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.Arrays;  
020import java.util.Collection;  
021import java.util.Comparator;  
022import java.util.Enumeration;  
023import java.util.Iterator;  
024import java.util.List;  
025  
026import org.apache.commons.collections4.iterators.SingletonIterator;  
027  
028/\*\*  
029 \* A FluentIterable provides a powerful yet simple API for manipulating  
030 \* Iterable instances in a fluent manner.  
031 \* <p>  
032 \* A FluentIterable can be created either from an Iterable or from a set  
033 \* of elements. The following types of methods are provided:  
034 \* </p>  
035 \* <ul>  
036 \* <li>fluent methods which return a new {@code FluentIterable} instance,  
037 \* providing a view of the original iterable (e.g. filter(Predicate));  
038 \* <li>conversion methods which copy the FluentIterable's contents into a  
039 \* new collection or array (e.g. toList());  
040 \* <li>utility methods which answer questions about the FluentIterable's  
041 \* contents (e.g. size(), anyMatch(Predicate)).  
042 \* <li>  
043 \* </ul>  
044 \* <p>  
045 \* The following example outputs the first 3 even numbers in the range [1, 10]  
046 \* into a list:  
047 \* </p>  
048 \* <pre>  
049 \* List<String> result =  
050 \* FluentIterable  
051 \* .of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
052 \* .filter(new Predicate<Integer>() {  
053 \* public boolean evaluate(Integer number) {  
054 \* return number % 2 == 0;  
055 \* }  
056 \* )  
057 \* .transform(TransformerUtils.stringValueTransformer())  
058 \* .limit(3)  
059 \* .toList();  
060 \* </pre>  
061 \* The resulting list will contain the following elements:  
062 \* <pre>[2, 4, 6]</pre>  
063 \*  
064 \* @param <E> the element type  
065 \* @since 4.1  
066 \*/  
067public class FluentIterable<E> implements Iterable<E> {  
068  
069 /\*\* A reference to the wrapped iterable. \*/  
070 private final Iterable<E> iterable;  
071  
072 // Static factory methods  
073 // ----------------------------------------------------------------------  
074  
075 /\*\*  
076 \* Creates a new empty FluentIterable.  
077 \*  
078 \* @param <T> the element type  
079 \* @return a new empty FluentIterable  
080 \*/  
081 public static <T> FluentIterable<T> empty() {  
082 return IterableUtils.EMPTY\_ITERABLE;  
083 }  
084  
085 /\*\*  
086 \* Creates a new FluentIterable of the single provided element.  
087 \* <p>  
088 \* The returned iterable's iterator does not support {@code remove()}.  
089 \*  
090 \* @param <T> the element type  
091 \* @param singleton the singleton element  
092 \* @return a new FluentIterable containing the singleton  
093 \*/  
094 public static <T> FluentIterable<T> of(final T singleton) {  
095 return of(IteratorUtils.asIterable(new SingletonIterator<>(singleton, false)));  
096 }  
097  
098 /\*\*  
099 \* Creates a new FluentIterable from the provided elements.  
100 \* <p>  
101 \* The returned iterable's iterator does not support {@code remove()}.  
102 \*  
103 \* @param <T> the element type  
104 \* @param elements the elements to be contained in the FluentIterable  
105 \* @return a new FluentIterable containing the provided elements  
106 \*/  
107 public static <T> FluentIterable<T> of(final T... elements) {  
108 return of(Arrays.asList(elements));  
109 }  
110  
111 /\*\*  
112 \* Construct a new FluentIterable from the provided iterable. If the  
113 \* iterable is already an instance of FluentIterable, the instance  
114 \* will be returned instead.  
115 \* <p>  
116 \* The returned iterable's iterator supports {@code remove()} when the  
117 \* corresponding input iterator supports it.  
118 \*  
119 \* @param <T> the element type  
120 \* @param iterable the iterable to wrap into a FluentIterable, may not be null  
121 \* @return a new FluentIterable wrapping the provided iterable  
122 \* @throws NullPointerException if iterable is null  
123 \*/  
124 public static <T> FluentIterable<T> of(final Iterable<T> iterable) {  
125 IterableUtils.checkNotNull(iterable);  
126 if (iterable instanceof FluentIterable<?>) {  
127 return (FluentIterable<T>) iterable;  
128 }  
129 return new FluentIterable<>(iterable);  
130 }  
131  
132 // Constructor  
133 // ----------------------------------------------------------------------  
134  
135 /\*\*  
136 \* Package-private constructor, used by IterableUtils.  
