001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.collections4;  
018  
019import java.lang.reflect.Array;  
020import java.util.ArrayList;  
021import java.util.Collection;  
022import java.util.Collections;  
023import java.util.Comparator;  
024import java.util.Enumeration;  
025import java.util.HashMap;  
026import java.util.HashSet;  
027import java.util.Iterator;  
028import java.util.List;  
029import java.util.ListIterator;  
030import java.util.Map;  
031import java.util.Set;  
032  
033import org.apache.commons.collections4.bag.HashBag;  
034import org.apache.commons.collections4.collection.PredicatedCollection;  
035import org.apache.commons.collections4.collection.SynchronizedCollection;  
036import org.apache.commons.collections4.collection.TransformedCollection;  
037import org.apache.commons.collections4.collection.UnmodifiableBoundedCollection;  
038import org.apache.commons.collections4.collection.UnmodifiableCollection;  
039import org.apache.commons.collections4.functors.TruePredicate;  
040import org.apache.commons.collections4.iterators.CollatingIterator;  
041import org.apache.commons.collections4.iterators.PermutationIterator;  
042  
043/\*\*  
044 \* Provides utility methods and decorators for {@link Collection} instances.  
045 \* <p>  
046 \* Various utility methods might put the input objects into a Set/Map/Bag. In case  
047 \* the input objects override {@link Object#equals(Object)}, it is mandatory that  
048 \* the general contract of the {@link Object#hashCode()} method is maintained.  
049 \* </p>  
050 \* <p>  
051 \* NOTE: From 4.0, method parameters will take {@link Iterable} objects when possible.  
052 \* </p>  
053 \*  
054 \* @since 1.0  
055 \*/  
056public class CollectionUtils {  
057  
058 /\*\*  
059 \* Helper class to easily access cardinality properties of two collections.  
060 \* @param <O> the element type  
061 \*/  
062 private static class CardinalityHelper<O> {  
063  
064 /\*\* Contains the cardinality for each object in collection A. \*/  
065 final Map<O, Integer> cardinalityA;  
066  
067 /\*\* Contains the cardinality for each object in collection B. \*/  
068 final Map<O, Integer> cardinalityB;  
069  
070 /\*\*  
071 \* Create a new CardinalityHelper for two collections.  
072 \* @param a the first collection  
073 \* @param b the second collection  
074 \*/  
075 public CardinalityHelper(final Iterable<? extends O> a, final Iterable<? extends O> b) {  
076 cardinalityA = CollectionUtils.<O>getCardinalityMap(a);  
077 cardinalityB = CollectionUtils.<O>getCardinalityMap(b);  
078 }  
079  
080 /\*\*  
081 \* Returns the maximum frequency of an object.  
082 \* @param obj the object  
083 \* @return the maximum frequency of the object  
084 \*/  
085 public final int max(final Object obj) {  
086 return Math.max(freqA(obj), freqB(obj));  
087 }  
088  
089 /\*\*  
090 \* Returns the minimum frequency of an object.  
091 \* @param obj the object  
092 \* @return the minimum frequency of the object  
093 \*/  
094 public final int min(final Object obj) {  
095 return Math.min(freqA(obj), freqB(obj));  
096 }  
097  
098 /\*\*  
099 \* Returns the frequency of this object in collection A.  
100 \* @param obj the object  
101 \* @return the frequency of the object in collection A  
102 \*/  
103 public int freqA(final Object obj) {  
104 return getFreq(obj, cardinalityA);  
105 }  
106  
107 /\*\*  
108 \* Returns the frequency of this object in collection B.  
109 \* @param obj the object  
110 \* @return the frequency of the object in collection B  
111 \*/  
112 public int freqB(final Object obj) {  
113 return getFreq(obj, cardinalityB);  
114 }  
115  
116 private int getFreq(final Object obj, final Map<?, Integer> freqMap) {  
117 final Integer count = freqMap.get(obj);  
118 if (count != null) {  
119 return count.intValue();  
120 }  
121 return 0;  
122 }  
123 }  
124  
125 /\*\*  
126 \* Helper class for set-related operations, e.g. union, subtract, intersection.  
127 \* @param <O> the element type  
128 \*/  
129 private static class SetOperationCardinalityHelper<O> extends CardinalityHelper<O> implements Iterable<O> {  
130  
131 /\*\* Contains the unique elements of the two collections. \*/  
132 private final Set<O> elements;  
133  
134 /\*\* Output collection. \*/  
135 private final List<O> newList;  
136  
137 /\*\*  
138 \* Create a new set operation helper from the two collections.  
139 \* @param a the first collection  
140 \* @param b the second collection  
141 \*/  
142 public SetOperationCardinalityHelper(final Iterable<? extends O> a, final Iterable<? extends O> b) {  
143 super(a, b);  
144 elements = new HashSet<>();  
145 addAll(elements, a);  
146 addAll(elements, b);  
147 // the resulting list must contain at least each unique element, but may grow  
148 newList = new ArrayList<>(elements.size());  
149 }  
150  
151 @Override  
152 public Iterator<O> iterator() {  
153 return elements.iterator();  
154 }  
155  
156 /\*\*  
157 \* Add the object {@code count} times to the result collection.  
158 \* @param obj the object to add  
159 \* @param count the count  
160 \*/  
161 public void setCardinality(final O obj, final int count) {  
162 for (int i = 0; i < count; i++) {  
163 newList.add(obj);  
164 }  
165 }  
166  
167 /\*\*  
168 \* Returns the resulting collection.  
169 \* @return the result  
170 \*/  
171 public Collection<O> list() {  
172 return newList;  
173 }  
174  
175 }  
176  
177 /\*\*  
178 \* An empty unmodifiable collection.  
179 \* The JDK provides empty Set and List implementations which could be used for  
180 \* this purpose. However they could be cast to Set or List which might be  
181 \* undesirable. This implementation only implements Collection.  
182 \*/  
183 @SuppressWarnings("rawtypes") // we deliberately use the raw type here  
184 public static final Collection EMPTY\_COLLECTION = Collections.emptyList();  
185  
186 /\*\*  
187 \* <code>CollectionUtils</code> should not normally be instantiated.  
188 \*/  
189 private CollectionUtils() {}  
190  
191 /\*\*  
192 \* Returns the immutable EMPTY\_COLLECTION with generic type safety.  
193 \*  
194 \* @see #EMPTY\_COLLECTION  
195 \* @since 4.0  
196 \* @param <T> the element type  
197 \* @return immutable empty collection  
198 \*/  
199 @SuppressWarnings("unchecked") // OK, empty collection is compatible with any type  
200 public static <T> Collection<T> emptyCollection() {  
201 return EMPTY\_COLLECTION;  
202 }  
203  
204 /\*\*  
205 \* Returns an immutable empty collection if the argument is <code>null</code>,  
206 \* or the argument itself otherwise.  
207 \*  
208 \* @param <T> the element type  
209 \* @param collection the collection, possibly <code>null</code>  
210 \* @return an empty collection if the argument is <code>null</code>  
211 \*/  
212 public static <T> Collection<T> emptyIfNull(final Collection<T> collection) {  
213 return collection == null ? CollectionUtils.<T>emptyCollection() : collection;  
214 }  
215  
216 /\*\*  
217 \* Returns a {@link Collection} containing the union of the given  
218 \* {@link Iterable}s.  
219 \* <p>  
220 \* The cardinality of each element in the returned {@link Collection} will  
221 \* be equal to the maximum of the cardinality of that element in the two  
222 \* given {@link Iterable}s.  
223 \* </p>  
224 \*  
225 \* @param a the first collection, must not be null  
226 \* @param b the second collection, must not be null  
227 \* @param <O> the generic type that is able to represent the types contained  
228 \* in both input collections.  
229 \* @return the union of the two collections  
230 \* @see Collection#addAll  
231 \*/  
232 public static <O> Collection<O> union(final Iterable<? extends O> a, final Iterable<? extends O> b) {  
233 final SetOperationCardinalityHelper<O> helper = new SetOperationCardinalityHelper<>(a, b);  
234 for (final O obj : helper) {  
235 helper.setCardinality(obj, helper.max(obj));  
236 }  
237 return helper.list();  
238 }  
239  
240 /\*\*  
241 \* Returns a {@link Collection} containing the intersection of the given  
242 \* {@link Iterable}s.  
243 \* <p>  
244 \* The cardinality of each element in the returned {@link Collection} will  
245 \* be equal to the minimum of the cardinality of that element in the two  
246 \* given {@link Iterable}s.  
247 \* </p>  
248 \*  
249 \* @param a the first collection, must not be null  
250 \* @param b the second collection, must not be null  
251 \* @param <O> the generic type that is able to represent the types contained  
252 \* in both input collections.  
253 \* @return the intersection of the two collections  
254 \* @see Collection#retainAll  
255 \* @see #containsAny  
256 \*/  
257 public static <O> Collection<O> intersection(final Iterable<? extends O> a, final Iterable<? extends O> b) {  
258 final SetOperationCardinalityHelper<O> helper = new SetOperationCardinalityHelper<>(a, b);  
259 for (final O obj : helper) {  
260 helper.setCardinality(obj, helper.min(obj));  
261 }  
262 return helper.list();  
263 }  
264  
265 /\*\*  
266 \* Returns a {@link Collection} containing the exclusive disjunction  
267 \* (symmetric difference) of the given {@link Iterable}s.  
268 \* <p>  
269 \* The cardinality of each element <i>e</i> in the returned  
270 \* {@link Collection} will be equal to  
271 \* <code>max(cardinality(<i>e</i>,<i>a</i>),cardinality(<i>e</i>,<i>b</i>)) - min(cardinality(<i>e</i>,<i>a</i>),  
272 \* cardinality(<i>e</i>,<i>b</i>))</code>.  
273 \* </p>  
274 \* <p>  
275 \* This is equivalent to  
276 \* {@code {@link #subtract subtract}({@link #union union(a,b)},{@link #intersection intersection(a,b)})}  
277 \* or  
278 \* {@code {@link #union union}({@link #subtract subtract(a,b)},{@link #subtract subtract(b,a)})}.  
279 \* </p>  
280 \*  
281 \* @param a the first collection, must not be null  
282 \* @param b the second collection, must not be null  
283 \* @param <O> the generic type that is able to represent the types contained  
284 \* in both input collections.  
285 \* @return the symmetric difference of the two collections  
286 \*/  
287 public static <O> Collection<O> disjunction(final Iterable<? extends O> a, final Iterable<? extends O> b) {  
288 final SetOperationCardinalityHelper<O> helper = new SetOperationCardinalityHelper<>(a, b);  
289 for (final O obj : helper) {  
290 helper.setCardinality(obj, helper.max(obj) - helper.min(obj));  
291 }  
292 return helper.list();  
293 }  
294  
295 /\*\*  
296 \* Returns a new {@link Collection} containing {@code <i>a</i> - <i>b</i>}.  
