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.lang.reflect.InvocationTargetException;  
021import java.lang.reflect.Method;  
022import java.util.ArrayList;  
023import java.util.Collection;  
024import java.util.Comparator;  
025import java.util.Dictionary;  
026import java.util.Enumeration;  
027import java.util.Iterator;  
028import java.util.List;  
029import java.util.ListIterator;  
030import java.util.Map;  
031  
032import org.apache.commons.collections4.functors.EqualPredicate;  
033import org.apache.commons.collections4.iterators.ArrayIterator;  
034import org.apache.commons.collections4.iterators.ArrayListIterator;  
035import org.apache.commons.collections4.iterators.BoundedIterator;  
036import org.apache.commons.collections4.iterators.CollatingIterator;  
037import org.apache.commons.collections4.iterators.EmptyIterator;  
038import org.apache.commons.collections4.iterators.EmptyListIterator;  
039import org.apache.commons.collections4.iterators.EmptyMapIterator;  
040import org.apache.commons.collections4.iterators.EmptyOrderedIterator;  
041import org.apache.commons.collections4.iterators.EmptyOrderedMapIterator;  
042import org.apache.commons.collections4.iterators.EnumerationIterator;  
043import org.apache.commons.collections4.iterators.FilterIterator;  
044import org.apache.commons.collections4.iterators.FilterListIterator;  
045import org.apache.commons.collections4.iterators.IteratorChain;  
046import org.apache.commons.collections4.iterators.IteratorEnumeration;  
047import org.apache.commons.collections4.iterators.IteratorIterable;  
048import org.apache.commons.collections4.iterators.ListIteratorWrapper;  
049import org.apache.commons.collections4.iterators.LoopingIterator;  
050import org.apache.commons.collections4.iterators.LoopingListIterator;  
051import org.apache.commons.collections4.iterators.NodeListIterator;  
052import org.apache.commons.collections4.iterators.ObjectArrayIterator;  
053import org.apache.commons.collections4.iterators.ObjectArrayListIterator;  
054import org.apache.commons.collections4.iterators.ObjectGraphIterator;  
055import org.apache.commons.collections4.iterators.PeekingIterator;  
056import org.apache.commons.collections4.iterators.PushbackIterator;  
057import org.apache.commons.collections4.iterators.SingletonIterator;  
058import org.apache.commons.collections4.iterators.SingletonListIterator;  
059import org.apache.commons.collections4.iterators.SkippingIterator;  
060import org.apache.commons.collections4.iterators.TransformIterator;  
061import org.apache.commons.collections4.iterators.UnmodifiableIterator;  
062import org.apache.commons.collections4.iterators.UnmodifiableListIterator;  
063import org.apache.commons.collections4.iterators.UnmodifiableMapIterator;  
064import org.apache.commons.collections4.iterators.ZippingIterator;  
065import org.w3c.dom.Node;  
066import org.w3c.dom.NodeList;  
067  
068/\*\*  
069 \* Provides static utility methods and decorators for {@link Iterator}  
070 \* instances. The implementations are provided in the iterators subpackage.  
071 \*  
072 \* @since 2.1  
073 \*/  
074public class IteratorUtils {  
075 // validation is done in this class in certain cases because the  
076 // public classes allow invalid states  
077  
078 /\*\*  
079 \* An iterator over no elements.  
080 \*/  
081 @SuppressWarnings("rawtypes")  
082 public static final ResettableIterator EMPTY\_ITERATOR = EmptyIterator.RESETTABLE\_INSTANCE;  
083  
084 /\*\*  
085 \* A list iterator over no elements.  
086 \*/  
087 @SuppressWarnings("rawtypes")  
088 public static final ResettableListIterator EMPTY\_LIST\_ITERATOR = EmptyListIterator.RESETTABLE\_INSTANCE;  
089  
090 /\*\*  
091 \* An ordered iterator over no elements.  
092 \*/  
093 @SuppressWarnings("rawtypes")  
094 public static final OrderedIterator EMPTY\_ORDERED\_ITERATOR = EmptyOrderedIterator.INSTANCE;  
095  
096 /\*\*  
097 \* A map iterator over no elements.  
098 \*/  
099 @SuppressWarnings("rawtypes")  
100 public static final MapIterator EMPTY\_MAP\_ITERATOR = EmptyMapIterator.INSTANCE;  
101  
102 /\*\*  
103 \* An ordered map iterator over no elements.  
104 \*/  
105 @SuppressWarnings("rawtypes")  
106 public static final OrderedMapIterator EMPTY\_ORDERED\_MAP\_ITERATOR = EmptyOrderedMapIterator.INSTANCE;  
107  
108 /\*\*  
109 \* Default prefix used while converting an Iterator to its String representation.  
110 \*/  
111 private static final String DEFAULT\_TOSTRING\_PREFIX = "[";  
112  
113 /\*\*  
114 \* Default suffix used while converting an Iterator to its String representation.  
115 \*/  
116 private static final String DEFAULT\_TOSTRING\_SUFFIX = "]";  
117  
118 /\*\*  
119 \* Default delimiter used to delimit elements while converting an Iterator  
120 \* to its String representation.  
121 \*/  
122 private static final String DEFAULT\_TOSTRING\_DELIMITER = ", ";  
123  
124 /\*\*  
125 \* IteratorUtils is not normally instantiated.  
126 \*/  
127 private IteratorUtils() {}  
128  
129 // Empty  
130 //-----------------------------------------------------------------------  
131 /\*\*  
132 \* Gets an empty iterator.  
133 \* <p>  
134 \* This iterator is a valid iterator object that will iterate over nothing.  
135 \*  
136 \* @param <E> the element type  
137 \* @return an iterator over nothing  
138 \*/  
139 public static <E> ResettableIterator<E> emptyIterator() {  
140 return EmptyIterator.<E>resettableEmptyIterator();  
141 }  
142  
143 /\*\*  
144 \* Gets an empty list iterator.  
145 \* <p>  
146 \* This iterator is a valid list iterator object that will iterate  
147 \* over nothing.  
148 \*  
149 \* @param <E> the element type  
150 \* @return a list iterator over nothing  
151 \*/  
152 public static <E> ResettableListIterator<E> emptyListIterator() {  
153 return EmptyListIterator.<E>resettableEmptyListIterator();  
154 }  
155  
156 /\*\*  
157 \* Gets an empty ordered iterator.  
158 \* <p>  
159 \* This iterator is a valid iterator object that will iterate  
160 \* over nothing.  
161 \*  
162 \* @param <E> the element type  
163 \* @return an ordered iterator over nothing  
164 \*/  
165 public static <E> OrderedIterator<E> emptyOrderedIterator() {  
166 return EmptyOrderedIterator.<E>emptyOrderedIterator();  
167 }  
168  
169 /\*\*  
170 \* Gets an empty map iterator.  
171 \* <p>  
172 \* This iterator is a valid map iterator object that will iterate  
173 \* over nothing.  
174 \*  
175 \* @param <K> the key type  
176 \* @param <V> the value type  
177 \* @return a map iterator over nothing  
178 \*/  
179 public static <K, V> MapIterator<K, V> emptyMapIterator() {  
180 return EmptyMapIterator.<K, V>emptyMapIterator();  
181 }  
182  
183 /\*\*  
184 \* Gets an empty ordered map iterator.  
185 \* <p>  
186 \* This iterator is a valid map iterator object that will iterate  
187 \* over nothing.  
188 \*  
189 \* @param <K> the key type  
190 \* @param <V> the value type  
191 \* @return a map iterator over nothing  
192 \*/  
193 public static <K, V> OrderedMapIterator<K, V> emptyOrderedMapIterator() {  
194 return EmptyOrderedMapIterator.<K, V>emptyOrderedMapIterator();  
195 }  
196  
197 // Singleton  
198 //-----------------------------------------------------------------------  
199 /\*\*  
200 \* Gets a singleton iterator.  
201 \* <p>  
202 \* This iterator is a valid iterator object that will iterate over  
203 \* the specified object.  