137 \*/  
138 FluentIterable() {  
139 this.iterable = this;  
140 }  
141  
142 /\*\*  
143 \* Create a new FluentIterable by wrapping the provided iterable.  
144 \* @param iterable the iterable to wrap  
145 \*/  
146 private FluentIterable(final Iterable<E> iterable) {  
147 this.iterable = iterable;  
148 }  
149  
150 // fluent construction methods  
151 // ----------------------------------------------------------------------  
152  
153 /\*\*  
154 \* Returns a new FluentIterable whose iterator will first traverse  
155 \* the elements of the current iterable, followed by the provided  
156 \* elements.  
157 \*  
158 \* @param elements the elements to append to the iterable  
159 \* @return a new iterable, combining this iterable with the elements  
160 \*/  
161 public FluentIterable<E> append(final E... elements) {  
162 return append(Arrays.asList(elements));  
163 }  
164  
165 /\*\*  
166 \* Returns a new FluentIterable whose iterator will first traverse  
167 \* the elements of the current iterable, followed by the elements  
168 \* of the provided iterable.  
169 \*  
170 \* @param other the other iterable to combine, may not be null  
171 \* @return a new iterable, combining this iterable with other  
172 \* @throws NullPointerException if other is null  
173 \*/  
174 public FluentIterable<E> append(final Iterable<? extends E> other) {  
175 return of(IterableUtils.chainedIterable(iterable, other));  
176 }  
177  
178 /\*\*  
179 \* Returns a new FluentIterable whose iterator will traverse the  
180 \* elements of the current and provided iterable in natural order.  
181 \* <p>  
182 \* Example: natural ordering  
183 \* <ul>  
184 \* <li>this contains elements [1, 3, 5, 7]  
185 \* <li>other contains elements [2, 4, 6, 8]  
186 \* </ul>  
187 \* <p>  
188 \* The returned iterable will traverse the elements in the following  
189 \* order: [1, 2, 3, 4, 5, 6, 7, 8]  
190 \*  
191 \* @param other the other iterable to collate, may not be null  
192 \* @return a new iterable, collating this iterable with the other in natural order  
193 \* @throws NullPointerException if other is null  
194 \* @see org.apache.commons.collections4.iterators.CollatingIterator  
195 \*/  
196 public FluentIterable<E> collate(final Iterable<? extends E> other) {  
197 return of(IterableUtils.collatedIterable(iterable, other));  
198 }  
199  
200 /\*\*  
201 \* Returns a new FluentIterable whose iterator will traverse the  
202 \* elements of the current and provided iterable according to the  
203 \* ordering defined by an comparator.  
204 \* <p>  
205 \* Example: descending order  
206 \* <ul>  
207 \* <li>this contains elements [7, 5, 3, 1]  
208 \* <li>other contains elements [8, 6, 4, 2]  
209 \* </ul>  
210 \* <p>  
211 \* The returned iterable will traverse the elements in the following  
212 \* order: [8, 7, 6, 5, 4, 3, 2, 1]  
213 \*  
214 \* @param comparator the comparator to define an ordering, may be null,  
215 \* in which case natural ordering will be used  
216 \* @param other the other iterable to collate, may not be null  
217 \* @return a new iterable, collating this iterable with the other in natural order  
218 \* @throws NullPointerException if other is null  
219 \* @see org.apache.commons.collections4.iterators.CollatingIterator  
220 \*/  
221 public FluentIterable<E> collate(final Iterable<? extends E> other,  
222 final Comparator<? super E> comparator) {  
223 return of(IterableUtils.collatedIterable(comparator, iterable, other));  
224 }  
225  
226 /\*\*  
227 \* This method fully traverses an iterator of this iterable and returns  
228 \* a new iterable with the same contents, but without any reference  
229 \* to the originating iterables and/or iterators.  
230 \* <p>  
231 \* Calling this method is equivalent to:  
232 \* <pre>  
233 \* FluentIterable<E> someIterable = ...;  
234 \* FluentIterable.of(someIterable.toList());  
235 \* </pre>  
236 \*  
237 \* @return a new iterable with the same contents as this iterable  
238 \*/  
239 public FluentIterable<E> eval() {  
240 return of(toList());  
241 }  
242  
243 /\*\*  
244 \* Returns a new FluentIterable whose iterator will only return  
245 \* elements from this iterable matching the provided predicate.  
246 \*  
247 \* @param predicate the predicate used to filter elements  
248 \* @return a new iterable, providing a filtered view of this iterable  
249 \* @throws NullPointerException if predicate is null  
250 \*/  
251 public FluentIterable<E> filter(final Predicate<? super E> predicate) {  
252 return of(IterableUtils.filteredIterable(iterable, predicate));  
253 }  
254  
255 /\*\*  
256 \* Returns a new FluentIterable whose iterator will return at most  
257 \* the provided maximum number of elements from this iterable.  