297 \* The cardinality of each element <i>e</i> in the returned {@link Collection}  
298 \* will be the cardinality of <i>e</i> in <i>a</i> minus the cardinality  
299 \* of <i>e</i> in <i>b</i>, or zero, whichever is greater.  
300 \*  
301 \* @param a the collection to subtract from, must not be null  
302 \* @param b the collection to subtract, must not be null  
303 \* @param <O> the generic type that is able to represent the types contained  
304 \* in both input collections.  
305 \* @return a new collection with the results  
306 \* @see Collection#removeAll  
307 \*/  
308 public static <O> Collection<O> subtract(final Iterable<? extends O> a, final Iterable<? extends O> b) {  
309 final Predicate<O> p = TruePredicate.truePredicate();  
310 return subtract(a, b, p);  
311 }  
312  
313 /\*\*  
314 \* Returns a new {@link Collection} containing <i>a</i> minus a subset of  
315 \* <i>b</i>. Only the elements of <i>b</i> that satisfy the predicate  
316 \* condition, <i>p</i> are subtracted from <i>a</i>.  
317 \*  
318 \* <p>  
319 \* The cardinality of each element <i>e</i> in the returned {@link Collection}  
320 \* that satisfies the predicate condition will be the cardinality of <i>e</i> in <i>a</i>  
321 \* minus the cardinality of <i>e</i> in <i>b</i>, or zero, whichever is greater.  
322 \* </p>  
323 \* <p>  
324 \* The cardinality of each element <i>e</i> in the returned {@link Collection} that does <b>not</b>  
325 \* satisfy the predicate condition will be equal to the cardinality of <i>e</i> in <i>a</i>.  
326 \* </p>  
327 \*  
328 \* @param a the collection to subtract from, must not be null  
329 \* @param b the collection to subtract, must not be null  
330 \* @param p the condition used to determine which elements of <i>b</i> are  
331 \* subtracted.  
332 \* @param <O> the generic type that is able to represent the types contained  
333 \* in both input collections.  
334 \* @return a new collection with the results  
335 \* @since 4.0  
336 \* @see Collection#removeAll  
337 \*/  
338 public static <O> Collection<O> subtract(final Iterable<? extends O> a,  
339 final Iterable<? extends O> b,  
340 final Predicate<O> p) {  
341 final ArrayList<O> list = new ArrayList<>();  
342 final HashBag<O> bag = new HashBag<>();  
343 for (final O element : b) {  
344 if (p.evaluate(element)) {  
345 bag.add(element);  
346 }  
347 }  
348 for (final O element : a) {  
349 if (!bag.remove(element, 1)) {  
350 list.add(element);  
351 }  
352 }  
353 return list;  
354 }  
355  
356 /\*\*  
357 \* Returns <code>true</code> iff all elements of {@code coll2} are also contained  
358 \* in {@code coll1}. The cardinality of values in {@code coll2} is not taken into account,  
359 \* which is the same behavior as {@link Collection#containsAll(Collection)}.  
360 \* <p>  
361 \* In other words, this method returns <code>true</code> iff the  
362 \* {@link #intersection} of <i>coll1</i> and <i>coll2</i> has the same cardinality as  
363 \* the set of unique values from {@code coll2}. In case {@code coll2} is empty, {@code true}  
364 \* will be returned.  
365 \* </p>  
366 \* <p>  
367 \* This method is intended as a replacement for {@link Collection#containsAll(Collection)}  
368 \* with a guaranteed runtime complexity of {@code O(n + m)}. Depending on the type of  
369 \* {@link Collection} provided, this method will be much faster than calling  
370 \* {@link Collection#containsAll(Collection)} instead, though this will come at the  
371 \* cost of an additional space complexity O(n).  
372 \* </p>  
373 \*  
374 \* @param coll1 the first collection, must not be null  
375 \* @param coll2 the second collection, must not be null  
376 \* @return <code>true</code> iff the intersection of the collections has the same cardinality  
377 \* as the set of unique elements from the second collection  
378 \* @since 4.0  
379 \*/  
380 public static boolean containsAll(final Collection<?> coll1, final Collection<?> coll2) {  
381 if (coll2.isEmpty()) {  
382 return true;  
383 }  
384 final Iterator<?> it = coll1.iterator();  
385 final Set<Object> elementsAlreadySeen = new HashSet<>();  
386 for (final Object nextElement : coll2) {  
387 if (elementsAlreadySeen.contains(nextElement)) {  
388 continue;  
389 }  
390  
391 boolean foundCurrentElement = false;  
392 while (it.hasNext()) {  
393 final Object p = it.next();  
394 elementsAlreadySeen.add(p);  
395 if (nextElement == null ? p == null : nextElement.equals(p)) {  
396 foundCurrentElement = true;  
397 break;  
398 }  
399 }  
400  
401 if (!foundCurrentElement) {  
402 return false;  
403 }  
404 }  
405 return true;  
406 }  
407  
408 /\*\*  
409 \* Returns <code>true</code> iff at least one element is in both collections.  
410 \* <p>  
411 \* In other words, this method returns <code>true</code> iff the  
412 \* {@link #intersection} of <i>coll1</i> and <i>coll2</i> is not empty.  
413 \* </p>  
414 \*  
415 \* @param <T> the type of object to lookup in <code>coll1</code>.  
416 \* @param coll1 the first collection, must not be null  
417 \* @param coll2 the second collection, must not be null  
418 \* @return <code>true</code> iff the intersection of the collections is non-empty  
419 \* @since 4.2  
420 \* @see #intersection  
421 \*/  
422 public static <T> boolean containsAny(final Collection<?> coll1, @SuppressWarnings("unchecked") final T... coll2) {  
423 if (coll1.size() < coll2.length) {  
424 for (final Object aColl1 : coll1) {  
425 if (ArrayUtils.contains(coll2, aColl1)) {  
426 return true;  
427 }  
428 }  
429 } else {  
430 for (final Object aColl2 : coll2) {  
431 if (coll1.contains(aColl2)) {  
432 return true;  
433 }  
434 }  
435 }  
436 return false;  
437 }  
438  
439 /\*\*  
440 \* Returns <code>true</code> iff at least one element is in both collections.  
441 \* <p>  
442 \* In other words, this method returns <code>true</code> iff the  
443 \* {@link #intersection} of <i>coll1</i> and <i>coll2</i> is not empty.  
444 \* </p>  
445 \*  
446 \* @param coll1 the first collection, must not be null  
447 \* @param coll2 the second collection, must not be null  
448 \* @return <code>true</code> iff the intersection of the collections is non-empty  
449 \* @since 2.1  
450 \* @see #intersection  
451 \*/  
452 public static boolean containsAny(final Collection<?> coll1, final Collection<?> coll2) {  
453 if (coll1.size() < coll2.size()) {  
454 for (final Object aColl1 : coll1) {  
455 if (coll2.contains(aColl1)) {  
456 return true;  
457 }  
458 }  
459 } else {  
460 for (final Object aColl2 : coll2) {  
461 if (coll1.contains(aColl2)) {  
462 return true;  
463 }  
464 }  
465 }  
466 return false;  
467 }  
468  
469 /\*\*  
470 \* Returns a {@link Map} mapping each unique element in the given  
471 \* {@link Collection} to an {@link Integer} representing the number  
472 \* of occurrences of that element in the {@link Collection}.  
473 \* <p>  
474 \* Only those elements present in the collection will appear as  
475 \* keys in the map.  
476 \* </p>  
477 \*  
478 \* @param <O> the type of object in the returned {@link Map}. This is a super type of <I>.  
479 \* @param coll the collection to get the cardinality map for, must not be null  
480 \* @return the populated cardinality map  
481 \*/  
482 public static <O> Map<O, Integer> getCardinalityMap(final Iterable<? extends O> coll) {  
483 final Map<O, Integer> count = new HashMap<>();  
484 for (final O obj : coll) {  
485 final Integer c = count.get(obj);  
486 if (c == null) {  
487 count.put(obj, Integer.valueOf(1));  
488 } else {  
489 count.put(obj, Integer.valueOf(c.intValue() + 1));  
490 }  
491 }  
492 return count;  
493 }  
494  
495 /\*\*  
496 \* Returns {@code true} iff <i>a</i> is a sub-collection of <i>b</i>,  
497 \* that is, iff the cardinality of <i>e</i> in <i>a</i> is less than or  
498 \* equal to the cardinality of <i>e</i> in <i>b</i>, for each element <i>e</i>  
499 \* in <i>a</i>.  
500 \*  
501 \* @param a the first (sub?) collection, must not be null  
502 \* @param b the second (super?) collection, must not be null  
503 \* @return <code>true</code> iff <i>a</i> is a sub-collection of <i>b</i>  
504 \* @see #isProperSubCollection  
505 \* @see Collection#containsAll  
506 \*/  
507 public static boolean isSubCollection(final Collection<?> a, final Collection<?> b) {  
508 final CardinalityHelper<Object> helper = new CardinalityHelper<>(a, b);  
509 for (final Object obj : a) {  
510 if (helper.freqA(obj) > helper.freqB(obj)) {  
511 return false;  
512 }  
513 }  
514 return true;  
515 }  
516  
517 /\*\*  
518 \* Returns {@code true} iff <i>a</i> is a <i>proper</i> sub-collection of <i>b</i>,  
519 \* that is, iff the cardinality of <i>e</i> in <i>a</i> is less  
520 \* than or equal to the cardinality of <i>e</i> in <i>b</i>,  
521 \* for each element <i>e</i> in <i>a</i>, and there is at least one  
522 \* element <i>f</i> such that the cardinality of <i>f</i> in <i>b</i>  
523 \* is strictly greater than the cardinality of <i>f</i> in <i>a</i>.  
524 \* <p>  
525 \* The implementation assumes  
526 \* </p>  
527 \* <ul>  
528 \* <li><code>a.size()</code> and <code>b.size()</code> represent the  
529 \* total cardinality of <i>a</i> and <i>b</i>, resp. </li>  
530 \* <li><code>a.size() < Integer.MAXVALUE</code></li>  
531 \* </ul>  
532 \*  
533 \* @param a the first (sub?) collection, must not be null  
534 \* @param b the second (super?) collection, must not be null  
535 \* @return <code>true</code> iff <i>a</i> is a <i>proper</i> sub-collection of <i>b</i>  
536 \* @see #isSubCollection  
537 \* @see Collection#containsAll  
538 \*/  
539 public static boolean isProperSubCollection(final Collection<?> a, final Collection<?> b) {  
540 return a.size() < b.size() && CollectionUtils.isSubCollection(a, b);  
541 }  
542  
543 /\*\*  
544 \* Returns {@code true} iff the given {@link Collection}s contain  
545 \* exactly the same elements with exactly the same cardinalities.  
