          * or -1 if there is no such index.  (Returns -1 if
909          * <tt>target.size() > source.size()</tt>.)
910          *
911          * <p>This implementation uses the "brute force" technique of scanning
```

```
912         * over the source list, looking for a match with the target at each
913         * location in turn.
914         *
915         * @param source the list in which to search for the first occurrence
916         *       of <tt>target</tt>.
917         * @param target the list to search for as a subList of <tt>source</tt>.
918         * @return the starting position of the first occurrence of the specified
919         *         target list within the specified source list, or -1 if there
920         *         is no such occurrence.
921         * @since  1.4
922         */
923        public static int indexOfSubList(List<?> source, List<?> target) {
924            int sourceSize = source.size();
925            int targetSize = target.size();
926            int maxCandidate = sourceSize - targetSize;
927
928            if (sourceSize < INDEXOFSUBLIST_THRESHOLD ||
929                (source instanceof RandomAccess&&target instanceof RandomAccess)) {
930            nextCand:
931                for (int candidate = 0; candidate <= maxCandidate; candidate++) {
932                    for (int i=0, j=candidate; i<targetSize; i++, j++)
933                        if (!eq(target.get(i), source.get(j)))
934                            continue nextCand;  // Element mismatch, try next cand
935                    return candidate;  // All elements of candidate matched target
936                }
937            } else {  // Iterator version of above algorithm
938                ListIterator<?> si = source.listIterator();
939            nextCand:
940                for (int candidate = 0; candidate <= maxCandidate; candidate++) {
941                    ListIterator<?> ti = target.listIterator();
942                    for (int i=0; i<targetSize; i++) {
943                        if (!eq(ti.next(), si.next())) {
944                            // Back up source iterator to next candidate
945                            for (int j=0; j<i; j++)
946                                si.previous();
947                            continue nextCand;
948                        }
949                    }
950                    return candidate;
951                }
952            }
953            return -1;  // No candidate matched the target
954        }
955
956        /**
957         * 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) {
```

```
1050            if (c==null)
1051                throw new NullPointerException();
1052            this.c = c;
1053        }
1054
1055        public int size()                   {return c.size();}
1056        public boolean isEmpty()            {return c.isEmpty();}
1057        public boolean contains(Object o)   {return c.contains(o);}
1058        public Object[] toArray()           {return c.toArray();}
1059        public <T> T[] toArray(T[] a)       {return c.toArray(a);}
1060        public String toString()            {return c.toString();}
1061
1062        public Iterator<E> iterator() {
1063            return new Iterator<E>() {
1064                private final Iterator<? extends E> i = c.iterator();
1065
1066                public boolean hasNext() {return i.hasNext();}
1067                public E next()          {return i.next();}
1068                public void remove() {
1069                    throw new UnsupportedOperationException();
1070                }
1071            };
1072        }
1073
1074        public boolean add(E e) {
1075            throw new UnsupportedOperationException();
1076        }
1077        public boolean remove(Object o) {
1078            throw new UnsupportedOperationException();
1079        }
1080
1081        public boolean containsAll(Collection<?> coll) {
1082            return c.containsAll(coll);
1083        }
1084        public boolean addAll(Collection<? extends E> coll) {
1085            throw new UnsupportedOperationException();
1086        }
1087        public boolean removeAll(Collection<?> coll) {
1088            throw new UnsupportedOperationException();
1089        }
1090        public boolean retainAll(Collection<?> coll) {
1091            throw new UnsupportedOperationException();
1092        }
1093        public void clear() {
1094            throw new UnsupportedOperationException();
1095        }
```

```
1096        }
1097
1098        /**
1099         * Returns an unmodifiable view of the specified set.  This method allows
1100         * modules to provide users with "read-only" access to internal sets.
1101         * Query operations on the returned set "read through" to the specified
1102         * set, and attempts to modify the returned set, whether direct or via its
1103         * iterator, result in an <tt>UnsupportedOperationException</tt>.<p>
1104         *
1105         * The returned set will be serializable if the specified set
1106         * is serializable.
1107         *
1108         * @param  s the set for which an unmodifiable view is to be returned.
1109         * @return an unmodifiable view of the specified set.
1110         */
1111        public static <T> Set<T> unmodifiableSet(Set<? extends T> s) {
1112            return new UnmodifiableSet<>(s);
1113        }
1114
1115        /**
1116         * @serial include
1117         */
1118        static class UnmodifiableSet<E> extends UnmodifiableCollection<E>
1119                                        implements Set<E>, Serializable {
1120            private static final long serialVersionUID = -9215047833775013803L;
1121
1122            UnmodifiableSet(Set<? extends E> s)     {super(s);}
1123            public boolean equals(Object o) {return o == this || c.equals(o);}
1124            public int hashCode()           {return c.hashCode();}
1125        }
1126
1127        /**
1128         * Returns an unmodifiable view of the specified sorted set.  This method
1129         * allows modules to provide users with "read-only" access to internal
1130         * sorted sets.  Query operations on the returned sorted set "read
1131         * through" to the specified sorted set.  Attempts to modify the returned
1132         * sorted set, whether direct, via its iterator, or via its
1133         * <tt>subSet</tt>, <tt>headSet</tt>, or <tt>tailSet</tt> views, result in
1134         * an <tt>UnsupportedOperationException</tt>.<p>
1135         *
1136         * The returned sorted set will be serializable if the specified sorted set
1137         * is serializable.
1138         *
1139         * @param s the sorted set for which an unmodifiable view is to be
1140         *        returned.
1141         * @return an unmodifiable view of the specified sorted set.
```

```
1142         */
1143        public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s) {
1144            return new UnmodifiableSortedSet<>(s);
1145        }
1146
1147        /**
1148         * @serial include
1149         */
1150        static class UnmodifiableSortedSet<E>
1151                                 extends UnmodifiableSet<E>
1152                                 implements SortedSet<E>, Serializable {
1153            private static final long serialVersionUID = -4929149591599911165L;
1154            private final SortedSet<E> ss;
1155
1156            UnmodifiableSortedSet(SortedSet<E> s) {super(s); ss = s;}
1157
1158            public Comparator<? super E> comparator() {return ss.comparator();}
1159
1160            public SortedSet<E> subSet(E fromElement, E toElement) {
1161                return new UnmodifiableSortedSet<>(ss.subSet(fromElement,toElement));
1162            }
1163            public SortedSet<E> headSet(E toElement) {
1164                return new UnmodifiableSortedSet<>(ss.headSet(toElement));
1165            }
1166            public SortedSet<E> tailSet(E fromElement) {
1167                return new UnmodifiableSortedSet<>(ss.tailSet(fromElement));
1168            }
1169
1170            public E first()                    {return ss.first();}
1171            public E last()                     {return ss.last();}
1172        }
1173
1174        /**
1175         * Returns an unmodifiable view of the specified list.  This method allows
1176         * modules to provide users with "read-only" access to internal
1177         * lists.  Query operations on the returned list "read through" to the
1178         * specified list, and attempts to modify the returned list, whether
1179         * direct or via its iterator, result in an
1180         * <tt>UnsupportedOperationException</tt>.<p>
1181         *
1182         * The returned list will be serializable if the specified list
1183         * is serializable. Similarly, the returned list will implement
1184         * {@link RandomAccess} if the specified list does.
1185         *
1186         * @param  list the list for which an unmodifiable view is to be returned.
1187         * @return an unmodifiable view of the specified list.
```

```
1188         */
1189        public static <T> List<T> unmodifiableList(List<? extends T> list) {
1190            return (list instanceof RandomAccess ?
1191                    new UnmodifiableRandomAccessList<>(list) :
1192                    new UnmodifiableList<>(list));
1193        }
1194
1195        /**
1196         * @serial include
1197         */
1198        static class UnmodifiableList<E> extends UnmodifiableCollection<E>
1199                                         implements List<E> {
1200            private static final long serialVersionUID = -283967356065247728L;
1201            final List<? extends E> list;
1202
1203            UnmodifiableList(List<? extends E> list) {
1204                super(list);
1205                this.list = list;
1206            }
1207
1208            public boolean equals(Object o) {return o == this || list.equals(o);}
1209            public int hashCode()           {return list.hashCode();}
1210
1211            public E get(int index) {return list.get(index);}
1212            public E set(int index, E element) {
1213                throw new UnsupportedOperationException();
1214            }
1215            public void add(int index, E element) {
1216                throw new UnsupportedOperationException();
1217            }
1218            public E remove(int index) {
1219                throw new UnsupportedOperationException();
1220            }
1221            public int indexOf(Object o)            {return list.indexOf(o);}
1222            public int lastIndexOf(Object o)        {return list.lastIndexOf(o);}
1223            public boolean addAll(int index, Collection<? extends E> c) {
1224                throw new UnsupportedOperationException();
1225            }
1226            public ListIterator<E> listIterator()   {return listIterator(0);}
1227
1228            public ListIterator<E> listIterator(final int index) {
1229                return new ListIterator<E>() {
1230                    private final ListIterator<? extends E> i
1231                        = list.listIterator(index);
1232
1233                    public boolean hasNext()     {return i.hasNext();}
```

```
1234                public E next()             {return i.next();}
1235                public boolean hasPrevious() {return i.hasPrevious();}
1236                public E previous()         {return i.previous();}
1237                public int nextIndex()      {return i.nextIndex();}
1238                public int previousIndex()  {return i.previousIndex();}
1239
1240                public void remove() {
1241                    throw new UnsupportedOperationException();
1242                }
1243                public void set(E e) {
1244                    throw new UnsupportedOperationException();
1245                }
1246                public void add(E e) {
1247                    throw new UnsupportedOperationException();
1248                }
1249            };
1250        }
1251
1252        public List<E> subList(int fromIndex, int toIndex) {
1253            return new UnmodifiableList<>(list.subList(fromIndex, toIndex));
1254        }
1255
1256        /**
1257         * UnmodifiableRandomAccessList instances are serialized as
1258         * UnmodifiableList instances to allow them to be deserialized
1259         * in pre-1.4 JREs (which do not have UnmodifiableRandomAccessList).
1260         * This method inverts the transformation.  As a beneficial
1261         * side-effect, it also grafts the RandomAccess marker onto
1262         * UnmodifiableList instances that were serialized in pre-1.4 JREs.
1263         *
1264         * Note: Unfortunately, UnmodifiableRandomAccessList instances
1265         * serialized in 1.4.1 and deserialized in 1.4 will become
1266         * UnmodifiableList instances, as this method was missing in 1.4.
1267         */
1268        private Object readResolve() {
1269            return (list instanceof RandomAccess
1270                    ? new UnmodifiableRandomAccessList<>(list)
1271                    : this);
1272        }
1273    }
1274
1275    /**
1276     * @serial include
1277     */
1278    static class UnmodifiableRandomAccessList<E> extends UnmodifiableList<E>
1279                                                 implements RandomAccess
```

```
1280        {
1281            UnmodifiableRandomAccessList(List<? extends E> list) {
1282                super(list);
1283            }
1284
1285            public List<E> subList(int fromIndex, int toIndex) {
1286                return new UnmodifiableRandomAccessList<>(
1287                    list.subList(fromIndex, toIndex));
1288            }
1289
1290            private static final long serialVersionUID = -2542308836966382001L;
1291
1292            /**
1293             * Allows instances to be deserialized in pre-1.4 JREs (which do
1294             * not have UnmodifiableRandomAccessList).  UnmodifiableList has
1295             * a readResolve method that inverts this transformation upon
1296             * deserialization.
1297             */
1298            private Object writeReplace() {
1299                return new UnmodifiableList<>(list);
1300            }
1301        }
1302
1303        /**
1304         * Returns an unmodifiable view of the specified map.  This method
1305         * allows modules to provide users with "read-only" access to internal
1306         * maps.  Query operations on the returned map "read through"
1307         * to the specified map, and attempts to modify the returned
1308         * map, whether direct or via its collection views, result in an
1309         * <tt>UnsupportedOperationException</tt>.<p>
1310         *
1311         * The returned map will be serializable if the specified map
1312         * is serializable.
1313         *
1314         * @param  m the map for which an unmodifiable view is to be returned.
1315         * @return an unmodifiable view of the specified map.
1316         */
1317        public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m) {
1318            return new UnmodifiableMap<>(m);
1319        }
1320
1321        /**
1322         * @serial include
1323         */
1324        private static class UnmodifiableMap<K,V> implements Map<K,V>, Serializable {
1325            private static final long serialVersionUID = -1034234728574286014L;
```

