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.util.ArrayList;  
020import java.util.Collection;  
021import java.util.Collections;  
022import java.util.Comparator;  
023import java.util.Iterator;  
024import java.util.List;  
025import java.util.Set;  
026  
027import org.apache.commons.collections4.functors.EqualPredicate;  
028import org.apache.commons.collections4.iterators.LazyIteratorChain;  
029import org.apache.commons.collections4.iterators.ReverseListIterator;  
030import org.apache.commons.collections4.iterators.UniqueFilterIterator;  
031  
032/\*\*  
033 \* Provides utility methods and decorators for {@link Iterable} instances.  
034 \* <p>  
035 \* <b>Note</b>: this util class has been designed for fail-fast argument checking.  
036 \* </p>  
037 \* <ul>  
038 \* <li>  
039 \* all decorator methods are <b>NOT</b> null-safe wrt the provided Iterable argument, i.e.  
040 \* they will throw a {@link NullPointerException} if a null Iterable is passed as argument.  
041 \* <li>  
042 \* all other utility methods are null-safe wrt the provided Iterable argument, i.e. they will  
043 \* treat a null Iterable the same way as an empty one. Other arguments which are null,  
044 \* e.g. a {@link Predicate}, will result in a {@link NullPointerException}. Exception: passing  
045 \* a null {@link Comparator} is equivalent to a Comparator with natural ordering.  
046 \* </ul>  
047 \*  
048 \* @since 4.1  
049 \*/  
050public class IterableUtils {  
051  
052 /\*\*  
053 \* An empty iterable.  
054 \*/  
055 @SuppressWarnings("rawtypes")  
056 static final FluentIterable EMPTY\_ITERABLE = new FluentIterable<Object>() {  
057 @Override  
058 public Iterator<Object> iterator() {  
059 return IteratorUtils.emptyIterator();  
060 }  
061 };  
062  
063 // Empty  
064 // ----------------------------------------------------------------------  
065  
066 /\*\*  
067 \* Gets an empty iterable.  
068 \* <p>  
069 \* This iterable does not contain any elements.  
070 \*  
071 \* @param <E> the element type  
072 \* @return an empty iterable  
073 \*/  
074 @SuppressWarnings("unchecked") // OK, empty collection is compatible with any type  
075 public static <E> Iterable<E> emptyIterable() {  
076 return EMPTY\_ITERABLE;  
077 }  
078  
079 // Chained  
080 // ----------------------------------------------------------------------  
081  
082 /\*\*  
083 \* Combines two iterables into a single iterable.  
084 \* <p>  
085 \* The returned iterable has an iterator that traverses the elements in {@code a},  
086 \* followed by the elements in {@code b}. The source iterators are not polled until  
087 \* necessary.  
088 \* <p>  
089 \* The returned iterable's iterator supports {@code remove()} when the corresponding  
090 \* input iterator supports it.  
091 \*  
092 \* @param <E> the element type  
093 \* @param a the first iterable, may not be null  
094 \* @param b the second iterable, may not be null  
095 \* @return a new iterable, combining the provided iterables  
096 \* @throws NullPointerException if either a or b is null  
097 \*/  
098 @SuppressWarnings("unchecked")  
099 public static <E> Iterable<E> chainedIterable(final Iterable<? extends E> a,  
100 final Iterable<? extends E> b) {  
101 return chainedIterable(new Iterable[] {a, b});  
102 }  
103  
104 /\*\*  
105 \* Combines three iterables into a single iterable.  
106 \* <p>  
107 \* The returned iterable has an iterator that traverses the elements in {@code a},  
108 \* followed by the elements in {@code b} and {@code c}. The source iterators are  
109 \* not polled until necessary.  
110 \* <p>  
111 \* The returned iterable's iterator supports {@code remove()} when the corresponding  
112 \* input iterator supports it.  
113 \*  
114 \* @param <E> the element type  
115 \* @param a the first iterable, may not be null  
116 \* @param b the second iterable, may not be null  
117 \* @param c the third iterable, may not be null  
118 \* @return a new iterable, combining the provided iterables  
119 \* @throws NullPointerException if either of the provided iterables is null  
120 \*/  
121 @SuppressWarnings("unchecked")  
122 public static <E> Iterable<E> chainedIterable(final Iterable<? extends E> a,  
123 final Iterable<? extends E> b,  
124 final Iterable<? extends E> c) {  
125 return chainedIterable(new Iterable[] {a, b, c});  
126 }  
127  
128 /\*\*  
129 \* Combines four iterables into a single iterable.  
130 \* <p>  
131 \* The returned iterable has an iterator that traverses the elements in {@code a},  
132 \* followed by the elements in {@code b}, {@code c} and {@code d}. The source  
133 \* iterators are not polled until necessary.  
134 \* <p>  
135 \* The returned iterable's iterator supports {@code remove()} when the corresponding  
136 \* input iterator supports it.  
137 \*  
138 \* @param <E> the element type  
139 \* @param a the first iterable, may not be null  
140 \* @param b the second iterable, may not be null  
141 \* @param c the third iterable, may not be null  
142 \* @param d the fourth iterable, may not be null  
143 \* @return a new iterable, combining the provided iterables  
144 \* @throws NullPointerException if either of the provided iterables is null  
145 \*/  
146 @SuppressWarnings("unchecked")  
147 public static <E> Iterable<E> chainedIterable(final Iterable<? extends E> a,  
148 final Iterable<? extends E> b,  
149 final Iterable<? extends E> c,  
150 final Iterable<? extends E> d) {  
151 return chainedIterable(new Iterable[] {a, b, c, d});  
152 }  
153  
154 /\*\*  
155 \* Combines the provided iterables into a single iterable.  