546 \* <p>  
547 \* That is, iff the cardinality of <i>e</i> in <i>a</i> is  
548 \* equal to the cardinality of <i>e</i> in <i>b</i>,  
549 \* for each element <i>e</i> in <i>a</i> or <i>b</i>.  
550 \* </p>  
551 \*  
552 \* @param a the first collection, must not be null  
553 \* @param b the second collection, must not be null  
554 \* @return <code>true</code> iff the collections contain the same elements with the same cardinalities.  
555 \*/  
556 public static boolean isEqualCollection(final Collection<?> a, final Collection<?> b) {  
557 if(a.size() != b.size()) {  
558 return false;  
559 }  
560 final CardinalityHelper<Object> helper = new CardinalityHelper<>(a, b);  
561 if(helper.cardinalityA.size() != helper.cardinalityB.size()) {  
562 return false;  
563 }  
564 for( final Object obj : helper.cardinalityA.keySet()) {  
565 if(helper.freqA(obj) != helper.freqB(obj)) {  
566 return false;  
567 }  
568 }  
569 return true;  
570 }  
571  
572 /\*\*  
573 \* Returns {@code true} iff the given {@link Collection}s contain  
574 \* exactly the same elements with exactly the same cardinalities.  
575 \* <p>  
576 \* That is, iff the cardinality of <i>e</i> in <i>a</i> is  
577 \* equal to the cardinality of <i>e</i> in <i>b</i>,  
578 \* for each element <i>e</i> in <i>a</i> or <i>b</i>.  
579 \* </p>  
580 \* <p>  
581 \* <b>Note:</b> from version 4.1 onwards this method requires the input  
582 \* collections and equator to be of compatible type (using bounded wildcards).  
583 \* Providing incompatible arguments (e.g. by casting to their rawtypes)  
584 \* will result in a {@code ClassCastException} thrown at runtime.  
585 \* </p>  
586 \*  
587 \* @param <E> the element type  
588 \* @param a the first collection, must not be null  
589 \* @param b the second collection, must not be null  
590 \* @param equator the Equator used for testing equality  
591 \* @return <code>true</code> iff the collections contain the same elements with the same cardinalities.  
592 \* @throws NullPointerException if the equator is null  
593 \* @since 4.0  
594 \*/  
595 public static <E> boolean isEqualCollection(final Collection<? extends E> a,  
596 final Collection<? extends E> b,  
597 final Equator<? super E> equator) {  
598 if (equator == null) {  
599 throw new NullPointerException("Equator must not be null.");  
600 }  
601  
602 if(a.size() != b.size()) {  
603 return false;  
604 }  
605  
606 @SuppressWarnings({ "unchecked", "rawtypes" })  
607 final Transformer<E, ?> transformer = new Transformer() {  
608 @Override  
609 public EquatorWrapper<?> transform(final Object input) {  
610 return new EquatorWrapper(equator, input);  
611 }  
612 };  
613  
614 return isEqualCollection(collect(a, transformer), collect(b, transformer));  
615 }  
616  
617 /\*\*  
618 \* Wraps another object and uses the provided Equator to implement  
619 \* {@link #equals(Object)} and {@link #hashCode()}.  
620 \* <p>  
621 \* This class can be used to store objects into a Map.  
622 \* </p>  
623 \*  
624 \* @param <O> the element type  
625 \* @since 4.0  
626 \*/  
627 private static class EquatorWrapper<O> {  
628 private final Equator<? super O> equator;  
629 private final O object;  
630  
631 public EquatorWrapper(final Equator<? super O> equator, final O object) {  
632 this.equator = equator;  
633 this.object = object;  
634 }  
635  
636 public O getObject() {  
637 return object;  
638 }  
639  
640 @Override  
641 public boolean equals(final Object obj) {  
642 if (!(obj instanceof EquatorWrapper)) {  
643 return false;  
644 }  
645 @SuppressWarnings("unchecked")  
646 final EquatorWrapper<O> otherObj = (EquatorWrapper<O>) obj;  
647 return equator.equate(object, otherObj.getObject());  
648 }  
649  
650 @Override  
651 public int hashCode() {  
652 return equator.hash(object);  
653 }  
654 }  
655  
656 /\*\*  
657 \* Returns the number of occurrences of <i>obj</i> in <i>coll</i>.  
658 \*  
659 \* @param obj the object to find the cardinality of  
660 \* @param coll the {@link Iterable} to search  
661 \* @param <O> the type of object that the {@link Iterable} may contain.  
662 \* @return the number of occurrences of obj in coll  
663 \* @throws NullPointerException if coll is null  
664 \* @deprecated since 4.1, use {@link IterableUtils#frequency(Iterable, Object)} instead.  
665 \* Be aware that the order of parameters has changed.  
666 \*/  
667 @Deprecated  
668 public static <O> int cardinality(final O obj, final Iterable<? super O> coll) {  
669 if (coll == null) {  
670 throw new NullPointerException("coll must not be null.");  
671 }  
672 return IterableUtils.frequency(coll, obj);  
673 }  
674  
675 /\*\*  
676 \* Finds the first element in the given collection which matches the given predicate.  
677 \* <p>  
678 \* If the input collection or predicate is null, or no element of the collection  
679 \* matches the predicate, null is returned.  
680 \* </p>  
681 \*  
682 \* @param <T> the type of object the {@link Iterable} contains  
683 \* @param collection the collection to search, may be null  
684 \* @param predicate the predicate to use, may be null  
685 \* @return the first element of the collection which matches the predicate or null if none could be found  
686 \* @deprecated since 4.1, use {@link IterableUtils#find(Iterable, Predicate)} instead  
687 \*/  
688 @Deprecated  
689 public static <T> T find(final Iterable<T> collection, final Predicate<? super T> predicate) {  
690 return predicate != null ? IterableUtils.find(collection, predicate) : null;  
691 }  
692  
693 /\*\*  
694 \* Executes the given closure on each element in the collection.  
695 \* <p>  
696 \* If the input collection or closure is null, there is no change made.  
697 \* </p>  
698 \*  
699 \* @param <T> the type of object the {@link Iterable} contains  
700 \* @param <C> the closure type  
701 \* @param collection the collection to get the input from, may be null  
702 \* @param closure the closure to perform, may be null  
703 \* @return closure  
704 \* @deprecated since 4.1, use {@link IterableUtils#forEach(Iterable, Closure)} instead  
705 \*/  
706 @Deprecated  
707 public static <T, C extends Closure<? super T>> C forAllDo(final Iterable<T> collection, final C closure) {  
708 if (closure != null) {  
709 IterableUtils.forEach(collection, closure);  
710 }  
711 return closure;  
712 }  
713  
714 /\*\*  
715 \* Executes the given closure on each element in the collection.  
716 \* <p>  
717 \* If the input collection or closure is null, there is no change made.  
718 \* </p>  
719 \*  
720 \* @param <T> the type of object the {@link Iterator} contains  
721 \* @param <C> the closure type  
722 \* @param iterator the iterator to get the input from, may be null  
723 \* @param closure the closure to perform, may be null  
724 \* @return closure  
725 \* @since 4.0  
726 \* @deprecated since 4.1, use {@link IteratorUtils#forEach(Iterator, Closure)} instead  
727 \*/  
728 @Deprecated  
729 public static <T, C extends Closure<? super T>> C forAllDo(final Iterator<T> iterator, final C closure) {  
730 if (closure != null) {  
731 IteratorUtils.forEach(iterator, closure);  
732 }  
733 return closure;  
734 }  
735  
736 /\*\*  
737 \* Executes the given closure on each but the last element in the collection.  
738 \* <p>  
739 \* If the input collection or closure is null, there is no change made.  
740 \* </p>  
741 \*  
742 \* @param <T> the type of object the {@link Iterable} contains  
743 \* @param <C> the closure type  
744 \* @param collection the collection to get the input from, may be null  
745 \* @param closure the closure to perform, may be null  
746 \* @return the last element in the collection, or null if either collection or closure is null  
747 \* @since 4.0  
748 \* @deprecated since 4.1, use {@link IterableUtils#forEachButLast(Iterable, Closure)} instead  
749 \*/  
750 @Deprecated  
751 public static <T, C extends Closure<? super T>> T forAllButLastDo(final Iterable<T> collection,  
752 final C closure) {  
753 return closure != null ? IterableUtils.forEachButLast(collection, closure) : null;  
754 }  
755  
756 /\*\*  
757 \* Executes the given closure on each but the last element in the collection.  
758 \* <p>  
759 \* If the input collection or closure is null, there is no change made.  
760 \* </p>  
761 \*  
762 \* @param <T> the type of object the {@link Collection} contains  
763 \* @param <C> the closure type  
764 \* @param iterator the iterator to get the input from, may be null  
765 \* @param closure the closure to perform, may be null  
766 \* @return the last element in the collection, or null if either iterator or closure is null  
767 \* @since 4.0  
768 \* @deprecated since 4.1, use {@link IteratorUtils#forEachButLast(Iterator, Closure)} instead  
769 \*/  
770 @Deprecated  
771 public static <T, C extends Closure<? super T>> T forAllButLastDo(final Iterator<T> iterator, final C closure) {  
772 return closure != null ? IteratorUtils.forEachButLast(iterator, closure) : null;  
773 }  
774  
775 /\*\*  
776 \* Filter the collection by applying a Predicate to each element. If the  
777 \* predicate returns false, remove the element.  
778 \* <p>  
779 \* If the input collection or predicate is null, there is no change made.  
780 \* </p>  
781 \*  
782 \* @param <T> the type of object the {@link Iterable} contains  
783 \* @param collection the collection to get the input from, may be null  
784 \* @param predicate the predicate to use as a filter, may be null  
785 \* @return true if the collection is modified by this call, false otherwise.  
786 \*/  
787 public static <T> boolean filter(final Iterable<T> collection, final Predicate<? super T> predicate) {  
788 boolean result = false;  
789 if (collection != null && predicate != null) {  
790 for (final Iterator<T> it = collection.iterator(); it.hasNext();) {  
791 if (!predicate.evaluate(it.next())) {  
792 it.remove();  
793 result = true;  
794 }  
795 }  
796 }  
797 return result;  
798 }  
799  
800 /\*\*  
801 \* Filter the collection by applying a Predicate to each element. If the  
802 \* predicate returns true, remove the element.  
803 \* <p>  
804 \* This is equivalent to <code>filter(collection, PredicateUtils.notPredicate(predicate))</code>  
805 \* if predicate is != null.  
