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.set;  
018  
019import java.io.Serializable;  
020import java.lang.reflect.Array;  
021import java.util.ArrayList;  
022import java.util.Collection;  
023import java.util.HashSet;  
024import java.util.Iterator;  
025import java.util.List;  
026import java.util.Set;  
027import java.util.Objects;  
028import java.util.function.Predicate;  
029  
030import org.apache.commons.collections4.CollectionUtils;  
031import org.apache.commons.collections4.iterators.EmptyIterator;  
032import org.apache.commons.collections4.iterators.IteratorChain;  
033import org.apache.commons.collections4.list.UnmodifiableList;  
034  
035/\*\*  
036 \* Decorates a set of other sets to provide a single unified view.  
037 \* <p>  
038 \* Changes made to this set will actually be made on the decorated set.  
039 \* Add operations require the use of a pluggable strategy.  
040 \* If no strategy is provided then add is unsupported.  
041 \* </p>  
042 \* <p>  
043 \* From version 4.0, this class does not extend  
044 \* {@link org.apache.commons.collections4.collection.CompositeCollection CompositeCollection}  
045 \* anymore due to its input restrictions (only accepts Sets).  
046 \* See <a href="https://issues.apache.org/jira/browse/COLLECTIONS-424">COLLECTIONS-424</a>  
047 \* for more details.  
048 \* </p>  
049 \*  
050 \* @param <E> the type of the elements in this set  
051 \* @since 3.0  
052 \*/  
053public class CompositeSet<E> implements Set<E>, Serializable {  
054  
055 /\*\* Serialization version \*/  
056 private static final long serialVersionUID = 5185069727540378940L;  
057  
058 /\*\* SetMutator to handle changes to the collection \*/  
059 private SetMutator<E> mutator;  
060  
061 /\*\* Sets in the composite \*/  
062 private final List<Set<E>> all = new ArrayList<>();  
063  
064 /\*\*  
065 \* Create an empty CompositeSet.  
066 \*/  
067 public CompositeSet() {  
068 super();  
069 }  
070  
071 /\*\*  
072 \* Create a CompositeSet with just <code>set</code> composited.  
073 \*  
074 \* @param set the initial set in the composite  
075 \*/  
076 public CompositeSet(final Set<E> set) {  
077 super();  
078 addComposited(set);  
079 }  
080  
081 /\*\*  
082 \* Create a composite set with sets as the initial set of composited Sets.  
083 \*  
084 \* @param sets the initial sets in the composite  
085 \*/  
086 public CompositeSet(final Set<E>... sets) {  
087 super();  
088 addComposited(sets);  
089 }  
090  
091 //-----------------------------------------------------------------------  
092 /\*\*  
093 \* Gets the size of this composite set.  
094 \* <p>  
095 \* This implementation calls <code>size()</code> on each set.  
096 \*  
097 \* @return total number of elements in all contained containers  
098 \*/  
099 @Override  
100 public int size() {  
101 int size = 0;  
102 for (final Set<E> item : all) {  
103 size += item.size();  
104 }  
105 return size;  
106 }  
107  
108 /\*\*  
109 \* Checks whether this composite set is empty.  
110 \* <p>  
111 \* This implementation calls <code>isEmpty()</code> on each set.  
112 \*  
113 \* @return true if all of the contained sets are empty  
114 \*/  
115 @Override  
116 public boolean isEmpty() {  
117 for (final Set<E> item : all) {  
118 if (item.isEmpty() == false) {  
119 return false;  
120 }  
121 }  
122 return true;  
123 }  
124  
125 /\*\*  
126 \* Checks whether this composite set contains the object.  
127 \* <p>  
128 \* This implementation calls <code>contains()</code> on each set.  
129 \*  
130 \* @param obj the object to search for  
131 \* @return true if obj is contained in any of the contained sets  
132 \*/  
133 @Override  
134 public boolean contains(final Object obj) {  
135 for (final Set<E> item : all) {  
136 if (item.contains(obj)) {  
137 return true;  
138 }  
139 }  
140 return false;  
141 }  
142  
143 /\*\*  
144 \* Gets an iterator over all the sets in this composite.  
145 \* <p>  
146 \* This implementation uses an <code>IteratorChain</code>.  
147 \*  
148 \* @return an <code>IteratorChain</code> instance which supports  
149 \* <code>remove()</code>. Iteration occurs over contained collections in  
150 \* the order they were added, but this behavior should not be relied upon.  