156 \* <p>  
157 \* The returned iterable has an iterator that traverses the elements in the order  
158 \* of the arguments, i.e. iterables[0], iterables[1], .... The source iterators  
159 \* are not polled until necessary.  
160 \* <p>  
161 \* The returned iterable's iterator supports {@code remove()} when the corresponding  
162 \* input iterator supports it.  
163 \*  
164 \* @param <E> the element type  
165 \* @param iterables the iterables to combine, may not be null  
166 \* @return a new iterable, combining the provided iterables  
167 \* @throws NullPointerException if either of the provided iterables is null  
168 \*/  
169 public static <E> Iterable<E> chainedIterable(final Iterable<? extends E>... iterables) {  
170 checkNotNull(iterables);  
171 return new FluentIterable<E>() {  
172 @Override  
173 public Iterator<E> iterator() {  
174 return new LazyIteratorChain<E>() {  
175 @Override  
176 protected Iterator<? extends E> nextIterator(final int count) {  
177 if (count > iterables.length) {  
178 return null;  
179 }  
180 return iterables[count - 1].iterator();  
181 }  
182 };  
183 }  
184 };  
185 }  
186  
187 // Collated  
188 // ----------------------------------------------------------------------  
189  
190 /\*\*  
191 \* Combines the two provided iterables into an ordered iterable using  
192 \* natural ordering.  
193 \* <p>  
194 \* The returned iterable's iterator supports {@code remove()} when the  
195 \* corresponding input iterator supports it.  
196 \*  
197 \* @param <E> the element type  
198 \* @param a the first iterable, may not be null  
199 \* @param b the second iterable, may not be null  
200 \* @return a filtered view on the specified iterable  
201 \* @throws NullPointerException if either of the provided iterables is null  
202 \*/  
203 public static <E> Iterable<E> collatedIterable(final Iterable<? extends E> a,  
204 final Iterable<? extends E> b) {  
205 checkNotNull(a, b);  
206 return new FluentIterable<E>() {  
207 @Override  
208 public Iterator<E> iterator() {  
209 return IteratorUtils.collatedIterator(null, a.iterator(), b.iterator());  
210 }  
211 };  
212 }  
213  
214 /\*\*  
215 \* Combines the two provided iterables into an ordered iterable using the  
216 \* provided comparator. If the comparator is null, natural ordering will be  
217 \* used.  
218 \* <p>  
219 \* The returned iterable's iterator supports {@code remove()} when the  
220 \* corresponding input iterator supports it.  
221 \*  
222 \* @param <E> the element type  
223 \* @param comparator the comparator defining an ordering over the elements,  
224 \* may be null, in which case natural ordering will be used  
225 \* @param a the first iterable, may not be null  
226 \* @param b the second iterable, may not be null  
227 \* @return a filtered view on the specified iterable  
228 \* @throws NullPointerException if either of the provided iterables is null  
229 \*/  
230 public static <E> Iterable<E> collatedIterable(final Comparator<? super E> comparator,  
231 final Iterable<? extends E> a,  
232 final Iterable<? extends E> b) {  
233 checkNotNull(a, b);  
234 return new FluentIterable<E>() {  
235 @Override  
236 public Iterator<E> iterator() {  
237 return IteratorUtils.collatedIterator(comparator, a.iterator(), b.iterator());  
238 }  
239 };  
240 }  
241  
242 // Filtered  
243 // ----------------------------------------------------------------------  
244  
245 /\*\*  
246 \* Returns a view of the given iterable that only contains elements matching  
247 \* the provided predicate.  
248 \* <p>  
249 \* The returned iterable's iterator supports {@code remove()} when the  
250 \* corresponding input iterator supports it.  
251 \*  
252 \* @param <E> the element type  
253 \* @param iterable the iterable to filter, may not be null  
254 \* @param predicate the predicate used to filter elements, may not be null  
255 \* @return a filtered view on the specified iterable  
256 \* @throws NullPointerException if either iterable or predicate is null  
257 \*/  
258 public static <E> Iterable<E> filteredIterable(final Iterable<E> iterable,  
259 final Predicate<? super E> predicate) {  
260 checkNotNull(iterable);  
261 if (predicate == null) {  
262 throw new NullPointerException("Predicate must not be null.");  
263 }  
264 return new FluentIterable<E>() {  
265 @Override  
266 public Iterator<E> iterator() {  
267 return IteratorUtils.filteredIterator(emptyIteratorIfNull(iterable), predicate);  
268 }  
269 };  
270 }  
271  
272 // Bounded  
273 // ----------------------------------------------------------------------  
274  
275 /\*\*  
276 \* Returns a view of the given iterable that contains at most the given number  
277 \* of elements.  
278 \* <p>  
279 \* The returned iterable's iterator supports {@code remove()} when the corresponding  
280 \* input iterator supports it.  
281 \*  
282 \* @param <E> the element type  
283 \* @param iterable the iterable to limit, may not be null  
284 \* @param maxSize the maximum number of elements, must not be negative  
285 \* @return a bounded view on the specified iterable  
286 \* @throws IllegalArgumentException if maxSize is negative  
287 \* @throws NullPointerException if iterable is null  
288 \*/  
289 public static <E> Iterable<E> boundedIterable(final Iterable<E> iterable, final long maxSize) {  
290 checkNotNull(iterable);  
291 if (maxSize < 0) {  
292 throw new IllegalArgumentException("MaxSize parameter must not be negative.");  
293 }  
294  
295 return new FluentIterable<E>() {  
296 @Override  
297 public Iterator<E> iterator() {  
298 return IteratorUtils.boundedIterator(iterable.iterator(), maxSize);  
299 }  
300 };  
301 }  
302  
303 // Looping  
304 // ----------------------------------------------------------------------  
305  
306 /\*\*  
307 \* Returns a view of the given iterable which will cycle infinitely over  
308 \* its elements.  