```
1326
1327            private final Map<? extends K, ? extends V> m;
1328
1329            UnmodifiableMap(Map<? extends K, ? extends V> m) {
1330                if (m==null)
1331                    throw new NullPointerException();
1332                this.m = m;
1333            }
1334
1335            public int size()                        {return m.size();}
1336            public boolean isEmpty()                 {return m.isEmpty();}
1337            public boolean containsKey(Object key)   {return m.containsKey(key);}
1338            public boolean containsValue(Object val) {return m.containsValue(val);}
1339            public V get(Object key)                 {return m.get(key);}
1340
1341            public V put(K key, V value) {
1342                throw new UnsupportedOperationException();
1343            }
1344            public V remove(Object key) {
1345                throw new UnsupportedOperationException();
1346            }
1347            public void putAll(Map<? extends K, ? extends V> m) {
1348                throw new UnsupportedOperationException();
1349            }
1350            public void clear() {
1351                throw new UnsupportedOperationException();
1352            }
1353
1354            private transient Set<K> keySet = null;
1355            private transient Set<Map.Entry<K,V>> entrySet = null;
1356            private transient Collection<V> values = null;
1357
1358            public Set<K> keySet() {
1359                if (keySet==null)
1360                    keySet = unmodifiableSet(m.keySet());
1361                return keySet;
1362            }
1363
1364            public Set<Map.Entry<K,V>> entrySet() {
1365                if (entrySet==null)
1366                    entrySet = new UnmodifiableEntrySet<>(m.entrySet());
1367                return entrySet;
1368            }
1369
1370            public Collection<V> values() {
1371                if (values==null)
```

```
1372                values = unmodifiableCollection(m.values());
1373            return values;
1374        }
1375
1376        public boolean equals(Object o) {return o == this || m.equals(o);}
1377        public int hashCode()           {return m.hashCode();}
1378        public String toString()        {return m.toString();}
1379
1380        /**
1381         * We need this class in addition to UnmodifiableSet as
1382         * Map.Entries themselves permit modification of the backing Map
1383         * via their setValue operation.  This class is subtle: there are
1384         * many possible attacks that must be thwarted.
1385         *
1386         * @serial include
1387         */
1388        static class UnmodifiableEntrySet<K,V>
1389            extends UnmodifiableSet<Map.Entry<K,V>> {
1390            private static final long serialVersionUID = 7854390611657943733L;
1391
1392            UnmodifiableEntrySet(Set<? extends Map.Entry<? extends K, ? extends V>> s) {
1393                super((Set)s);
1394            }
1395            public Iterator<Map.Entry<K,V>> iterator() {
1396                return new Iterator<Map.Entry<K,V>>() {
1397                    private final Iterator<? extends Map.Entry<? extends K, ? extends V>> i
1398
1399                    public boolean hasNext() {
1400                        return i.hasNext();
1401                    }
1402                    public Map.Entry<K,V> next() {
1403                        return new UnmodifiableEntry<>(i.next());
1404                    }
1405                    public void remove() {
1406                        throw new UnsupportedOperationException();
1407                    }
1408                };
1409            }
1410
1411            public Object[] toArray() {
1412                Object[] a = c.toArray();
1413                for (int i=0; i<a.length; i++)
1414                    a[i] = new UnmodifiableEntry<>((Map.Entry<K,V>)a[i]);
1415                return a;
1416            }
1417
```

```
1418            public <T> T[] toArray(T[] a) {
1419                // We don't pass a to c.toArray, to avoid window of
1420                // vulnerability wherein an unscrupulous multithreaded client
1421                // could get his hands on raw (unwrapped) Entries from c.
1422                Object[] arr = c.toArray(a.length==0 ? a : Arrays.copyOf(a, 0));
1423
1424                for (int i=0; i<arr.length; i++)
1425                    arr[i] = new UnmodifiableEntry<>((Map.Entry<K,V>)arr[i]);
1426
1427                if (arr.length > a.length)
1428                    return (T[])arr;
1429
1430                System.arraycopy(arr, 0, a, 0, arr.length);
1431                if (a.length > arr.length)
1432                    a[arr.length] = null;
1433                return a;
1434            }
1435
1436            /**
1437             * This method is overridden to protect the backing set against
1438             * an object with a nefarious equals function that senses
1439             * that the equality-candidate is Map.Entry and calls its
1440             * setValue method.
1441             */
1442            public boolean contains(Object o) {
1443                if (!(o instanceof Map.Entry))
1444                    return false;
1445                return c.contains(
1446                    new UnmodifiableEntry<>((Map.Entry<?,?>) o));
1447            }
1448
1449            /**
1450             * The next two methods are overridden to protect against
1451             * an unscrupulous List whose contains(Object o) method senses
1452             * when o is a Map.Entry, and calls o.setValue.
1453             */
1454            public boolean containsAll(Collection<?> coll) {
1455                for (Object e : coll) {
1456                    if (!contains(e)) // Invokes safe contains() above
1457                        return false;
1458                }
1459                return true;
1460            }
1461            public boolean equals(Object o) {
1462                if (o == this)
1463                    return true;
```

```
1464
1465                if (!(o instanceof Set))
1466                    return false;
1467                Set s = (Set) o;
1468                if (s.size() != c.size())
1469                    return false;
1470                return containsAll(s); // Invokes safe containsAll() above
1471            }
1472
1473            /**
1474             * This "wrapper class" serves two purposes: it prevents
1475             * the client from modifying the backing Map, by short-circuiting
1476             * the setValue method, and it protects the backing Map against
1477             * an ill-behaved Map.Entry that attempts to modify another
1478             * Map Entry when asked to perform an equality check.
1479             */
1480            private static class UnmodifiableEntry<K,V> implements Map.Entry<K,V> {
1481                private Map.Entry<? extends K, ? extends V> e;
1482
1483                UnmodifiableEntry(Map.Entry<? extends K, ? extends V> e) {this.e = e;}
1484
1485                public K getKey()        {return e.getKey();}
1486                public V getValue()      {return e.getValue();}
1487                public V setValue(V value) {
1488                    throw new UnsupportedOperationException();
1489                }
1490                public int hashCode()    {return e.hashCode();}
1491                public boolean equals(Object o) {
1492                    if (this == o)
1493                        return true;
1494                    if (!(o instanceof Map.Entry))
1495                        return false;
1496                    Map.Entry t = (Map.Entry)o;
1497                    return eq(e.getKey(),   t.getKey()) &&
1498                           eq(e.getValue(), t.getValue());
1499                }
1500                public String toString() {return e.toString();}
1501            }
1502        }
1503    }
1504
1505    /**
1506     * Returns an unmodifiable view of the specified sorted map.  This method
1507     * allows modules to provide users with "read-only" access to internal
1508     * sorted maps.  Query operations on the returned sorted map "read through"
1509     * to the specified sorted map.  Attempts to modify the returned
```

```
1510         * sorted map, whether direct, via its collection views, or via its
1511         * <tt>subMap</tt>, <tt>headMap</tt>, or <tt>tailMap</tt> views, result in
1512         * an <tt>UnsupportedOperationException</tt>.<p>
1513         *
1514         * The returned sorted map will be serializable if the specified sorted map
1515         * is serializable.
1516         *
1517         * @param m the sorted map for which an unmodifiable view is to be
1518         *        returned.
1519         * @return an unmodifiable view of the specified sorted map.
1520         */
1521        public static <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K, ? extends V> m) {
1522            return new UnmodifiableSortedMap<>(m);
1523        }
1524
1525        /**
1526         * @serial include
1527         */
1528        static class UnmodifiableSortedMap<K,V>
1529              extends UnmodifiableMap<K,V>
1530              implements SortedMap<K,V>, Serializable {
1531            private static final long serialVersionUID = -8806743815996713206L;
1532
1533            private final SortedMap<K, ? extends V> sm;
1534
1535            UnmodifiableSortedMap(SortedMap<K, ? extends V> m) {super(m); sm = m;}
1536
1537            public Comparator<? super K> comparator() {return sm.comparator();}
1538
1539            public SortedMap<K,V> subMap(K fromKey, K toKey) {
1540                return new UnmodifiableSortedMap<>(sm.subMap(fromKey, toKey));
1541            }
1542            public SortedMap<K,V> headMap(K toKey) {
1543                return new UnmodifiableSortedMap<>(sm.headMap(toKey));
1544            }
1545            public SortedMap<K,V> tailMap(K fromKey) {
1546                return new UnmodifiableSortedMap<>(sm.tailMap(fromKey));
1547            }
1548
1549            public K firstKey()           {return sm.firstKey();}
1550            public K lastKey()            {return sm.lastKey();}
1551        }
1552
1553
1554        // Synch Wrappers
1555
```

```
1556        /**
1557         * Returns a synchronized (thread-safe) collection backed by the specified
1558         * collection.  In order to guarantee serial access, it is critical that
1559         * <strong>all</strong> access to the backing collection is accomplished
1560         * through the returned collection.<p>
1561         *
1562         * It is imperative that the user manually synchronize on the returned
1563         * collection when iterating over it:
1564         * <pre>
1565         *  Collection c = Collections.synchronizedCollection(myCollection);
1566         *     ...
1567         *  synchronized (c) {
1568         *      Iterator i = c.iterator(); // Must be in the synchronized block
1569         *      while (i.hasNext())
1570         *         foo(i.next());
1571         *  }
1572         * </pre>
1573         * Failure to follow this advice may result in non-deterministic behavior.
1574         *
1575         * <p>The returned collection does <i>not</i> pass the <tt>hashCode</tt>
1576         * and <tt>equals</tt> operations through to the backing collection, but
1577         * relies on <tt>Object</tt>'s equals and hashCode methods.  This is
1578         * necessary to preserve the contracts of these operations in the case
1579         * that the backing collection is a set or a list.<p>
1580         *
1581         * The returned collection will be serializable if the specified collection
1582         * is serializable.
1583         *
1584         * @param  c the collection to be "wrapped" in a synchronized collection.
1585         * @return a synchronized view of the specified collection.
1586         */
1587        public static <T> Collection<T> synchronizedCollection(Collection<T> c) {
1588            return new SynchronizedCollection<>(c);
1589        }
1590
1591        static <T> Collection<T> synchronizedCollection(Collection<T> c, Object mutex) {
1592            return new SynchronizedCollection<>(c, mutex);
1593        }
1594
1595        /**
1596         * @serial include
1597         */
1598        static class SynchronizedCollection<E> implements Collection<E>, Serializable {
1599            private static final long serialVersionUID = 3053995032091335093L;
1600
1601            final Collection<E> c;  // Backing Collection
```

```
1602        final Object mutex;     // Object on which to synchronize
1603
1604        SynchronizedCollection(Collection<E> c) {
1605            if (c==null)
1606                throw new NullPointerException();
1607            this.c = c;
1608            mutex = this;
1609        }
1610        SynchronizedCollection(Collection<E> c, Object mutex) {
1611            this.c = c;
1612            this.mutex = mutex;
1613        }
1614
1615        public int size() {
1616            synchronized (mutex) {return c.size();}
1617        }
1618        public boolean isEmpty() {
1619            synchronized (mutex) {return c.isEmpty();}
1620        }
1621        public boolean contains(Object o) {
1622            synchronized (mutex) {return c.contains(o);}
1623        }
1624        public Object[] toArray() {
1625            synchronized (mutex) {return c.toArray();}
1626        }
1627        public <T> T[] toArray(T[] a) {
1628            synchronized (mutex) {return c.toArray(a);}
1629        }
1630
1631        public Iterator<E> iterator() {
1632            return c.iterator(); // Must be manually synched by user!
1633        }
1634
1635        public boolean add(E e) {
1636            synchronized (mutex) {return c.add(e);}
1637        }
1638        public boolean remove(Object o) {
1639            synchronized (mutex) {return c.remove(o);}
1640        }
1641
1642        public boolean containsAll(Collection<?> coll) {
1643            synchronized (mutex) {return c.containsAll(coll);}
1644        }
1645        public boolean addAll(Collection<? extends E> coll) {
1646            synchronized (mutex) {return c.addAll(coll);}
1647        }
```