151 \* @see IteratorChain  
152 \*/  
153 @Override  
154 public Iterator<E> iterator() {  
155 if (all.isEmpty()) {  
156 return EmptyIterator.<E>emptyIterator();  
157 }  
158 final IteratorChain<E> chain = new IteratorChain<>();  
159 for (final Set<E> item : all) {  
160 chain.addIterator(item.iterator());  
161 }  
162 return chain;  
163 }  
164  
165 /\*\*  
166 \* Returns an array containing all of the elements in this composite.  
167 \*  
168 \* @return an object array of all the elements in the collection  
169 \*/  
170 @Override  
171 public Object[] toArray() {  
172 final Object[] result = new Object[size()];  
173 int i = 0;  
174 for (final Iterator<E> it = iterator(); it.hasNext(); i++) {  
175 result[i] = it.next();  
176 }  
177 return result;  
178 }  
179  
180 /\*\*  
181 \* Returns an object array, populating the supplied array if possible.  
182 \* See <code>Collection</code> interface for full details.  
183 \*  
184 \* @param <T> the type of the elements in the collection  
185 \* @param array the array to use, populating if possible  
186 \* @return an array of all the elements in the collection  
187 \*/  
188 @Override  
189 @SuppressWarnings("unchecked")  
190 public <T> T[] toArray(final T[] array) {  
191 final int size = size();  
192 Object[] result = null;  
193 if (array.length >= size) {  
194 result = array;  
195 } else {  
196 result = (Object[]) Array.newInstance(array.getClass().getComponentType(), size);  
197 }  
198  
199 int offset = 0;  
200 for (final Collection<E> item : all) {  
201 for (final E e : item) {  
202 result[offset++] = e;  
203 }  
204 }  
205 if (result.length > size) {  
206 result[size] = null;  
207 }  
208 return (T[]) result;  
209 }  
210  
211 /\*\*  
212 \* Adds an object to the collection, throwing UnsupportedOperationException  
213 \* unless a SetMutator strategy is specified.  
214 \*  
215 \* @param obj the object to add  
216 \* @return {@code true} if the collection was modified  
217 \* @throws UnsupportedOperationException if SetMutator hasn't been set or add is unsupported  
218 \* @throws ClassCastException if the object cannot be added due to its type  
219 \* @throws NullPointerException if the object cannot be added because its null  
220 \* @throws IllegalArgumentException if the object cannot be added  
221 \*/  
222 @Override  
223 public boolean add(final E obj) {  
224 if (mutator == null) {  
225 throw new UnsupportedOperationException(  
226 "add() is not supported on CompositeSet without a SetMutator strategy");  
227 }  
228 return mutator.add(this, all, obj);  
229 }  
230  
231 /\*\*  
232 \* If a <code>CollectionMutator</code> is defined for this CompositeSet then this  
233 \* method will be called anyway.  
234 \*  
235 \* @param obj object to be removed  
236 \* @return true if the object is removed, false otherwise  
237 \*/  
238 @Override  
239 public boolean remove(final Object obj) {  
240 for (final Set<E> set : getSets()) {  
241 if (set.contains(obj)) {  
242 return set.remove(obj);  
243 }  
244 }  
245 return false;  
246 }  
247  
248 /\*\*  
249 \* Checks whether this composite contains all the elements in the specified collection.  
250 \* <p>  
251 \* This implementation calls <code>contains()</code> for each element in the  
252 \* specified collection.  
253 \*  
254 \* @param coll the collection to check for  
255 \* @return true if all elements contained  
256 \*/  
257 @Override  
258 public boolean containsAll(final Collection<?> coll) {  
259 if (coll == null) {  
260 return false;  
261 }  
262 for (final Object item : coll) {  
263 if (contains(item) == false) {  
264 return false;  
265 }  
266 }  
267 return true;  
268 }  
269  
270 /\*\*  
271 \* Adds a collection of elements to this composite, throwing  
272 \* UnsupportedOperationException unless a SetMutator strategy is specified.  