309 \* <p>  
310 \* The returned iterable's iterator supports {@code remove()} if  
311 \* {@code iterable.iterator()} does. After {@code remove()} is called, subsequent  
312 \* cycles omit the removed element, which is no longer in {@code iterable}. The  
313 \* iterator's {@code hasNext()} method returns {@code true} until {@code iterable}  
314 \* is empty.  
315 \*  
316 \* @param <E> the element type  
317 \* @param iterable the iterable to loop, may not be null  
318 \* @return a view of the iterable, providing an infinite loop over its elements  
319 \* @throws NullPointerException if iterable is null  
320 \*/  
321 public static <E> Iterable<E> loopingIterable(final Iterable<E> iterable) {  
322 checkNotNull(iterable);  
323 return new FluentIterable<E>() {  
324 @Override  
325 public Iterator<E> iterator() {  
326 return new LazyIteratorChain<E>() {  
327 @Override  
328 protected Iterator<? extends E> nextIterator(final int count) {  
329 if (IterableUtils.isEmpty(iterable)) {  
330 return null;  
331 }  
332 return iterable.iterator();  
333 }  
334 };  
335 }  
336 };  
337 }  
338  
339 // Reversed  
340 // ----------------------------------------------------------------------  
341  
342 /\*\*  
343 \* Returns a reversed view of the given iterable.  
344 \* <p>  
345 \* In case the provided iterable is a {@link List} instance, a  
346 \* {@link ReverseListIterator} will be used to reverse the traversal  
347 \* order, otherwise an intermediate {@link List} needs to be created.  
348 \* <p>  
349 \* The returned iterable's iterator supports {@code remove()} if the  
350 \* provided iterable is a {@link List} instance.  
351 \*  
352 \* @param <E> the element type  
353 \* @param iterable the iterable to use, may not be null  
354 \* @return a reversed view of the specified iterable  
355 \* @throws NullPointerException if iterable is null  
356 \* @see ReverseListIterator  
357 \*/  
358 public static <E> Iterable<E> reversedIterable(final Iterable<E> iterable) {  
359 checkNotNull(iterable);  
360 return new FluentIterable<E>() {  
361 @Override  
362 public Iterator<E> iterator() {  
363 final List<E> list = (iterable instanceof List<?>) ?  
364 (List<E>) iterable :  
365 IteratorUtils.toList(iterable.iterator());  
366 return new ReverseListIterator<>(list);  
367 }  
368 };  
369 }  
370  
371 // Skipping  
372 // ----------------------------------------------------------------------  
373  
374 /\*\*  
375 \* Returns a view of the given iterable that skips the first N elements.  
376 \* <p>  
377 \* The returned iterable's iterator supports {@code remove()} when the corresponding  
378 \* input iterator supports it.  
379 \*  
380 \* @param <E> the element type  
381 \* @param iterable the iterable to use, may not be null  
382 \* @param elementsToSkip the number of elements to skip from the start, must not be negative  
383 \* @return a view of the specified iterable, skipping the first N elements  
384 \* @throws IllegalArgumentException if elementsToSkip is negative  
385 \* @throws NullPointerException if iterable is null  
386 \*/  
387 public static <E> Iterable<E> skippingIterable(final Iterable<E> iterable, final long elementsToSkip) {  
388 checkNotNull(iterable);  
389 if (elementsToSkip < 0) {  
390 throw new IllegalArgumentException("ElementsToSkip parameter must not be negative.");  
391 }  
392  
393 return new FluentIterable<E>() {  
394 @Override  
395 public Iterator<E> iterator() {  
396 return IteratorUtils.skippingIterator(iterable.iterator(), elementsToSkip);  
397 }  
398 };  
399 }  
400  
401 // Transformed  
402 // ----------------------------------------------------------------------  
403  
404 /\*\*  
405 \* Returns a transformed view of the given iterable where all of its elements  
406 \* have been transformed by the provided transformer.  
407 \* <p>  
408 \* The returned iterable's iterator supports {@code remove()} when the corresponding  
409 \* input iterator supports it.  
410 \*  
411 \* @param <I> the input element type  
412 \* @param <O> the output element type  
413 \* @param iterable the iterable to transform, may not be null  
414 \* @param transformer the transformer, must not be null  
415 \* @return a transformed view of the specified iterable  
416 \* @throws NullPointerException if either iterable or transformer is null  
417 \*/  
418 public static <I, O> Iterable<O> transformedIterable(final Iterable<I> iterable,  
419 final Transformer<? super I, ? extends O> transformer) {  
420 checkNotNull(iterable);  
421 if (transformer == null) {  
422 throw new NullPointerException("Transformer must not be null.");  
423 }  
424 return new FluentIterable<O>() {  
425 @Override  
426 public Iterator<O> iterator() {  
427 return IteratorUtils.transformedIterator(iterable.iterator(), transformer);  
428 }  
429 };  
430 }  
431  
432 // Unique  
433 // ----------------------------------------------------------------------  
434  
435 /\*\*  
436 \* Returns a unique view of the given iterable.  
437 \* <p>  
438 \* The returned iterable's iterator supports {@code remove()} when the  
439 \* corresponding input iterator supports it. Calling {@code remove()}  
440 \* will only remove a single element from the underlying iterator.  