806 \* </p>  
807 \* <p>  
808 \* If the input collection or predicate is null, there is no change made.  
809 \* </p>  
810 \*  
811 \* @param <T> the type of object the {@link Iterable} contains  
812 \* @param collection the collection to get the input from, may be null  
813 \* @param predicate the predicate to use as a filter, may be null  
814 \* @return true if the collection is modified by this call, false otherwise.  
815 \*/  
816 public static <T> boolean filterInverse(final Iterable<T> collection, final Predicate<? super T> predicate) {  
817 return filter(collection, predicate == null ? null : PredicateUtils.notPredicate(predicate));  
818 }  
819  
820 /\*\*  
821 \* Transform the collection by applying a Transformer to each element.  
822 \* <p>  
823 \* If the input collection or transformer is null, there is no change made.  
824 \* </p>  
825 \* <p>  
826 \* This routine is best for Lists, for which set() is used to do the  
827 \* transformations "in place." For other Collections, clear() and addAll()  
828 \* are used to replace elements.  
829 \* </p>  
830 \* <p>  
831 \* If the input collection controls its input, such as a Set, and the  
832 \* Transformer creates duplicates (or are otherwise invalid), the collection  
833 \* may reduce in size due to calling this method.  
834 \* </p>  
835 \*  
836 \* @param <C> the type of object the {@link Collection} contains  
837 \* @param collection the {@link Collection} to get the input from, may be null  
838 \* @param transformer the transformer to perform, may be null  
839 \*/  
840 public static <C> void transform(final Collection<C> collection,  
841 final Transformer<? super C, ? extends C> transformer) {  
842  
843 if (collection != null && transformer != null) {  
844 if (collection instanceof List<?>) {  
845 final List<C> list = (List<C>) collection;  
846 for (final ListIterator<C> it = list.listIterator(); it.hasNext();) {  
847 it.set(transformer.transform(it.next()));  
848 }  
849 } else {  
850 final Collection<C> resultCollection = collect(collection, transformer);  
851 collection.clear();  
852 collection.addAll(resultCollection);  
853 }  
854 }  
855 }  
856  
857 /\*\*  
858 \* Counts the number of elements in the input collection that match the  
859 \* predicate.  
860 \* <p>  
861 \* A <code>null</code> collection or predicate matches no elements.  
862 \* </p>  
863 \*  
864 \* @param <C> the type of object the {@link Iterable} contains  
865 \* @param input the {@link Iterable} to get the input from, may be null  
866 \* @param predicate the predicate to use, may be null  
867 \* @return the number of matches for the predicate in the collection  
868 \* @deprecated since 4.1, use {@link IterableUtils#countMatches(Iterable, Predicate)} instead  
869 \*/  
870 @Deprecated  
871 public static <C> int countMatches(final Iterable<C> input, final Predicate<? super C> predicate) {  
872 return predicate == null ? 0 : (int) IterableUtils.countMatches(input, predicate);  
873 }  
874  
875 /\*\*  
876 \* Answers true if a predicate is true for at least one element of a  
877 \* collection.  
878 \* <p>  
879 \* A <code>null</code> collection or predicate returns false.  
880 \* </p>  
881 \*  
882 \* @param <C> the type of object the {@link Iterable} contains  
883 \* @param input the {@link Iterable} to get the input from, may be null  
884 \* @param predicate the predicate to use, may be null  
885 \* @return true if at least one element of the collection matches the predicate  
886 \* @deprecated since 4.1, use {@link IterableUtils#matchesAny(Iterable, Predicate)} instead  
887 \*/  
888 @Deprecated  
889 public static <C> boolean exists(final Iterable<C> input, final Predicate<? super C> predicate) {  
890 return predicate != null && IterableUtils.matchesAny(input, predicate);  
891 }  
892  
893 /\*\*  
894 \* Answers true if a predicate is true for every element of a  
895 \* collection.  
896 \*  
897 \* <p>  
898 \* A <code>null</code> predicate returns false.  
899 \* </p>  
900 \* <p>  
901 \* A <code>null</code> or empty collection returns true.  
902 \* </p>  
903 \*  
904 \* @param <C> the type of object the {@link Iterable} contains  
905 \* @param input the {@link Iterable} to get the input from, may be null  
906 \* @param predicate the predicate to use, may be null  
907 \* @return true if every element of the collection matches the predicate or if the  
908 \* collection is empty, false otherwise  
909 \* @since 4.0  
910 \* @deprecated since 4.1, use {@link IterableUtils#matchesAll(Iterable, Predicate)} instead  
911 \*/  
912 @Deprecated  
913 public static <C> boolean matchesAll(final Iterable<C> input, final Predicate<? super C> predicate) {  
914 return predicate != null && IterableUtils.matchesAll(input, predicate);  
915 }  
916  
917 /\*\*  
918 \* Selects all elements from input collection which match the given  
919 \* predicate into an output collection.  
920 \* <p>  
921 \* A <code>null</code> predicate matches no elements.  
922 \* </p>  
923 \*  
924 \* @param <O> the type of object the {@link Iterable} contains  
925 \* @param inputCollection the collection to get the input from, may not be null  
926 \* @param predicate the predicate to use, may be null  
927 \* @return the elements matching the predicate (new list)  
928 \* @throws NullPointerException if the input collection is null  
929 \*/  
930 public static <O> Collection<O> select(final Iterable<? extends O> inputCollection,  
931 final Predicate<? super O> predicate) {  
932 final Collection<O> answer = inputCollection instanceof Collection<?> ?  
933 new ArrayList<>(((Collection<?>) inputCollection).size()) : new ArrayList<>();  
934 return select(inputCollection, predicate, answer);  
935 }  
936  
937 /\*\*  
938 \* Selects all elements from input collection which match the given  
939 \* predicate and adds them to outputCollection.  
940 \* <p>  
941 \* If the input collection or predicate is null, there is no change to the  
942 \* output collection.  
943 \* </p>  
944 \*  
945 \* @param <O> the type of object the {@link Iterable} contains  
946 \* @param <R> the type of the output {@link Collection}  
947 \* @param inputCollection the collection to get the input from, may be null  
948 \* @param predicate the predicate to use, may be null  
949 \* @param outputCollection the collection to output into, may not be null if the inputCollection  
950 \* and predicate or not null  
951 \* @return the outputCollection  
952 \*/  
953 public static <O, R extends Collection<? super O>> R select(final Iterable<? extends O> inputCollection,  
954 final Predicate<? super O> predicate, final R outputCollection) {  
955  
956 if (inputCollection != null && predicate != null) {  
957 for (final O item : inputCollection) {  
958 if (predicate.evaluate(item)) {  
959 outputCollection.add(item);  
960 }  
961 }  
962 }  
963 return outputCollection;  
964 }  
965  
966 /\*\*  
967 \* Selects all elements from inputCollection into an output and rejected collection,  
968 \* based on the evaluation of the given predicate.  
969 \* <p>  
970 \* Elements matching the predicate are added to the <code>outputCollection</code>,  
971 \* all other elements are added to the <code>rejectedCollection</code>.  
972 \* </p>  
973 \* <p>  
974 \* If the input predicate is <code>null</code>, no elements are added to  
975 \* <code>outputCollection</code> or <code>rejectedCollection</code>.  
976 \* </p>  
977 \* <p>  
978 \* Note: calling the method is equivalent to the following code snippet:  
979 \* </p>  
980 \* <pre>  
981 \* select(inputCollection, predicate, outputCollection);  
982 \* selectRejected(inputCollection, predicate, rejectedCollection);  
983 \* </pre>  
984 \*  
985 \* @param <O> the type of object the {@link Iterable} contains  
986 \* @param <R> the type of the output {@link Collection}  
987 \* @param inputCollection the collection to get the input from, may be null  
988 \* @param predicate the predicate to use, may be null  
989 \* @param outputCollection the collection to output selected elements into, may not be null if the  
990 \* inputCollection and predicate are not null  
991 \* @param rejectedCollection the collection to output rejected elements into, may not be null if the  
992 \* inputCollection or predicate are not null  
993 \* @return the outputCollection  
994 \* @since 4.1  
995 \*/  
996 public static <O, R extends Collection<? super O>> R select(final Iterable<? extends O> inputCollection,  
997 final Predicate<? super O> predicate, final R outputCollection, final R rejectedCollection) {  
998  
999 if (inputCollection != null && predicate != null) {  
1000 for (final O element : inputCollection) {  
1001 if (predicate.evaluate(element)) {  
1002 outputCollection.add(element);  
1003 } else {  
1004 rejectedCollection.add(element);  
1005 }  
1006 }  
1007 }  
1008 return outputCollection;  
1009 }  
1010  
1011 /\*\*  
1012 \* Selects all elements from inputCollection which don't match the given  
1013 \* predicate into an output collection.  
1014 \* <p>  
1015 \* If the input predicate is <code>null</code>, the result is an empty  
1016 \* list.  
1017 \* </p>  
1018 \*  
1019 \* @param <O> the type of object the {@link Iterable} contains  
1020 \* @param inputCollection the collection to get the input from, may not be null  
1021 \* @param predicate the predicate to use, may be null  
1022 \* @return the elements <b>not</b> matching the predicate (new list)  
1023 \* @throws NullPointerException if the input collection is null  
1024 \*/  
1025 public static <O> Collection<O> selectRejected(final Iterable<? extends O> inputCollection,  
1026 final Predicate<? super O> predicate) {  
1027 final Collection<O> answer = inputCollection instanceof Collection<?> ?  
1028 new ArrayList<>(((Collection<?>) inputCollection).size()) : new ArrayList<>();  
1029 return selectRejected(inputCollection, predicate, answer);  
1030 }  
1031  
1032 /\*\*  
1033 \* Selects all elements from inputCollection which don't match the given  
1034 \* predicate and adds them to outputCollection.  
1035 \* <p>  
1036 \* If the input predicate is <code>null</code>, no elements are added to  
1037 \* <code>outputCollection</code>.  
1038 \* </p>  
1039 \*  
1040 \* @param <O> the type of object the {@link Iterable} contains  
1041 \* @param <R> the type of the output {@link Collection}  
1042 \* @param inputCollection the collection to get the input from, may be null  
1043 \* @param predicate the predicate to use, may be null  
1044 \* @param outputCollection the collection to output into, may not be null if the inputCollection  
1045 \* and predicate or not null  
1046 \* @return outputCollection  
1047 \*/  
1048 public static <O, R extends Collection<? super O>> R selectRejected(final Iterable<? extends O> inputCollection,  
1049 final Predicate<? super O> predicate, final R outputCollection) {  
1050  
1051 if (inputCollection != null && predicate != null) {  
1052 for (final O item : inputCollection) {  
1053 if (!predicate.evaluate(item)) {  
1054 outputCollection.add(item);  
1055 }  
1056 }  
1057 }  
1058 return outputCollection;  
1059 }  
1060  
1061 /\*\*  
1062 \* Returns a new Collection containing all elements of the input collection  
1063 \* transformed by the given transformer.  