```
1648            public boolean removeAll(Collection<?> coll) {
1649                synchronized (mutex) {return c.removeAll(coll);}
1650            }
1651            public boolean retainAll(Collection<?> coll) {
1652                synchronized (mutex) {return c.retainAll(coll);}
1653            }
1654            public void clear() {
1655                synchronized (mutex) {c.clear();}
1656            }
1657            public String toString() {
1658                synchronized (mutex) {return c.toString();}
1659            }
1660            private void writeObject(ObjectOutputStream s) throws IOException {
1661                synchronized (mutex) {s.defaultWriteObject();}
1662            }
1663        }
1664
1665        /**
1666         * Returns a synchronized (thread-safe) set backed by the specified
1667         * set.  In order to guarantee serial access, it is critical that
1668         * <strong>all</strong> access to the backing set is accomplished
1669         * through the returned set.<p>
1670         *
1671         * It is imperative that the user manually synchronize on the returned
1672         * set when iterating over it:
1673         * <pre>
1674         *  Set s = Collections.synchronizedSet(new HashSet());
1675         *      ...
1676         *  synchronized (s) {
1677         *      Iterator i = s.iterator(); // Must be in the synchronized block
1678         *      while (i.hasNext())
1679         *          foo(i.next());
1680         *  }
1681         * </pre>
1682         * Failure to follow this advice may result in non-deterministic behavior.
1683         *
1684         * <p>The returned set will be serializable if the specified set is
1685         * serializable.
1686         *
1687         * @param  s the set to be "wrapped" in a synchronized set.
1688         * @return a synchronized view of the specified set.
1689         */
1690        public static <T> Set<T> synchronizedSet(Set<T> s) {
1691            return new SynchronizedSet<>(s);
1692        }
1693
```

```
1694        static <T> Set<T> synchronizedSet(Set<T> s, Object mutex) {
1695            return new SynchronizedSet<>(s, mutex);
1696        }
1697
1698        /**
1699         * @serial include
1700         */
1701        static class SynchronizedSet<E>
1702              extends SynchronizedCollection<E>
1703              implements Set<E> {
1704            private static final long serialVersionUID = 487447009682186044L;
1705
1706            SynchronizedSet(Set<E> s) {
1707                super(s);
1708            }
1709            SynchronizedSet(Set<E> s, Object mutex) {
1710                super(s, mutex);
1711            }
1712
1713            public boolean equals(Object o) {
1714                if (this == o)
1715                    return true;
1716                synchronized (mutex) {return c.equals(o);}
1717            }
1718            public int hashCode() {
1719                synchronized (mutex) {return c.hashCode();}
1720            }
1721        }
1722
1723        /**
1724         * Returns a synchronized (thread-safe) sorted set backed by the specified
1725         * sorted set.  In order to guarantee serial access, it is critical that
1726         * <strong>all</strong> access to the backing sorted set is accomplished
1727         * through the returned sorted set (or its views).<p>
1728         *
1729         * It is imperative that the user manually synchronize on the returned
1730         * sorted set when iterating over it or any of its <tt>subSet</tt>,
1731         * <tt>headSet</tt>, or <tt>tailSet</tt> views.
1732         * <pre>
1733         *  SortedSet s = Collections.synchronizedSortedSet(new TreeSet());
1734         *      ...
1735         *  synchronized (s) {
1736         *      Iterator i = s.iterator(); // Must be in the synchronized block
1737         *      while (i.hasNext())
1738         *          foo(i.next());
1739         *  }
```

```
1740         * </pre>
1741         * or:
1742         * <pre>
1743         *  SortedSet s = Collections.synchronizedSortedSet(new TreeSet());
1744         *  SortedSet s2 = s.headSet(foo);
1745         *      ...
1746         *  synchronized (s) {  // Note: s, not s2!!!
1747         *      Iterator i = s2.iterator(); // Must be in the synchronized block
1748         *      while (i.hasNext())
1749         *          foo(i.next());
1750         *  }
1751         * </pre>
1752         * Failure to follow this advice may result in non-deterministic behavior.
1753         *
1754         * <p>The returned sorted set will be serializable if the specified
1755         * sorted set is serializable.
1756         *
1757         * @param  s the sorted set to be "wrapped" in a synchronized sorted set.
1758         * @return a synchronized view of the specified sorted set.
1759         */
1760        public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s) {
1761            return new SynchronizedSortedSet<>(s);
1762        }
1763
1764        /**
1765         * @serial include
1766         */
1767        static class SynchronizedSortedSet<E>
1768            extends SynchronizedSet<E>
1769            implements SortedSet<E>
1770        {
1771            private static final long serialVersionUID = 8695801310862127406L;
1772
1773            private final SortedSet<E> ss;
1774
1775            SynchronizedSortedSet(SortedSet<E> s) {
1776                super(s);
1777                ss = s;
1778            }
1779            SynchronizedSortedSet(SortedSet<E> s, Object mutex) {
1780                super(s, mutex);
1781                ss = s;
1782            }
1783
1784            public Comparator<? super E> comparator() {
1785                synchronized (mutex) {return ss.comparator();}
```

```
1786            }
1787
1788            public SortedSet<E> subSet(E fromElement, E toElement) {
1789                synchronized (mutex) {
1790                    return new SynchronizedSortedSet<>(
1791                        ss.subSet(fromElement, toElement), mutex);
1792                }
1793            }
1794            public SortedSet<E> headSet(E toElement) {
1795                synchronized (mutex) {
1796                    return new SynchronizedSortedSet<>(ss.headSet(toElement), mutex);
1797                }
1798            }
1799            public SortedSet<E> tailSet(E fromElement) {
1800                synchronized (mutex) {
1801                   return new SynchronizedSortedSet<>(ss.tailSet(fromElement),mutex);
1802                }
1803            }
1804
1805            public E first() {
1806                synchronized (mutex) {return ss.first();}
1807            }
1808            public E last() {
1809                synchronized (mutex) {return ss.last();}
1810            }
1811        }
1812
1813        /**
1814         * Returns a synchronized (thread-safe) list backed by the specified
1815         * list.  In order to guarantee serial access, it is critical that
1816         * <strong>all</strong> access to the backing list is accomplished
1817         * through the returned list.<p>
1818         *
1819         * It is imperative that the user manually synchronize on the returned
1820         * list when iterating over it:
1821         * <pre>
1822         *  List list = Collections.synchronizedList(new ArrayList());
1823         *      ...
1824         *  synchronized (list) {
1825         *      Iterator i = list.iterator(); // Must be in synchronized block
1826         *      while (i.hasNext())
1827         *          foo(i.next());
1828         *  }
1829         * </pre>
1830         * Failure to follow this advice may result in non-deterministic behavior.
1831         *
```

```
1832         * <p>The returned list will be serializable if the specified list is
1833         * serializable.
1834         *
1835         * @param  list the list to be "wrapped" in a synchronized list.
1836         * @return a synchronized view of the specified list.
1837         */
1838        public static <T> List<T> synchronizedList(List<T> list) {
1839            return (list instanceof RandomAccess ?
1840                    new SynchronizedRandomAccessList<>(list) :
1841                    new SynchronizedList<>(list));
1842        }
1843
1844        static <T> List<T> synchronizedList(List<T> list, Object mutex) {
1845            return (list instanceof RandomAccess ?
1846                    new SynchronizedRandomAccessList<>(list, mutex) :
1847                    new SynchronizedList<>(list, mutex));
1848        }
1849
1850        /**
1851         * @serial include
1852         */
1853        static class SynchronizedList<E>
1854            extends SynchronizedCollection<E>
1855            implements List<E> {
1856            private static final long serialVersionUID = -7754090372962971524L;
1857
1858            final List<E> list;
1859
1860            SynchronizedList(List<E> list) {
1861                super(list);
1862                this.list = list;
1863            }
1864            SynchronizedList(List<E> list, Object mutex) {
1865                super(list, mutex);
1866                this.list = list;
1867            }
1868
1869            public boolean equals(Object o) {
1870                if (this == o)
1871                    return true;
1872                synchronized (mutex) {return list.equals(o);}
1873            }
1874            public int hashCode() {
1875                synchronized (mutex) {return list.hashCode();}
1876            }
1877
```

```
1878            public E get(int index) {
1879                synchronized (mutex) {return list.get(index);}
1880            }
1881            public E set(int index, E element) {
1882                synchronized (mutex) {return list.set(index, element);}
1883            }
1884            public void add(int index, E element) {
1885                synchronized (mutex) {list.add(index, element);}
1886            }
1887            public E remove(int index) {
1888                synchronized (mutex) {return list.remove(index);}
1889            }
1890
1891            public int indexOf(Object o) {
1892                synchronized (mutex) {return list.indexOf(o);}
1893            }
1894            public int lastIndexOf(Object o) {
1895                synchronized (mutex) {return list.lastIndexOf(o);}
1896            }
1897
1898            public boolean addAll(int index, Collection<? extends E> c) {
1899                synchronized (mutex) {return list.addAll(index, c);}
1900            }
1901
1902            public ListIterator<E> listIterator() {
1903                return list.listIterator(); // Must be manually synched by user
1904            }
1905
1906            public ListIterator<E> listIterator(int index) {
1907                return list.listIterator(index); // Must be manually synched by user
1908            }
1909
1910            public List<E> subList(int fromIndex, int toIndex) {
1911                synchronized (mutex) {
1912                    return new SynchronizedList<>(list.subList(fromIndex, toIndex),
1913                                                  mutex);
1914                }
1915            }
1916
1917            /**
1918             * SynchronizedRandomAccessList instances are serialized as
1919             * SynchronizedList instances to allow them to be deserialized
1920             * in pre-1.4 JREs (which do not have SynchronizedRandomAccessList).
1921             * This method inverts the transformation.  As a beneficial
1922             * side-effect, it also grafts the RandomAccess marker onto
1923             * SynchronizedList instances that were serialized in pre-1.4 JREs.
```

```
1924            *
1925            * Note: Unfortunately, SynchronizedRandomAccessList instances
1926            * serialized in 1.4.1 and deserialized in 1.4 will become
1927            * SynchronizedList instances, as this method was missing in 1.4.
1928            */
1929           private Object readResolve() {
1930               return (list instanceof RandomAccess
1931                       ? new SynchronizedRandomAccessList<>(list)
1932                       : this);
1933           }
1934       }
1935
1936       /**
1937        * @serial include
1938        */
1939       static class SynchronizedRandomAccessList<E>
1940           extends SynchronizedList<E>
1941           implements RandomAccess {
1942
1943           SynchronizedRandomAccessList(List<E> list) {
1944               super(list);
1945           }
1946
1947           SynchronizedRandomAccessList(List<E> list, Object mutex) {
1948               super(list, mutex);
1949           }
1950
1951           public List<E> subList(int fromIndex, int toIndex) {
1952               synchronized (mutex) {
1953                   return new SynchronizedRandomAccessList<>(
1954                       list.subList(fromIndex, toIndex), mutex);
1955               }
1956           }
1957
1958           private static final long serialVersionUID = 1530674583602358482L;
1959
1960           /**
1961            * Allows instances to be deserialized in pre-1.4 JREs (which do
1962            * not have SynchronizedRandomAccessList).  SynchronizedList has
1963            * a readResolve method that inverts this transformation upon
1964            * deserialization.
1965            */
1966           private Object writeReplace() {
1967               return new SynchronizedList<>(list);
1968           }
1969       }
```