441 \*  
442 \* @param <E> the element type  
443 \* @param iterable the iterable to use, may not be null  
444 \* @return a unique view of the specified iterable  
445 \* @throws NullPointerException if iterable is null  
446 \*/  
447 public static <E> Iterable<E> uniqueIterable(final Iterable<E> iterable) {  
448 checkNotNull(iterable);  
449 return new FluentIterable<E>() {  
450 @Override  
451 public Iterator<E> iterator() {  
452 return new UniqueFilterIterator<>(iterable.iterator());  
453 }  
454 };  
455 }  
456  
457 // Unmodifiable  
458 // ----------------------------------------------------------------------  
459  
460 /\*\*  
461 \* Returns an unmodifiable view of the given iterable.  
462 \* <p>  
463 \* The returned iterable's iterator does not support {@code remove()}.  
464 \*  
465 \* @param <E> the element type  
466 \* @param iterable the iterable to use, may not be null  
467 \* @return an unmodifiable view of the specified iterable  
468 \* @throws NullPointerException if iterable is null  
469 \*/  
470 public static <E> Iterable<E> unmodifiableIterable(final Iterable<E> iterable) {  
471 checkNotNull(iterable);  
472 if (iterable instanceof UnmodifiableIterable<?>) {  
473 return iterable;  
474 }  
475 return new UnmodifiableIterable<>(iterable);  
476 }  
477  
478 /\*\*  
479 \* Inner class to distinguish unmodifiable instances.  
480 \*/  
481 private static final class UnmodifiableIterable<E> extends FluentIterable<E> {  
482 private final Iterable<E> unmodifiable;  
483  
484 public UnmodifiableIterable(final Iterable<E> iterable) {  
485 super();  
486 this.unmodifiable = iterable;  
487 }  
488  
489 @Override  
490 public Iterator<E> iterator() {  
491 return IteratorUtils.unmodifiableIterator(unmodifiable.iterator());  
492 }  
493 }  
494  
495 // Zipping  
496 // ----------------------------------------------------------------------  
497  
498 /\*\*  
499 \* Interleaves two iterables into a single iterable.  
500 \* <p>  
501 \* The returned iterable has an iterator that traverses the elements in {@code a}  
502 \* and {@code b} in alternating order. The source iterators are not polled until  
503 \* necessary.  
504 \* <p>  
505 \* The returned iterable's iterator supports {@code remove()} when the corresponding  
506 \* input iterator supports it.  
507 \*  
508 \* @param <E> the element type  
509 \* @param a the first iterable, may not be null  
510 \* @param b the second iterable, may not be null  
511 \* @return a new iterable, interleaving the provided iterables  
512 \* @throws NullPointerException if either a or b is null  
513 \*/  
514 public static <E> Iterable<E> zippingIterable(final Iterable<? extends E> a,  
515 final Iterable<? extends E> b) {  
516 checkNotNull(a);  
517 checkNotNull(b);  
518 return new FluentIterable<E>() {  
519 @Override  
520 public Iterator<E> iterator() {  
521 return IteratorUtils.zippingIterator(a.iterator(), b.iterator());  
522 }  
523 };  
524 }  
525  
526 /\*\*  
527 \* Interleaves two iterables into a single iterable.  
528 \* <p>  
529 \* The returned iterable has an iterator that traverses the elements in {@code a}  
530 \* and {@code b} in alternating order. The source iterators are not polled until  
531 \* necessary.  
532 \* <p>  
533 \* The returned iterable's iterator supports {@code remove()} when the corresponding  
534 \* input iterator supports it.  
535 \*  
536 \* @param <E> the element type  
537 \* @param first the first iterable, may not be null  
538 \* @param others the array of iterables to interleave, may not be null  
539 \* @return a new iterable, interleaving the provided iterables  
540 \* @throws NullPointerException if either of the provided iterables is null  
541 \*/  
542 public static <E> Iterable<E> zippingIterable(final Iterable<? extends E> first,  
543 final Iterable<? extends E>... others) {  
544 checkNotNull(first);  
545 checkNotNull(others);  
546 return new FluentIterable<E>() {  
547 @Override  
548 public Iterator<E> iterator() {  
549 @SuppressWarnings("unchecked") // safe  
550 final  
551 Iterator<? extends E>[] iterators = new Iterator[others.length + 1];  
552 iterators[0] = first.iterator();  
553 for (int i = 0; i < others.length; i++) {  
554 iterators[i + 1] = others[i].iterator();  
555 }  
556 return IteratorUtils.zippingIterator(iterators);  
557 }  
558 };  
559 }  
560  
561 // Utility methods  
562 // ----------------------------------------------------------------------  
563  
564 /\*\*  
565 \* Returns an immutable empty iterable if the argument is null,  
566 \* or the argument itself otherwise.  
567 \*  
568 \* @param <E> the element type  
569 \* @param iterable the iterable, may be null  
570 \* @return an empty iterable if the argument is null  
571 \*/  
572 public static <E> Iterable<E> emptyIfNull(final Iterable<E> iterable) {  
573 return iterable == null ? IterableUtils.<E>emptyIterable() : iterable;  
574 }  
575  
576 /\*\*  
577 \* Applies the closure to each element of the provided iterable.  
578 \*  
579 \* @param <E> the element type  
580 \* @param iterable the iterator to use, may be null  
581 \* @param closure the closure to apply to each element, may not be null  
582 \* @throws NullPointerException if closure is null  
583 \*/  
584 public static <E> void forEach(final Iterable<E> iterable, final Closure<? super E> closure) {  
585 IteratorUtils.forEach(emptyIteratorIfNull(iterable), closure);  
586 }  
587  
588 /\*\*  
589 \* Executes the given closure on each but the last element in the iterable.  
590 \* <p>  
591 \* If the input iterable is null no change is made.  