204 \*  
205 \* @param <E> the element type  
206 \* @param object the single object over which to iterate  
207 \* @return a singleton iterator over the object  
208 \*/  
209 public static <E> ResettableIterator<E> singletonIterator(final E object) {  
210 return new SingletonIterator<>(object);  
211 }  
212  
213 /\*\*  
214 \* Gets a singleton list iterator.  
215 \* <p>  
216 \* This iterator is a valid list iterator object that will iterate over  
217 \* the specified object.  
218 \*  
219 \* @param <E> the element type  
220 \* @param object the single object over which to iterate  
221 \* @return a singleton list iterator over the object  
222 \*/  
223 public static <E> ListIterator<E> singletonListIterator(final E object) {  
224 return new SingletonListIterator<>(object);  
225 }  
226  
227 // Arrays  
228 //-----------------------------------------------------------------------  
229 /\*\*  
230 \* Gets an iterator over an object array.  
231 \*  
232 \* @param <E> the element type  
233 \* @param array the array over which to iterate  
234 \* @return an iterator over the array  
235 \* @throws NullPointerException if array is null  
236 \*/  
237 public static <E> ResettableIterator<E> arrayIterator(final E... array) {  
238 return new ObjectArrayIterator<>(array);  
239 }  
240  
241 /\*\*  
242 \* Gets an iterator over an object or primitive array.  
243 \* <p>  
244 \* This method will handle primitive arrays as well as object arrays.  
245 \* The primitives will be wrapped in the appropriate wrapper class.  
246 \*  
247 \* @param <E> the element type  
248 \* @param array the array over which to iterate  
249 \* @return an iterator over the array  
250 \* @throws IllegalArgumentException if the array is not an array  
251 \* @throws NullPointerException if array is null  
252 \*/  
253 public static <E> ResettableIterator<E> arrayIterator(final Object array) {  
254 return new ArrayIterator<>(array);  
255 }  
256  
257 /\*\*  
258 \* Gets an iterator over the end part of an object array.  
259 \*  
260 \* @param <E> the element type  
261 \* @param array the array over which to iterate  
262 \* @param start the index to start iterating at  
263 \* @return an iterator over part of the array  
264 \* @throws IndexOutOfBoundsException if start is less than zero or greater  
265 \* than the length of the array  
266 \* @throws NullPointerException if array is null  
267 \*/  
268 public static <E> ResettableIterator<E> arrayIterator(final E[] array, final int start) {  
269 return new ObjectArrayIterator<>(array, start);  
270 }  
271  
272 /\*\*  
273 \* Gets an iterator over the end part of an object or primitive array.  
274 \* <p>  
275 \* This method will handle primitive arrays as well as object arrays.  
276 \* The primitives will be wrapped in the appropriate wrapper class.  
277 \*  
278 \* @param <E> the element type  
279 \* @param array the array over which to iterate  
280 \* @param start the index to start iterating at  
281 \* @return an iterator over part of the array  
282 \* @throws IllegalArgumentException if the array is not an array  
283 \* @throws IndexOutOfBoundsException if start is less than zero or greater  
284 \* than the length of the array  
285 \* @throws NullPointerException if array is null  
286 \*/  
287 public static <E> ResettableIterator<E> arrayIterator(final Object array, final int start) {  
288 return new ArrayIterator<>(array, start);  
289 }  
290  
291 /\*\*  
292 \* Gets an iterator over part of an object array.  
293 \*  
294 \* @param <E> the element type  
295 \* @param array the array over which to iterate  
296 \* @param start the index to start iterating at  
297 \* @param end the index to finish iterating at  
298 \* @return an iterator over part of the array  
299 \* @throws IndexOutOfBoundsException if array bounds are invalid  
300 \* @throws IllegalArgumentException if end is before start  
301 \* @throws NullPointerException if array is null  
302 \*/  
303 public static <E> ResettableIterator<E> arrayIterator(final E[] array, final int start, final int end) {  
304 return new ObjectArrayIterator<>(array, start, end);  
305 }  
306  
307 /\*\*  
308 \* Gets an iterator over part of an object or primitive array.  
309 \* <p>  
310 \* This method will handle primitive arrays as well as object arrays.  
311 \* The primitives will be wrapped in the appropriate wrapper class.  
312 \*  
313 \* @param <E> the element type  
314 \* @param array the array over which to iterate  
315 \* @param start the index to start iterating at  
316 \* @param end the index to finish iterating at  
317 \* @return an iterator over part of the array  
318 \* @throws IllegalArgumentException if the array is not an array or end is before start  
319 \* @throws IndexOutOfBoundsException if array bounds are invalid  
320 \* @throws NullPointerException if array is null  
321 \*/  
322 public static <E> ResettableIterator<E> arrayIterator(final Object array, final int start, final int end) {  
323 return new ArrayIterator<>(array, start, end);  
324 }  
325  
326 //-----------------------------------------------------------------------  
327 /\*\*  
328 \* Gets a list iterator over an object array.  
329 \*  
330 \* @param <E> the element type  
331 \* @param array the array over which to iterate  
332 \* @return a list iterator over the array  
333 \* @throws NullPointerException if array is null  
334 \*/  
335 public static <E> ResettableListIterator<E> arrayListIterator(final E... array) {  
336 return new ObjectArrayListIterator<>(array);  
337 }  
338  
339 /\*\*  
340 \* Gets a list iterator over an object or primitive array.  
341 \* <p>  
342 \* This method will handle primitive arrays as well as object arrays.  
343 \* The primitives will be wrapped in the appropriate wrapper class.  
344 \*  
345 \* @param <E> the element type  
346 \* @param array the array over which to iterate  
347 \* @return a list iterator over the array  
348 \* @throws IllegalArgumentException if the array is not an array  
349 \* @throws NullPointerException if array is null  
350 \*/  
351 public static <E> ResettableListIterator<E> arrayListIterator(final Object array) {  
352 return new ArrayListIterator<>(array);  
353 }  
354  
355 /\*\*  
356 \* Gets a list iterator over the end part of an object array.  
357 \*  
358 \* @param <E> the element type  
359 \* @param array the array over which to iterate  
360 \* @param start the index to start iterating at  
361 \* @return a list iterator over part of the array  
362 \* @throws IndexOutOfBoundsException if start is less than zero  
363 \* @throws NullPointerException if array is null  
364 \*/  
365 public static <E> ResettableListIterator<E> arrayListIterator(final E[] array, final int start) {  
366 return new ObjectArrayListIterator<>(array, start);  
367 }  
368  
369 /\*\*  
370 \* Gets a list iterator over the end part of an object or primitive array.  
371 \* <p>  
372 \* This method will handle primitive arrays as well as object arrays.  
373 \* The primitives will be wrapped in the appropriate wrapper class.  
374 \*  
375 \* @param <E> the element type  
376 \* @param array the array over which to iterate  
377 \* @param start the index to start iterating at  
378 \* @return a list iterator over part of the array  
379 \* @throws IllegalArgumentException if the array is not an array  
380 \* @throws IndexOutOfBoundsException if start is less than zero  
381 \* @throws NullPointerException if array is null  
382 \*/  
383 public static <E> ResettableListIterator<E> arrayListIterator(final Object array, final int start) {  
384 return new ArrayListIterator<>(array, start);  
385 }  
386  
387 /\*\*  
388 \* Gets a list iterator over part of an object array.  
389 \*  
390 \* @param <E> the element type  
391 \* @param array the array over which to iterate  
392 \* @param start the index to start iterating at  
393 \* @param end the index to finish iterating at  
394 \* @return a list iterator over part of the array  
395 \* @throws IndexOutOfBoundsException if array bounds are invalid  
396 \* @throws IllegalArgumentException if end is before start  
397 \* @throws NullPointerException if array is null  
398 \*/  
399 public static <E> ResettableListIterator<E> arrayListIterator(final E[] array, final int start, final int end) {  
400 return new ObjectArrayListIterator<>(array, start, end);  
401 }  
402  
403 /\*\*  
404 \* Gets a list iterator over part of an object or primitive array.  