1064 \* <p>  
1065 \* If the input collection or transformer is null, the result is an empty list.  
1066 \* </p>  
1067 \*  
1068 \* @param <I> the type of object in the input collection  
1069 \* @param <O> the type of object in the output collection  
1070 \* @param inputCollection the collection to get the input from, may not be null  
1071 \* @param transformer the transformer to use, may be null  
1072 \* @return the transformed result (new list)  
1073 \* @throws NullPointerException if the input collection is null  
1074 \*/  
1075 public static <I, O> Collection<O> collect(final Iterable<I> inputCollection,  
1076 final Transformer<? super I, ? extends O> transformer) {  
1077 final Collection<O> answer = inputCollection instanceof Collection<?> ?  
1078 new ArrayList<>(((Collection<?>) inputCollection).size()) : new ArrayList<>();  
1079 return collect(inputCollection, transformer, answer);  
1080 }  
1081  
1082 /\*\*  
1083 \* Transforms all elements from the input iterator with the given transformer  
1084 \* and adds them to the output collection.  
1085 \* <p>  
1086 \* If the input iterator or transformer is null, the result is an empty list.  
1087 \* </p>  
1088 \*  
1089 \* @param <I> the type of object in the input collection  
1090 \* @param <O> the type of object in the output collection  
1091 \* @param inputIterator the iterator to get the input from, may be null  
1092 \* @param transformer the transformer to use, may be null  
1093 \* @return the transformed result (new list)  
1094 \*/  
1095 public static <I, O> Collection<O> collect(final Iterator<I> inputIterator,  
1096 final Transformer<? super I, ? extends O> transformer) {  
1097 return collect(inputIterator, transformer, new ArrayList<O>());  
1098 }  
1099  
1100 /\*\*  
1101 \* Transforms all elements from input collection with the given transformer  
1102 \* and adds them to the output collection.  
1103 \* <p>  
1104 \* If the input collection or transformer is null, there is no change to the  
1105 \* output collection.  
1106 \* </p>  
1107 \*  
1108 \* @param <I> the type of object in the input collection  
1109 \* @param <O> the type of object in the output collection  
1110 \* @param <R> the type of the output collection  
1111 \* @param inputCollection the collection to get the input from, may be null  
1112 \* @param transformer the transformer to use, may be null  
1113 \* @param outputCollection the collection to output into, may not be null if inputCollection  
1114 \* and transformer are not null  
1115 \* @return the output collection with the transformed input added  
1116 \* @throws NullPointerException if the outputCollection is null and both, inputCollection and  
1117 \* transformer are not null  
1118 \*/  
1119 public static <I, O, R extends Collection<? super O>> R collect(final Iterable<? extends I> inputCollection,  
1120 final Transformer<? super I, ? extends O> transformer, final R outputCollection) {  
1121 if (inputCollection != null) {  
1122 return collect(inputCollection.iterator(), transformer, outputCollection);  
1123 }  
1124 return outputCollection;  
1125 }  
1126  
1127 /\*\*  
1128 \* Transforms all elements from the input iterator with the given transformer  
1129 \* and adds them to the output collection.  
1130 \* <p>  
1131 \* If the input iterator or transformer is null, there is no change to the  
1132 \* output collection.  
1133 \* </p>  
1134 \*  
1135 \* @param <I> the type of object in the input collection  
1136 \* @param <O> the type of object in the output collection  
1137 \* @param <R> the type of the output collection  
1138 \* @param inputIterator the iterator to get the input from, may be null  
1139 \* @param transformer the transformer to use, may be null  
1140 \* @param outputCollection the collection to output into, may not be null if inputIterator  
1141 \* and transformer are not null  
1142 \* @return the outputCollection with the transformed input added  
1143 \* @throws NullPointerException if the output collection is null and both, inputIterator and  
1144 \* transformer are not null  
1145 \*/  
1146 public static <I, O, R extends Collection<? super O>> R collect(final Iterator<? extends I> inputIterator,  
1147 final Transformer<? super I, ? extends O> transformer, final R outputCollection) {  
1148 if (inputIterator != null && transformer != null) {  
1149 while (inputIterator.hasNext()) {  
1150 final I item = inputIterator.next();  
1151 final O value = transformer.transform(item);  
1152 outputCollection.add(value);  
1153 }  
1154 }  
1155 return outputCollection;  
1156 }  
1157  
1158 //-----------------------------------------------------------------------  
1159 /\*\*  
1160 \* Adds an element to the collection unless the element is null.  
1161 \*  
1162 \* @param <T> the type of object the {@link Collection} contains  
1163 \* @param collection the collection to add to, must not be null  
1164 \* @param object the object to add, if null it will not be added  
1165 \* @return true if the collection changed  
1166 \* @throws NullPointerException if the collection is null  
1167 \* @since 3.2  
1168 \*/  
1169 public static <T> boolean addIgnoreNull(final Collection<T> collection, final T object) {  
1170 if (collection == null) {  
1171 throw new NullPointerException("The collection must not be null");  
1172 }  
1173 return object != null && collection.add(object);  
1174 }  
1175  
1176 /\*\*  
1177 \* Adds all elements in the {@link Iterable} to the given collection. If the  
1178 \* {@link Iterable} is a {@link Collection} then it is cast and will be  
1179 \* added using {@link Collection#addAll(Collection)} instead of iterating.  
1180 \*  
1181 \* @param <C> the type of object the {@link Collection} contains  
1182 \* @param collection the collection to add to, must not be null  
1183 \* @param iterable the iterable of elements to add, must not be null  
1184 \* @return a boolean indicating whether the collection has changed or not.  
1185 \* @throws NullPointerException if the collection or iterator is null  
1186 \*/  
1187 public static <C> boolean addAll(final Collection<C> collection, final Iterable<? extends C> iterable) {  
1188 if (iterable instanceof Collection<?>) {  
1189 return collection.addAll((Collection<? extends C>) iterable);  
1190 }  
1191 return addAll(collection, iterable.iterator());  
1192 }  
1193  
1194 /\*\*  
1195 \* Adds all elements in the iteration to the given collection.  
1196 \*  
1197 \* @param <C> the type of object the {@link Collection} contains  
1198 \* @param collection the collection to add to, must not be null  
1199 \* @param iterator the iterator of elements to add, must not be null  
1200 \* @return a boolean indicating whether the collection has changed or not.  
1201 \* @throws NullPointerException if the collection or iterator is null  
1202 \*/  
1203 public static <C> boolean addAll(final Collection<C> collection, final Iterator<? extends C> iterator) {  
1204 boolean changed = false;  
1205 while (iterator.hasNext()) {  
1206 changed |= collection.add(iterator.next());  
1207 }  
1208 return changed;  
1209 }  
1210  
1211 /\*\*  
1212 \* Adds all elements in the enumeration to the given collection.  
1213 \*  
1214 \* @param <C> the type of object the {@link Collection} contains  
1215 \* @param collection the collection to add to, must not be null  
1216 \* @param enumeration the enumeration of elements to add, must not be null  
1217 \* @return {@code true} if the collections was changed, {@code false} otherwise  
1218 \* @throws NullPointerException if the collection or enumeration is null  
1219 \*/  
1220 public static <C> boolean addAll(final Collection<C> collection, final Enumeration<? extends C> enumeration) {  
1221 boolean changed = false;  
1222 while (enumeration.hasMoreElements()) {  
1223 changed |= collection.add(enumeration.nextElement());  
1224 }  
1225 return changed;  
1226 }  
1227  
1228 /\*\*  
1229 \* Adds all elements in the array to the given collection.  
1230 \*  
1231 \* @param <C> the type of object the {@link Collection} contains  
1232 \* @param collection the collection to add to, must not be null  
1233 \* @param elements the array of elements to add, must not be null  
1234 \* @return {@code true} if the collection was changed, {@code false} otherwise  
1235 \* @throws NullPointerException if the collection or array is null  
1236 \*/  
1237 public static <C> boolean addAll(final Collection<C> collection, final C... elements) {  
1238 boolean changed = false;  
1239 for (final C element : elements) {  
1240 changed |= collection.add(element);  
1241 }  
1242 return changed;  
1243 }  
1244  
1245 /\*\*  
1246 \* Returns the <code>index</code>-th value in {@link Iterator}, throwing  
1247 \* <code>IndexOutOfBoundsException</code> if there is no such element.  
1248 \* <p>  
1249 \* The Iterator is advanced to <code>index</code> (or to the end, if  
1250 \* <code>index</code> exceeds the number of entries) as a side effect of this method.  
1251 \* </p>  
1252 \*  
1253 \* @param iterator the iterator to get a value from  
1254 \* @param index the index to get  
1255 \* @param <T> the type of object in the {@link Iterator}  
1256 \* @return the object at the specified index  
1257 \* @throws IndexOutOfBoundsException if the index is invalid  
1258 \* @throws IllegalArgumentException if the object type is invalid  
1259 \* @deprecated since 4.1, use {@code IteratorUtils.get(Iterator, int)} instead  
1260 \*/  
1261 @Deprecated  
1262 public static <T> T get(final Iterator<T> iterator, final int index) {  
1263 return IteratorUtils.get(iterator, index);  
1264 }  
1265  
1266 /\*\*  
1267 \* Ensures an index is not negative.  
1268 \* @param index the index to check.  
1269 \* @throws IndexOutOfBoundsException if the index is negative.  
1270 \*/  
1271 static void checkIndexBounds(final int index) {  
1272 if (index < 0) {  
1273 throw new IndexOutOfBoundsException("Index cannot be negative: " + index);  
1274 }  
1275 }  
1276  
1277 /\*\*  
1278 \* Returns the <code>index</code>-th value in the <code>iterable</code>'s {@link Iterator}, throwing  
1279 \* <code>IndexOutOfBoundsException</code> if there is no such element.  
1280 \* <p>  
1281 \* If the {@link Iterable} is a {@link List}, then it will use {@link List#get(int)}.  
1282 \* </p>  
1283 \*  
1284 \* @param iterable the {@link Iterable} to get a value from  
1285 \* @param index the index to get  
1286 \* @param <T> the type of object in the {@link Iterable}.  