592 \*  
593 \* @param <E> the type of object the {@link Iterable} contains  
594 \* @param iterable the iterable to get the input from, may be null  
595 \* @param closure the closure to perform, may not be null  
596 \* @return the last element in the iterable, or null if iterable is null or empty  
597 \*/  
598 public static <E> E forEachButLast(final Iterable<E> iterable, final Closure<? super E> closure) {  
599 return IteratorUtils.forEachButLast(emptyIteratorIfNull(iterable), closure);  
600 }  
601  
602 /\*\*  
603 \* Finds the first element in the given iterable which matches the given predicate.  
604 \* <p>  
605 \* A <code>null</code> or empty iterator returns null.  
606 \*  
607 \* @param <E> the element type  
608 \* @param iterable the iterable to search, may be null  
609 \* @param predicate the predicate to use, may not be null  
610 \* @return the first element of the iterable which matches the predicate or null if none could be found  
611 \* @throws NullPointerException if predicate is null  
612 \*/  
613 public static <E> E find(final Iterable<E> iterable, final Predicate<? super E> predicate) {  
614 return IteratorUtils.find(emptyIteratorIfNull(iterable), predicate);  
615 }  
616  
617 /\*\*  
618 \* Returns the index of the first element in the specified iterable that  
619 \* matches the given predicate.  
620 \* <p>  
621 \* A <code>null</code> or empty iterable returns -1.  
622 \*  
623 \* @param <E> the element type  
624 \* @param iterable the iterable to search, may be null  
625 \* @param predicate the predicate to use, may not be null  
626 \* @return the index of the first element which matches the predicate or -1 if none matches  
627 \* @throws NullPointerException if predicate is null  
628 \*/  
629 public static <E> int indexOf(final Iterable<E> iterable, final Predicate<? super E> predicate) {  
630 return IteratorUtils.indexOf(emptyIteratorIfNull(iterable), predicate);  
631 }  
632  
633 /\*\*  
634 \* Answers true if a predicate is true for every element of an iterable.  
635 \* <p>  
636 \* A <code>null</code> or empty iterable returns true.  
637 \*  
638 \* @param <E> the type of object the {@link Iterable} contains  
639 \* @param iterable the {@link Iterable} to use, may be null  
640 \* @param predicate the predicate to use, may not be null  
641 \* @return true if every element of the collection matches the predicate or if the  
642 \* collection is empty, false otherwise  
643 \* @throws NullPointerException if predicate is null  
644 \*/  
645 public static <E> boolean matchesAll(final Iterable<E> iterable, final Predicate<? super E> predicate) {  
646 return IteratorUtils.matchesAll(emptyIteratorIfNull(iterable), predicate);  
647 }  
648  
649 /\*\*  
650 \* Answers true if a predicate is true for any element of the iterable.  
651 \* <p>  
652 \* A <code>null</code> or empty iterable returns false.  
653 \*  
654 \* @param <E> the type of object the {@link Iterable} contains  
655 \* @param iterable the {@link Iterable} to use, may be null  
656 \* @param predicate the predicate to use, may not be null  
657 \* @return true if any element of the collection matches the predicate, false otherwise  
658 \* @throws NullPointerException if predicate is null  
659 \*/  
660 public static <E> boolean matchesAny(final Iterable<E> iterable, final Predicate<? super E> predicate) {  
661 return IteratorUtils.matchesAny(emptyIteratorIfNull(iterable), predicate);  
662 }  
663  
664 /\*\*  
665 \* Counts the number of elements in the input iterable that match the predicate.  
666 \* <p>  
667 \* A <code>null</code> iterable matches no elements.  
668 \*  
669 \* @param <E> the type of object the {@link Iterable} contains  
670 \* @param input the {@link Iterable} to get the input from, may be null  
671 \* @param predicate the predicate to use, may not be null  
672 \* @return the number of matches for the predicate in the collection  
673 \* @throws NullPointerException if predicate is null  
674 \*/  
675 public static <E> long countMatches(final Iterable<E> input, final Predicate<? super E> predicate) {  
676 if (predicate == null) {  
677 throw new NullPointerException("Predicate must not be null.");  
678 }  
679 return size(filteredIterable(emptyIfNull(input), predicate));  
680 }  
681  
682 /\*\*  
683 \* Answers true if the provided iterable is empty.  
684 \* <p>  
685 \* A <code>null</code> iterable returns true.  
686 \*  
687 \* @param iterable the {@link Iterable to use}, may be null  
688 \* @return true if the iterable is null or empty, false otherwise  
689 \*/  
690 public static boolean isEmpty(final Iterable<?> iterable) {  
691 if (iterable instanceof Collection<?>) {  
692 return ((Collection<?>) iterable).isEmpty();  
693 }  
694 return IteratorUtils.isEmpty(emptyIteratorIfNull(iterable));  
695 }  
696  
697 /\*\*  
698 \* Checks if the object is contained in the given iterable.  
699 \* <p>  
700 \* A <code>null</code> or empty iterable returns false.  
701 \*  
702 \* @param <E> the type of object the {@link Iterable} contains  
703 \* @param iterable the iterable to check, may be null  
704 \* @param object the object to check  
705 \* @return true if the object is contained in the iterable, false otherwise  
706 \*/  
707 public static <E> boolean contains(final Iterable<E> iterable, final Object object) {  
708 if (iterable instanceof Collection<?>) {  
709 return ((Collection<E>) iterable).contains(object);  
710 }  
711 return IteratorUtils.contains(emptyIteratorIfNull(iterable), object);  
712 }  
713  
714 /\*\*  
715 \* Checks if the object is contained in the given iterable. Object equality  
716 \* is tested with an {@code equator} unlike {@link #contains(Iterable, Object)}  
717 \* which uses {@link Object#equals(Object)}.  