273 \*  
274 \* @param coll the collection to add  
275 \* @return true if the composite was modified  
276 \* @throws UnsupportedOperationException if SetMutator hasn't been set or add is unsupported  
277 \* @throws ClassCastException if the object cannot be added due to its type  
278 \* @throws NullPointerException if the object cannot be added because its null  
279 \* @throws IllegalArgumentException if the object cannot be added  
280 \*/  
281 @Override  
282 public boolean addAll(final Collection<? extends E> coll) {  
283 if (mutator == null) {  
284 throw new UnsupportedOperationException(  
285 "addAll() is not supported on CompositeSet without a SetMutator strategy");  
286 }  
287 return mutator.addAll(this, all, coll);  
288 }  
289  
290 /\*\*  
291 \* @since 4.4  
292 \*/  
293 @Override  
294 public boolean removeIf(Predicate<? super E> filter) {  
295 if (Objects.isNull(filter)) {  
296 return false;  
297 }  
298 boolean changed = false;  
299 for (final Collection<E> item : all) {  
300 changed |= item.removeIf(filter);  
301 }  
302 return changed;  
303 }  
304  
305 /\*\*  
306 \* Removes the elements in the specified collection from this composite set.  
307 \* <p>  
308 \* This implementation calls <code>removeAll</code> on each collection.  
309 \*  
310 \* @param coll the collection to remove  
311 \* @return true if the composite was modified  
312 \* @throws UnsupportedOperationException if removeAll is unsupported  
313 \*/  
314 @Override  
315 public boolean removeAll(final Collection<?> coll) {  
316 if (CollectionUtils.isEmpty(coll)) {  
317 return false;  
318 }  
319 boolean changed = false;  
320 for (final Collection<E> item : all) {  
321 changed |= item.removeAll(coll);  
322 }  
323 return changed;  
324 }  
325  
326 /\*\*  
327 \* Retains all the elements in the specified collection in this composite set,  
328 \* removing all others.  
329 \* <p>  
330 \* This implementation calls <code>retainAll()</code> on each collection.  
331 \*  
332 \* @param coll the collection to remove  
333 \* @return true if the composite was modified  
334 \* @throws UnsupportedOperationException if retainAll is unsupported  
335 \*/  
336 @Override  
337 public boolean retainAll(final Collection<?> coll) {  
338 boolean changed = false;  
339 for (final Collection<E> item : all) {  
340 changed |= item.retainAll(coll);  
341 }  
342 return changed;  
343 }  
344  
345 /\*\*  
346 \* Removes all of the elements from this composite set.  
347 \* <p>  
348 \* This implementation calls <code>clear()</code> on each set.  
349 \*  
350 \* @throws UnsupportedOperationException if clear is unsupported  
351 \*/  
352 @Override  
353 public void clear() {  
354 for (final Collection<E> coll : all) {  
355 coll.clear();  
356 }  
357 }  
358  
359 //-----------------------------------------------------------------------  
360 /\*\*  
361 \* Specify a SetMutator strategy instance to handle changes.  
362 \*  
363 \* @param mutator the mutator to use  
364 \*/  
365 public void setMutator(final SetMutator<E> mutator) {  
366 this.mutator = mutator;  
367 }  
368  
369 /\*\*  
370 \* Add a Set to this composite.  
371 \*  
372 \* @param set the set to add  
373 \* @throws IllegalArgumentException if a SetMutator is set, but fails to resolve a collision  
374 \* @throws UnsupportedOperationException if there is no SetMutator set  
375 \* @throws NullPointerException if {@code set} is null  
376 \* @see SetMutator  
377 \*/  
378 public synchronized void addComposited(final Set<E> set) {  
379 if (set != null) {  
380 for (final Set<E> existingSet : getSets()) {  
381 final Collection<E> intersects = CollectionUtils.intersection(existingSet, set);  
382 if (intersects.size() > 0) {  
383 if (this.mutator == null) {  
384 throw new UnsupportedOperationException(  
385 "Collision adding composited set with no SetMutator set");  
386 }  
387 getMutator().resolveCollision(this, existingSet, set, intersects);  
388 if (CollectionUtils.intersection(existingSet, set).size() > 0) {  
389 throw new IllegalArgumentException(  
390 "Attempt to add illegal entry unresolved by SetMutator.resolveCollision()");  
391 }  
392 }  
393 }  
394 all.add(set);  
395 }  
396 }  
397  
398 /\*\*  
399 \* Add these Sets to the list of sets in this composite.  