405 \* <p>  
406 \* This method will handle primitive arrays as well as object arrays.  
407 \* The primitives will be wrapped in the appropriate wrapper class.  
408 \*  
409 \* @param <E> the element type  
410 \* @param array the array over which to iterate  
411 \* @param start the index to start iterating at  
412 \* @param end the index to finish iterating at  
413 \* @return a list iterator over part of the array  
414 \* @throws IllegalArgumentException if the array is not an array or end is before start  
415 \* @throws IndexOutOfBoundsException if array bounds are invalid  
416 \* @throws NullPointerException if array is null  
417 \*/  
418 public static <E> ResettableListIterator<E> arrayListIterator(final Object array, final int start, final int end) {  
419 return new ArrayListIterator<>(array, start, end);  
420 }  
421  
422 // Bounded  
423 //-----------------------------------------------------------------------  
424 /\*\*  
425 \* Decorates the specified iterator to return at most the given number  
426 \* of elements.  
427 \*  
428 \* @param <E> the element type  
429 \* @param iterator the iterator to decorate  
430 \* @param max the maximum number of elements returned by this iterator  
431 \* @return a new bounded iterator  
432 \* @throws NullPointerException if the iterator is null  
433 \* @throws IllegalArgumentException if max is negative  
434 \* @since 4.1  
435 \*/  
436 public static <E> BoundedIterator<E> boundedIterator(final Iterator<? extends E> iterator, final long max) {  
437 return boundedIterator(iterator, 0, max);  
438 }  
439  
440 /\*\*  
441 \* Decorates the specified iterator to return at most the given number  
442 \* of elements, skipping all elements until the iterator reaches the  
443 \* position at {@code offset}.  
444 \* <p>  
445 \* The iterator is immediately advanced until it reaches the position at  
446 \* {@code offset}, incurring O(n) time.  
447 \*  
448 \* @param <E> the element type  
449 \* @param iterator the iterator to decorate  
450 \* @param offset the index of the first element of the decorated iterator to return  
451 \* @param max the maximum number of elements returned by this iterator  
452 \* @return a new bounded iterator  
453 \* @throws NullPointerException if the iterator is null  
454 \* @throws IllegalArgumentException if either offset or max is negative  
455 \* @since 4.1  
456 \*/  
457 public static <E> BoundedIterator<E> boundedIterator(final Iterator<? extends E> iterator,  
458 final long offset, final long max) {  
459 return new BoundedIterator<>(iterator, offset, max);  
460 }  
461  
462 // Unmodifiable  
463 //-----------------------------------------------------------------------  
464 /\*\*  
465 \* Gets an immutable version of an {@link Iterator}. The returned object  
466 \* will always throw an {@link UnsupportedOperationException} for  
467 \* the {@link Iterator#remove} method.  
468 \*  
469 \* @param <E> the element type  
470 \* @param iterator the iterator to make immutable  
471 \* @return an immutable version of the iterator  
472 \*/  
473 public static <E> Iterator<E> unmodifiableIterator(final Iterator<E> iterator) {  
474 return UnmodifiableIterator.unmodifiableIterator(iterator);  
475 }  
476  
477 /\*\*  
478 \* Gets an immutable version of a {@link ListIterator}. The returned object  
479 \* will always throw an {@link UnsupportedOperationException} for  
480 \* the {@link Iterator#remove}, {@link ListIterator#add} and  
481 \* {@link ListIterator#set} methods.  
482 \*  
483 \* @param <E> the element type  
484 \* @param listIterator the iterator to make immutable  
485 \* @return an immutable version of the iterator  
486 \*/  
487 public static <E> ListIterator<E> unmodifiableListIterator(final ListIterator<E> listIterator) {  
488 return UnmodifiableListIterator.umodifiableListIterator(listIterator);  
489 }  
490  
491 /\*\*  
492 \* Gets an immutable version of a {@link MapIterator}. The returned object  
493 \* will always throw an {@link UnsupportedOperationException} for  
494 \* the {@link Iterator#remove}, {@link MapIterator#setValue(Object)} methods.  
495 \*  
496 \* @param <K> the key type  
497 \* @param <V> the value type  
498 \* @param mapIterator the iterator to make immutable  
499 \* @return an immutable version of the iterator  
500 \*/  
501 public static <K, V> MapIterator<K, V> unmodifiableMapIterator(final MapIterator<K, V> mapIterator) {  
502 return UnmodifiableMapIterator.unmodifiableMapIterator(mapIterator);  
503 }  
504  
505 // Chained  
506 //-----------------------------------------------------------------------  
507  
508 /\*\*  
509 \* Gets an iterator that iterates through two {@link Iterator}s  
510 \* one after another.  
511 \*  
512 \* @param <E> the element type  
513 \* @param iterator1 the first iterator to use, not null  
514 \* @param iterator2 the second iterator to use, not null  
515 \* @return a combination iterator over the iterators  
516 \* @throws NullPointerException if either iterator is null  
517 \*/  
518 public static <E> Iterator<E> chainedIterator(final Iterator<? extends E> iterator1,  
519 final Iterator<? extends E> iterator2) {  
520 // keep a version with two iterators to avoid the following warning in client code (Java 5 & 6)  
521 // "A generic array of E is created for a varargs parameter"  
522 return new IteratorChain<>(iterator1, iterator2);  
523 }  
524  
525 /\*\*  
526 \* Gets an iterator that iterates through an array of {@link Iterator}s  
527 \* one after another.  
528 \*  
529 \* @param <E> the element type  
530 \* @param iterators the iterators to use, not null or empty or contain nulls  
531 \* @return a combination iterator over the iterators  
532 \* @throws NullPointerException if iterators array is null or contains a null  
533 \*/  
534 public static <E> Iterator<E> chainedIterator(final Iterator<? extends E>... iterators) {  
535 return new IteratorChain<>(iterators);  
536 }  
537  
538 /\*\*  
539 \* Gets an iterator that iterates through a collections of {@link Iterator}s  
540 \* one after another.  
541 \*  
542 \* @param <E> the element type  
543 \* @param iterators the iterators to use, not null or empty or contain nulls  
544 \* @return a combination iterator over the iterators  
545 \* @throws NullPointerException if iterators collection is null or contains a null  
546 \* @throws ClassCastException if the iterators collection contains the wrong object type  
547 \*/  
548 public static <E> Iterator<E> chainedIterator(final Collection<Iterator<? extends E>> iterators) {  
549 return new IteratorChain<>(iterators);  
550 }  
551  
552 // Collated  
553 //-----------------------------------------------------------------------  
554 /\*\*  
555 \* Gets an iterator that provides an ordered iteration over the elements  
556 \* contained in a collection of ordered {@link Iterator}s.  
557 \* <p>  
558 \* Given two ordered {@link Iterator}s <code>A</code> and <code>B</code>,  
559 \* the {@link Iterator#next()} method will return the lesser of  
560 \* <code>A.next()</code> and <code>B.next()</code>.  
561 \* <p>  
562 \* The comparator is optional. If null is specified then natural order is used.  
563 \*  
564 \* @param <E> the element type  
565 \* @param comparator the comparator to use, may be null for natural order  
566 \* @param iterator1 the first iterators to use, not null  
567 \* @param iterator2 the first iterators to use, not null  
568 \* @return a combination iterator over the iterators  
569 \* @throws NullPointerException if either iterator is null  
570 \*/  
571 public static <E> Iterator<E> collatedIterator(final Comparator<? super E> comparator,  
572 final Iterator<? extends E> iterator1,  
573 final Iterator<? extends E> iterator2) {  
574 @SuppressWarnings("unchecked")  
575 final Comparator<E> comp =  
576 comparator == null ? ComparatorUtils.NATURAL\_COMPARATOR : (Comparator<E>) comparator;  
577 return new CollatingIterator<>(comp, iterator1, iterator2);  
578 }  
579  
580 /\*\*  
581 \* Gets an iterator that provides an ordered iteration over the elements  
582 \* contained in an array of {@link Iterator}s.  