258 \*  
259 \* @param maxSize the maximum number of elements  
260 \* @return a new iterable, providing a bounded view of this iterable  
261 \* @throws IllegalArgumentException if maxSize is negative  
262 \*/  
263 public FluentIterable<E> limit(final long maxSize) {  
264 return of(IterableUtils.boundedIterable(iterable, maxSize));  
265 }  
266  
267 /\*\*  
268 \* Returns a new FluentIterable whose iterator will loop infinitely  
269 \* over the elements from this iterable.  
270 \*  
271 \* @return a new iterable, providing a looping view of this iterable  
272 \*/  
273 public FluentIterable<E> loop() {  
274 return of(IterableUtils.loopingIterable(iterable));  
275 }  
276  
277 /\*\*  
278 \* Returns a new FluentIterable whose iterator will traverse the  
279 \* elements from this iterable in reverse order.  
280 \*  
281 \* @return a new iterable, providing a reversed view of this iterable  
282 \*/  
283 public FluentIterable<E> reverse() {  
284 return of(IterableUtils.reversedIterable(iterable));  
285 }  
286  
287 /\*\*  
288 \* Returns a new FluentIterable whose iterator will skip the first  
289 \* N elements from this iterable.  
290 \*  
291 \* @param elementsToSkip the number of elements to skip  
292 \* @return a new iterable, providing a view of this iterable by skipping  
293 \* the first N elements  
294 \* @throws IllegalArgumentException if elementsToSkip is negative  
295 \*/  
296 public FluentIterable<E> skip(final long elementsToSkip) {  
297 return of(IterableUtils.skippingIterable(iterable, elementsToSkip));  
298 }  
299  
300 /\*\*  
301 \* Returns a new FluentIterable whose iterator will return all elements  
302 \* of this iterable transformed by the provided transformer.  
303 \*  
304 \* @param <O> the output element type  
305 \* @param transformer the transformer applied to each element  
306 \* @return a new iterable, providing a transformed view of this iterable  
307 \* @throws NullPointerException if transformer is null  
308 \*/  
309 public <O> FluentIterable<O> transform(final Transformer<? super E, ? extends O> transformer) {  
310 return of(IterableUtils.transformedIterable(iterable, transformer));  
311 }  
312  
313 /\*\*  
314 \* Returns a new FluentIterable whose iterator will return a unique view  
315 \* of this iterable.  
316 \*  
317 \* @return a new iterable, providing a unique view of this iterable  
318 \*/  
319 public FluentIterable<E> unique() {  
320 return of(IterableUtils.uniqueIterable(iterable));  
321 }  
322  
323 /\*\*  
324 \* Returns a new FluentIterable whose iterator will return an unmodifiable  
325 \* view of this iterable.  
326 \*  
327 \* @return a new iterable, providing an unmodifiable view of this iterable  
328 \*/  
329 public FluentIterable<E> unmodifiable() {  
330 return of(IterableUtils.unmodifiableIterable(iterable));  
331 }  
332  
333 /\*\*  
334 \* Returns a new FluentIterable whose iterator will traverse  
335 \* the elements of this iterable and the other iterable in  
336 \* alternating order.  
337 \*  
338 \* @param other the other iterable to interleave, may not be null  
339 \* @return a new iterable, interleaving this iterable with others  
340 \* @throws NullPointerException if other is null  
341 \*/  
342 public FluentIterable<E> zip(final Iterable<? extends E> other) {  
343 return of(IterableUtils.zippingIterable(iterable, other));  
344 }  
345  
346 /\*\*  
347 \* Returns a new FluentIterable whose iterator will traverse  
348 \* the elements of this iterable and the other iterables in  
349 \* alternating order.  
350 \*  
351 \* @param others the iterables to interleave, may not be null  
352 \* @return a new iterable, interleaving this iterable with others  
353 \* @throws NullPointerException if either of the provided iterables is null  
354 \*/  
355 public FluentIterable<E> zip(final Iterable<? extends E>... others) {  
356 return of(IterableUtils.zippingIterable(iterable, others));  
357 }  
358  
359 // convenience methods  
360 // ----------------------------------------------------------------------  
361  
362 /\*\* {@inheritDoc} \*/  
363 @Override  
364 public Iterator<E> iterator() {  
365 return iterable.iterator();  
366 }  
367  
368 /\*\*  
369 \* Returns an Enumeration that will enumerate all elements contained  
370 \* in this iterable.  