400 \*  
401 \* @param set1 the first Set to be appended to the composite  
402 \* @param set2 the second Set to be appended to the composite  
403 \*/  
404 public void addComposited(final Set<E> set1, final Set<E> set2) {  
405 addComposited(set1);  
406 addComposited(set2);  
407 }  
408  
409 /\*\*  
410 \* Add these Sets to the list of sets in this composite  
411 \*  
412 \* @param sets the Sets to be appended to the composite  
413 \*/  
414 public void addComposited(final Set<E>... sets) {  
415 if (sets != null) {  
416 for (final Set<E> set : sets) {  
417 addComposited(set);  
418 }  
419 }  
420 }  
421  
422 /\*\*  
423 \* Removes a set from those being decorated in this composite.  
424 \*  
425 \* @param set set to be removed  
426 \*/  
427 public void removeComposited(final Set<E> set) {  
428 all.remove(set);  
429 }  
430  
431 //-----------------------------------------------------------------------  
432 /\*\*  
433 \* Returns a new Set containing all of the elements.  
434 \*  
435 \* @return A new HashSet containing all of the elements in this composite.  
436 \* The new collection is <i>not</i> backed by this composite.  
437 \*/  
438 public Set<E> toSet() {  
439 return new HashSet<>(this);  
440 }  
441  
442 /\*\*  
443 \* Gets the sets being decorated.  
444 \*  
445 \* @return Unmodifiable list of all sets in this composite.  
446 \*/  
447 public List<Set<E>> getSets() {  
448 return UnmodifiableList.unmodifiableList(all);  
449 }  
450  
451 /\*\*  
452 \* Get the set mutator to be used for this CompositeSet.  
453 \* @return the set mutator  
454 \*/  
455 protected SetMutator<E> getMutator() {  
456 return mutator;  
457 }  
458  
459 /\*\*  
460 \* {@inheritDoc}  
461 \* @see java.util.Set#equals  
462 \*/  
463 @Override  
464 public boolean equals(final Object obj) {  
465 if (obj instanceof Set) {  
466 final Set<?> set = (Set<?>) obj;  
467 return set.size() == this.size() && set.containsAll(this);  
468 }  
469 return false;  
470 }  
471  
472 /\*\*  
473 \* {@inheritDoc}  
474 \* @see java.util.Set#hashCode  
475 \*/  
476 @Override  
477 public int hashCode() {  
478 int code = 0;  
479 for (final E e : this) {  
480 code += e == null ? 0 : e.hashCode();  
481 }  
482 return code;  
483 }  
484  
485 /\*\*  
486 \* Define callbacks for mutation operations.  
487 \*/  
488 public interface SetMutator<E> extends Serializable {  
489  
490 /\*\*  
491 \* Called when an object is to be added to the composite.  
492 \*  
493 \* @param composite the CompositeSet being changed  
494 \* @param sets all of the Set instances in this CompositeSet  
495 \* @param obj the object being added  
496 \* @return true if the collection is changed  
497 \* @throws UnsupportedOperationException if add is unsupported  
498 \* @throws ClassCastException if the object cannot be added due to its type  
499 \* @throws NullPointerException if the object cannot be added because its null  
500 \* @throws IllegalArgumentException if the object cannot be added  
501 \*/  
502 boolean add(CompositeSet<E> composite, List<Set<E>> sets, E obj);  
503  
504 /\*\*  
505 \* Called when a collection is to be added to the composite.  
506 \*  
507 \* @param composite the CompositeSet being changed  
508 \* @param sets all of the Set instances in this CompositeSet  
509 \* @param coll the collection being added  
510 \* @return true if the collection is changed  
511 \* @throws UnsupportedOperationException if add is unsupported  
512 \* @throws ClassCastException if the object cannot be added due to its type  
513 \* @throws NullPointerException if the object cannot be added because its null  
514 \* @throws IllegalArgumentException if the object cannot be added  
515 \*/  
516 boolean addAll(CompositeSet<E> composite,  
517 List<Set<E>> sets,  
518 Collection<? extends E> coll);  
519  
520 /\*\*  
521 \* Called when a Set is added to the CompositeSet and there is a  
522 \* collision between existing and added sets.  
523 \* <p>  
524 \* If <code>added</code> and <code>existing</code> still have any intersects  
525 \* after this method returns an IllegalArgumentException will be thrown.  
526 \*  
527 \* @param comp the CompositeSet being modified  
528 \* @param existing the Set already existing in the composite  
529 \* @param added the Set being added to the composite  
530 \* @param intersects the intersection of the existing and added sets  
531 \*/  
532 void resolveCollision(CompositeSet<E> comp,  
533 Set<E> existing,  
534 Set<E> added,  
535 Collection<E> intersects);  
536 }  
537}