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.list;  
018  
019import java.lang.reflect.InvocationTargetException;  
020import java.util.ArrayList;  
021import java.util.Collection;  
022import java.util.HashSet;  
023import java.util.Iterator;  
024import java.util.List;  
025import java.util.ListIterator;  
026import java.util.Set;  
027import java.util.function.Predicate;  
028  
029import org.apache.commons.collections4.ListUtils;  
030import org.apache.commons.collections4.iterators.AbstractIteratorDecorator;  
031import org.apache.commons.collections4.iterators.AbstractListIteratorDecorator;  
032import org.apache.commons.collections4.set.UnmodifiableSet;  
033  
034/\*\*  
035 \* Decorates a <code>List</code> to ensure that no duplicates are present much  
036 \* like a <code>Set</code>.  
037 \* <p>  
038 \* The <code>List</code> interface makes certain assumptions/requirements. This  
039 \* implementation breaks these in certain ways, but this is merely the result of  
040 \* rejecting duplicates. Each violation is explained in the method, but it  
041 \* should not affect you. Bear in mind that Sets require immutable objects to  
042 \* function correctly.  
043 \* </p>  
044 \* <p>  
045 \* The {@link org.apache.commons.collections4.set.ListOrderedSet ListOrderedSet}  
046 \* class provides an alternative approach, by wrapping an existing Set and  
047 \* retaining insertion order in the iterator.  
048 \* </p>  
049 \* <p>  
050 \* This class is Serializable from Commons Collections 3.1.  
051 \* </p>  
052 \*  
053 \* @since 3.0  
054 \*/  
055public class SetUniqueList<E> extends AbstractSerializableListDecorator<E> {  
056  
057 /\*\* Serialization version. \*/  
058 private static final long serialVersionUID = 7196982186153478694L;  
059  
060 /\*\* Internal Set to maintain uniqueness. \*/  
061 private final Set<E> set;  
062  
063 /\*\*  
064 \* Factory method to create a SetList using the supplied list to retain order.  
065 \* <p>  
066 \* If the list contains duplicates, these are removed (first indexed one  
067 \* kept). A <code>HashSet</code> is used for the set behaviour.  
068 \*  
069 \* @param <E> the element type  
070 \* @param list the list to decorate, must not be null  
071 \* @return a new {@link SetUniqueList}  
072 \* @throws NullPointerException if list is null  
073 \* @since 4.0  
074 \*/  
075 public static <E> SetUniqueList<E> setUniqueList(final List<E> list) {  
076 if (list == null) {  
077 throw new NullPointerException("List must not be null");  
078 }  
079 if (list.isEmpty()) {  
080 return new SetUniqueList<>(list, new HashSet<E>());  
081 }  
082 final List<E> temp = new ArrayList<>(list);  
083 list.clear();  
084 final SetUniqueList<E> sl = new SetUniqueList<>(list, new HashSet<E>());  
085 sl.addAll(temp);  
086 return sl;  
087 }  
088  
089 // -----------------------------------------------------------------------  
090 /\*\*  
091 \* Constructor that wraps (not copies) the List and specifies the set to use.  
092 \* <p>  
093 \* The set and list must both be correctly initialised to the same elements.  
094 \*  
095 \* @param set the set to decorate, must not be null  
096 \* @param list the list to decorate, must not be null  
097 \* @throws NullPointerException if set or list is null  
098 \*/  
099 protected SetUniqueList(final List<E> list, final Set<E> set) {  
100 super(list);  
101 if (set == null) {  
102 throw new NullPointerException("Set must not be null");  
103 }  
104 this.set = set;  
105 }  
106  
107 // -----------------------------------------------------------------------  
108 /\*\*  
109 \* Gets an unmodifiable view as a Set.  
110 \*  
111 \* @return an unmodifiable set view  
112 \*/  
113 public Set<E> asSet() {  
114 return UnmodifiableSet.unmodifiableSet(set);  
115 }  
116  
117 // -----------------------------------------------------------------------  
118 /\*\*  
119 \* Adds an element to the list if it is not already present.  
120 \* <p>  
121 \* <i>(Violation)</i> The <code>List</code> interface requires that this  
122 \* method returns <code>true</code> always. However this class may return  
123 \* <code>false</code> because of the <code>Set</code> behaviour.  
124 \*  
125 \* @param object the object to add  
126 \* @return true if object was added  
127 \*/  
128 @Override  
129 public boolean add(final E object) {  
130 // gets initial size  
131 final int sizeBefore = size();  
132  
133 // adds element if unique  
134 add(size(), object);  
135  
136 // compares sizes to detect if collection changed  
137 return sizeBefore != size();  
138 }  
139  
140 /\*\*  
141 \* Adds an element to a specific index in the list if it is not already  
142 \* present.  