371 \*  
372 \* @return an Enumeration over the elements of this iterable  
373 \*/  
374 public Enumeration<E> asEnumeration() {  
375 return IteratorUtils.asEnumeration(iterator());  
376 }  
377  
378 /\*\*  
379 \* Checks if all elements contained in this iterable are matching the  
380 \* provided predicate.  
381 \* <p>  
382 \* A <code>null</code> or empty iterable returns true.  
383 \*  
384 \* @param predicate the predicate to use, may not be null  
385 \* @return true if all elements contained in this iterable match the predicate,  
386 \* false otherwise  
387 \* @throws NullPointerException if predicate is null  
388 \*/  
389 public boolean allMatch(final Predicate<? super E> predicate) {  
390 return IterableUtils.matchesAll(iterable, predicate);  
391 }  
392  
393 /\*\*  
394 \* Checks if this iterable contains any element matching the provided predicate.  
395 \* <p>  
396 \* A <code>null</code> or empty iterable returns false.  
397 \*  
398 \* @param predicate the predicate to use, may not be null  
399 \* @return true if at least one element contained in this iterable matches the predicate,  
400 \* false otherwise  
401 \* @throws NullPointerException if predicate is null  
402 \*/  
403 public boolean anyMatch(final Predicate<? super E> predicate) {  
404 return IterableUtils.matchesAny(iterable, predicate);  
405 }  
406  
407 /\*\*  
408 \* Checks if this iterable is empty.  
409 \*  
410 \* @return true if this iterable does not contain any elements, false otherwise  
411 \*/  
412 public boolean isEmpty() {  
413 return IterableUtils.isEmpty(iterable);  
414 }  
415  
416 /\*\*  
417 \* Checks if the object is contained in this iterable.  
418 \*  
419 \* @param object the object to check  
420 \* @return true if the object is contained in this iterable, false otherwise  
421 \*/  
422 public boolean contains(final Object object) {  
423 return IterableUtils.contains(iterable, object);  
424 }  
425  
426 /\*\*  
427 \* Applies the closure to all elements contained in this iterable.  
428 \*  
429 \* @param closure the closure to apply to each element, may not be null  
430 \* @throws NullPointerException if closure is null  
431 \*/  
432 public void forEach(final Closure<? super E> closure) {  
433 IterableUtils.forEach(iterable, closure);  
434 }  
435  
436 /\*\*  
437 \* Returns the element at the provided position in this iterable.  
438 \* In order to return the element, an iterator needs to be traversed  
439 \* up to the requested position.  
440 \*  
441 \* @param position the position of the element to return  
442 \* @return the element  
443 \* @throws IndexOutOfBoundsException if the provided position is outside the  
444 \* valid range of this iterable: [0, size)  
445 \*/  
446 public E get(final int position) {  
447 return IterableUtils.get(iterable, position);  
448 }  
449  
450 /\*\*  
451 \* Returns the number of elements that are contained in this iterable.  
452 \* In order to determine the size, an iterator needs to be traversed.  
453 \*  
454 \* @return the size of this iterable  
455 \*/  
456 public int size() {  
457 return IterableUtils.size(iterable);  
458 }  
459  
460 /\*\*  
461 \* Traverses an iterator of this iterable and adds all elements  
462 \* to the provided collection.  
463 \*  
464 \* @param collection the collection to add the elements  
465 \* @throws NullPointerException if collection is null  
466 \*/  
467 public void copyInto(final Collection<? super E> collection) {  
468 if (collection == null) {  
469 throw new NullPointerException("Collection must not be null");  
470 }  
471 CollectionUtils.addAll(collection, iterable);  
472 }  
473  
474 /\*\*  
475 \* Returns an array containing all elements of this iterable by traversing  
476 \* its iterator.  
477 \*  
478 \* @param arrayClass the class of array to create  
479 \* @return an array of the iterable contents  
480 \* @throws ArrayStoreException if arrayClass is invalid  
481 \*/  
482 public E[] toArray(final Class<E> arrayClass) {  
483 return IteratorUtils.toArray(iterator(), arrayClass);  
484 }  
485  
486 /\*\*  
487 \* Returns a mutable list containing all elements of this iterable  
488 \* by traversing its iterator.  
489 \* <p>  
490 \* The returned list is guaranteed to be mutable.  
491 \*  
492 \* @return a list of the iterable contents  
493 \*/  
494 public List<E> toList() {  
495 return IterableUtils.toList(iterable);  
496 }  
497  
498 /\*\* {@inheritDoc} \*/  
499 @Override  
500 public String toString() {  
501 return IterableUtils.toString(iterable);  
502 }  
503  
504}