```
1970
1971        /**
1972         * Returns a synchronized (thread-safe) map backed by the specified
1973         * map.  In order to guarantee serial access, it is critical that
1974         * <strong>all</strong> access to the backing map is accomplished
1975         * through the returned map.<p>
1976         *
1977         * It is imperative that the user manually synchronize on the returned
1978         * map when iterating over any of its collection views:
1979         * <pre>
1980         *  Map m = Collections.synchronizedMap(new HashMap());
1981         *      ...
1982         *  Set s = m.keySet();  // Needn't be in synchronized block
1983         *      ...
1984         *  synchronized (m) {  // Synchronizing on m, not s!
1985         *      Iterator i = s.iterator(); // Must be in synchronized block
1986         *      while (i.hasNext())
1987         *          foo(i.next());
1988         *  }
1989         * </pre>
1990         * Failure to follow this advice may result in non-deterministic behavior.
1991         *
1992         * <p>The returned map will be serializable if the specified map is
1993         * serializable.
1994         *
1995         * @param  m the map to be "wrapped" in a synchronized map.
1996         * @return a synchronized view of the specified map.
1997         */
1998        public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {
1999            return new SynchronizedMap<>(m);
2000        }
2001
2002        /**
2003         * @serial include
2004         */
2005        private static class SynchronizedMap<K,V>
2006            implements Map<K,V>, Serializable {
2007            private static final long serialVersionUID = 1978198479659022715L;
2008
2009            private final Map<K,V> m;     // Backing Map
2010            final Object      mutex;        // Object on which to synchronize
2011
2012            SynchronizedMap(Map<K,V> m) {
2013                if (m==null)
2014                    throw new NullPointerException();
2015                this.m = m;
```

```
2016            mutex = this;
2017        }
2018
2019        SynchronizedMap(Map<K,V> m, Object mutex) {
2020            this.m = m;
2021            this.mutex = mutex;
2022        }
2023
2024        public int size() {
2025            synchronized (mutex) {return m.size();}
2026        }
2027        public boolean isEmpty() {
2028            synchronized (mutex) {return m.isEmpty();}
2029        }
2030        public boolean containsKey(Object key) {
2031            synchronized (mutex) {return m.containsKey(key);}
2032        }
2033        public boolean containsValue(Object value) {
2034            synchronized (mutex) {return m.containsValue(value);}
2035        }
2036        public V get(Object key) {
2037            synchronized (mutex) {return m.get(key);}
2038        }
2039
2040        public V put(K key, V value) {
2041            synchronized (mutex) {return m.put(key, value);}
2042        }
2043        public V remove(Object key) {
2044            synchronized (mutex) {return m.remove(key);}
2045        }
2046        public void putAll(Map<? extends K, ? extends V> map) {
2047            synchronized (mutex) {m.putAll(map);}
2048        }
2049        public void clear() {
2050            synchronized (mutex) {m.clear();}
2051        }
2052
2053        private transient Set<K> keySet = null;
2054        private transient Set<Map.Entry<K,V>> entrySet = null;
2055        private transient Collection<V> values = null;
2056
2057        public Set<K> keySet() {
2058            synchronized (mutex) {
2059                if (keySet==null)
2060                    keySet = new SynchronizedSet<>(m.keySet(), mutex);
2061                return keySet;
```

```
2062            }
2063        }
2064
2065        public Set<Map.Entry<K,V>> entrySet() {
2066            synchronized (mutex) {
2067                if (entrySet==null)
2068                    entrySet = new SynchronizedSet<>(m.entrySet(), mutex);
2069                return entrySet;
2070            }
2071        }
2072
2073        public Collection<V> values() {
2074            synchronized (mutex) {
2075                if (values==null)
2076                    values = new SynchronizedCollection<>(m.values(), mutex);
2077                return values;
2078            }
2079        }
2080
2081        public boolean equals(Object o) {
2082            if (this == o)
2083                return true;
2084            synchronized (mutex) {return m.equals(o);}
2085        }
2086        public int hashCode() {
2087            synchronized (mutex) {return m.hashCode();}
2088        }
2089        public String toString() {
2090            synchronized (mutex) {return m.toString();}
2091        }
2092        private void writeObject(ObjectOutputStream s) throws IOException {
2093            synchronized (mutex) {s.defaultWriteObject();}
2094        }
2095    }
2096
2097    /**
2098     * Returns a synchronized (thread-safe) sorted map backed by the specified
2099     * sorted map.  In order to guarantee serial access, it is critical that
2100     * <strong>all</strong> access to the backing sorted map is accomplished
2101     * through the returned sorted map (or its views).<p>
2102     *
2103     * It is imperative that the user manually synchronize on the returned
2104     * sorted map when iterating over any of its collection views, or the
2105     * collections views of any of its <tt>subMap</tt>, <tt>headMap</tt> or
2106     * <tt>tailMap</tt> views.
2107     * <pre>
```

```
2108         *  SortedMap m = Collections.synchronizedSortedMap(new TreeMap());
2109         *      ...
2110         *  Set s = m.keySet();  // Needn't be in synchronized block
2111         *      ...
2112         *  synchronized (m) {  // Synchronizing on m, not s!
2113         *      Iterator i = s.iterator(); // Must be in synchronized block
2114         *      while (i.hasNext())
2115         *          foo(i.next());
2116         *  }
2117         * </pre>
2118         * or:
2119         * <pre>
2120         *  SortedMap m = Collections.synchronizedSortedMap(new TreeMap());
2121         *  SortedMap m2 = m.subMap(foo, bar);
2122         *      ...
2123         *  Set s2 = m2.keySet();  // Needn't be in synchronized block
2124         *      ...
2125         *  synchronized (m) {  // Synchronizing on m, not m2 or s2!
2126         *      Iterator i = s.iterator(); // Must be in synchronized block
2127         *      while (i.hasNext())
2128         *          foo(i.next());
2129         *  }
2130         * </pre>
2131         * Failure to follow this advice may result in non-deterministic behavior.
2132         *
2133         * <p>The returned sorted map will be serializable if the specified
2134         * sorted map is serializable.
2135         *
2136         * @param  m the sorted map to be "wrapped" in a synchronized sorted map.
2137         * @return a synchronized view of the specified sorted map.
2138         */
2139        public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m) {
2140            return new SynchronizedSortedMap<>(m);
2141        }
2142
2143
2144        /**
2145         * @serial include
2146         */
2147        static class SynchronizedSortedMap<K,V>
2148            extends SynchronizedMap<K,V>
2149            implements SortedMap<K,V>
2150        {
2151            private static final long serialVersionUID = -8798146769416483793L;
2152
2153            private final SortedMap<K,V> sm;
```

```
2154
2155            SynchronizedSortedMap(SortedMap<K,V> m) {
2156                super(m);
2157                sm = m;
2158            }
2159            SynchronizedSortedMap(SortedMap<K,V> m, Object mutex) {
2160                super(m, mutex);
2161                sm = m;
2162            }
2163
2164            public Comparator<? super K> comparator() {
2165                synchronized (mutex) {return sm.comparator();}
2166            }
2167
2168            public SortedMap<K,V> subMap(K fromKey, K toKey) {
2169                synchronized (mutex) {
2170                    return new SynchronizedSortedMap<>(
2171                        sm.subMap(fromKey, toKey), mutex);
2172                }
2173            }
2174            public SortedMap<K,V> headMap(K toKey) {
2175                synchronized (mutex) {
2176                    return new SynchronizedSortedMap<>(sm.headMap(toKey), mutex);
2177                }
2178            }
2179            public SortedMap<K,V> tailMap(K fromKey) {
2180                synchronized (mutex) {
2181                   return new SynchronizedSortedMap<>(sm.tailMap(fromKey),mutex);
2182                }
2183            }
2184
2185            public K firstKey() {
2186                synchronized (mutex) {return sm.firstKey();}
2187            }
2188            public K lastKey() {
2189                synchronized (mutex) {return sm.lastKey();}
2190            }
2191        }
2192
2193        // Dynamically typesafe collection wrappers
2194
2195        /**
2196         * Returns a dynamically typesafe view of the specified collection.
2197         * Any attempt to insert an element of the wrong type will result in an
2198         * immediate {@link ClassCastException}.  Assuming a collection
2199         * contains no incorrectly typed elements prior to the time a
```

```
2200      * dynamically typesafe view is generated, and that all subsequent
2201      * access to the collection takes place through the view, it is
2202      * <i>guaranteed</i> that the collection cannot contain an incorrectly
2203      * typed element.
2204      *
2205      * <p>The generics mechanism in the language provides compile-time
2206      * (static) type checking, but it is possible to defeat this mechanism
2207      * with unchecked casts.  Usually this is not a problem, as the compiler
2208      * issues warnings on all such unchecked operations.  There are, however,
2209      * times when static type checking alone is not sufficient.  For example,
2210      * suppose a collection is passed to a third-party library and it is
2211      * imperative that the library code not corrupt the collection by
2212      * inserting an element of the wrong type.
2213      *
2214      * <p>Another use of dynamically typesafe views is debugging.  Suppose a
2215      * program fails with a {@code ClassCastException}, indicating that an
2216      * incorrectly typed element was put into a parameterized collection.
2217      * Unfortunately, the exception can occur at any time after the erroneous
2218      * element is inserted, so it typically provides little or no information
2219      * as to the real source of the problem.  If the problem is reproducible,
2220      * one can quickly determine its source by temporarily modifying the
2221      * program to wrap the collection with a dynamically typesafe view.
2222      * For example, this declaration:
2223      *  <pre> {@code
2224      *     Collection<String> c = new HashSet<String>();
2225      * }</pre>
2226      * may be replaced temporarily by this one:
2227      *  <pre> {@code
2228      *     Collection<String> c = Collections.checkedCollection(
2229      *         new HashSet<String>(), String.class);
2230      * }</pre>
2231      * Running the program again will cause it to fail at the point where
2232      * an incorrectly typed element is inserted into the collection, clearly
2233      * identifying the source of the problem.  Once the problem is fixed, the
2234      * modified declaration may be reverted back to the original.
2235      *
2236      * <p>The returned collection does <i>not</i> pass the hashCode and equals
2237      * operations through to the backing collection, but relies on
2238      * {@code Object}'s {@code equals} and {@code hashCode} methods.  This
2239      * is necessary to preserve the contracts of these operations in the case
2240      * that the backing collection is a set or a list.
2241      *
2242      * <p>The returned collection will be serializable if the specified
2243      * collection is serializable.
2244      *
2245      * <p>Since {@code null} is considered to be a value of any reference
```

```
2246         * type, the returned collection permits insertion of null elements
2247         * whenever the backing collection does.
2248         *
2249         * @param c the collection for which a dynamically typesafe view is to be
2250         *          returned
2251         * @param type the type of element that {@code c} is permitted to hold
2252         * @return a dynamically typesafe view of the specified collection
2253         * @since 1.5
2254         */
2255        public static <E> Collection<E> checkedCollection(Collection<E> c,
2256                                                          Class<E> type) {
2257            return new CheckedCollection<>(c, type);
2258        }
2259
2260        @SuppressWarnings("unchecked")
2261        static <T> T[] zeroLengthArray(Class<T> type) {
2262            return (T[]) Array.newInstance(type, 0);
2263        }
2264
2265        /**
2266         * @serial include
2267         */
2268        static class CheckedCollection<E> implements Collection<E>, Serializable {
2269            private static final long serialVersionUID = 1578914078182001775L;
2270
2271            final Collection<E> c;
2272            final Class<E> type;
2273
2274            void typeCheck(Object o) {
2275                if (o != null && !type.isInstance(o))
2276                    throw new ClassCastException(badElementMsg(o));
2277            }
2278
2279            private String badElementMsg(Object o) {
2280                return "Attempt to insert " + o.getClass() +
2281                    " element into collection with element type " + type;
2282            }
2283
2284            CheckedCollection(Collection<E> c, Class<E> type) {
2285                if (c==null || type == null)
2286                    throw new NullPointerException();
2287                this.c = c;
2288                this.type = type;
2289            }
2290
2291            public int size()                   { return c.size(); }
```

