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.collection;  
018  
019import java.io.Serializable;  
020import java.lang.reflect.Array;  
021import java.util.ArrayList;  
022import java.util.Collection;  
023import java.util.Iterator;  
024import java.util.List;  
025import java.util.Objects;  
026import java.util.function.Predicate;  
027  
028import org.apache.commons.collections4.CollectionUtils;  
029import org.apache.commons.collections4.iterators.EmptyIterator;  
030import org.apache.commons.collections4.iterators.IteratorChain;  
031import org.apache.commons.collections4.list.UnmodifiableList;  
032  
033/\*\*  
034 \* Decorates a collection of other collections to provide a single unified view.  
035 \* <p>  
036 \* Changes made to this collection will actually be made on the decorated collection.  
037 \* Add and remove operations require the use of a pluggable strategy. If no  
038 \* strategy is provided then add and remove are unsupported.  
039 \* </p>  
040 \* @param <E> the type of the elements in the collection  
041 \* @since 3.0  
042 \*/  
043public class CompositeCollection<E> implements Collection<E>, Serializable {  
044  
045 /\*\* Serialization version \*/  
046 private static final long serialVersionUID = 8417515734108306801L;  
047  
048 /\*\* CollectionMutator to handle changes to the collection \*/  
049 private CollectionMutator<E> mutator;  
050  
051 /\*\* Collections in the composite \*/  
052 private final List<Collection<E>> all = new ArrayList<>();  
053  
054 /\*\*  
055 \* Create an empty CompositeCollection.  
056 \*/  
057 public CompositeCollection() {  
058 super();  
059 }  
060  
061 /\*\*  
062 \* Create a Composite Collection with one collection.  
063 \*  
064 \* @param compositeCollection the Collection to be appended to the composite  
065 \*/  
066 public CompositeCollection(final Collection<E> compositeCollection) {  
067 super();  
068 addComposited(compositeCollection);  
069 }  
070  
071 /\*\*  
072 \* Create a Composite Collection with two collections.  
073 \*  
074 \* @param compositeCollection1 the Collection to be appended to the composite  
075 \* @param compositeCollection2 the Collection to be appended to the composite  
076 \*/  
077 public CompositeCollection(final Collection<E> compositeCollection1,  
078 final Collection<E> compositeCollection2) {  
079 super();  
080 addComposited(compositeCollection1, compositeCollection2);  
081 }  
082  
083 /\*\*  
084 \* Create a Composite Collection with an array of collections.  
085 \*  
086 \* @param compositeCollections the collections to composite  
087 \*/  
088 public CompositeCollection(final Collection<E>... compositeCollections) {  
089 super();  
090 addComposited(compositeCollections);  
091 }  
092  
093 //-----------------------------------------------------------------------  
094 /\*\*  
095 \* Gets the size of this composite collection.  
096 \* <p>  
097 \* This implementation calls <code>size()</code> on each collection.  
098 \* </p>  
099 \* @return total number of elements in all contained containers  
100 \*/  
101 @Override  
102 public int size() {  
103 int size = 0;  
104 for (final Collection<E> item : all) {  
105 size += item.size();  
106 }  
107 return size;  
108 }  
109  
110 /\*\*  
111 \* Checks whether this composite collection is empty.  
112 \* <p>  
113 \* This implementation calls <code>isEmpty()</code> on each collection.  
114 \* </p>  
115 \* @return true if all of the contained collections are empty  
116 \*/  
117 @Override  
118 public boolean isEmpty() {  
119 for (final Collection<E> item : all) {  
120 if (item.isEmpty() == false) {  
121 return false;  
122 }  
123 }  
124 return true;  
125 }  
126  
127 /\*\*  
128 \* Checks whether this composite collection contains the object.  
129 \* <p>  
130 \* This implementation calls <code>contains()</code> on each collection.  
131 \* </p>  
132 \* @param obj the object to search for  
133 \* @return true if obj is contained in any of the contained collections  
134 \*/  
135 @Override  
136 public boolean contains(final Object obj) {  
137 for (final Collection<E> item : all) {  
138 if (item.contains(obj)) {  
139 return true;  
140 }  
141 }  
142 return false;  
143 }  
144  
145 /\*\*  
146 \* Gets an iterator over all the collections in this composite.  
147 \* <p>  
148 \* This implementation uses an <code>IteratorChain</code>.  