718 \* <p>  
719 \* A <code>null</code> or empty iterable returns false.  
720 \* A <code>null</code> object will not be passed to the equator, instead a  
721 \* {@link org.apache.commons.collections4.functors.NullPredicate NullPredicate}  
722 \* will be used.  
723 \*  
724 \* @param <E> the type of object the {@link Iterable} contains  
725 \* @param iterable the iterable to check, may be null  
726 \* @param object the object to check  
727 \* @param equator the equator to use to check, may not be null  
728 \* @return true if the object is contained in the iterable, false otherwise  
729 \* @throws NullPointerException if equator is null  
730 \*/  
731 public static <E> boolean contains(final Iterable<? extends E> iterable, final E object,  
732 final Equator<? super E> equator) {  
733 if (equator == null) {  
734 throw new NullPointerException("Equator must not be null.");  
735 }  
736 return matchesAny(iterable, EqualPredicate.equalPredicate(object, equator));  
737 }  
738  
739 /\*\*  
740 \* Returns the number of occurrences of the provided object in the iterable.  
741 \*  
742 \* @param <E> the element type that the {@link Iterable} may contain  
743 \* @param <T> the element type of the object to find  
744 \* @param iterable the {@link Iterable} to search  
745 \* @param obj the object to find the cardinality of  
746 \* @return the number of occurrences of obj in iterable  
747 \*/  
748 public static <E, T extends E> int frequency(final Iterable<E> iterable, final T obj) {  
749 if (iterable instanceof Set<?>) {  
750 return ((Set<E>) iterable).contains(obj) ? 1 : 0;  
751 }  
752 if (iterable instanceof Bag<?>) {  
753 return ((Bag<E>) iterable).getCount(obj);  
754 }  
755 return size(filteredIterable(emptyIfNull(iterable), EqualPredicate.<E>equalPredicate(obj)));  
756 }  
757  
758 /\*\*  
759 \* Returns the <code>index</code>-th value in the <code>iterable</code>'s {@link Iterator}, throwing  
760 \* <code>IndexOutOfBoundsException</code> if there is no such element.  
761 \* <p>  
762 \* If the {@link Iterable} is a {@link List}, then it will use {@link List#get(int)}.  
763 \*  
764 \* @param <T> the type of object in the {@link Iterable}.  
765 \* @param iterable the {@link Iterable} to get a value from, may be null  
766 \* @param index the index to get  
767 \* @return the object at the specified index  
768 \* @throws IndexOutOfBoundsException if the index is invalid  
769 \*/  
770 public static <T> T get(final Iterable<T> iterable, final int index) {  
771 CollectionUtils.checkIndexBounds(index);  
772 if (iterable instanceof List<?>) {  
773 return ((List<T>) iterable).get(index);  
774 }  
775 return IteratorUtils.get(emptyIteratorIfNull(iterable), index);  
776 }  
777  
778 /\*\*  
779 \* Shortcut for {@code get(iterator, 0)}.  
780 \* <p>  
781 \* Returns the <code>first</code> value in the <code>iterable</code>'s {@link Iterator}, throwing  
782 \* <code>IndexOutOfBoundsException</code> if there is no such element.  
783 \* </p>  
784 \* <p>  
785 \* If the {@link Iterable} is a {@link List}, then it will use {@link List#get(int)}.  
786 \* </p>  
787 \*  
788 \* @param <T> the type of object in the {@link Iterable}.  
789 \* @param iterable the {@link Iterable} to get a value from, may be null  
790 \* @return the first object  
791 \* @throws IndexOutOfBoundsException if the request is invalid  
792 \* @since 4.2  
793 \*/  
794 public static <T> T first(final Iterable<T> iterable) {  
795 return get(iterable, 0);  
796 }  
797  
798 /\*\*  
799 \* Returns the number of elements contained in the given iterator.  
800 \* <p>  
801 \* A <code>null</code> or empty iterator returns {@code 0}.  
802 \*  
803 \* @param iterable the iterable to check, may be null  
804 \* @return the number of elements contained in the iterable  
805 \*/  
806 public static int size(final Iterable<?> iterable) {  
807 if (iterable instanceof Collection<?>) {  
808 return ((Collection<?>) iterable).size();  
809 }  
810 return IteratorUtils.size(emptyIteratorIfNull(iterable));  
811 }  
812  
813 /\*\*  
814 \* Partitions all elements from iterable into separate output collections,  
815 \* based on the evaluation of the given predicate.  
816 \* <p>  
817 \* For each predicate, the result will contain a list holding all elements of the  
818 \* input iterable matching the predicate. The last list will hold all elements  
819 \* which didn't match any predicate:  
820 \* <pre>  
821 \* [C1, R] = partition(I, P1) with  
822 \* I = input  
823 \* P1 = first predicate  
824 \* C1 = collection of elements matching P1  
825 \* R = collection of elements rejected by all predicates  
826 \* </pre>  
827 \* <p>  
828 \* If the input iterable is <code>null</code>, the same is returned as for an  
829 \* empty iterable.  
830 \* <p>  
831 \* Example: for an input list [1, 2, 3, 4, 5] calling partition with a predicate [x < 3]  
832 \* will result in the following output: [[1, 2], [3, 4, 5]].  