143 \* <p>  
144 \* <i>(Violation)</i> The <code>List</code> interface makes the assumption  
145 \* that the element is always inserted. This may not happen with this  
146 \* implementation.  
147 \*  
148 \* @param index the index to insert at  
149 \* @param object the object to add  
150 \*/  
151 @Override  
152 public void add(final int index, final E object) {  
153 // adds element if it is not contained already  
154 if (set.contains(object) == false) {  
155 set.add(object);  
156 super.add(index, object);  
157 }  
158 }  
159  
160 /\*\*  
161 \* Adds a collection of objects to the end of the list avoiding duplicates.  
162 \* <p>  
163 \* Only elements that are not already in this list will be added, and  
164 \* duplicates from the specified collection will be ignored.  
165 \* <p>  
166 \* <i>(Violation)</i> The <code>List</code> interface makes the assumption  
167 \* that the elements are always inserted. This may not happen with this  
168 \* implementation.  
169 \*  
170 \* @param coll the collection to add in iterator order  
171 \* @return true if this collection changed  
172 \*/  
173 @Override  
174 public boolean addAll(final Collection<? extends E> coll) {  
175 return addAll(size(), coll);  
176 }  
177  
178 /\*\*  
179 \* Adds a collection of objects a specific index in the list avoiding  
180 \* duplicates.  
181 \* <p>  
182 \* Only elements that are not already in this list will be added, and  
183 \* duplicates from the specified collection will be ignored.  
184 \* <p>  
185 \* <i>(Violation)</i> The <code>List</code> interface makes the assumption  
186 \* that the elements are always inserted. This may not happen with this  
187 \* implementation.  
188 \*  
189 \* @param index the index to insert at  
190 \* @param coll the collection to add in iterator order  
191 \* @return true if this collection changed  
192 \*/  
193 @Override  
194 public boolean addAll(final int index, final Collection<? extends E> coll) {  
195 final List<E> temp = new ArrayList<>();  
196 for (final E e : coll) {  
197 if (set.add(e)) {  
198 temp.add(e);  
199 }  
200 }  
201 return super.addAll(index, temp);  
202 }  
203  
204 // -----------------------------------------------------------------------  
205 /\*\*  
206 \* Sets the value at the specified index avoiding duplicates.  
207 \* <p>  
208 \* The object is set into the specified index. Afterwards, any previous  
209 \* duplicate is removed. If the object is not already in the list then a  
210 \* normal set occurs. If it is present, then the old version is removed.  
211 \*  
212 \* @param index the index to insert at  
213 \* @param object the object to set  
214 \* @return the previous object  
215 \*/  
216 @Override  
217 public E set(final int index, final E object) {  
218 final int pos = indexOf(object);  
219 final E removed = super.set(index, object);  
220  
221 if (pos != -1 && pos != index) {  
222 // the object is already in the unique list  
223 // (and it hasn't been swapped with itself)  
224 super.remove(pos); // remove the duplicate by index  
225 }  
226  
227 set.remove(removed); // remove the item deleted by the set  
228 set.add(object); // add the new item to the unique set  
229  
230 return removed; // return the item deleted by the set  
231 }  
232  
233 @Override  
234 public boolean remove(final Object object) {  
235 final boolean result = set.remove(object);  
236 if (result) {  
237 super.remove(object);  
238 }  
239 return result;  
240 }  
241  
242 @Override  
243 public E remove(final int index) {  
244 final E result = super.remove(index);  
245 set.remove(result);  
246 return result;  
247 }  
248  
249 /\*\*  
250 \* @since 4.4  
251 \*/  
252 @Override  
253 public boolean removeIf(Predicate<? super E> filter) {  
254 boolean result = super.removeIf(filter);  
255 set.removeIf(filter);  
256 return result;  
257 }  
258  
259 @Override  
260 public boolean removeAll(final Collection<?> coll) {  
261 boolean result = false;  
262 for (final Object name : coll) {  
263 result |= remove(name);  
264 }  
265 return result;  
266 }  
267  
268 /\*\*  
269 \* {@inheritDoc}  
270 \* <p>  
271 \* This implementation iterates over the elements of this list, checking  
272 \* each element in turn to see if it's contained in <code>coll</code>.  
273 \* If it's not contained, it's removed from this list. As a consequence,  
274 \* it is advised to use a collection type for <code>coll</code> that provides  
275 \* a fast (e.g. O(1)) implementation of {@link Collection#contains(Object)}.  