149 \* </p>  
150 \* @return an <code>IteratorChain</code> instance which supports  
151 \* <code>remove()</code>. Iteration occurs over contained collections in  
152 \* the order they were added, but this behavior should not be relied upon.  
153 \* @see IteratorChain  
154 \*/  
155 @Override  
156 public Iterator<E> iterator() {  
157 if (all.isEmpty()) {  
158 return EmptyIterator.<E>emptyIterator();  
159 }  
160 final IteratorChain<E> chain = new IteratorChain<>();  
161 for (final Collection<E> item : all) {  
162 chain.addIterator(item.iterator());  
163 }  
164 return chain;  
165 }  
166  
167 /\*\*  
168 \* Returns an array containing all of the elements in this composite.  
169 \*  
170 \* @return an object array of all the elements in the collection  
171 \*/  
172 @Override  
173 public Object[] toArray() {  
174 final Object[] result = new Object[size()];  
175 int i = 0;  
176 for (final Iterator<E> it = iterator(); it.hasNext(); i++) {  
177 result[i] = it.next();  
178 }  
179 return result;  
180 }  
181  
182 /\*\*  
183 \* Returns an object array, populating the supplied array if possible.  
184 \* See <code>Collection</code> interface for full details.  
185 \*  
186 \* @param <T> the type of the elements in the collection  
187 \* @param array the array to use, populating if possible  
188 \* @return an array of all the elements in the collection  
189 \*/  
190 @Override  
191 @SuppressWarnings("unchecked")  
192 public <T> T[] toArray(final T[] array) {  
193 final int size = size();  
194 Object[] result = null;  
195 if (array.length >= size) {  
196 result = array;  
197 } else {  
198 result = (Object[]) Array.newInstance(array.getClass().getComponentType(), size);  
199 }  
200  
201 int offset = 0;  
202 for (final Collection<E> item : all) {  
203 for (final E e : item) {  
204 result[offset++] = e;  
205 }  
206 }  
207 if (result.length > size) {  
208 result[size] = null;  
209 }  
210 return (T[]) result;  
211 }  
212  
213 /\*\*  
214 \* Adds an object to the collection, throwing UnsupportedOperationException  
215 \* unless a CollectionMutator strategy is specified.  
216 \*  
217 \* @param obj the object to add  
218 \* @return {@code true} if the collection was modified  
219 \* @throws UnsupportedOperationException if CollectionMutator hasn't been set  
220 \* @throws UnsupportedOperationException if add is unsupported  
221 \* @throws ClassCastException if the object cannot be added due to its type  
222 \* @throws NullPointerException if the object cannot be added because its null  
223 \* @throws IllegalArgumentException if the object cannot be added  
224 \*/  
225 @Override  
226 public boolean add(final E obj) {  
227 if (mutator == null) {  
228 throw new UnsupportedOperationException(  
229 "add() is not supported on CompositeCollection without a CollectionMutator strategy");  
230 }  
231 return mutator.add(this, all, obj);  
232 }  
233  
234 /\*\*  
235 \* Removes an object from the collection, throwing UnsupportedOperationException  
236 \* unless a CollectionMutator strategy is specified.  
237 \*  
238 \* @param obj the object being removed  
239 \* @return true if the collection is changed  
240 \* @throws UnsupportedOperationException if removed is unsupported  
241 \* @throws ClassCastException if the object cannot be removed due to its type  
242 \* @throws NullPointerException if the object cannot be removed because its null  
243 \* @throws IllegalArgumentException if the object cannot be removed  
244 \*/  
245 @Override  
246 public boolean remove(final Object obj) {  
247 if (mutator == null) {  
248 throw new UnsupportedOperationException(  
249 "remove() is not supported on CompositeCollection without a CollectionMutator strategy");  
250 }  
251 return mutator.remove(this, all, obj);  
252 }  
253  
254 /\*\*  
255 \* Checks whether this composite contains all the elements in the specified collection.  
256 \* <p>  
257 \* This implementation calls <code>contains()</code> for each element in the  
258 \* specified collection.  
259 \* </p>  
260 \* @param coll the collection to check for  
261 \* @return true if all elements contained  
262 \*/  
263 @Override  
264 public boolean containsAll(final Collection<?> coll) {  
265 if (coll == null) {  
266 return false;  
267 }  
268 for (final Object item : coll) {  
269 if (contains(item) == false) {  
270 return false;  
271 }  
272 }  
273 return true;  
274 }  
275  
276 /\*\*  
277 \* Adds a collection of elements to this collection, throwing  
278 \* UnsupportedOperationException unless a CollectionMutator strategy is specified.  