583 \* <p>  
584 \* Given two ordered {@link Iterator}s <code>A</code> and <code>B</code>,  
585 \* the {@link Iterator#next()} method will return the lesser of  
586 \* <code>A.next()</code> and <code>B.next()</code> and so on.  
587 \* <p>  
588 \* The comparator is optional. If null is specified then natural order is used.  
589 \*  
590 \* @param <E> the element type  
591 \* @param comparator the comparator to use, may be null for natural order  
592 \* @param iterators the iterators to use, not null or empty or contain nulls  
593 \* @return a combination iterator over the iterators  
594 \* @throws NullPointerException if iterators array is null or contains a null value  
595 \*/  
596 public static <E> Iterator<E> collatedIterator(final Comparator<? super E> comparator,  
597 final Iterator<? extends E>... iterators) {  
598 @SuppressWarnings("unchecked")  
599 final Comparator<E> comp =  
600 comparator == null ? ComparatorUtils.NATURAL\_COMPARATOR : (Comparator<E>) comparator;  
601 return new CollatingIterator<>(comp, iterators);  
602 }  
603  
604 /\*\*  
605 \* Gets an iterator that provides an ordered iteration over the elements  
606 \* contained in a collection of {@link Iterator}s.  
607 \* <p>  
608 \* Given two ordered {@link Iterator}s <code>A</code> and <code>B</code>,  
609 \* the {@link Iterator#next()} method will return the lesser of  
610 \* <code>A.next()</code> and <code>B.next()</code> and so on.  
611 \* <p>  
612 \* The comparator is optional. If null is specified then natural order is used.  
613 \*  
614 \* @param <E> the element type  
615 \* @param comparator the comparator to use, may be null for natural order  
616 \* @param iterators the iterators to use, not null or empty or contain nulls  
617 \* @return a combination iterator over the iterators  
618 \* @throws NullPointerException if iterators collection is null or contains a null  
619 \* @throws ClassCastException if the iterators collection contains the wrong object type  
620 \*/  
621 public static <E> Iterator<E> collatedIterator(final Comparator<? super E> comparator,  
622 final Collection<Iterator<? extends E>> iterators) {  
623 @SuppressWarnings("unchecked")  
624 final Comparator<E> comp =  
625 comparator == null ? ComparatorUtils.NATURAL\_COMPARATOR : (Comparator<E>) comparator;  
626 return new CollatingIterator<>(comp, iterators);  
627 }  
628  
629 // Object Graph  
630 //-----------------------------------------------------------------------  
631 /\*\*  
632 \* Gets an iterator that operates over an object graph.  
633 \* <p>  
634 \* This iterator can extract multiple objects from a complex tree-like object graph.  
635 \* The iteration starts from a single root object.  
636 \* It uses a <code>Transformer</code> to extract the iterators and elements.  
637 \* Its main benefit is that no intermediate <code>List</code> is created.  
638 \* <p>  
639 \* For example, consider an object graph:  
640 \* <pre>  
641 \* |- Branch -- Leaf  
642 \* | \- Leaf  
643 \* |- Tree | /- Leaf  
644 \* | |- Branch -- Leaf  
645 \* Forest | \- Leaf  
646 \* | |- Branch -- Leaf  
647 \* | | \- Leaf  
648 \* |- Tree | /- Leaf  
649 \* |- Branch -- Leaf  
650 \* |- Branch -- Leaf</pre>  
651 \* The following <code>Transformer</code>, used in this class, will extract all  
652 \* the Leaf objects without creating a combined intermediate list:  
653 \* <pre>  
654 \* public Object transform(Object input) {  
655 \* if (input instanceof Forest) {  
656 \* return ((Forest) input).treeIterator();  
657 \* }  
658 \* if (input instanceof Tree) {  
659 \* return ((Tree) input).branchIterator();  
660 \* }  
661 \* if (input instanceof Branch) {  
662 \* return ((Branch) input).leafIterator();  
663 \* }  
664 \* if (input instanceof Leaf) {  
665 \* return input;  
666 \* }  
667 \* throw new ClassCastException();  
668 \* }</pre>  
669 \* <p>  
670 \* Internally, iteration starts from the root object. When next is called,  
671 \* the transformer is called to examine the object. The transformer will return  
672 \* either an iterator or an object. If the object is an Iterator, the next element  
673 \* from that iterator is obtained and the process repeats. If the element is an object  
674 \* it is returned.  
675 \* <p>  
676 \* Under many circumstances, linking Iterators together in this manner is  
677 \* more efficient (and convenient) than using nested for loops to extract a list.  
678 \*  
679 \* @param <E> the element type  
680 \* @param root the root object to start iterating from, null results in an empty iterator  
681 \* @param transformer the transformer to use, see above, null uses no effect transformer  
682 \* @return a new object graph iterator  
683 \* @since 3.1  
684 \*/  
685 public static <E> Iterator<E> objectGraphIterator(final E root,  
686 final Transformer<? super E, ? extends E> transformer) {  
687 return new ObjectGraphIterator<>(root, transformer);  
688 }  
689  
690 // Transformed  
691 //-----------------------------------------------------------------------  
692 /\*\*  
693 \* Gets an iterator that transforms the elements of another iterator.  
694 \* <p>  
695 \* The transformation occurs during the next() method and the underlying  
696 \* iterator is unaffected by the transformation.  
697 \*  
698 \* @param <I> the input type  
699 \* @param <O> the output type  
700 \* @param iterator the iterator to use, not null  
701 \* @param transform the transform to use, not null  
702 \* @return a new transforming iterator  
703 \* @throws NullPointerException if either parameter is null  
704 \*/  
705 public static <I, O> Iterator<O> transformedIterator(final Iterator<? extends I> iterator,  
706 final Transformer<? super I, ? extends O> transform) {  
707  
708 if (iterator == null) {  
709 throw new NullPointerException("Iterator must not be null");  
710 }  
711 if (transform == null) {  
712 throw new NullPointerException("Transformer must not be null");  
713 }  
714 return new TransformIterator<>(iterator, transform);  
715 }  
716  
717 // Filtered  
718 //-----------------------------------------------------------------------  
719 /\*\*  
720 \* Gets an iterator that filters another iterator.  
721 \* <p>  
722 \* The returned iterator will only return objects that match the specified  
723 \* filtering predicate.  
724 \*  
725 \* @param <E> the element type  
726 \* @param iterator the iterator to use, not null  
727 \* @param predicate the predicate to use as a filter, not null  
728 \* @return a new filtered iterator  
729 \* @throws NullPointerException if either parameter is null  
730 \*/  
731 public static <E> Iterator<E> filteredIterator(final Iterator<? extends E> iterator,  
732 final Predicate<? super E> predicate) {  
733 if (iterator == null) {  
734 throw new NullPointerException("Iterator must not be null");  
735 }  
736 if (predicate == null) {  
737 throw new NullPointerException("Predicate must not be null");  
738 }  
739 return new FilterIterator<>(iterator, predicate);  
740 }  
741  
742 /\*\*  
743 \* Gets a list iterator that filters another list iterator.  
744 \* <p>  
745 \* The returned iterator will only return objects that match the specified  
746 \* filtering predicate.  
747 \*  
748 \* @param <E> the element type  
749 \* @param listIterator the list iterator to use, not null  
750 \* @param predicate the predicate to use as a filter, not null  
751 \* @return a new filtered iterator  
752 \* @throws NullPointerException if either parameter is null  
753 \*/  
754 public static <E> ListIterator<E> filteredListIterator(final ListIterator<? extends E> listIterator,  
755 final Predicate<? super E> predicate) {  
756  
757 if (listIterator == null) {  
758 throw new NullPointerException("ListIterator must not be null");  
759 }  
760 if (predicate == null) {  
761 throw new NullPointerException("Predicate must not be null");  
762 }  
763 return new FilterListIterator<>(listIterator, predicate);  
764 }  
765  
766 // Looping  
767 //-----------------------------------------------------------------------  
768 /\*\*  
769 \* Gets an iterator that loops continuously over the supplied collection.  