```
2292            public boolean isEmpty()          { return c.isEmpty(); }
2293            public boolean contains(Object o) { return c.contains(o); }
2294            public Object[] toArray()         { return c.toArray(); }
2295            public <T> T[] toArray(T[] a)     { return c.toArray(a); }
2296            public String toString()          { return c.toString(); }
2297            public boolean remove(Object o)   { return c.remove(o); }
2298            public void clear()               {        c.clear(); }
2299
2300            public boolean containsAll(Collection<?> coll) {
2301                return c.containsAll(coll);
2302            }
2303            public boolean removeAll(Collection<?> coll) {
2304                return c.removeAll(coll);
2305            }
2306            public boolean retainAll(Collection<?> coll) {
2307                return c.retainAll(coll);
2308            }
2309
2310            public Iterator<E> iterator() {
2311                final Iterator<E> it = c.iterator();
2312                return new Iterator<E>() {
2313                    public boolean hasNext() { return it.hasNext(); }
2314                    public E next()          { return it.next(); }
2315                    public void remove()     {        it.remove(); }};
2316            }
2317
2318            public boolean add(E e) {
2319                typeCheck(e);
2320                return c.add(e);
2321            }
2322
2323            private E[] zeroLengthElementArray = null; // Lazily initialized
2324
2325            private E[] zeroLengthElementArray() {
2326                return zeroLengthElementArray != null ? zeroLengthElementArray :
2327                    (zeroLengthElementArray = zeroLengthArray(type));
2328            }
2329
2330            @SuppressWarnings("unchecked")
2331            Collection<E> checkedCopyOf(Collection<? extends E> coll) {
2332                Object[] a = null;
2333                try {
2334                    E[] z = zeroLengthElementArray();
2335                    a = coll.toArray(z);
2336                    // Defend against coll violating the toArray contract
2337                    if (a.getClass() != z.getClass())
```

```
                    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
```

```
2384         * @param type the type of element that {@code s} is permitted to hold
2385         * @return a dynamically typesafe view of the specified set
2386         * @since 1.5
2387         */
2388        public static <E> Set<E> checkedSet(Set<E> s, Class<E> type) {
2389            return new CheckedSet<>(s, type);
2390        }
2391
2392        /**
2393         * @serial include
2394         */
2395        static class CheckedSet<E> extends CheckedCollection<E>
2396                                   implements Set<E>, Serializable
2397        {
2398            private static final long serialVersionUID = 4694047833775013803L;
2399
2400            CheckedSet(Set<E> s, Class<E> elementType) { super(s, elementType); }
2401
2402            public boolean equals(Object o) { return o == this || c.equals(o); }
2403            public int hashCode()           { return c.hashCode(); }
2404        }
2405
2406        /**
2407         * Returns a dynamically typesafe view of the specified sorted set.
2408         * Any attempt to insert an element of the wrong type will result in an
2409         * immediate {@link ClassCastException}.  Assuming a sorted set
2410         * contains no incorrectly typed elements prior to the time a
2411         * dynamically typesafe view is generated, and that all subsequent
2412         * access to the sorted set takes place through the view, it is
2413         * <i>guaranteed</i> that the sorted set cannot contain an incorrectly
2414         * typed element.
2415         *
2416         * <p>A discussion of the use of dynamically typesafe views may be
2417         * found in the documentation for the {@link #checkedCollection
2418         * checkedCollection} method.
2419         *
2420         * <p>The returned sorted set will be serializable if the specified sorted
2421         * set is serializable.
2422         *
2423         * <p>Since {@code null} is considered to be a value of any reference
2424         * type, the returned sorted set permits insertion of null elements
2425         * whenever the backing sorted set does.
2426         *
2427         * @param s the sorted set for which a dynamically typesafe view is to be
2428         *          returned
2429         * @param type the type of element that {@code s} is permitted to hold
```

```
2430         * @return a dynamically typesafe view of the specified sorted set
2431         * @since 1.5
2432         */
2433        public static <E> SortedSet<E> checkedSortedSet(SortedSet<E> s,
2434                                                        Class<E> type) {
2435            return new CheckedSortedSet<>(s, type);
2436        }
2437
2438        /**
2439         * @serial include
2440         */
2441        static class CheckedSortedSet<E> extends CheckedSet<E>
2442            implements SortedSet<E>, Serializable
2443        {
2444            private static final long serialVersionUID = 1599911165492914959L;
2445            private final SortedSet<E> ss;
2446
2447            CheckedSortedSet(SortedSet<E> s, Class<E> type) {
2448                super(s, type);
2449                ss = s;
2450            }
2451
2452            public Comparator<? super E> comparator() { return ss.comparator(); }
2453            public E first()                  { return ss.first(); }
2454            public E last()                   { return ss.last(); }
2455
2456            public SortedSet<E> subSet(E fromElement, E toElement) {
2457                return checkedSortedSet(ss.subSet(fromElement,toElement), type);
2458            }
2459            public SortedSet<E> headSet(E toElement) {
2460                return checkedSortedSet(ss.headSet(toElement), type);
2461            }
2462            public SortedSet<E> tailSet(E fromElement) {
2463                return checkedSortedSet(ss.tailSet(fromElement), type);
2464            }
2465        }
2466
2467        /**
2468         * Returns a dynamically typesafe view of the specified list.
2469         * Any attempt to insert an element of the wrong type will result in
2470         * an immediate {@link ClassCastException}.  Assuming a list contains
2471         * no incorrectly typed elements prior to the time a dynamically typesafe
2472         * view is generated, and that all subsequent access to the list
2473         * takes place through the view, it is <i>guaranteed</i> that the
2474         * list cannot contain an incorrectly typed element.
2475         *
```

```
2476         * <p>A discussion of the use of dynamically typesafe views may be
2477         * found in the documentation for the {@link #checkedCollection
2478         * checkedCollection} method.
2479         *
2480         * <p>The returned list will be serializable if the specified list
2481         * is serializable.
2482         *
2483         * <p>Since {@code null} is considered to be a value of any reference
2484         * type, the returned list permits insertion of null elements whenever
2485         * the backing list does.
2486         *
2487         * @param list the list for which a dynamically typesafe view is to be
2488         *             returned
2489         * @param type the type of element that {@code list} is permitted to hold
2490         * @return a dynamically typesafe view of the specified list
2491         * @since 1.5
2492         */
2493        public static <E> List<E> checkedList(List<E> list, Class<E> type) {
2494            return (list instanceof RandomAccess ?
2495                    new CheckedRandomAccessList<>(list, type) :
2496                    new CheckedList<>(list, type));
2497        }
2498
2499        /**
2500         * @serial include
2501         */
2502        static class CheckedList<E>
2503            extends CheckedCollection<E>
2504            implements List<E>
2505        {
2506            private static final long serialVersionUID = 65247728283967356L;
2507            final List<E> list;
2508
2509            CheckedList(List<E> list, Class<E> type) {
2510                super(list, type);
2511                this.list = list;
2512            }
2513
2514            public boolean equals(Object o)  { return o == this || list.equals(o); }
2515            public int hashCode()            { return list.hashCode(); }
2516            public E get(int index)          { return list.get(index); }
2517            public E remove(int index)       { return list.remove(index); }
2518            public int indexOf(Object o)     { return list.indexOf(o); }
2519            public int lastIndexOf(Object o) { return list.lastIndexOf(o); }
2520
2521            public E set(int index, E element) {
```

```
2522                typeCheck(element);
2523                return list.set(index, element);
2524            }
2525
2526            public void add(int index, E element) {
2527                typeCheck(element);
2528                list.add(index, element);
2529            }
2530
2531            public boolean addAll(int index, Collection<? extends E> c) {
2532                return list.addAll(index, checkedCopyOf(c));
2533            }
2534            public ListIterator<E> listIterator()   { return listIterator(0); }
2535
2536            public ListIterator<E> listIterator(final int index) {
2537                final ListIterator<E> i = list.listIterator(index);
2538
2539                return new ListIterator<E>() {
2540                    public boolean hasNext()     { return i.hasNext(); }
2541                    public E next()              { return i.next(); }
2542                    public boolean hasPrevious() { return i.hasPrevious(); }
2543                    public E previous()          { return i.previous(); }
2544                    public int nextIndex()       { return i.nextIndex(); }
2545                    public int previousIndex()   { return i.previousIndex(); }
2546                    public void remove()         {        i.remove(); }
2547
2548                    public void set(E e) {
2549                        typeCheck(e);
2550                        i.set(e);
2551                    }
2552
2553                    public void add(E e) {
2554                        typeCheck(e);
2555                        i.add(e);
2556                    }
2557                };
2558            }
2559
2560            public List<E> subList(int fromIndex, int toIndex) {
2561                return new CheckedList<>(list.subList(fromIndex, toIndex), type);
2562            }
2563        }
2564
2565        /**
2566         * @serial include
2567         */
```

```
2568        static class CheckedRandomAccessList<E> extends CheckedList<E>
2569                                                implements RandomAccess
2570        {
2571            private static final long serialVersionUID = 1638200125423088369L;
2572
2573            CheckedRandomAccessList(List<E> list, Class<E> type) {
2574                super(list, type);
2575            }
2576
2577            public List<E> subList(int fromIndex, int toIndex) {
2578                return new CheckedRandomAccessList<>(
2579                    list.subList(fromIndex, toIndex), type);
2580            }
2581        }
2582
2583        /**
2584         * Returns a dynamically typesafe view of the specified map.
2585         * Any attempt to insert a mapping whose key or value have the wrong
2586         * type will result in an immediate {@link ClassCastException}.
2587         * Similarly, any attempt to modify the value currently associated with
2588         * a key will result in an immediate {@link ClassCastException},
2589         * whether the modification is attempted directly through the map
2590         * itself, or through a {@link Map.Entry} instance obtained from the
2591         * map's {@link Map#entrySet() entry set} view.
2592         *
2593         * <p>Assuming a map contains no incorrectly typed keys or values
2594         * prior to the time a dynamically typesafe view is generated, and
2595         * that all subsequent access to the map takes place through the view
2596         * (or one of its collection views), it is <i>guaranteed</i> that the
2597         * map cannot contain an incorrectly typed key or value.
2598         *
2599         * <p>A discussion of the use of dynamically typesafe views may be
2600         * found in the documentation for the {@link #checkedCollection
2601         * checkedCollection} method.
2602         *
2603         * <p>The returned map will be serializable if the specified map is
2604         * serializable.
2605         *
2606         * <p>Since {@code null} is considered to be a value of any reference
2607         * type, the returned map permits insertion of null keys or values
2608         * whenever the backing map does.
2609         *
2610         * @param m the map for which a dynamically typesafe view is to be
2611         *          returned
2612         * @param keyType the type of key that {@code m} is permitted to hold
2613         * @param valueType the type of value that {@code m} is permitted to hold
```

```
2614         * @return a dynamically typesafe view of the specified map
2615         * @since 1.5
2616         */
2617        public static <K, V> Map<K, V> checkedMap(Map<K, V> m,
2618                                                  Class<K> keyType,
2619                                                  Class<V> valueType) {
2620            return new CheckedMap<>(m, keyType, valueType);
2621        }
2622
2623
2624        /**
2625         * @serial include
2626         */
2627        private static class CheckedMap<K,V>
2628            implements Map<K,V>, Serializable
2629        {
2630            private static final long serialVersionUID = 5742860141034234728L;
2631
2632            private final Map<K, V> m;
2633            final Class<K> keyType;
2634            final Class<V> valueType;
2635
2636            private void typeCheck(Object key, Object value) {
2637                if (key != null && !keyType.isInstance(key))
2638                    throw new ClassCastException(badKeyMsg(key));
2639
2640                if (value != null && !valueType.isInstance(value))
2641                    throw new ClassCastException(badValueMsg(value));
2642            }
2643
2644            private String badKeyMsg(Object key) {
2645                return "Attempt to insert " + key.getClass() +
2646                    " key into map with key type " + keyType;
2647            }
2648
2649            private String badValueMsg(Object value) {
2650                return "Attempt to insert " + value.getClass() +
2651                    " value into map with value type " + valueType;
2652            }
2653
2654            CheckedMap(Map<K, V> m, Class<K> keyType, Class<V> valueType) {
2655                if (m == null || keyType == null || valueType == null)
2656                    throw new NullPointerException();
2657                this.m = m;
2658                this.keyType = keyType;
2659                this.valueType = valueType;
```