279 \*  
280 \* @param coll the collection to add  
281 \* @return true if the collection was modified  
282 \* @throws UnsupportedOperationException if CollectionMutator hasn't been set  
283 \* @throws UnsupportedOperationException if add is unsupported  
284 \* @throws ClassCastException if the object cannot be added due to its type  
285 \* @throws NullPointerException if the object cannot be added because its null  
286 \* @throws IllegalArgumentException if the object cannot be added  
287 \*/  
288 @Override  
289 public boolean addAll(final Collection<? extends E> coll) {  
290 if (mutator == null) {  
291 throw new UnsupportedOperationException(  
292 "addAll() is not supported on CompositeCollection without a CollectionMutator strategy");  
293 }  
294 return mutator.addAll(this, all, coll);  
295 }  
296  
297 /\*\*  
298 \* Removes the elements in the specified collection from this composite collection.  
299 \* <p>  
300 \* This implementation calls <code>removeAll</code> on each collection.  
301 \* </p>  
302 \* @param coll the collection to remove  
303 \* @return true if the collection was modified  
304 \* @throws UnsupportedOperationException if removeAll is unsupported  
305 \*/  
306 @Override  
307 public boolean removeAll(final Collection<?> coll) {  
308 if (CollectionUtils.isEmpty(coll)) {  
309 return false;  
310 }  
311 boolean changed = false;  
312 for (final Collection<E> item : all) {  
313 changed |= item.removeAll(coll);  
314 }  
315 return changed;  
316 }  
317  
318 /\*\*  
319 \* Removes all of the elements of this collection that satisfy the given predicate from this composite collection.  
320 \* <p>  
321 \* This implementation calls <code>removeIf</code> on each collection.  
322 \* </p>  
323 \* @param filter a predicate which returns true for elements to be removed  
324 \* @return true if the collection was modified  
325 \* @throws UnsupportedOperationException if removeIf is unsupported  
326 \* @since 4.4  
327 \*/  
328 @Override  
329 public boolean removeIf(final Predicate<? super E> filter) {  
330 if (Objects.isNull(filter)) {  
331 return false;  
332 }  
333 boolean changed = false;  
334 for (final Collection<E> item : all) {  
335 changed |= item.removeIf(filter);  
336 }  
337 return changed;  
338 }  
339  
340 /\*\*  
341 \* Retains all the elements in the specified collection in this composite collection,  
342 \* removing all others.  
343 \* <p>  
344 \* This implementation calls <code>retainAll()</code> on each collection.  
345 \* </p>  
346 \* @param coll the collection to remove  
347 \* @return true if the collection was modified  
348 \* @throws UnsupportedOperationException if retainAll is unsupported  
349 \*/  
350 @Override  
351 public boolean retainAll(final Collection<?> coll) {  
352 boolean changed = false;  
353 if (coll != null) {  
354 for (final Collection<E> item : all) {  
355 changed |= item.retainAll(coll);  
356 }  
357 }  
358 return changed;  
359 }  
360  
361 /\*\*  
362 \* Removes all of the elements from this collection .  
363 \* <p>  
364 \* This implementation calls <code>clear()</code> on each collection.  
365 \* </p>  
366 \* @throws UnsupportedOperationException if clear is unsupported  
367 \*/  
368 @Override  
369 public void clear() {  
370 for (final Collection<E> coll : all) {  
371 coll.clear();  
372 }  
373 }  
374  
375 //-----------------------------------------------------------------------  
376 /\*\*  
377 \* Specify a CollectionMutator strategy instance to handle changes.  