1287 \* @return the object at the specified index  
1288 \* @throws IndexOutOfBoundsException if the index is invalid  
1289 \* @deprecated since 4.1, use {@code IterableUtils.get(Iterable, int)} instead  
1290 \*/  
1291 @Deprecated  
1292 public static <T> T get(final Iterable<T> iterable, final int index) {  
1293 return IterableUtils.get(iterable, index);  
1294 }  
1295  
1296 /\*\*  
1297 \* Returns the <code>index</code>-th value in <code>object</code>, throwing  
1298 \* <code>IndexOutOfBoundsException</code> if there is no such element or  
1299 \* <code>IllegalArgumentException</code> if <code>object</code> is not an  
1300 \* instance of one of the supported types.  
1301 \* <p>  
1302 \* The supported types, and associated semantics are:  
1303 \* </p>  
1304 \* <ul>  
1305 \* <li> Map -- the value returned is the <code>Map.Entry</code> in position  
1306 \* <code>index</code> in the map's <code>entrySet</code> iterator,  
1307 \* if there is such an entry.</li>  
1308 \* <li> List -- this method is equivalent to the list's get method.</li>  
1309 \* <li> Array -- the <code>index</code>-th array entry is returned,  
1310 \* if there is such an entry; otherwise an <code>IndexOutOfBoundsException</code>  
1311 \* is thrown.</li>  
1312 \* <li> Collection -- the value returned is the <code>index</code>-th object  
1313 \* returned by the collection's default iterator, if there is such an element.</li>  
1314 \* <li> Iterator or Enumeration -- the value returned is the  
1315 \* <code>index</code>-th object in the Iterator/Enumeration, if there  
1316 \* is such an element. The Iterator/Enumeration is advanced to  
1317 \* <code>index</code> (or to the end, if <code>index</code> exceeds the  
1318 \* number of entries) as a side effect of this method.</li>  
1319 \* </ul>  
1320 \*  
1321 \* @param object the object to get a value from  
1322 \* @param index the index to get  
1323 \* @return the object at the specified index  
1324 \* @throws IndexOutOfBoundsException if the index is invalid  
1325 \* @throws IllegalArgumentException if the object type is invalid  
1326 \*/  
1327 public static Object get(final Object object, final int index) {  
1328 final int i = index;  
1329 if (i < 0) {  
1330 throw new IndexOutOfBoundsException("Index cannot be negative: " + i);  
1331 }  
1332 if (object instanceof Map<?,?>) {  
1333 final Map<?, ?> map = (Map<?, ?>) object;  
1334 final Iterator<?> iterator = map.entrySet().iterator();  
1335 return IteratorUtils.get(iterator, i);  
1336 } else if (object instanceof Object[]) {  
1337 return ((Object[]) object)[i];  
1338 } else if (object instanceof Iterator<?>) {  
1339 final Iterator<?> it = (Iterator<?>) object;  
1340 return IteratorUtils.get(it, i);  
1341 } else if (object instanceof Iterable<?>) {  
1342 final Iterable<?> iterable = (Iterable<?>) object;  
1343 return IterableUtils.get(iterable, i);  
1344 } else if (object instanceof Enumeration<?>) {  
1345 final Enumeration<?> it = (Enumeration<?>) object;  
1346 return EnumerationUtils.get(it, i);  
1347 } else if (object == null) {  
1348 throw new IllegalArgumentException("Unsupported object type: null");  
1349 } else {  
1350 try {  
1351 return Array.get(object, i);  
1352 } catch (final IllegalArgumentException ex) {  
1353 throw new IllegalArgumentException("Unsupported object type: " + object.getClass().getName());  
1354 }  
1355 }  
1356 }  
1357  
1358 /\*\*  
1359 \* Returns the <code>index</code>-th <code>Map.Entry</code> in the <code>map</code>'s <code>entrySet</code>,  
1360 \* throwing <code>IndexOutOfBoundsException</code> if there is no such element.  
1361 \*  
1362 \* @param <K> the key type in the {@link Map}  
1363 \* @param <V> the key type in the {@link Map}  
1364 \* @param map the object to get a value from  
1365 \* @param index the index to get  
1366 \* @return the object at the specified index  
1367 \* @throws IndexOutOfBoundsException if the index is invalid  
1368 \*/  
1369 public static <K,V> Map.Entry<K, V> get(final Map<K,V> map, final int index) {  
1370 checkIndexBounds(index);  
1371 return get(map.entrySet(), index);  
1372 }  
1373  
1374 /\*\*  
1375 \* Gets the size of the collection/iterator specified.  
1376 \* <p>  
1377 \* This method can handles objects as follows  
1378 \* </p>  
1379 \* <ul>  
1380 \* <li>Collection - the collection size  
1381 \* <li>Map - the map size  
1382 \* <li>Array - the array size  
1383 \* <li>Iterator - the number of elements remaining in the iterator  
1384 \* <li>Enumeration - the number of elements remaining in the enumeration  
1385 \* </ul>  
1386 \*  
1387 \* @param object the object to get the size of, may be null  
1388 \* @return the size of the specified collection or 0 if the object was null  
1389 \* @throws IllegalArgumentException thrown if object is not recognized  
1390 \* @since 3.1  
1391 \*/  
1392 public static int size(final Object object) {  
1393 if (object == null) {  
1394 return 0;  
1395 }  
1396 int total = 0;  
1397 if (object instanceof Map<?,?>) {  
1398 total = ((Map<?, ?>) object).size();  
1399 } else if (object instanceof Collection<?>) {  
1400 total = ((Collection<?>) object).size();  
1401 } else if (object instanceof Iterable<?>) {  
1402 total = IterableUtils.size((Iterable<?>) object);  
1403 } else if (object instanceof Object[]) {  
1404 total = ((Object[]) object).length;  
1405 } else if (object instanceof Iterator<?>) {  
1406 total = IteratorUtils.size((Iterator<?>) object);  
1407 } else if (object instanceof Enumeration<?>) {  
1408 final Enumeration<?> it = (Enumeration<?>) object;  
1409 while (it.hasMoreElements()) {  
1410 total++;  
1411 it.nextElement();  
1412 }  
1413 } else {  
1414 try {  
1415 total = Array.getLength(object);  
1416 } catch (final IllegalArgumentException ex) {  
1417 throw new IllegalArgumentException("Unsupported object type: " + object.getClass().getName());  
1418 }  
1419 }  
1420 return total;  
1421 }  
1422  
1423 /\*\*  
1424 \* Checks if the specified collection/array/iterator is empty.  
1425 \* <p>  
1426 \* This method can handles objects as follows  
1427 \* </p>  
1428 \* <ul>  
1429 \* <li>Collection - via collection isEmpty  
1430 \* <li>Map - via map isEmpty  
1431 \* <li>Array - using array size  
1432 \* <li>Iterator - via hasNext  
1433 \* <li>Enumeration - via hasMoreElements  
1434 \* </ul>  
1435 \* <p>  
1436 \* Note: This method is named to avoid clashing with  
1437 \* {@link #isEmpty(Collection)}.  
1438 \* </p>  
1439 \*  
1440 \* @param object the object to get the size of, may be null  
1441 \* @return true if empty or null  
1442 \* @throws IllegalArgumentException thrown if object is not recognized  
1443 \* @since 3.2  
1444 \*/  
1445 public static boolean sizeIsEmpty(final Object object) {  
1446 if (object == null) {  
1447 return true;  
1448 } else if (object instanceof Collection<?>) {  
1449 return ((Collection<?>) object).isEmpty();  
1450 } else if (object instanceof Iterable<?>) {  
1451 return IterableUtils.isEmpty((Iterable<?>) object);  
1452 } else if (object instanceof Map<?, ?>) {  
1453 return ((Map<?, ?>) object).isEmpty();  
1454 } else if (object instanceof Object[]) {  
1455 return ((Object[]) object).length == 0;  
1456 } else if (object instanceof Iterator<?>) {  
1457 return ((Iterator<?>) object).hasNext() == false;  
1458 } else if (object instanceof Enumeration<?>) {  
1459 return ((Enumeration<?>) object).hasMoreElements() == false;  
1460 } else {  
1461 try {  
1462 return Array.getLength(object) == 0;  
1463 } catch (final IllegalArgumentException ex) {  
1464 throw new IllegalArgumentException("Unsupported object type: " + object.getClass().getName());  
1465 }  
1466 }  
1467 }  
1468  
1469 //-----------------------------------------------------------------------  
1470 /\*\*  
1471 \* Null-safe check if the specified collection is empty.  
1472 \* <p>  
1473 \* Null returns true.  
1474 \* </p>  
1475 \*  
1476 \* @param coll the collection to check, may be null  
1477 \* @return true if empty or null  
1478 \* @since 3.2  
1479 \*/  
1480 public static boolean isEmpty(final Collection<?> coll) {  
1481 return coll == null || coll.isEmpty();  
1482 }  
1483  
1484 /\*\*  
1485 \* Null-safe check if the specified collection is not empty.  
1486 \* <p>  
1487 \* Null returns false.  
1488 \* </p>  
1489 \*  
1490 \* @param coll the collection to check, may be null  
1491 \* @return true if non-null and non-empty  
1492 \* @since 3.2  
1493 \*/  
1494 public static boolean isNotEmpty(final Collection<?> coll) {  
1495 return !isEmpty(coll);  
1496 }  
1497  
1498 //-----------------------------------------------------------------------  
1499 /\*\*  
1500 \* Reverses the order of the given array.  
1501 \*  
1502 \* @param array the array to reverse  
1503 \*/  
1504 public static void reverseArray(final Object[] array) {  
1505 int i = 0;  
1506 int j = array.length - 1;  
1507 Object tmp;  
1508  
1509 while (j > i) {  
1510 tmp = array[j];  
1511 array[j] = array[i];  
1512 array[i] = tmp;  
1513 j--;  
1514 i++;  
1515 }  
1516 }  
1517  
1518 /\*\*  
1519 \* Returns true if no more elements can be added to the Collection.  
1520 \* <p>  
1521 \* This method uses the {@link BoundedCollection} interface to determine the  
1522 \* full status. If the collection does not implement this interface then  
1523 \* false is returned.  
1524 \* </p>  
1525 \* <p>  
1526 \* The collection does not have to implement this interface directly.  
1527 \* If the collection has been decorated using the decorators subpackage  
1528 \* then these will be removed to access the BoundedCollection.  
1529 \* </p>  
1530 \*  
1531 \* @param coll the collection to check  
1532 \* @return true if the BoundedCollection is full  
1533 \* @throws NullPointerException if the collection is null  
1534 \*/  
1535 public static boolean isFull(final Collection<? extends Object> coll) {  
1536 if (coll == null) {  
1537 throw new NullPointerException("The collection must not be null");  
1538 }  
1539 if (coll instanceof BoundedCollection) {  
1540 return ((BoundedCollection<?>) coll).isFull();  
1541 }  
1542 try {  
1543 final BoundedCollection<?> bcoll =  
1544 UnmodifiableBoundedCollection.unmodifiableBoundedCollection(coll);  
1545 return bcoll.isFull();  
1546 } catch (final IllegalArgumentException ex) {  
1547 return false;  
1548 }  
1549 }  
1550  
1551 /\*\*  
1552 \* Get the maximum number of elements that the Collection can contain.  