833 \*  
834 \* @param <O> the type of object the {@link Iterable} contains  
835 \* @param iterable the iterable to partition, may be null  
836 \* @param predicate the predicate to use, may not be null  
837 \* @return a list containing the output collections  
838 \* @throws NullPointerException if predicate is null  
839 \*/  
840 public static <O> List<List<O>> partition(final Iterable<? extends O> iterable,  
841 final Predicate<? super O> predicate) {  
842 if (predicate == null) {  
843 throw new NullPointerException("Predicate must not be null.");  
844 }  
845 @SuppressWarnings({ "unchecked", "rawtypes" }) // safe  
846 final Factory<List<O>> factory = FactoryUtils.instantiateFactory((Class) ArrayList.class);  
847 @SuppressWarnings("unchecked") // safe  
848 final Predicate<? super O>[] predicates = new Predicate[] { predicate };  
849 return partition(iterable, factory, predicates);  
850 }  
851  
852 /\*\*  
853 \* Partitions all elements from iterable into separate output collections,  
854 \* based on the evaluation of the given predicates.  
855 \* <p>  
856 \* For each predicate, the result will contain a list holding all elements of the  
857 \* input iterable matching the predicate. The last list will hold all elements  
858 \* which didn't match any predicate:  
859 \* <pre>  
860 \* [C1, C2, R] = partition(I, P1, P2) with  
861 \* I = input  
862 \* P1 = first predicate  
863 \* P2 = second predicate  
864 \* C1 = collection of elements matching P1  
865 \* C2 = collection of elements matching P2  
866 \* R = collection of elements rejected by all predicates  
867 \* </pre>  
868 \* <p>  
869 \* <b>Note</b>: elements are only added to the output collection of the first matching  
870 \* predicate, determined by the order of arguments.  
871 \* <p>  
872 \* If the input iterable is <code>null</code>, the same is returned as for an  
873 \* empty iterable.  
874 \* <p>  
875 \* Example: for an input list [1, 2, 3, 4, 5] calling partition with predicates [x < 3]  
876 \* and [x < 5] will result in the following output: [[1, 2], [3, 4], [5]].  
877 \*  
878 \* @param <O> the type of object the {@link Iterable} contains  
879 \* @param iterable the collection to get the input from, may be null  
880 \* @param predicates the predicates to use, may not be null  
881 \* @return a list containing the output collections  
882 \* @throws NullPointerException if any predicate is null  
883 \*/  
884 public static <O> List<List<O>> partition(final Iterable<? extends O> iterable,  
885 final Predicate<? super O>... predicates) {  
886  
887 @SuppressWarnings({ "unchecked", "rawtypes" }) // safe  
888 final Factory<List<O>> factory = FactoryUtils.instantiateFactory((Class) ArrayList.class);  
889 return partition(iterable, factory, predicates);  
890 }  
891  
892 /\*\*  
893 \* Partitions all elements from iterable into separate output collections,  
894 \* based on the evaluation of the given predicates.  
895 \* <p>  
896 \* For each predicate, the returned list will contain a collection holding  
897 \* all elements of the input iterable matching the predicate. The last collection  
898 \* contained in the list will hold all elements which didn't match any predicate:  
899 \* <pre>  
900 \* [C1, C2, R] = partition(I, P1, P2) with  
901 \* I = input  
902 \* P1 = first predicate  
903 \* P2 = second predicate  
904 \* C1 = collection of elements matching P1  
905 \* C2 = collection of elements matching P2  
906 \* R = collection of elements rejected by all predicates  
907 \* </pre>  
908 \* <p>  
909 \* <b>Note</b>: elements are only added to the output collection of the first matching  
910 \* predicate, determined by the order of arguments.  
911 \* <p>  
912 \* If the input iterable is <code>null</code>, the same is returned as for an  
913 \* empty iterable.  
914 \* If no predicates have been provided, all elements of the input collection  
915 \* will be added to the rejected collection.  
916 \* <p>  
917 \* Example: for an input list [1, 2, 3, 4, 5] calling partition with predicates [x < 3]  
918 \* and [x < 5] will result in the following output: [[1, 2], [3, 4], [5]].  
919 \*  
920 \* @param <O> the type of object the {@link Iterable} contains  
921 \* @param <R> the type of the output {@link Collection}  
922 \* @param iterable the collection to get the input from, may be null  
923 \* @param partitionFactory the factory used to create the output collections  
924 \* @param predicates the predicates to use, may not be null  
925 \* @return a list containing the output collections  
926 \* @throws NullPointerException if any predicate is null  
927 \*/  
928 public static <O, R extends Collection<O>> List<R> partition(final Iterable<? extends O> iterable,  
929 final Factory<R> partitionFactory, final Predicate<? super O>... predicates) {  
930  
931 if (iterable == null) {  
932 final Iterable<O> empty = emptyIterable();  
933 return partition(empty, partitionFactory, predicates);  
934 }  
935  
936 if (predicates == null) {  
937 throw new NullPointerException("Predicates must not be null.");  
938 }  
939  
940 for (final Predicate<?> p : predicates) {  
941 if (p == null) {  
942 throw new NullPointerException("Predicate must not be null.");  
943 }  
944 }  
945  
946 if (predicates.length < 1) {  
947 // return the entire input collection as a single partition  
948 final R singlePartition = partitionFactory.create();  
949 CollectionUtils.addAll(singlePartition, iterable);  
950 return Collections.singletonList(singlePartition);  
951 }  
952  
953 // create the empty partitions  
954 final int numberOfPredicates = predicates.length;  
955 final int numberOfPartitions = numberOfPredicates + 1;  
956 final List<R> partitions = new ArrayList<>(numberOfPartitions);  
957 for (int i = 0; i < numberOfPartitions; ++i) {  
958 partitions.add(partitionFactory.create());  
959 }  
960  
961 // for each element in inputCollection:  
962 // find the first predicate that evaluates to true.  
963 // if there is a predicate, add the element to the corresponding partition.  
964 // if there is no predicate, add it to the last, catch-all partition.  