378 \*  
379 \* @param mutator the mutator to use  
380 \*/  
381 public void setMutator(final CollectionMutator<E> mutator) {  
382 this.mutator = mutator;  
383 }  
384  
385 /\*\*  
386 \* Add these Collections to the list of collections in this composite  
387 \*  
388 \* @param compositeCollection the Collection to be appended to the composite  
389 \*/  
390 public void addComposited(final Collection<E> compositeCollection) {  
391 if (compositeCollection != null) {  
392 all.add(compositeCollection);  
393 }  
394 }  
395  
396 /\*\*  
397 \* Add these Collections to the list of collections in this composite  
398 \*  
399 \* @param compositeCollection1 the Collection to be appended to the composite  
400 \* @param compositeCollection2 the Collection to be appended to the composite  
401 \*/  
402 public void addComposited(final Collection<E> compositeCollection1,  
403 final Collection<E> compositeCollection2) {  
404 if (compositeCollection1 != null) {  
405 all.add(compositeCollection1);  
406 }  
407 if (compositeCollection2 != null) {  
408 all.add(compositeCollection2);  
409 }  
410 }  
411  
412 /\*\*  
413 \* Add these Collections to the list of collections in this composite  
414 \*  
415 \* @param compositeCollections the Collections to be appended to the composite  
416 \*/  
417 public void addComposited(final Collection<E>... compositeCollections) {  
418 for (Collection<E> compositeCollection : compositeCollections) {  
419 if (compositeCollection != null) {  
420 all.add(compositeCollection);  
421 }  
422 }  
423 }  
424  
425 /\*\*  
426 \* Removes a collection from the those being decorated in this composite.  
427 \*  
428 \* @param coll collection to be removed  
429 \*/  
430 public void removeComposited(final Collection<E> coll) {  
431 all.remove(coll);  
432 }  
433  
434 //-----------------------------------------------------------------------  
435 /\*\*  
436 \* Returns a new collection containing all of the elements  
437 \*  
438 \* @return A new ArrayList containing all of the elements in this composite.  
439 \* The new collection is <i>not</i> backed by this composite.  
440 \*/  
441 public Collection<E> toCollection() {  
442 return new ArrayList<>(this);  
443 }  
444  
445 /\*\*  
446 \* Gets the collections being decorated.  
447 \*  
448 \* @return Unmodifiable list of all collections in this composite.  
449 \*/  
450 public List<Collection<E>> getCollections() {  
451 return UnmodifiableList.unmodifiableList(all);  
452 }  
453  
454 /\*\*  
455 \* Get the collection mutator to be used for this CompositeCollection.  
456 \* @return CollectionMutator<E>  
457 \*/  
458 protected CollectionMutator<E> getMutator() {  
459 return mutator;  
460 }  
461  
462 //-----------------------------------------------------------------------  
463 /\*\*  
464 \* Pluggable strategy to handle changes to the composite.  
465 \*  
466 \* @param <E> the element being held in the collection  
467 \*/  
468 public interface CollectionMutator<E> extends Serializable {  
469  
470 /\*\*  
471 \* Called when an object is to be added to the composite.  
472 \*  
473 \* @param composite the CompositeCollection being changed  
474 \* @param collections all of the Collection instances in this CompositeCollection  
475 \* @param obj the object being added  
476 \* @return true if the collection is changed  
477 \* @throws UnsupportedOperationException if add is unsupported  
478 \* @throws ClassCastException if the object cannot be added due to its type  
479 \* @throws NullPointerException if the object cannot be added because its null  
480 \* @throws IllegalArgumentException if the object cannot be added  
481 \*/  
482 boolean add(CompositeCollection<E> composite, List<Collection<E>> collections, E obj);  
483  
484 /\*\*  
485 \* Called when a collection is to be added to the composite.  
486 \*  
487 \* @param composite the CompositeCollection being changed  
488 \* @param collections all of the Collection instances in this CompositeCollection  
489 \* @param coll the collection being added  
490 \* @return true if the collection is changed  
491 \* @throws UnsupportedOperationException if add is unsupported  
492 \* @throws ClassCastException if the object cannot be added due to its type  
493 \* @throws NullPointerException if the object cannot be added because its null  
494 \* @throws IllegalArgumentException if the object cannot be added  
495 \*/  
496 boolean addAll(CompositeCollection<E> composite,  
497 List<Collection<E>> collections,  
498 Collection<? extends E> coll);  
499  
500 /\*\*  
501 \* Called when an object is to be removed to the composite.  
502 \*  
503 \* @param composite the CompositeCollection being changed  
504 \* @param collections all of the Collection instances in this CompositeCollection  
505 \* @param obj the object being removed  
506 \* @return true if the collection is changed  
507 \* @throws UnsupportedOperationException if removed is unsupported  
508 \* @throws ClassCastException if the object cannot be removed due to its type  
509 \* @throws NullPointerException if the object cannot be removed because its null  
510 \* @throws IllegalArgumentException if the object cannot be removed  
511 \*/  
512 boolean remove(CompositeCollection<E> composite,  
513 List<Collection<E>> collections,  
514 Object obj);  
515  
516 }  
517  
518}  
519