770 \* <p>  
771 \* The iterator will only stop looping if the remove method is called  
772 \* enough times to empty the collection, or if the collection is empty  
773 \* to start with.  
774 \*  
775 \* @param <E> the element type  
776 \* @param coll the collection to iterate over, not null  
777 \* @return a new looping iterator  
778 \* @throws NullPointerException if the collection is null  
779 \*/  
780 public static <E> ResettableIterator<E> loopingIterator(final Collection<? extends E> coll) {  
781 if (coll == null) {  
782 throw new NullPointerException("Collection must not be null");  
783 }  
784 return new LoopingIterator<>(coll);  
785 }  
786  
787 /\*\*  
788 \* Gets an iterator that loops continuously over the supplied list.  
789 \* <p>  
790 \* The iterator will only stop looping if the remove method is called  
791 \* enough times to empty the list, or if the list is empty to start with.  
792 \*  
793 \* @param <E> the element type  
794 \* @param list the list to iterate over, not null  
795 \* @return a new looping iterator  
796 \* @throws NullPointerException if the list is null  
797 \* @since 3.2  
798 \*/  
799 public static <E> ResettableListIterator<E> loopingListIterator(final List<E> list) {  
800 if (list == null) {  
801 throw new NullPointerException("List must not be null");  
802 }  
803 return new LoopingListIterator<>(list);  
804 }  
805  
806 // org.w3c.dom.NodeList iterators  
807 //-----------------------------------------------------------------------  
808 /\*\*  
809 \* Gets an {@link Iterator} that wraps the specified {@link NodeList}.  
810 \* The returned {@link Iterator} can be used for a single iteration.  
811 \*  
812 \* @param nodeList the node list to use, may not be null  
813 \* @return a new, single use {@link Iterator}  
814 \* @throws NullPointerException if nodeList is null  
815 \* @since 4.0  
816 \*/  
817 public static NodeListIterator nodeListIterator(final NodeList nodeList) {  
818 if (nodeList == null) {  
819 throw new NullPointerException("NodeList must not be null");  
820 }  
821 return new NodeListIterator(nodeList);  
822 }  
823  
824 /\*\*  
825 \* Gets an {@link Iterator} that wraps the specified node's childNodes.  
826 \* The returned {@link Iterator} can be used for a single iteration.  
827 \* <p>  
828 \* Convenience method, allows easy iteration over NodeLists:  
829 \* <pre>  
830 \* Iterator<Node> iterator = IteratorUtils.nodeListIterator(node);  
831 \* for(Node childNode : IteratorUtils.asIterable(iterator)) {  
832 \* ...  
833 \* }  
834 \* </pre>  
835 \*  
836 \* @param node the node to use, may not be null  
837 \* @return a new, single use {@link Iterator}  
838 \* @throws NullPointerException if node is null  
839 \* @since 4.0  
840 \*/  
841 public static NodeListIterator nodeListIterator(final Node node) {  
842 if (node == null) {  
843 throw new NullPointerException("Node must not be null");  
844 }  
845 return new NodeListIterator(node);  
846 }  
847  
848 // Peeking  
849 //-----------------------------------------------------------------------  
850  
851 /\*\*  
852 \* Gets an iterator that supports one-element lookahead.  
853 \*  
854 \* @param <E> the element type  
855 \* @param iterator the iterator to decorate, not null  
856 \* @return a peeking iterator  
857 \* @throws NullPointerException if the iterator is null  
858 \* @since 4.0  
859 \*/  
860 public static <E> Iterator<E> peekingIterator(final Iterator<? extends E> iterator) {  
861 return PeekingIterator.peekingIterator(iterator);  
862 }  
863  
864 // Pushback  
865 //-----------------------------------------------------------------------  
866  
867 /\*\*  
868 \* Gets an iterator that supports pushback of elements.  
869 \*  
870 \* @param <E> the element type  
871 \* @param iterator the iterator to decorate, not null  
872 \* @return a pushback iterator  
873 \* @throws NullPointerException if the iterator is null  
874 \* @since 4.0  
875 \*/  
876 public static <E> Iterator<E> pushbackIterator(final Iterator<? extends E> iterator) {  
877 return PushbackIterator.pushbackIterator(iterator);  
878 }  
879  
880 // Skipping  
881 //-----------------------------------------------------------------------  
882 /\*\*  
883 \* Decorates the specified iterator to skip the first N elements.  
884 \*  
885 \* @param <E> the element type  
886 \* @param iterator the iterator to decorate  
887 \* @param offset the first number of elements to skip  
888 \* @return a new skipping iterator  
889 \* @throws NullPointerException if the iterator is null  
890 \* @throws IllegalArgumentException if offset is negative  
891 \* @since 4.1  
892 \*/  
893 public static <E> SkippingIterator<E> skippingIterator(final Iterator<E> iterator, final long offset) {  
894 return new SkippingIterator<>(iterator, offset);  
895 }  
896  
897 // Zipping  
898 //-----------------------------------------------------------------------  
899 /\*\*  
900 \* Returns an iterator that interleaves elements from the decorated iterators.  
901 \*  
902 \* @param <E> the element type  
903 \* @param a the first iterator to interleave  
904 \* @param b the second iterator to interleave  
905 \* @return an iterator, interleaving the decorated iterators  
906 \* @throws NullPointerException if any iterator is null  
907 \* @since 4.1  
908 \*/  
909 public static <E> ZippingIterator<E> zippingIterator(final Iterator<? extends E> a,  
910 final Iterator<? extends E> b) {  
911 return new ZippingIterator<>(a, b);  
912 }  
913  
914 /\*\*  
915 \* Returns an iterator that interleaves elements from the decorated iterators.  
916 \*  
917 \* @param <E> the element type  
918 \* @param a the first iterator to interleave  
919 \* @param b the second iterator to interleave  
920 \* @param c the third iterator to interleave  
921 \* @return an iterator, interleaving the decorated iterators  
922 \* @throws NullPointerException if any iterator is null  
923 \* @since 4.1  
924 \*/  
925 public static <E> ZippingIterator<E> zippingIterator(final Iterator<? extends E> a,  
926 final Iterator<? extends E> b,  
927 final Iterator<? extends E> c) {  
928 return new ZippingIterator<>(a, b, c);  
929 }  
930  
931 /\*\*  
932 \* Returns an iterator that interleaves elements from the decorated iterators.  
933 \*  
934 \* @param <E> the element type  
935 \* @param iterators the array of iterators to interleave  
936 \* @return an iterator, interleaving the decorated iterators  
937 \* @throws NullPointerException if any iterator is null  
938 \* @since 4.1  
939 \*/  
940 public static <E> ZippingIterator<E> zippingIterator(final Iterator<? extends E>... iterators) {  
941 return new ZippingIterator<>(iterators);  
942 }  
943  
944 // Views  
945 //-----------------------------------------------------------------------  
946 /\*\*  
947 \* Gets an iterator that provides an iterator view of the given enumeration.  
948 \*  
949 \* @param <E> the element type  
950 \* @param enumeration the enumeration to use, may not be null  
951 \* @return a new iterator  
952 \* @throws NullPointerException if enumeration is null  
953 \*/  
954 public static <E> Iterator<E> asIterator(final Enumeration<? extends E> enumeration) {  
955 if (enumeration == null) {  
956 throw new NullPointerException("Enumeration must not be null");  
957 }  
958 return new EnumerationIterator<>(enumeration);  
959 }  
960  
961 /\*\*  
962 \* Gets an iterator that provides an iterator view of the given enumeration  
963 \* that will remove elements from the specified collection.  