```
2660        }
2661
2662        public int size()                       { return m.size(); }
2663        public boolean isEmpty()                { return m.isEmpty(); }
2664        public boolean containsKey(Object key) { return m.containsKey(key); }
2665        public boolean containsValue(Object v) { return m.containsValue(v); }
2666        public V get(Object key)                { return m.get(key); }
2667        public V remove(Object key)             { return m.remove(key); }
2668        public void clear()                     { m.clear(); }
2669        public Set<K> keySet()                  { return m.keySet(); }
2670        public Collection<V> values()           { return m.values(); }
2671        public boolean equals(Object o)         { return o == this || m.equals(o); }
2672        public int hashCode()                   { return m.hashCode(); }
2673        public String toString()                { return m.toString(); }
2674
2675        public V put(K key, V value) {
2676            typeCheck(key, value);
2677            return m.put(key, value);
2678        }
2679
2680        @SuppressWarnings("unchecked")
2681        public void putAll(Map<? extends K, ? extends V> t) {
2682            // Satisfy the following goals:
2683            // - good diagnostics in case of type mismatch
2684            // - all-or-nothing semantics
2685            // - protection from malicious t
2686            // - correct behavior if t is a concurrent map
2687            Object[] entries = t.entrySet().toArray();
2688            List<Map.Entry<K,V>> checked = new ArrayList<>(entries.length);
2689            for (Object o : entries) {
2690                Map.Entry<?,?> e = (Map.Entry<?,?>) o;
2691                Object k = e.getKey();
2692                Object v = e.getValue();
2693                typeCheck(k, v);
2694                checked.add(
2695                    new AbstractMap.SimpleImmutableEntry<>((K) k, (V) v));
2696            }
2697            for (Map.Entry<K,V> e : checked)
2698                m.put(e.getKey(), e.getValue());
2699        }
2700
2701        private transient Set<Map.Entry<K,V>> entrySet = null;
2702
2703        public Set<Map.Entry<K,V>> entrySet() {
2704            if (entrySet==null)
2705                entrySet = new CheckedEntrySet<>(m.entrySet(), valueType);
```

```
2706            return entrySet;
2707        }
2708
2709        /**
2710         * We need this class in addition to CheckedSet as Map.Entry permits
2711         * modification of the backing Map via the setValue operation.  This
2712         * class is subtle: there are many possible attacks that must be
2713         * thwarted.
2714         *
2715         * @serial exclude
2716         */
2717        static class CheckedEntrySet<K,V> implements Set<Map.Entry<K,V>> {
2718            private final Set<Map.Entry<K,V>> s;
2719            private final Class<V> valueType;
2720
2721            CheckedEntrySet(Set<Map.Entry<K, V>> s, Class<V> valueType) {
2722                this.s = s;
2723                this.valueType = valueType;
2724            }
2725
2726            public int size()        { return s.size(); }
2727            public boolean isEmpty() { return s.isEmpty(); }
2728            public String toString() { return s.toString(); }
2729            public int hashCode()    { return s.hashCode(); }
2730            public void clear()      {        s.clear(); }
2731
2732            public boolean add(Map.Entry<K, V> e) {
2733                throw new UnsupportedOperationException();
2734            }
2735            public boolean addAll(Collection<? extends Map.Entry<K, V>> coll) {
2736                throw new UnsupportedOperationException();
2737            }
2738
2739            public Iterator<Map.Entry<K,V>> iterator() {
2740                final Iterator<Map.Entry<K, V>> i = s.iterator();
2741                final Class<V> valueType = this.valueType;
2742
2743                return new Iterator<Map.Entry<K,V>>() {
2744                    public boolean hasNext() { return i.hasNext(); }
2745                    public void remove()     { i.remove(); }
2746
2747                    public Map.Entry<K,V> next() {
2748                        return checkedEntry(i.next(), valueType);
2749                    }
2750                };
2751            }
```

```
2752
2753            @SuppressWarnings("unchecked")
2754            public Object[] toArray() {
2755                Object[] source = s.toArray();
2756
2757                /*
2758                 * Ensure that we don't get an ArrayStoreException even if
2759                 * s.toArray returns an array of something other than Object
2760                 */
2761                Object[] dest = (CheckedEntry.class.isInstance(
2762                    source.getClass().getComponentType()) ? source :
2763                                new Object[source.length]);
2764
2765                for (int i = 0; i < source.length; i++)
2766                    dest[i] = checkedEntry((Map.Entry<K,V>)source[i],
2767                                           valueType);
2768                return dest;
2769            }
2770
2771            @SuppressWarnings("unchecked")
2772            public <T> T[] toArray(T[] a) {
2773                // We don't pass a to s.toArray, to avoid window of
2774                // vulnerability wherein an unscrupulous multithreaded client
2775                // could get his hands on raw (unwrapped) Entries from s.
2776                T[] arr = s.toArray(a.length==0 ? a : Arrays.copyOf(a, 0));
2777
2778                for (int i=0; i<arr.length; i++)
2779                    arr[i] = (T) checkedEntry((Map.Entry<K,V>)arr[i],
2780                                              valueType);
2781                if (arr.length > a.length)
2782                    return arr;
2783
2784                System.arraycopy(arr, 0, a, 0, arr.length);
2785                if (a.length > arr.length)
2786                    a[arr.length] = null;
2787                return a;
2788            }
2789
2790            /**
2791             * This method is overridden to protect the backing set against
2792             * an object with a nefarious equals function that senses
2793             * that the equality-candidate is Map.Entry and calls its
2794             * setValue method.
2795             */
2796            public boolean contains(Object o) {
2797                if (!(o instanceof Map.Entry))
```

```
2798                    return false;
2799                Map.Entry<?,?> e = (Map.Entry<?,?>) o;
2800                return s.contains(
2801                    (e instanceof CheckedEntry) ? e : checkedEntry(e, valueType));
2802            }
2803
2804            /**
2805             * The bulk collection methods are overridden to protect
2806             * against an unscrupulous collection whose contains(Object o)
2807             * method senses when o is a Map.Entry, and calls o.setValue.
2808             */
2809            public boolean containsAll(Collection<?> c) {
2810                for (Object o : c)
2811                    if (!contains(o)) // Invokes safe contains() above
2812                        return false;
2813                return true;
2814            }
2815
2816            public boolean remove(Object o) {
2817                if (!(o instanceof Map.Entry))
2818                    return false;
2819                return s.remove(new AbstractMap.SimpleImmutableEntry
2820                                <>((Map.Entry<?,?>)o));
2821            }
2822
2823            public boolean removeAll(Collection<?> c) {
2824                return batchRemove(c, false);
2825            }
2826            public boolean retainAll(Collection<?> c) {
2827                return batchRemove(c, true);
2828            }
2829            private boolean batchRemove(Collection<?> c, boolean complement) {
2830                boolean modified = false;
2831                Iterator<Map.Entry<K,V>> it = iterator();
2832                while (it.hasNext()) {
2833                    if (c.contains(it.next()) != complement) {
2834                        it.remove();
2835                        modified = true;
2836                    }
2837                }
2838                return modified;
2839            }
2840
2841            public boolean equals(Object o) {
2842                if (o == this)
2843                    return true;
```

```
            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}.
     *
```

```
2982         * <li>{@link Iterator#next next} always throws {@link
2983         * NoSuchElementException}.
2984         *
2985         * <li>{@link Iterator#remove remove} always throws {@link
2986         * IllegalStateException}.
2987         *
2988         * </ul>
2989         *
2990         * <p>Implementations of this method are permitted, but not
2991         * required, to return the same object from multiple invocations.
2992         *
2993         * @return an empty iterator
2994         * @since 1.7
2995         */
2996        @SuppressWarnings("unchecked")
2997        public static <T> Iterator<T> emptyIterator() {
2998            return (Iterator<T>) EmptyIterator.EMPTY_ITERATOR;
2999        }
3000
3001        private static class EmptyIterator<E> implements Iterator<E> {
3002            static final EmptyIterator<Object> EMPTY_ITERATOR
3003                = new EmptyIterator<>();
3004
3005            public boolean hasNext() { return false; }
3006            public E next() { throw new NoSuchElementException(); }
3007            public void remove() { throw new IllegalStateException(); }
3008        }
3009
3010        /**
3011         * Returns a list iterator that has no elements.  More precisely,
3012         *
3013         * <ul compact>
3014         *
3015         * <li>{@link Iterator#hasNext hasNext} and {@link
3016         * ListIterator#hasPrevious hasPrevious} always return {@code
3017         * false}.
3018         *
3019         * <li>{@link Iterator#next next} and {@link ListIterator#previous
3020         * previous} always throw {@link NoSuchElementException}.
3021         *
3022         * <li>{@link Iterator#remove remove} and {@link ListIterator#set
3023         * set} always throw {@link IllegalStateException}.
3024         *
3025         * <li>{@link ListIterator#add add} always throws {@link
3026         * UnsupportedOperationException}.
3027         *
```

```
3028         * <li>{@link ListIterator#nextIndex nextIndex} always returns
3029         * {@code 0} .
3030         *
3031         * <li>{@link ListIterator#previousIndex previousIndex} always
3032         * returns {@code -1}.
3033         *
3034         * </ul>
3035         *
3036         * <p>Implementations of this method are permitted, but not
3037         * required, to return the same object from multiple invocations.
3038         *
3039         * @return an empty list iterator
3040         * @since 1.7
3041         */
3042        @SuppressWarnings("unchecked")
3043        public static <T> ListIterator<T> emptyListIterator() {
3044            return (ListIterator<T>) EmptyListIterator.EMPTY_ITERATOR;
3045        }
3046
3047        private static class EmptyListIterator<E>
3048            extends EmptyIterator<E>
3049            implements ListIterator<E>
3050        {
3051            static final EmptyListIterator<Object> EMPTY_ITERATOR
3052                = new EmptyListIterator<>();
3053
3054            public boolean hasPrevious() { return false; }
3055            public E previous() { throw new NoSuchElementException(); }
3056            public int nextIndex()     { return 0; }
3057            public int previousIndex() { return -1; }
3058            public void set(E e) { throw new IllegalStateException(); }
3059            public void add(E e) { throw new UnsupportedOperationException(); }
3060        }
3061
3062        /**
3063         * Returns an enumeration that has no elements.  More precisely,
3064         *
3065         * <ul compact>
3066         *
3067         * <li>{@link Enumeration#hasMoreElements hasMoreElements} always
3068         * returns {@code false}.
3069         *
3070         * <li> {@link Enumeration#nextElement nextElement} always throws
3071         * {@link NoSuchElementException}.
3072         *
3073         * </ul>
```

```
3074         *
3075         * <p>Implementations of this method are permitted, but not
3076         * required, to return the same object from multiple invocations.
3077         *
3078         * @return an empty enumeration
3079         * @since 1.7
3080         */
3081        @SuppressWarnings("unchecked")
3082        public static <T> Enumeration<T> emptyEnumeration() {
3083            return (Enumeration<T>) EmptyEnumeration.EMPTY_ENUMERATION;
3084        }
3085
3086        private static class EmptyEnumeration<E> implements Enumeration<E> {
3087            static final EmptyEnumeration<Object> EMPTY_ENUMERATION
3088                = new EmptyEnumeration<>();
3089
3090            public boolean hasMoreElements() { return false; }
3091            public E nextElement() { throw new NoSuchElementException(); }
3092        }
3093
3094        /**
3095         * The empty set (immutable).  This set is serializable.
3096         *
3097         * @see #emptySet()
3098         */
3099        @SuppressWarnings("unchecked")
3100        public static final Set EMPTY_SET = new EmptySet<>();
3101
3102        /**
3103         * Returns the empty set (immutable).  This set is serializable.
3104         * Unlike the like-named field, this method is parameterized.
3105         *
3106         * <p>This example illustrates the type-safe way to obtain an empty set:
3107         * <pre>
3108         *     Set&lt;String&gt; s = Collections.emptySet();
3109         * </pre>
3110         * Implementation note:  Implementations of this method need not
3111         * create a separate <tt>Set</tt> object for each call.   Using this
3112         * method is likely to have comparable cost to using the like-named
3113         * field.  (Unlike this method, the field does not provide type safety.)
3114         *
3115         * @see #EMPTY_SET
3116         * @since 1.5
3117         */
3118        @SuppressWarnings("unchecked")
3119        public static final <T> Set<T> emptySet() {
```