965 for (final O element : iterable) {  
966 boolean elementAssigned = false;  
967 for (int i = 0; i < numberOfPredicates; ++i) {  
968 if (predicates[i].evaluate(element)) {  
969 partitions.get(i).add(element);  
970 elementAssigned = true;  
971 break;  
972 }  
973 }  
974  
975 if (!elementAssigned) {  
976 // no predicates evaluated to true  
977 // add element to last partition  
978 partitions.get(numberOfPredicates).add(element);  
979 }  
980 }  
981  
982 return partitions;  
983 }  
984  
985 /\*\*  
986 \* Gets a new list with the contents of the provided iterable.  
987 \*  
988 \* @param <E> the element type  
989 \* @param iterable the iterable to use, may be null  
990 \* @return a list of the iterator contents  
991 \*/  
992 public static <E> List<E> toList(final Iterable<E> iterable) {  
993 return IteratorUtils.toList(emptyIteratorIfNull(iterable));  
994 }  
995  
996 /\*\*  
997 \* Returns a string representation of the elements of the specified iterable.  
998 \* <p>  
999 \* The string representation consists of a list of the iterable's elements,  
1000 \* enclosed in square brackets ({@code "[]"}). Adjacent elements are separated  
1001 \* by the characters {@code ", "} (a comma followed by a space). Elements are  
1002 \* converted to strings as by {@code String.valueOf(Object)}.  
1003 \*  
1004 \* @param <E> the element type  
1005 \* @param iterable the iterable to convert to a string, may be null  
1006 \* @return a string representation of {@code iterable}  
1007 \*/  
1008 public static <E> String toString(final Iterable<E> iterable) {  
1009 return IteratorUtils.toString(emptyIteratorIfNull(iterable));  
1010 }  
1011  
1012 /\*\*  
1013 \* Returns a string representation of the elements of the specified iterable.  
1014 \* <p>  
1015 \* The string representation consists of a list of the iterable's elements,  
1016 \* enclosed in square brackets ({@code "[]"}). Adjacent elements are separated  
1017 \* by the characters {@code ", "} (a comma followed by a space). Elements are  
1018 \* converted to strings as by using the provided {@code transformer}.  
1019 \*  
1020 \* @param <E> the element type  
1021 \* @param iterable the iterable to convert to a string, may be null  
1022 \* @param transformer the transformer used to get a string representation of an element  
1023 \* @return a string representation of {@code iterable}  
1024 \* @throws NullPointerException if {@code transformer} is null  
1025 \*/  
1026 public static <E> String toString(final Iterable<E> iterable,  
1027 final Transformer<? super E, String> transformer) {  
1028 if (transformer == null) {  
1029 throw new NullPointerException("Transformer must not be null.");  
1030 }  
1031 return IteratorUtils.toString(emptyIteratorIfNull(iterable), transformer);  
1032 }  
1033  
1034 /\*\*  
1035 \* Returns a string representation of the elements of the specified iterable.  
1036 \* <p>  
1037 \* The string representation consists of a list of the iterable's elements,  
1038 \* enclosed by the provided {@code prefix} and {@code suffix}. Adjacent elements  
1039 \* are separated by the provided {@code delimiter}. Elements are converted to  
1040 \* strings as by using the provided {@code transformer}.  
1041 \*  
1042 \* @param <E> the element type  
1043 \* @param iterable the iterable to convert to a string, may be null  
1044 \* @param transformer the transformer used to get a string representation of an element  
1045 \* @param delimiter the string to delimit elements  
1046 \* @param prefix the prefix, prepended to the string representation  
1047 \* @param suffix the suffix, appended to the string representation  
1048 \* @return a string representation of {@code iterable}  
1049 \* @throws NullPointerException if either transformer, delimiter, prefix or suffix is null  
1050 \*/  
1051 public static <E> String toString(final Iterable<E> iterable,  
1052 final Transformer<? super E, String> transformer,  
1053 final String delimiter,  
1054 final String prefix,  
1055 final String suffix) {  
1056 return IteratorUtils.toString(emptyIteratorIfNull(iterable),  
1057 transformer, delimiter, prefix, suffix);  
1058 }  
1059  
1060 // Helper methods  
1061 // ----------------------------------------------------------------------  
1062  
1063 /\*\*  
1064 \* Fail-fast check for null arguments.  
1065 \*  
1066 \* @param iterable the iterable to check  
1067 \* @throws NullPointerException if iterable is null  
1068 \*/  
1069 static void checkNotNull(final Iterable<?> iterable) {  
1070 if (iterable == null) {  
1071 throw new NullPointerException("Iterable must not be null.");  
1072 }  
1073 }  
1074  
1075 /\*\*  
1076 \* Fail-fast check for null arguments.  
1077 \*  
1078 \* @param iterables the iterables to check  
1079 \* @throws NullPointerException if the argument or any of its contents is null  
1080 \*/  
1081 static void checkNotNull(final Iterable<?>... iterables) {  
1082 if (iterables == null) {  
1083 throw new NullPointerException("Iterables must not be null.");  
1084 }  
1085 for (final Iterable<?> iterable : iterables) {  
1086 checkNotNull(iterable);  
1087 }  
1088 }  
1089  
1090 /\*\*  
1091 \* Returns an empty iterator if the argument is <code>null</code>,  
1092 \* or {@code iterable.iterator()} otherwise.  
1093 \*  
1094 \* @param <E> the element type  
1095 \* @param iterable the iterable, possibly <code>null</code>  
1096 \* @return an empty iterator if the argument is <code>null</code>  
1097 \*/  
1098 private static <E> Iterator<E> emptyIteratorIfNull(final Iterable<E> iterable) {  
1099 return iterable != null ? iterable.iterator() : IteratorUtils.<E>emptyIterator();  
1100 }  
1101  
1102}