964 \*  
965 \* @param <E> the element type  
966 \* @param enumeration the enumeration to use, may not be null  
967 \* @param removeCollection the collection to remove elements from, may not be null  
968 \* @return a new iterator  
969 \* @throws NullPointerException if enumeration or removeCollection is null  
970 \*/  
971 public static <E> Iterator<E> asIterator(final Enumeration<? extends E> enumeration,  
972 final Collection<? super E> removeCollection) {  
973 if (enumeration == null) {  
974 throw new NullPointerException("Enumeration must not be null");  
975 }  
976 if (removeCollection == null) {  
977 throw new NullPointerException("Collection must not be null");  
978 }  
979 return new EnumerationIterator<>(enumeration, removeCollection);  
980 }  
981  
982 /\*\*  
983 \* Gets an enumeration that wraps an iterator.  
984 \*  
985 \* @param <E> the element type  
986 \* @param iterator the iterator to use, may not be null  
987 \* @return a new enumeration  
988 \* @throws NullPointerException if iterator is null  
989 \*/  
990 public static <E> Enumeration<E> asEnumeration(final Iterator<? extends E> iterator) {  
991 if (iterator == null) {  
992 throw new NullPointerException("Iterator must not be null");  
993 }  
994 return new IteratorEnumeration<>(iterator);  
995 }  
996  
997 /\*\*  
998 \* Gets an {@link Iterable} that wraps an iterator. The returned {@link Iterable} can be  
999 \* used for a single iteration.  
1000 \*  
1001 \* @param <E> the element type  
1002 \* @param iterator the iterator to use, may not be null  
1003 \* @return a new, single use {@link Iterable}  
1004 \* @throws NullPointerException if iterator is null  
1005 \*/  
1006 public static <E> Iterable<E> asIterable(final Iterator<? extends E> iterator) {  
1007 if (iterator == null) {  
1008 throw new NullPointerException("Iterator must not be null");  
1009 }  
1010 return new IteratorIterable<>(iterator, false);  
1011 }  
1012  
1013 /\*\*  
1014 \* Gets an iterable that wraps an iterator. The returned iterable can be  
1015 \* used for multiple iterations.  
1016 \*  
1017 \* @param <E> the element type  
1018 \* @param iterator the iterator to use, may not be null  
1019 \* @return a new, multiple use iterable  
1020 \* @throws NullPointerException if iterator is null  
1021 \*/  
1022 public static <E> Iterable<E> asMultipleUseIterable(final Iterator<? extends E> iterator) {  
1023 if (iterator == null) {  
1024 throw new NullPointerException("Iterator must not be null");  
1025 }  
1026 return new IteratorIterable<>(iterator, true);  
1027 }  
1028  
1029 /\*\*  
1030 \* Gets a list iterator based on a simple iterator.  
1031 \* <p>  
1032 \* As the wrapped Iterator is traversed, a LinkedList of its values is  
1033 \* cached, permitting all required operations of ListIterator.  
1034 \*  
1035 \* @param <E> the element type  
1036 \* @param iterator the iterator to use, may not be null  
1037 \* @return a new iterator  
1038 \* @throws NullPointerException if iterator parameter is null  
1039 \*/  
1040 public static <E> ListIterator<E> toListIterator(final Iterator<? extends E> iterator) {  
1041 if (iterator == null) {  
1042 throw new NullPointerException("Iterator must not be null");  
1043 }  
1044 return new ListIteratorWrapper<>(iterator);  
1045 }  
1046  
1047 /\*\*  
1048 \* Gets an array based on an iterator.  
1049 \* <p>  
1050 \* As the wrapped Iterator is traversed, an ArrayList of its values is  
1051 \* created. At the end, this is converted to an array.  
1052 \*  
1053 \* @param iterator the iterator to use, not null  
1054 \* @return an array of the iterator contents  
1055 \* @throws NullPointerException if iterator parameter is null  
1056 \*/  
1057 public static Object[] toArray(final Iterator<?> iterator) {  
1058 if (iterator == null) {  
1059 throw new NullPointerException("Iterator must not be null");  
1060 }  
1061 final List<?> list = toList(iterator, 100);  
1062 return list.toArray();  
1063 }  
1064  
1065 /\*\*  
1066 \* Gets an array based on an iterator.  
1067 \* <p>  
1068 \* As the wrapped Iterator is traversed, an ArrayList of its values is  
1069 \* created. At the end, this is converted to an array.  
1070 \*  
1071 \* @param <E> the element type  
1072 \* @param iterator the iterator to use, not null  
1073 \* @param arrayClass the class of array to create  
1074 \* @return an array of the iterator contents  
1075 \* @throws NullPointerException if iterator parameter or arrayClass is null  
1076 \* @throws ArrayStoreException if the arrayClass is invalid  
1077 \*/  
1078 public static <E> E[] toArray(final Iterator<? extends E> iterator, final Class<E> arrayClass) {  
1079 if (iterator == null) {  
1080 throw new NullPointerException("Iterator must not be null");  
1081 }  
1082 if (arrayClass == null) {  
1083 throw new NullPointerException("Array class must not be null");  
1084 }  
1085 final List<E> list = toList(iterator, 100);  
1086 @SuppressWarnings("unchecked")  
1087 final E[] array = (E[]) Array.newInstance(arrayClass, list.size());  
1088 return list.toArray(array);  
1089 }  
1090  
1091 /\*\*  
1092 \* Gets a list based on an iterator.  
1093 \* <p>  
1094 \* As the wrapped Iterator is traversed, an ArrayList of its values is  
1095 \* created. At the end, the list is returned.  
1096 \*  
1097 \* @param <E> the element type  
1098 \* @param iterator the iterator to use, not null  
1099 \* @return a list of the iterator contents  
1100 \* @throws NullPointerException if iterator parameter is null  
1101 \*/  
1102 public static <E> List<E> toList(final Iterator<? extends E> iterator) {  
1103 return toList(iterator, 10);  
1104 }  
1105  
1106 /\*\*  
1107 \* Gets a list based on an iterator.  
1108 \* <p>  
1109 \* As the wrapped Iterator is traversed, an ArrayList of its values is  
1110 \* created. At the end, the list is returned.  
1111 \*  
1112 \* @param <E> the element type  
1113 \* @param iterator the iterator to use, not null  
1114 \* @param estimatedSize the initial size of the ArrayList  
1115 \* @return a list of the iterator contents  
1116 \* @throws NullPointerException if iterator parameter is null  
1117 \* @throws IllegalArgumentException if the size is less than 1  
1118 \*/  
1119 public static <E> List<E> toList(final Iterator<? extends E> iterator, final int estimatedSize) {  
1120 if (iterator == null) {  
1121 throw new NullPointerException("Iterator must not be null");  
1122 }  
1123 if (estimatedSize < 1) {  
1124 throw new IllegalArgumentException("Estimated size must be greater than 0");  
1125 }  
1126 final List<E> list = new ArrayList<>(estimatedSize);  
1127 while (iterator.hasNext()) {  
1128 list.add(iterator.next());  
1129 }  
1130 return list;  
1131 }  
1132  
1133 /\*\*  
1134 \* Gets a suitable Iterator for the given object.  