```
3120            return (Set<T>) EMPTY_SET;
3121        }
3122
3123        /**
3124         * @serial include
3125         */
3126        private static class EmptySet<E>
3127            extends AbstractSet<E>
3128            implements Serializable
3129        {
3130            private static final long serialVersionUID = 1582296315990362920L;
3131
3132            public Iterator<E> iterator() { return emptyIterator(); }
3133
3134            public int size() {return 0;}
3135            public boolean isEmpty() {return true;}
3136
3137            public boolean contains(Object obj) {return false;}
3138            public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
3139
3140            public Object[] toArray() { return new Object[0]; }
3141
3142            public <T> T[] toArray(T[] a) {
3143                if (a.length > 0)
3144                    a[0] = null;
3145                return a;
3146            }
3147
3148            // Preserves singleton property
3149            private Object readResolve() {
3150                return EMPTY_SET;
3151            }
3152        }
3153
3154        /**
3155         * The empty list (immutable).  This list is serializable.
3156         *
3157         * @see #emptyList()
3158         */
3159        @SuppressWarnings("unchecked")
3160        public static final List EMPTY_LIST = new EmptyList<>();
3161
3162        /**
3163         * Returns the empty list (immutable).  This list is serializable.
3164         *
3165         * <p>This example illustrates the type-safe way to obtain an empty list:
```

```
3166         * <pre>
3167         *     List&lt;String&gt; s = Collections.emptyList();
3168         * </pre>
3169         * Implementation note:  Implementations of this method need not
3170         * create a separate <tt>List</tt> object for each call.   Using this
3171         * method is likely to have comparable cost to using the like-named
3172         * field.  (Unlike this method, the field does not provide type safety.)
3173         *
3174         * @see #EMPTY_LIST
3175         * @since 1.5
3176         */
3177        @SuppressWarnings("unchecked")
3178        public static final <T> List<T> emptyList() {
3179            return (List<T>) EMPTY_LIST;
3180        }
3181
3182        /**
3183         * @serial include
3184         */
3185        private static class EmptyList<E>
3186            extends AbstractList<E>
3187            implements RandomAccess, Serializable {
3188            private static final long serialVersionUID = 8842843931221139166L;
3189
3190            public Iterator<E> iterator() {
3191                return emptyIterator();
3192            }
3193            public ListIterator<E> listIterator() {
3194                return emptyListIterator();
3195            }
3196
3197            public int size() {return 0;}
3198            public boolean isEmpty() {return true;}
3199
3200            public boolean contains(Object obj) {return false;}
3201            public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
3202
3203            public Object[] toArray() { return new Object[0]; }
3204
3205            public <T> T[] toArray(T[] a) {
3206                if (a.length > 0)
3207                    a[0] = null;
3208                return a;
3209            }
3210
3211            public E get(int index) {
```

```
3212                throw new IndexOutOfBoundsException("Index: "+index);
3213            }
3214
3215            public boolean equals(Object o) {
3216                return (o instanceof List) && ((List<?>)o).isEmpty();
3217            }
3218
3219            public int hashCode() { return 1; }
3220
3221            // Preserves singleton property
3222            private Object readResolve() {
3223                return EMPTY_LIST;
3224            }
3225        }
3226
3227        /**
3228         * The empty map (immutable).  This map is serializable.
3229         *
3230         * @see #emptyMap()
3231         * @since 1.3
3232         */
3233        @SuppressWarnings("unchecked")
3234        public static final Map EMPTY_MAP = new EmptyMap<>();
3235
3236        /**
3237         * Returns the empty map (immutable).  This map is serializable.
3238         *
3239         * <p>This example illustrates the type-safe way to obtain an empty set:
3240         * <pre>
3241         *     Map&lt;String, Date&gt; s = Collections.emptyMap();
3242         * </pre>
3243         * Implementation note:  Implementations of this method need not
3244         * create a separate <tt>Map</tt> object for each call.   Using this
3245         * method is likely to have comparable cost to using the like-named
3246         * field.  (Unlike this method, the field does not provide type safety.)
3247         *
3248         * @see #EMPTY_MAP
3249         * @since 1.5
3250         */
3251        @SuppressWarnings("unchecked")
3252        public static final <K,V> Map<K,V> emptyMap() {
3253            return (Map<K,V>) EMPTY_MAP;
3254        }
3255
3256        /**
3257         * @serial include
```

```
3258         */
3259        private static class EmptyMap<K,V>
3260            extends AbstractMap<K,V>
3261            implements Serializable
3262        {
3263            private static final long serialVersionUID = 6428348081105594320L;
3264
3265            public int size()                               {return 0;}
3266            public boolean isEmpty()                        {return true;}
3267            public boolean containsKey(Object key)          {return false;}
3268            public boolean containsValue(Object value)      {return false;}
3269            public V get(Object key)                        {return null;}
3270            public Set<K> keySet()                          {return emptySet();}
3271            public Collection<V> values()                   {return emptySet();}
3272            public Set<Map.Entry<K,V>> entrySet()           {return emptySet();}
3273
3274            public boolean equals(Object o) {
3275                return (o instanceof Map) && ((Map<?,?>)o).isEmpty();
3276            }
3277
3278            public int hashCode()                           {return 0;}
3279
3280            // Preserves singleton property
3281            private Object readResolve() {
3282                return EMPTY_MAP;
3283            }
3284        }
3285
3286        // Singleton collections
3287
3288        /**
3289         * Returns an immutable set containing only the specified object.
3290         * The returned set is serializable.
3291         *
3292         * @param o the sole object to be stored in the returned set.
3293         * @return an immutable set containing only the specified object.
3294         */
3295        public static <T> Set<T> singleton(T o) {
3296            return new SingletonSet<>(o);
3297        }
3298
3299        static <E> Iterator<E> singletonIterator(final E e) {
3300            return new Iterator<E>() {
3301                private boolean hasNext = true;
3302                public boolean hasNext() {
3303                    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);
```

```
3350        }
3351
3352        /**
3353         * @serial include
3354         */
3355        private static class SingletonList<E>
3356            extends AbstractList<E>
3357            implements RandomAccess, Serializable {
3358
3359            private static final long serialVersionUID = 3093736618740652951L;
3360
3361            private final E element;
3362
3363            SingletonList(E obj)                {element = obj;}
3364
3365            public Iterator<E> iterator() {
3366                return singletonIterator(element);
3367            }
3368
3369            public int size()                   {return 1;}
3370
3371            public boolean contains(Object obj) {return eq(obj, element);}
3372
3373            public E get(int index) {
3374                if (index != 0)
3375                  throw new IndexOutOfBoundsException("Index: "+index+", Size: 1");
3376                return element;
3377            }
3378        }
3379
3380        /**
3381         * Returns an immutable map, mapping only the specified key to the
3382         * specified value.  The returned map is serializable.
3383         *
3384         * @param key the sole key to be stored in the returned map.
3385         * @param value the value to which the returned map maps <tt>key</tt>.
3386         * @return an immutable map containing only the specified key-value
3387         *         mapping.
3388         * @since 1.3
3389         */
3390        public static <K,V> Map<K,V> singletonMap(K key, V value) {
3391            return new SingletonMap<>(key, value);
3392        }
3393
3394        /**
3395         * @serial include
```

```
3396        */
3397       private static class SingletonMap<K,V>
3398             extends AbstractMap<K,V>
3399             implements Serializable {
3400           private static final long serialVersionUID = -6979724477215052911L;
3401
3402           private final K k;
3403           private final V v;
3404
3405           SingletonMap(K key, V value) {
3406               k = key;
3407               v = value;
3408           }
3409
3410           public int size()                         {return 1;}
3411
3412           public boolean isEmpty()                  {return false;}
3413
3414           public boolean containsKey(Object key)    {return eq(key, k);}
3415
3416           public boolean containsValue(Object value) {return eq(value, v);}
3417
3418           public V get(Object key)                  {return (eq(key, k) ? v : null);}
3419
3420           private transient Set<K> keySet = null;
3421           private transient Set<Map.Entry<K,V>> entrySet = null;
3422           private transient Collection<V> values = null;
3423
3424           public Set<K> keySet() {
3425               if (keySet==null)
3426                   keySet = singleton(k);
3427               return keySet;
3428           }
3429
3430           public Set<Map.Entry<K,V>> entrySet() {
3431               if (entrySet==null)
3432                   entrySet = Collections.<Map.Entry<K,V>>singleton(
3433                       new SimpleImmutableEntry<>(k, v));
3434               return entrySet;
3435           }
3436
3437           public Collection<V> values() {
3438               if (values==null)
3439                   values = singleton(v);
3440               return values;
3441           }
```

```
3442
3443        }
3444
3445        // Miscellaneous
3446
3447        /**
3448         * Returns an immutable list consisting of <tt>n</tt> copies of the
3449         * specified object.  The newly allocated data object is tiny (it contains
3450         * a single reference to the data object).  This method is useful in
3451         * combination with the <tt>List.addAll</tt> method to grow lists.
3452         * The returned list is serializable.
3453         *
3454         * @param  n the number of elements in the returned list.
3455         * @param  o the element to appear repeatedly in the returned list.
3456         * @return an immutable list consisting of <tt>n</tt> copies of the
3457         *         specified object.
3458         * @throws IllegalArgumentException if {@code n < 0}
3459         * @see    List#addAll(Collection)
3460         * @see    List#addAll(int, Collection)
3461         */
3462        public static <T> List<T> nCopies(int n, T o) {
3463            if (n < 0)
3464                throw new IllegalArgumentException("List length = " + n);
3465            return new CopiesList<>(n, o);
3466        }
3467
3468        /**
3469         * @serial include
3470         */
3471        private static class CopiesList<E>
3472            extends AbstractList<E>
3473            implements RandomAccess, Serializable
3474        {
3475            private static final long serialVersionUID = 2739099268398711800L;
3476
3477            final int n;
3478            final E element;
3479
3480            CopiesList(int n, E e) {
3481                assert n >= 0;
3482                this.n = n;
3483                element = e;
3484            }
3485
3486            public int size() {
3487                return n;
```

```
3488            }
3489
3490            public boolean contains(Object obj) {
3491                return n != 0 && eq(obj, element);
3492            }
3493
3494            public int indexOf(Object o) {
3495                return contains(o) ? 0 : -1;
3496            }
3497
3498            public int lastIndexOf(Object o) {
3499                return contains(o) ? n - 1 : -1;
3500            }
3501
3502            public E get(int index) {
3503                if (index < 0 || index >= n)
3504                    throw new IndexOutOfBoundsException("Index: "+index+
3505                                                        ", Size: "+n);
3506                return element;
3507            }
3508
3509            public Object[] toArray() {
3510                final Object[] a = new Object[n];
3511                if (element != null)
3512                    Arrays.fill(a, 0, n, element);
3513                return a;
3514            }
3515
3516            public <T> T[] toArray(T[] a) {
3517                final int n = this.n;
3518                if (a.length < n) {
3519                    a = (T[])java.lang.reflect.Array
3520                        .newInstance(a.getClass().getComponentType(), n);
3521                    if (element != null)
3522                        Arrays.fill(a, 0, n, element);
3523                } else {
3524                    Arrays.fill(a, 0, n, element);
3525                    if (a.length > n)
3526                        a[n] = null;
3527                }
3528                return a;
3529            }
3530
3531            public List<E> subList(int fromIndex, int toIndex) {
3532                if (fromIndex < 0)
3533                    throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);
```

```
            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);
        }
```