1553 \* <p>  
1554 \* This method uses the {@link BoundedCollection} interface to determine the  
1555 \* maximum size. If the collection does not implement this interface then  
1556 \* -1 is returned.  
1557 \* </p>  
1558 \* <p>  
1559 \* The collection does not have to implement this interface directly.  
1560 \* If the collection has been decorated using the decorators subpackage  
1561 \* then these will be removed to access the BoundedCollection.  
1562 \* </p>  
1563 \*  
1564 \* @param coll the collection to check  
1565 \* @return the maximum size of the BoundedCollection, -1 if no maximum size  
1566 \* @throws NullPointerException if the collection is null  
1567 \*/  
1568 public static int maxSize(final Collection<? extends Object> coll) {  
1569 if (coll == null) {  
1570 throw new NullPointerException("The collection must not be null");  
1571 }  
1572 if (coll instanceof BoundedCollection) {  
1573 return ((BoundedCollection<?>) coll).maxSize();  
1574 }  
1575 try {  
1576 final BoundedCollection<?> bcoll =  
1577 UnmodifiableBoundedCollection.unmodifiableBoundedCollection(coll);  
1578 return bcoll.maxSize();  
1579 } catch (final IllegalArgumentException ex) {  
1580 return -1;  
1581 }  
1582 }  
1583  
1584 //-----------------------------------------------------------------------  
1585 /\*\*  
1586 \* Merges two sorted Collections, a and b, into a single, sorted List  
1587 \* such that the natural ordering of the elements is retained.  
1588 \* <p>  
1589 \* Uses the standard O(n) merge algorithm for combining two sorted lists.  
1590 \* </p>  
1591 \*  
1592 \* @param <O> the element type  
1593 \* @param a the first collection, must not be null  
1594 \* @param b the second collection, must not be null  
1595 \* @return a new sorted List, containing the elements of Collection a and b  
1596 \* @throws NullPointerException if either collection is null  
1597 \* @since 4.0  
1598 \*/  
1599 public static <O extends Comparable<? super O>> List<O> collate(final Iterable<? extends O> a,  
1600 final Iterable<? extends O> b) {  
1601 return collate(a, b, ComparatorUtils.<O>naturalComparator(), true);  
1602 }  
1603  
1604 /\*\*  
1605 \* Merges two sorted Collections, a and b, into a single, sorted List  
1606 \* such that the natural ordering of the elements is retained.  
1607 \* <p>  
1608 \* Uses the standard O(n) merge algorithm for combining two sorted lists.  
1609 \* </p>  
1610 \*  
1611 \* @param <O> the element type  
1612 \* @param a the first collection, must not be null  
1613 \* @param b the second collection, must not be null  
1614 \* @param includeDuplicates if {@code true} duplicate elements will be retained, otherwise  
1615 \* they will be removed in the output collection  
1616 \* @return a new sorted List, containing the elements of Collection a and b  
1617 \* @throws NullPointerException if either collection is null  
1618 \* @since 4.0  
1619 \*/  
1620 public static <O extends Comparable<? super O>> List<O> collate(final Iterable<? extends O> a,  
1621 final Iterable<? extends O> b,  
1622 final boolean includeDuplicates) {  
1623 return collate(a, b, ComparatorUtils.<O>naturalComparator(), includeDuplicates);  
1624 }  
1625  
1626 /\*\*  
1627 \* Merges two sorted Collections, a and b, into a single, sorted List  
1628 \* such that the ordering of the elements according to Comparator c is retained.  
1629 \* <p>  
1630 \* Uses the standard O(n) merge algorithm for combining two sorted lists.  
1631 \* </p>  
1632 \*  
1633 \* @param <O> the element type  
1634 \* @param a the first collection, must not be null  
1635 \* @param b the second collection, must not be null  
1636 \* @param c the comparator to use for the merge.  
1637 \* @return a new sorted List, containing the elements of Collection a and b  
1638 \* @throws NullPointerException if either collection or the comparator is null  
1639 \* @since 4.0  
1640 \*/  
1641 public static <O> List<O> collate(final Iterable<? extends O> a, final Iterable<? extends O> b,  
1642 final Comparator<? super O> c) {  
1643 return collate(a, b, c, true);  
1644 }  
1645  
1646 /\*\*  
1647 \* Merges two sorted Collections, a and b, into a single, sorted List  
1648 \* such that the ordering of the elements according to Comparator c is retained.  
1649 \* <p>  
1650 \* Uses the standard O(n) merge algorithm for combining two sorted lists.  
1651 \* </p>  
1652 \*  
1653 \* @param <O> the element type  
1654 \* @param a the first collection, must not be null  
1655 \* @param b the second collection, must not be null  
1656 \* @param c the comparator to use for the merge.  
1657 \* @param includeDuplicates if {@code true} duplicate elements will be retained, otherwise  
1658 \* they will be removed in the output collection  
1659 \* @return a new sorted List, containing the elements of Collection a and b  
1660 \* @throws NullPointerException if either collection or the comparator is null  
1661 \* @since 4.0  
1662 \*/  
1663 public static <O> List<O> collate(final Iterable<? extends O> a, final Iterable<? extends O> b,  
1664 final Comparator<? super O> c, final boolean includeDuplicates) {  
1665  
1666 if (a == null || b == null) {  
1667 throw new NullPointerException("The collections must not be null");  
1668 }  
1669 if (c == null) {  
1670 throw new NullPointerException("The comparator must not be null");  
1671 }  
1672  
1673 // if both Iterables are a Collection, we can estimate the size  
1674 final int totalSize = a instanceof Collection<?> && b instanceof Collection<?> ?  
1675 Math.max(1, ((Collection<?>) a).size() + ((Collection<?>) b).size()) : 10;  
1676  
1677 final Iterator<O> iterator = new CollatingIterator<>(c, a.iterator(), b.iterator());  
1678 if (includeDuplicates) {  
1679 return IteratorUtils.toList(iterator, totalSize);  
1680 }  
1681 final ArrayList<O> mergedList = new ArrayList<>(totalSize);  
1682  
1683 O lastItem = null;  
1684 while (iterator.hasNext()) {  
1685 final O item = iterator.next();  
1686 if (lastItem == null || !lastItem.equals(item)) {  
1687 mergedList.add(item);  
1688 }  
1689 lastItem = item;  
1690 }  
1691  
1692 mergedList.trimToSize();  
1693 return mergedList;  
1694 }  
1695  
1696 //-----------------------------------------------------------------------  
1697  
1698 /\*\*  
1699 \* Returns a {@link Collection} of all the permutations of the input collection.  
1700 \* <p>  
1701 \* NOTE: the number of permutations of a given collection is equal to n!, where  
1702 \* n is the size of the collection. Thus, the resulting collection will become  
1703 \* <b>very</b> large for collections > 10 (e.g. 10! = 3628800, 15! = 1307674368000).  
1704 \* </p>  
1705 \* <p>  
1706 \* For larger collections it is advised to use a {@link PermutationIterator} to  
1707 \* iterate over all permutations.  
1708 \* </p>  
1709 \*  
1710 \* @see PermutationIterator  
1711 \*  
1712 \* @param <E> the element type  
1713 \* @param collection the collection to create permutations for, may not be null  
1714 \* @return an unordered collection of all permutations of the input collection  
1715 \* @throws NullPointerException if collection is null  
1716 \* @since 4.0  
1717 \*/  
1718 public static <E> Collection<List<E>> permutations(final Collection<E> collection) {  
1719 final PermutationIterator<E> it = new PermutationIterator<>(collection);  
1720 final Collection<List<E>> result = new ArrayList<>();  
1721 while (it.hasNext()) {  
1722 result.add(it.next());  
1723 }  
1724 return result;  
1725 }  
1726  
1727 //-----------------------------------------------------------------------  
1728 /\*\*  
1729 \* Returns a collection containing all the elements in <code>collection</code>  
1730 \* that are also in <code>retain</code>. The cardinality of an element <code>e</code>  
1731 \* in the returned collection is the same as the cardinality of <code>e</code>  
1732 \* in <code>collection</code> unless <code>retain</code> does not contain <code>e</code>, in which  
1733 \* case the cardinality is zero. This method is useful if you do not wish to modify  
1734 \* the collection <code>c</code> and thus cannot call <code>c.retainAll(retain);</code>.  
1735 \* <p>  
1736 \* This implementation iterates over <code>collection</code>, checking each element in  
1737 \* turn to see if it's contained in <code>retain</code>. If it's contained, it's added  
1738 \* to the returned list. As a consequence, it is advised to use a collection type for  
1739 \* <code>retain</code> that provides a fast (e.g. O(1)) implementation of  
1740 \* {@link Collection#contains(Object)}.  
1741 \* </p>  
1742 \*  
1743 \* @param <C> the type of object the {@link Collection} contains  
1744 \* @param collection the collection whose contents are the target of the #retailAll operation  
1745 \* @param retain the collection containing the elements to be retained in the returned collection  
1746 \* @return a <code>Collection</code> containing all the elements of <code>collection</code>  
1747 \* that occur at least once in <code>retain</code>.  
1748 \* @throws NullPointerException if either parameter is null  
1749 \* @since 3.2  
1750 \*/  
1751 public static <C> Collection<C> retainAll(final Collection<C> collection, final Collection<?> retain) {  
1752 return ListUtils.retainAll(collection, retain);  
1753 }  
1754  
1755 /\*\*  
1756 \* Returns a collection containing all the elements in  
1757 \* <code>collection</code> that are also in <code>retain</code>. The  
1758 \* cardinality of an element <code>e</code> in the returned collection is  
1759 \* the same as the cardinality of <code>e</code> in <code>collection</code>  
1760 \* unless <code>retain</code> does not contain <code>e</code>, in which case  
1761 \* the cardinality is zero. This method is useful if you do not wish to  
1762 \* modify the collection <code>c</code> and thus cannot call  
1763 \* <code>c.retainAll(retain);</code>.  
1764 \* <p>  
1765 \* Moreover this method uses an {@link Equator} instead of  
1766 \* {@link Object#equals(Object)} to determine the equality of the elements  
1767 \* in <code>collection</code> and <code>retain</code>. Hence this method is  
1768 \* useful in cases where the equals behavior of an object needs to be  
1769 \* modified without changing the object itself.  