1135 \* <p>  
1136 \* This method can handle objects as follows  
1137 \* <ul>  
1138 \* <li>null - empty iterator  
1139 \* <li>Iterator - returned directly  
1140 \* <li>Enumeration - wrapped  
1141 \* <li>Collection - iterator from collection returned  
1142 \* <li>Map - values iterator returned  
1143 \* <li>Dictionary - values (elements) enumeration returned as iterator  
1144 \* <li>array - iterator over array returned  
1145 \* <li>object with iterator() public method accessed by reflection  
1146 \* <li>object - singleton iterator  
1147 \* <li>NodeList - iterator over the list  
1148 \* <li>Node - iterator over the child nodes  
1149 \* </ul>  
1150 \*  
1151 \* @param obj the object to convert to an iterator  
1152 \* @return a suitable iterator, never null  
1153 \*/  
1154 public static Iterator<?> getIterator(final Object obj) {  
1155 if (obj == null) {  
1156 return emptyIterator();  
1157 }  
1158 if (obj instanceof Iterator) {  
1159 return (Iterator<?>) obj;  
1160 }  
1161 if (obj instanceof Iterable) {  
1162 return ((Iterable<?>) obj).iterator();  
1163 }  
1164 if (obj instanceof Object[]) {  
1165 return new ObjectArrayIterator<>((Object[]) obj);  
1166 }  
1167 if (obj instanceof Enumeration) {  
1168 return new EnumerationIterator<>((Enumeration<?>) obj);  
1169 }  
1170 if (obj instanceof Map) {  
1171 return ((Map<?, ?>) obj).values().iterator();  
1172 }  
1173 if (obj instanceof NodeList) {  
1174 return new NodeListIterator((NodeList) obj);  
1175 }  
1176 if (obj instanceof Node) {  
1177 return new NodeListIterator((Node) obj);  
1178 }  
1179 if (obj instanceof Dictionary) {  
1180 return new EnumerationIterator<>(((Dictionary<?, ?>) obj).elements());  
1181 } else if (obj.getClass().isArray()) {  
1182 return new ArrayIterator<>(obj);  
1183 }  
1184 try {  
1185 final Method method = obj.getClass().getMethod("iterator", (Class[]) null);  
1186 if (Iterator.class.isAssignableFrom(method.getReturnType())) {  
1187 final Iterator<?> it = (Iterator<?>) method.invoke(obj, (Object[]) null);  
1188 if (it != null) {  
1189 return it;  
1190 }  
1191 }  
1192 } catch (final RuntimeException e) { // NOPMD  
1193 // ignore  
1194 } catch (final NoSuchMethodException e) { // NOPMD  
1195 // ignore  
1196 } catch (final IllegalAccessException e) { // NOPMD  
1197 // ignore  
1198 } catch (final InvocationTargetException e) { // NOPMD  
1199 // ignore  
1200 }  
1201 return singletonIterator(obj);  
1202 }  
1203  
1204 // Utility methods  
1205 //-----------------------------------------------------------------------  
1206  
1207 /\*\*  
1208 \* Applies the closure to each element of the provided iterator.  
1209 \*  
1210 \* @param <E> the element type  
1211 \* @param iterator the iterator to use, may be null  
1212 \* @param closure the closure to apply to each element, may not be null  
1213 \* @throws NullPointerException if closure is null  
1214 \* @since 4.1  
1215 \*/  
1216 public static <E> void forEach(final Iterator<E> iterator, final Closure<? super E> closure) {  
1217 if (closure == null) {  
1218 throw new NullPointerException("Closure must not be null");  
1219 }  
1220  
1221 if (iterator != null) {  
1222 while (iterator.hasNext()) {  
1223 final E element = iterator.next();  
1224 closure.execute(element);  
1225 }  
1226 }  
1227 }  
1228  
1229 /\*\*  
1230 \* Executes the given closure on each but the last element in the iterator.  
1231 \* <p>  
1232 \* If the input iterator is null no change is made.  
1233 \*  
1234 \* @param <E> the type of object the {@link Iterator} contains  
1235 \* @param iterator the iterator to get the input from, may be null  
1236 \* @param closure the closure to perform, may not be null  
1237 \* @return the last element in the iterator, or null if iterator is null or empty  
1238 \* @throws NullPointerException if closure is null  
1239 \* @since 4.1  
1240 \*/  
1241 public static <E> E forEachButLast(final Iterator<E> iterator, final Closure<? super E> closure) {  
1242 if (closure == null) {  
1243 throw new NullPointerException("Closure must not be null.");  
1244 }  
1245 if (iterator != null) {  
1246 while (iterator.hasNext()) {  
1247 final E element = iterator.next();  
1248 if (iterator.hasNext()) {  
1249 closure.execute(element);  
1250 } else {  
1251 return element;  
1252 }  
1253 }  
1254 }  
1255 return null;  
1256 }  
1257  
1258 /\*\*  
1259 \* Finds the first element in the given iterator which matches the given predicate.  
1260 \* <p>  
1261 \* A <code>null</code> or empty iterator returns null.  
1262 \*  
1263 \* @param <E> the element type  
1264 \* @param iterator the iterator to search, may be null  
1265 \* @param predicate the predicate to use, may not be null  
1266 \* @return the first element of the iterator which matches the predicate or null if none could be found  
1267 \* @throws NullPointerException if predicate is null  
1268 \* @since 4.1  
1269 \*/  
1270 public static <E> E find(final Iterator<E> iterator, final Predicate<? super E> predicate) {  
1271 if (predicate == null) {  
1272 throw new NullPointerException("Predicate must not be null");  
1273 }  
1274  
1275 if (iterator != null) {  
1276 while (iterator.hasNext()) {  
1277 final E element = iterator.next();  
1278 if (predicate.evaluate(element)) {  
1279 return element;  
1280 }  
1281 }  
1282 }  
1283 return null;  
1284 }  
1285  
1286 /\*\*  
1287 \* Returns the index of the first element in the specified iterator that  
1288 \* matches the given predicate.  
1289 \* <p>  
1290 \* A <code>null</code> or empty iterator returns -1.  
1291 \*  
1292 \* @param <E> the element type  
1293 \* @param iterator the iterator to search, may be null  
1294 \* @param predicate the predicate to use, may not be null  
1295 \* @return the index of the first element which matches the predicate or -1 if none matches  
1296 \* @throws NullPointerException if predicate is null  
1297 \* @since 4.1  
1298 \*/  
1299 public static <E> int indexOf(final Iterator<E> iterator, final Predicate<? super E> predicate) {  
1300 if (predicate == null) {  
1301 throw new NullPointerException("Predicate must not be null");  
1302 }  
1303  
1304 if (iterator != null) {  
1305 for(int index = 0; iterator.hasNext(); index++) {  
1306 final E element = iterator.next();  
1307 if (predicate.evaluate(element)) {  
1308 return index;  
1309 }  
1310 }  
1311 }  
1312 return -1;  
1313 }  
1314  
1315 /\*\*  
1316 \* Answers true if a predicate is true for any element of the iterator.  
1317 \* <p>  
1318 \* A <code>null</code> or empty iterator returns false.  
1319 \*  
1320 \* @param <E> the type of object the {@link Iterator} contains  
1321 \* @param iterator the {@link Iterator} to use, may be null  
1322 \* @param predicate the predicate to use, may not be null  
1323 \* @return true if any element of the collection matches the predicate, false otherwise  
1324 \* @throws NullPointerException if predicate is null  
1325 \* @since 4.1  
1326 \*/  
1327 public static <E> boolean matchesAny(final Iterator<E> iterator, final Predicate<? super E> predicate) {  
1328 return indexOf(iterator, predicate) != -1;  
1329 }  
1330  
1331 /\*\*  
1332 \* Answers true if a predicate is true for every element of an iterator.  
1333 \* <p>  
1334 \* A <code>null</code> or empty iterator returns true.  
1335 \*  
1336 \* @param <E> the type of object the {@link Iterator} contains  
1337 \* @param iterator the {@link Iterator} to use, may be null  
1338 \* @param predicate the predicate to use, may not be null  
1339 \* @return true if every element of the collection matches the predicate or if the  
1340 \* collection is empty, false otherwise  
1341 \* @throws NullPointerException if predicate is null  
1342 \* @since 4.1  
1343 \*/  
1344 public static <E> boolean matchesAll(final Iterator<E> iterator, final Predicate<? super E> predicate) {  
1345 if (predicate == null) {  
1346 throw new NullPointerException("Predicate must not be null");  
1347 }  
1348  
1349 if (iterator != null) {  
1350 while (iterator.hasNext()) {  
1351 final E element = iterator.next();  
1352 if (!predicate.evaluate(element)) {  
1353 return false;  
1354 }  
1355 }  
1356 }  
1357 return true;  
1358 }  
1359  
1360 /\*\*  
1361 \* Checks if the given iterator is empty.  
1362 \* <p>  
1363 \* A <code>null</code> or empty iterator returns true.  