```
3580
3581            private Object readResolve() { return reverseOrder(); }
3582        }
3583
3584        /**
3585         * Returns a comparator that imposes the reverse ordering of the specified
3586         * comparator.  If the specified comparator is {@code null}, this method is
3587         * equivalent to {@link #reverseOrder()} (in other words, it returns a
3588         * comparator that imposes the reverse of the <em>natural ordering</em> on
3589         * a collection of objects that implement the Comparable interface).
3590         *
3591         * <p>The returned comparator is serializable (assuming the specified
3592         * comparator is also serializable or {@code null}).
3593         *
3594         * @param cmp a comparator who's ordering is to be reversed by the returned
3595         * comparator or {@code null}
3596         * @return A comparator that imposes the reverse ordering of the
3597         *         specified comparator.
3598         * @since 1.5
3599         */
3600        public static <T> Comparator<T> reverseOrder(Comparator<T> cmp) {
3601            if (cmp == null)
3602                return reverseOrder();
3603
3604            if (cmp instanceof ReverseComparator2)
3605                return ((ReverseComparator2<T>)cmp).cmp;
3606
3607            return new ReverseComparator2<>(cmp);
3608        }
3609
3610        /**
3611         * @serial include
3612         */
3613        private static class ReverseComparator2<T> implements Comparator<T>,
3614            Serializable
3615        {
3616            private static final long serialVersionUID = 4374092139857L;
3617
3618            /**
3619             * The comparator specified in the static factory.  This will never
3620             * be null, as the static factory returns a ReverseComparator
3621             * instance if its argument is null.
3622             *
3623             * @serial
3624             */
3625            final Comparator<T> cmp;
```

```
3626
3627            ReverseComparator2(Comparator<T> cmp) {
3628                assert cmp != null;
3629                this.cmp = cmp;
3630            }
3631
3632            public int compare(T t1, T t2) {
3633                return cmp.compare(t2, t1);
3634            }
3635
3636            public boolean equals(Object o) {
3637                return (o == this) ||
3638                    (o instanceof ReverseComparator2 &&
3639                     cmp.equals(((ReverseComparator2)o).cmp));
3640            }
3641
3642            public int hashCode() {
3643                return cmp.hashCode() ^ Integer.MIN_VALUE;
3644            }
3645        }
3646
3647        /**
3648         * Returns an enumeration over the specified collection.  This provides
3649         * interoperability with legacy APIs that require an enumeration
3650         * as input.
3651         *
3652         * @param c the collection for which an enumeration is to be returned.
3653         * @return an enumeration over the specified collection.
3654         * @see Enumeration
3655         */
3656        public static <T> Enumeration<T> enumeration(final Collection<T> c) {
3657            return new Enumeration<T>() {
3658                private final Iterator<T> i = c.iterator();
3659
3660                public boolean hasMoreElements() {
3661                    return i.hasNext();
3662                }
3663
3664                public T nextElement() {
3665                    return i.next();
3666                }
3667            };
3668        }
3669
3670        /**
3671         * Returns an array list containing the elements returned by the
```

```
3672         * specified enumeration in the order they are returned by the
3673         * enumeration.  This method provides interoperability between
3674         * legacy APIs that return enumerations and new APIs that require
3675         * collections.
3676         *
3677         * @param e enumeration providing elements for the returned
3678         *          array list
3679         * @return an array list containing the elements returned
3680         *         by the specified enumeration.
3681         * @since 1.4
3682         * @see Enumeration
3683         * @see ArrayList
3684         */
3685        public static <T> ArrayList<T> list(Enumeration<T> e) {
3686            ArrayList<T> l = new ArrayList<>();
3687            while (e.hasMoreElements())
3688                l.add(e.nextElement());
3689            return l;
3690        }
3691
3692        /**
3693         * Returns true if the specified arguments are equal, or both null.
3694         */
3695        static boolean eq(Object o1, Object o2) {
3696            return o1==null ? o2==null : o1.equals(o2);
3697        }
3698
3699        /**
3700         * Returns the number of elements in the specified collection equal to the
3701         * specified object.  More formally, returns the number of elements
3702         * <tt>e</tt> in the collection such that
3703         * <tt>(o == null ? e == null : o.equals(e))</tt>.
3704         *
3705         * @param c the collection in which to determine the frequency
3706         *     of <tt>o</tt>
3707         * @param o the object whose frequency is to be determined
3708         * @throws NullPointerException if <tt>c</tt> is null
3709         * @since 1.5
3710         */
3711        public static int frequency(Collection<?> c, Object o) {
3712            int result = 0;
3713            if (o == null) {
3714                for (Object e : c)
3715                    if (e == null)
3716                        result++;
3717            } else {
```

```
3718            for (Object e : c)
3719                if (o.equals(e))
3720                    result++;
3721        }
3722        return result;
3723    }
3724
3725    /**
3726     * Returns {@code true} if the two specified collections have no
3727     * elements in common.
3728     *
3729     * <p>Care must be exercised if this method is used on collections that
3730     * do not comply with the general contract for {@code Collection}.
3731     * Implementations may elect to iterate over either collection and test
3732     * for containment in the other collection (or to perform any equivalent
3733     * computation).  If either collection uses a nonstandard equality test
3734     * (as does a {@link SortedSet} whose ordering is not <em>compatible with
3735     * equals</em>, or the key set of an {@link IdentityHashMap}), both
3736     * collections must use the same nonstandard equality test, or the
3737     * result of this method is undefined.
3738     *
3739     * <p>Care must also be exercised when using collections that have
3740     * restrictions on the elements that they may contain. Collection
3741     * implementations are allowed to throw exceptions for any operation
3742     * involving elements they deem ineligible. For absolute safety the
3743     * specified collections should contain only elements which are
3744     * eligible elements for both collections.
3745     *
3746     * <p>Note that it is permissible to pass the same collection in both
3747     * parameters, in which case the method will return {@code true} if and
3748     * only if the collection is empty.
3749     *
3750     * @param c1 a collection
3751     * @param c2 a collection
3752     * @return {@code true} if the two specified collections have no
3753     * elements in common.
3754     * @throws NullPointerException if either collection is {@code null}.
3755     * @throws NullPointerException if one collection contains a {@code null}
3756     * element and {@code null} is not an eligible element for the other collection.
3757     * (<a href="Collection.html#optional-restrictions">optional</a>)
3758     * @throws ClassCastException if one collection contains an element that is
3759     * of a type which is ineligible for the other collection.
3760     * (<a href="Collection.html#optional-restrictions">optional</a>)
3761     * @since 1.5
3762     */
3763    public static boolean disjoint(Collection<?> c1, Collection<?> c2) {
```

```
        // 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;
            }
        }
```

```
3810
3811            // No common elements were found.
3812            return true;
3813        }
3814
3815        /**
3816         * Adds all of the specified elements to the specified collection.
3817         * Elements to be added may be specified individually or as an array.
3818         * The behavior of this convenience method is identical to that of
3819         * <tt>c.addAll(Arrays.asList(elements))</tt>, but this method is likely
3820         * to run significantly faster under most implementations.
3821         *
3822         * <p>When elements are specified individually, this method provides a
3823         * convenient way to add a few elements to an existing collection:
3824         * <pre>
3825         *     Collections.addAll(flavors, "Peaches 'n Plutonium", "Rocky Racoon");
3826         * </pre>
3827         *
3828         * @param c the collection into which <tt>elements</tt> are to be inserted
3829         * @param elements the elements to insert into <tt>c</tt>
3830         * @return <tt>true</tt> if the collection changed as a result of the call
3831         * @throws UnsupportedOperationException if <tt>c</tt> does not support
3832         *         the <tt>add</tt> operation
3833         * @throws NullPointerException if <tt>elements</tt> contains one or more
3834         *         null values and <tt>c</tt> does not permit null elements, or
3835         *         if <tt>c</tt> or <tt>elements</tt> are <tt>null</tt>
3836         * @throws IllegalArgumentException if some property of a value in
3837         *         <tt>elements</tt> prevents it from being added to <tt>c</tt>
3838         * @see Collection#addAll(Collection)
3839         * @since 1.5
3840         */
3841        @SafeVarargs
3842        public static <T> boolean addAll(Collection<? super T> c, T... elements) {
3843            boolean result = false;
3844            for (T element : elements)
3845                result |= c.add(element);
3846            return result;
3847        }
3848
3849        /**
3850         * Returns a set backed by the specified map.  The resulting set displays
3851         * the same ordering, concurrency, and performance characteristics as the
3852         * backing map.  In essence, this factory method provides a {@link Set}
3853         * implementation corresponding to any {@link Map} implementation.  There
3854         * is no need to use this method on a {@link Map} implementation that
3855         * already has a corresponding {@link Set} implementation (such as {@link
```

```
3856         * HashMap} or {@link TreeMap}).
3857         *
3858         * <p>Each method invocation on the set returned by this method results in
3859         * exactly one method invocation on the backing map or its <tt>keySet</tt>
3860         * view, with one exception.  The <tt>addAll</tt> method is implemented
3861         * as a sequence of <tt>put</tt> invocations on the backing map.
3862         *
3863         * <p>The specified map must be empty at the time this method is invoked,
3864         * and should not be accessed directly after this method returns.  These
3865         * conditions are ensured if the map is created empty, passed directly
3866         * to this method, and no reference to the map is retained, as illustrated
3867         * in the following code fragment:
3868         * <pre>
3869         *    Set&lt;Object&gt; weakHashSet = Collections.newSetFromMap(
3870         *        new WeakHashMap&lt;Object, Boolean&gt;());
3871         * </pre>
3872         *
3873         * @param map the backing map
3874         * @return the set backed by the map
3875         * @throws IllegalArgumentException if <tt>map</tt> is not empty
3876         * @since 1.6
3877         */
3878        public static <E> Set<E> newSetFromMap(Map<E, Boolean> map) {
3879            return new SetFromMap<>(map);
3880        }
3881
3882        /**
3883         * @serial include
3884         */
3885        private static class SetFromMap<E> extends AbstractSet<E>
3886            implements Set<E>, Serializable
3887        {
3888            private final Map<E, Boolean> m;  // The backing map
3889            private transient Set<E> s;       // Its keySet
3890
3891            SetFromMap(Map<E, Boolean> map) {
3892                if (!map.isEmpty())
3893                    throw new IllegalArgumentException("Map is non-empty");
3894                m = map;
3895                s = map.keySet();
3896            }
3897
3898            public void clear()               {        m.clear(); }
3899            public int size()                 { return m.size(); }
3900            public boolean isEmpty()          { return m.isEmpty(); }
3901            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
```

```
3948         */
3949        static class AsLIFOQueue<E> extends AbstractQueue<E>
3950            implements Queue<E>, Serializable {
3951            private static final long serialVersionUID = 1802017725587941708L;
3952            private final Deque<E> q;
3953            AsLIFOQueue(Deque<E> q)           { this.q = q; }
3954            public boolean add(E e)           { q.addFirst(e); return true; }
3955            public boolean offer(E e)         { return q.offerFirst(e); }
3956            public E poll()                   { return q.pollFirst(); }
3957            public E remove()                 { return q.removeFirst(); }
3958            public E peek()                   { return q.peekFirst(); }
3959            public E element()                { return q.getFirst(); }
3960            public void clear()               {        q.clear(); }
3961            public int size()                 { return q.size(); }
3962            public boolean isEmpty()          { return q.isEmpty(); }
3963            public boolean contains(Object o) { return q.contains(o); }
3964            public boolean remove(Object o)   { return q.remove(o); }
3965            public Iterator<E> iterator()     { return q.iterator(); }
3966            public Object[] toArray()         { return q.toArray(); }
3967            public <T> T[] toArray(T[] a)     { return q.toArray(a); }
3968            public String toString()          { return q.toString(); }
3969            public boolean containsAll(Collection<?> c) {return q.containsAll(c);}
3970            public boolean removeAll(Collection<?> c)   {return q.removeAll(c);}
3971            public boolean retainAll(Collection<?> c)   {return q.retainAll(c);}
3972            // We use inherited addAll; forwarding addAll would be wrong
3973        }
3974    }
```