1770 \* </p>  
1771 \*  
1772 \* @param <E> the type of object the {@link Collection} contains  
1773 \* @param collection the collection whose contents are the target of the {@code retainAll} operation  
1774 \* @param retain the collection containing the elements to be retained in the returned collection  
1775 \* @param equator the Equator used for testing equality  
1776 \* @return a <code>Collection</code> containing all the elements of <code>collection</code>  
1777 \* that occur at least once in <code>retain</code> according to the <code>equator</code>  
1778 \* @throws NullPointerException if any of the parameters is null  
1779 \* @since 4.1  
1780 \*/  
1781 public static <E> Collection<E> retainAll(final Iterable<E> collection,  
1782 final Iterable<? extends E> retain,  
1783 final Equator<? super E> equator) {  
1784  
1785 final Transformer<E, EquatorWrapper<E>> transformer = new Transformer<E, EquatorWrapper<E>>() {  
1786 @Override  
1787 public EquatorWrapper<E> transform(final E input) {  
1788 return new EquatorWrapper<>(equator, input);  
1789 }  
1790 };  
1791  
1792 final Set<EquatorWrapper<E>> retainSet =  
1793 collect(retain, transformer, new HashSet<EquatorWrapper<E>>());  
1794  
1795 final List<E> list = new ArrayList<>();  
1796 for (final E element : collection) {  
1797 if (retainSet.contains(new EquatorWrapper<>(equator, element))) {  
1798 list.add(element);  
1799 }  
1800 }  
1801 return list;  
1802 }  
1803  
1804 /\*\*  
1805 \* Removes the elements in <code>remove</code> from <code>collection</code>. That is, this  
1806 \* method returns a collection containing all the elements in <code>c</code>  
1807 \* that are not in <code>remove</code>. The cardinality of an element <code>e</code>  
1808 \* in the returned collection is the same as the cardinality of <code>e</code>  
1809 \* in <code>collection</code> unless <code>remove</code> contains <code>e</code>, in which  
1810 \* case the cardinality is zero. This method is useful if you do not wish to modify  
1811 \* the collection <code>c</code> and thus cannot call <code>collection.removeAll(remove);</code>.  
1812 \* <p>  
1813 \* This implementation iterates over <code>collection</code>, checking each element in  
1814 \* turn to see if it's contained in <code>remove</code>. If it's not contained, it's added  
1815 \* to the returned list. As a consequence, it is advised to use a collection type for  
1816 \* <code>remove</code> that provides a fast (e.g. O(1)) implementation of  
1817 \* {@link Collection#contains(Object)}.  
1818 \* </p>  
1819 \*  
1820 \* @param <E> the type of object the {@link Collection} contains  
1821 \* @param collection the collection from which items are removed (in the returned collection)  
1822 \* @param remove the items to be removed from the returned <code>collection</code>  
1823 \* @return a <code>Collection</code> containing all the elements of <code>collection</code> except  
1824 \* any elements that also occur in <code>remove</code>.  
1825 \* @throws NullPointerException if either parameter is null  
1826 \* @since 4.0 (method existed in 3.2 but was completely broken)  
1827 \*/  
1828 public static <E> Collection<E> removeAll(final Collection<E> collection, final Collection<?> remove) {  
1829 return ListUtils.removeAll(collection, remove);  
1830 }  
1831  
1832 /\*\*  
1833 \* Removes all elements in <code>remove</code> from <code>collection</code>.  
1834 \* That is, this method returns a collection containing all the elements in  
1835 \* <code>collection</code> that are not in <code>remove</code>. The  
1836 \* cardinality of an element <code>e</code> in the returned collection is  
1837 \* the same as the cardinality of <code>e</code> in <code>collection</code>  
1838 \* unless <code>remove</code> contains <code>e</code>, in which case the  
1839 \* cardinality is zero. This method is useful if you do not wish to modify  
1840 \* the collection <code>c</code> and thus cannot call  
1841 \* <code>collection.removeAll(remove)</code>.  
1842 \* <p>  
1843 \* Moreover this method uses an {@link Equator} instead of  
1844 \* {@link Object#equals(Object)} to determine the equality of the elements  
1845 \* in <code>collection</code> and <code>remove</code>. Hence this method is  
1846 \* useful in cases where the equals behavior of an object needs to be  
1847 \* modified without changing the object itself.  
1848 \* </p>  
1849 \*  
1850 \* @param <E> the type of object the {@link Collection} contains  
1851 \* @param collection the collection from which items are removed (in the returned collection)  
1852 \* @param remove the items to be removed from the returned collection  
1853 \* @param equator the Equator used for testing equality  
1854 \* @return a <code>Collection</code> containing all the elements of <code>collection</code>  
1855 \* except any element that if equal according to the <code>equator</code>  
1856 \* @throws NullPointerException if any of the parameters is null  
1857 \* @since 4.1  
1858 \*/  
1859 public static <E> Collection<E> removeAll(final Iterable<E> collection,  
1860 final Iterable<? extends E> remove,  
1861 final Equator<? super E> equator) {  
1862  
1863 final Transformer<E, EquatorWrapper<E>> transformer = new Transformer<E, EquatorWrapper<E>>() {  
1864 @Override  
1865 public EquatorWrapper<E> transform(final E input) {  
1866 return new EquatorWrapper<>(equator, input);  
1867 }  
1868 };  
1869  
1870 final Set<EquatorWrapper<E>> removeSet =  
1871 collect(remove, transformer, new HashSet<EquatorWrapper<E>>());  
1872  
1873 final List<E> list = new ArrayList<>();  
1874 for (final E element : collection) {  
1875 if (!removeSet.contains(new EquatorWrapper<>(equator, element))) {  
1876 list.add(element);  
1877 }  
1878 }  
1879 return list;  
1880 }  
1881  
1882 //-----------------------------------------------------------------------  
1883 /\*\*  
1884 \* Returns a synchronized collection backed by the given collection.  
1885 \* <p>  
1886 \* You must manually synchronize on the returned buffer's iterator to  
1887 \* avoid non-deterministic behavior:  
1888 \* </p>  
1889 \* <pre>  
1890 \* Collection c = CollectionUtils.synchronizedCollection(myCollection);  
1891 \* synchronized (c) {  
1892 \* Iterator i = c.iterator();  
1893 \* while (i.hasNext()) {  
1894 \* process (i.next());  
1895 \* }  
1896 \* }  
1897 \* </pre>  
1898 \* <p>  
1899 \* This method uses the implementation in the decorators subpackage.  
1900 \* </p>  
1901 \*  
1902 \* @param <C> the type of object the {@link Collection} contains  
1903 \* @param collection the collection to synchronize, must not be null  
1904 \* @return a synchronized collection backed by the given collection  
1905 \* @throws NullPointerException if the collection is null  
1906 \* @deprecated since 4.1, use {@link java.util.Collections#synchronizedCollection(Collection)} instead  
1907 \*/  
1908 @Deprecated  
1909 public static <C> Collection<C> synchronizedCollection(final Collection<C> collection) {  
1910 return SynchronizedCollection.synchronizedCollection(collection);  
1911 }  
1912  
1913 /\*\*  
1914 \* Returns an unmodifiable collection backed by the given collection.  
1915 \* <p>  
1916 \* This method uses the implementation in the decorators subpackage.  
1917 \* </p>  
1918 \*  
1919 \* @param <C> the type of object the {@link Collection} contains  
1920 \* @param collection the collection to make unmodifiable, must not be null  
1921 \* @return an unmodifiable collection backed by the given collection  
1922 \* @throws NullPointerException if the collection is null  
1923 \* @deprecated since 4.1, use {@link java.util.Collections#unmodifiableCollection(Collection)} instead  
1924 \*/  
1925 @Deprecated  
1926 public static <C> Collection<C> unmodifiableCollection(final Collection<? extends C> collection) {  
1927 return UnmodifiableCollection.unmodifiableCollection(collection);  
1928 }  
1929  
1930 /\*\*  
1931 \* Returns a predicated (validating) collection backed by the given collection.  
1932 \* <p>  
1933 \* Only objects that pass the test in the given predicate can be added to the collection.  
1934 \* Trying to add an invalid object results in an IllegalArgumentException.  
1935 \* It is important not to use the original collection after invoking this method,  
1936 \* as it is a backdoor for adding invalid objects.  
1937 \* </p>  
1938 \*  
1939 \* @param <C> the type of objects in the Collection.  
1940 \* @param collection the collection to predicate, must not be null  
1941 \* @param predicate the predicate for the collection, must not be null  
1942 \* @return a predicated collection backed by the given collection  
1943 \* @throws NullPointerException if the Collection is null  
1944 \*/  
1945 public static <C> Collection<C> predicatedCollection(final Collection<C> collection,  
1946 final Predicate<? super C> predicate) {  
1947 return PredicatedCollection.predicatedCollection(collection, predicate);  
1948 }  
1949  
1950 /\*\*  
1951 \* Returns a transformed bag backed by the given collection.  
1952 \* <p>  
1953 \* Each object is passed through the transformer as it is added to the  
1954 \* Collection. It is important not to use the original collection after invoking this  
1955 \* method, as it is a backdoor for adding untransformed objects.  
1956 \* </p>  
1957 \* <p>  
1958 \* Existing entries in the specified collection will not be transformed.  
1959 \* If you want that behaviour, see {@link TransformedCollection#transformedCollection}.  
1960 \* </p>  
1961 \*  
1962 \* @param <E> the type of object the {@link Collection} contains  
1963 \* @param collection the collection to predicate, must not be null  
1964 \* @param transformer the transformer for the collection, must not be null  
1965 \* @return a transformed collection backed by the given collection  
1966 \* @throws NullPointerException if the Collection or Transformer is null  
1967 \*/  
1968 public static <E> Collection<E> transformingCollection(final Collection<E> collection,  
1969 final Transformer<? super E, ? extends E> transformer) {  
1970 return TransformedCollection.transformingCollection(collection, transformer);  
1971 }  
1972  
1973 /\*\*  
1974 \* Extract the lone element of the specified Collection.  
1975 \*  
1976 \* @param <E> collection type  
1977 \* @param collection to read  
1978 \* @return sole member of collection  
1979 \* @throws NullPointerException if collection is null  
1980 \* @throws IllegalArgumentException if collection is empty or contains more than one element  
1981 \* @since 4.0  
1982 \*/  
1983 public static <E> E extractSingleton(final Collection<E> collection) {  
1984 if (collection == null) {  
1985 throw new NullPointerException("Collection must not be null.");  
1986 }  
1987 if (collection.size() != 1) {  
1988 throw new IllegalArgumentException("Can extract singleton only when collection size == 1");  
1989 }  
1990 return collection.iterator().next();  
1991 }  
1992}