1364 \*  
1365 \* @param iterator the {@link Iterator} to use, may be null  
1366 \* @return true if the iterator is exhausted or null, false otherwise  
1367 \* @since 4.1  
1368 \*/  
1369 public static boolean isEmpty(final Iterator<?> iterator) {  
1370 return iterator == null || !iterator.hasNext();  
1371 }  
1372  
1373 /\*\*  
1374 \* Checks if the object is contained in the given iterator.  
1375 \* <p>  
1376 \* A <code>null</code> or empty iterator returns false.  
1377 \*  
1378 \* @param <E> the type of object the {@link Iterator} contains  
1379 \* @param iterator the iterator to check, may be null  
1380 \* @param object the object to check  
1381 \* @return true if the object is contained in the iterator, false otherwise  
1382 \* @since 4.1  
1383 \*/  
1384 public static <E> boolean contains(final Iterator<E> iterator, final Object object) {  
1385 return matchesAny(iterator, EqualPredicate.equalPredicate(object));  
1386 }  
1387  
1388 /\*\*  
1389 \* Returns the <code>index</code>-th value in {@link Iterator}, throwing  
1390 \* <code>IndexOutOfBoundsException</code> if there is no such element.  
1391 \* <p>  
1392 \* The Iterator is advanced to <code>index</code> (or to the end, if  
1393 \* <code>index</code> exceeds the number of entries) as a side effect of this method.  
1394 \*  
1395 \* @param <E> the type of object in the {@link Iterator}  
1396 \* @param iterator the iterator to get a value from  
1397 \* @param index the index to get  
1398 \* @return the object at the specified index  
1399 \* @throws IndexOutOfBoundsException if the index is invalid  
1400 \* @since 4.1  
1401 \*/  
1402 public static <E> E get(final Iterator<E> iterator, final int index) {  
1403 int i = index;  
1404 CollectionUtils.checkIndexBounds(i);  
1405 while (iterator.hasNext()) {  
1406 i--;  
1407 if (i == -1) {  
1408 return iterator.next();  
1409 }  
1410 iterator.next();  
1411 }  
1412 throw new IndexOutOfBoundsException("Entry does not exist: " + i);  
1413 }  
1414  
1415 /\*\*  
1416 \* Shortcut for {@code get(iterator, 0)}.  
1417 \* <p>  
1418 \* Returns the <code>first</code> value in {@link Iterator}, throwing  
1419 \* <code>IndexOutOfBoundsException</code> if there is no such element.  
1420 \* </p>  
1421 \* <p>  
1422 \* The Iterator is advanced to <code>0</code> (or to the end, if  
1423 \* <code>0</code> exceeds the number of entries) as a side effect of this method.  
1424 \* </p>  
1425 \* @param <E> the type of object in the {@link Iterator}  
1426 \* @param iterator the iterator to get a value from  
1427 \* @return the first object  
1428 \* @throws IndexOutOfBoundsException if the request is invalid  
1429 \* @since 4.2  
1430 \*/  
1431 public static <E> E first(final Iterator<E> iterator) {  
1432 return get(iterator, 0);  
1433 }  
1434  
1435 /\*\*  
1436 \* Returns the number of elements contained in the given iterator.  
1437 \* <p>  
1438 \* A <code>null</code> or empty iterator returns {@code 0}.  
1439 \*  
1440 \* @param iterator the iterator to check, may be null  
1441 \* @return the number of elements contained in the iterator  
1442 \* @since 4.1  
1443 \*/  
1444 public static int size(final Iterator<?> iterator) {  
1445 int size = 0;  
1446 if (iterator != null) {  
1447 while (iterator.hasNext()) {  
1448 iterator.next();  
1449 size++;  
1450 }  
1451 }  
1452 return size;  
1453 }  
1454  
1455 /\*\*  
1456 \* Returns a string representation of the elements of the specified iterator.  
1457 \* <p>  
1458 \* The string representation consists of a list of the iterator's elements,  
1459 \* enclosed in square brackets ({@code "[]"}). Adjacent elements are separated  
1460 \* by the characters {@code ", "} (a comma followed by a space). Elements are  
1461 \* converted to strings as by {@code String.valueOf(Object)}.  
1462 \*  
1463 \* @param <E> the element type  
1464 \* @param iterator the iterator to convert to a string, may be null  
1465 \* @return a string representation of {@code iterator}  
1466 \* @since 4.1  
1467 \*/  
1468 public static <E> String toString(final Iterator<E> iterator) {  
1469 return toString(iterator, TransformerUtils.stringValueTransformer(),  
1470 DEFAULT\_TOSTRING\_DELIMITER, DEFAULT\_TOSTRING\_PREFIX,  
1471 DEFAULT\_TOSTRING\_SUFFIX);  
1472 }  
1473  
1474 /\*\*  
1475 \* Returns a string representation of the elements of the specified iterator.  
1476 \* <p>  
1477 \* The string representation consists of a list of the iterable's elements,  
1478 \* enclosed in square brackets ({@code "[]"}). Adjacent elements are separated  
1479 \* by the characters {@code ", "} (a comma followed by a space). Elements are  
1480 \* converted to strings as by using the provided {@code transformer}.  
1481 \*  
1482 \* @param <E> the element type  
1483 \* @param iterator the iterator to convert to a string, may be null  
1484 \* @param transformer the transformer used to get a string representation of an element  
1485 \* @return a string representation of {@code iterator}  
1486 \* @throws NullPointerException if {@code transformer} is null  
1487 \* @since 4.1  
1488 \*/  
1489 public static <E> String toString(final Iterator<E> iterator,  
1490 final Transformer<? super E, String> transformer) {  
1491 return toString(iterator, transformer, DEFAULT\_TOSTRING\_DELIMITER,  
1492 DEFAULT\_TOSTRING\_PREFIX, DEFAULT\_TOSTRING\_SUFFIX);  
1493 }  
1494  
1495 /\*\*  
1496 \* Returns a string representation of the elements of the specified iterator.  
1497 \* <p>  
1498 \* The string representation consists of a list of the iterator's elements,  
1499 \* enclosed by the provided {@code prefix} and {@code suffix}. Adjacent elements  
1500 \* are separated by the provided {@code delimiter}. Elements are converted to  
1501 \* strings as by using the provided {@code transformer}.  
1502 \*  
1503 \* @param <E> the element type  
1504 \* @param iterator the iterator to convert to a string, may be null  
1505 \* @param transformer the transformer used to get a string representation of an element  
1506 \* @param delimiter the string to delimit elements  
1507 \* @param prefix the prefix, prepended to the string representation  
1508 \* @param suffix the suffix, appended to the string representation  
1509 \* @return a string representation of {@code iterator}  
1510 \* @throws NullPointerException if either transformer, delimiter, prefix or suffix is null  
1511 \* @since 4.1  
1512 \*/  
1513 public static <E> String toString(final Iterator<E> iterator,  
1514 final Transformer<? super E, String> transformer,  
1515 final String delimiter,  
1516 final String prefix,  
1517 final String suffix) {  
1518 if (transformer == null) {  
1519 throw new NullPointerException("transformer may not be null");  
1520 }  
1521 if (delimiter == null) {  
1522 throw new NullPointerException("delimiter may not be null");  
1523 }  
1524 if (prefix == null) {  
1525 throw new NullPointerException("prefix may not be null");  
1526 }  
1527 if (suffix == null) {  
1528 throw new NullPointerException("suffix may not be null");  
1529 }  
1530 final StringBuilder stringBuilder = new StringBuilder(prefix);  
1531 if (iterator != null) {  
1532 while (iterator.hasNext()) {  
1533 final E element = iterator.next();  
1534 stringBuilder.append(transformer.transform(element));  
1535 stringBuilder.append(delimiter);  
1536 }  
1537 if(stringBuilder.length() > prefix.length()) {  
1538 stringBuilder.setLength(stringBuilder.length() - delimiter.length());  
1539 }  
1540 }  
1541 stringBuilder.append(suffix);  
1542 return stringBuilder.toString();  
1543 }  
1544  
1545}