276 \*/  
277 @Override  
278 public boolean retainAll(final Collection<?> coll) {  
279 final boolean result = set.retainAll(coll);  
280 if (result == false) {  
281 return false;  
282 }  
283 if (set.size() == 0) {  
284 super.clear();  
285 } else {  
286 // use the set as parameter for the call to retainAll to improve performance  
287 super.retainAll(set);  
288 }  
289 return result;  
290 }  
291  
292 @Override  
293 public void clear() {  
294 super.clear();  
295 set.clear();  
296 }  
297  
298 @Override  
299 public boolean contains(final Object object) {  
300 return set.contains(object);  
301 }  
302  
303 @Override  
304 public boolean containsAll(final Collection<?> coll) {  
305 return set.containsAll(coll);  
306 }  
307  
308 @Override  
309 public Iterator<E> iterator() {  
310 return new SetListIterator<>(super.iterator(), set);  
311 }  
312  
313 @Override  
314 public ListIterator<E> listIterator() {  
315 return new SetListListIterator<>(super.listIterator(), set);  
316 }  
317  
318 @Override  
319 public ListIterator<E> listIterator(final int index) {  
320 return new SetListListIterator<>(super.listIterator(index), set);  
321 }  
322  
323 /\*\*  
324 \* {@inheritDoc}  
325 \* <p>  
326 \* NOTE: from 4.0, an unmodifiable list will be returned, as changes to the  
327 \* subList can invalidate the parent list.  
328 \*/  
329 @Override  
330 public List<E> subList(final int fromIndex, final int toIndex) {  
331 final List<E> superSubList = super.subList(fromIndex, toIndex);  
332 final Set<E> subSet = createSetBasedOnList(set, superSubList);  
333 return ListUtils.unmodifiableList(new SetUniqueList<>(superSubList, subSet));  
334 }  
335  
336 /\*\*  
337 \* Create a new {@link Set} with the same type as the provided {@code set}  
338 \* and populate it with all elements of {@code list}.  
339 \*  
340 \* @param set the {@link Set} to be used as return type, must not be null  
341 \* @param list the {@link List} to populate the {@link Set}  
342 \* @return a new {@link Set} populated with all elements of the provided  
343 \* {@link List}  
344 \*/  
345 protected Set<E> createSetBasedOnList(final Set<E> set, final List<E> list) {  
346 Set<E> subSet;  
347 if (set.getClass().equals(HashSet.class)) {  
348 subSet = new HashSet<>(list.size());  
349 } else {  
350 try {  
351 subSet = set.getClass().getDeclaredConstructor(set.getClass()).newInstance(set);  
352 } catch (final InstantiationException  
353 | IllegalAccessException  
354 | InvocationTargetException  
355 | NoSuchMethodException ie) {  
356 subSet = new HashSet<>();  
357 }  
358 }  
359 return subSet;  
360 }  
361  
362 // -----------------------------------------------------------------------  
363 /\*\*  
364 \* Inner class iterator.  
365 \*/  
366 static class SetListIterator<E> extends AbstractIteratorDecorator<E> {  
367  
368 private final Set<E> set;  
369 private E last = null;  
370  
371 protected SetListIterator(final Iterator<E> it, final Set<E> set) {  
372 super(it);  
373 this.set = set;  
374 }  
375  
376 @Override  
377 public E next() {  
378 last = super.next();  
379 return last;  
380 }  
381  
382 @Override  
383 public void remove() {  
384 super.remove();  
385 set.remove(last);  
386 last = null;  
387 }  
388 }  
389  
390 /\*\*  
391 \* Inner class iterator.  
392 \*/  
393 static class SetListListIterator<E> extends  
394 AbstractListIteratorDecorator<E> {  
395  
396 private final Set<E> set;  
397 private E last = null;  
398  
399 protected SetListListIterator(final ListIterator<E> it, final Set<E> set) {  
400 super(it);  
401 this.set = set;  
402 }  
403  
404 @Override  
405 public E next() {  
406 last = super.next();  
407 return last;  
408 }  
409  
410 @Override  
411 public E previous() {  
412 last = super.previous();  
413 return last;  
414 }  
415  
416 @Override  
417 public void remove() {  
418 super.remove();  
419 set.remove(last);  
420 last = null;  
421 }  
422  
423 @Override  
424 public void add(final E object) {  
425 if (set.contains(object) == false) {  
426 super.add(object);  
427 set.add(object);  
428 }  
429 }  
430  
431 @Override  
432 public void set(final E object) {  
433 throw new UnsupportedOperationException("ListIterator does not support set");  
434 }  
435 }  
436  
437}