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.util.ArrayList;  
020import java.util.Collection;  
021import java.util.HashSet;  
022import java.util.Iterator;  
023import java.util.List;  
024import java.util.ListIterator;  
025import java.util.Set;  
026import java.util.Objects;  
027import java.util.function.Predicate;  
028  
029import org.apache.commons.collections4.CollectionUtils;  
030import org.apache.commons.collections4.OrderedIterator;  
031import org.apache.commons.collections4.functors.UniquePredicate;  
032import org.apache.commons.collections4.iterators.AbstractIteratorDecorator;  
033import org.apache.commons.collections4.list.UnmodifiableList;  
034  
035/\*\*  
036 \* Decorates another <code>Set</code> to ensure that the order of addition is  
037 \* retained and used by the iterator.  
038 \* <p>  
039 \* If an object is added to the set for a second time, it will remain in the  
040 \* original position in the iteration. The order can be observed from the set  
041 \* via the iterator or toArray methods.  
042 \* </p>  
043 \* <p>  
044 \* The ListOrderedSet also has various useful direct methods. These include many  
045 \* from <code>List</code>, such as <code>get(int)</code>,  
046 \* <code>remove(int)</code> and <code>indexOf(int)</code>. An unmodifiable  
047 \* <code>List</code> view of the set can be obtained via <code>asList()</code>.  
048 \* </p>  
049 \* <p>  
050 \* This class cannot implement the <code>List</code> interface directly as  
051 \* various interface methods (notably equals/hashCode) are incompatible with a  
052 \* set.  
053 \* </p>  
054 \* <p>  
055 \* This class is Serializable from Commons Collections 3.1.  
056 \* </p>  
057 \*  
058 \* @param <E> the type of the elements in this set  
059 \* @since 3.0  
060 \*/  
061public class ListOrderedSet<E>  
062 extends AbstractSerializableSetDecorator<E> {  
063  
064 /\*\* Serialization version \*/  
065 private static final long serialVersionUID = -228664372470420141L;  
066  
067 /\*\* Internal list to hold the sequence of objects \*/  
068 private final List<E> setOrder;  
069  
070 /\*\*  
071 \* Factory method to create an ordered set specifying the list and set to use.  
072 \* <p>  
073 \* The list and set must both be empty.  
074 \*  
075 \* @param <E> the element type  
076 \* @param set the set to decorate, must be empty and not null  
077 \* @param list the list to decorate, must be empty and not null  
078 \* @return a new ordered set  
079 \* @throws NullPointerException if set or list is null  
080 \* @throws IllegalArgumentException if either the set or list is not empty  
081 \* @since 4.0  
082 \*/  
083 public static <E> ListOrderedSet<E> listOrderedSet(final Set<E> set, final List<E> list) {  
084 if (set == null) {  
085 throw new NullPointerException("Set must not be null");  
086 }  
087 if (list == null) {  
088 throw new NullPointerException("List must not be null");  
089 }  
090 if (set.size() > 0 || list.size() > 0) {  
091 throw new IllegalArgumentException("Set and List must be empty");  
092 }  
093 return new ListOrderedSet<>(set, list);  
094 }  
095  
096 /\*\*  
097 \* Factory method to create an ordered set.  
098 \* <p>  
099 \* An <code>ArrayList</code> is used to retain order.  
100 \*  
101 \* @param <E> the element type  
102 \* @param set the set to decorate, must not be null  
103 \* @return a new ordered set  
104 \* @throws NullPointerException if set is null  
105 \* @since 4.0  
106 \*/  
107 public static <E> ListOrderedSet<E> listOrderedSet(final Set<E> set) {  
108 return new ListOrderedSet<>(set);  
109 }  
110  
111 /\*\*  
112 \* Factory method to create an ordered set using the supplied list to retain order.  
113 \* <p>  
114 \* A <code>HashSet</code> is used for the set behaviour.  
115 \* <p>  
116 \* NOTE: If the list contains duplicates, the duplicates are removed,  
117 \* altering the specified list.  
118 \*  
119 \* @param <E> the element type  
120 \* @param list the list to decorate, must not be null  
121 \* @return a new ordered set  
122 \* @throws NullPointerException if list is null  
123 \* @since 4.0  
124 \*/  
125 public static <E> ListOrderedSet<E> listOrderedSet(final List<E> list) {  
126 if (list == null) {  
127 throw new NullPointerException("List must not be null");  
128 }  
129 CollectionUtils.filter(list, UniquePredicate.uniquePredicate());  
130 final Set<E> set = new HashSet<>(list);  
131  
132 return new ListOrderedSet<>(set, list);  
133 }  
134  
135 // -----------------------------------------------------------------------  
136 /\*\*  
137 \* Constructs a new empty <code>ListOrderedSet</code> using a  
138 \* <code>HashSet</code> and an <code>ArrayList</code> internally.  
139 \*  
140 \* @since 3.1  
141 \*/  
142 public ListOrderedSet() {  
143 super(new HashSet<E>());  
144 setOrder = new ArrayList<>();  
145 }  
146  
147 /\*\*  
148 \* Constructor that wraps (not copies).  
149 \*  
150 \* @param set the set to decorate, must not be null  
151 \* @throws IllegalArgumentException if set is null  
152 \*/  
153 protected ListOrderedSet(final Set<E> set) {  
154 super(set);  
155 setOrder = new ArrayList<>(set);  
156 }  
157  
158 /\*\*  
159 \* Constructor that wraps (not copies) the Set and specifies the list to  
160 \* use.  
161 \* <p>  
162 \* The set and list must both be correctly initialised to the same elements.  
163 \*  
164 \* @param set the set to decorate, must not be null  
165 \* @param list the list to decorate, must not be null  
166 \* @throws NullPointerException if set or list is null  
167 \*/  
168 protected ListOrderedSet(final Set<E> set, final List<E> list) {  
169 super(set);  
170 if (list == null) {  
171 throw new NullPointerException("List must not be null");  
172 }  
173 setOrder = list;  
174 }  
175  
176 // -----------------------------------------------------------------------  
177 /\*\*  
178 \* Gets an unmodifiable view of the order of the Set.  
179 \*  
180 \* @return an unmodifiable list view  
181 \*/  
182 public List<E> asList() {  
183 return UnmodifiableList.unmodifiableList(setOrder);  
184 }  
185  
186 // -----------------------------------------------------------------------  
187 @Override  
188 public void clear() {  
189 decorated().clear();  
190 setOrder.clear();  
191 }  
192  
193 @Override  
194 public OrderedIterator<E> iterator() {  
195 return new OrderedSetIterator<>(setOrder.listIterator(), decorated());  
196 }  
197  
198 @Override  
199 public boolean add(final E object) {  
200 if (decorated().add(object)) {  
201 setOrder.add(object);  
202 return true;  
203 }  
204 return false;  
205 }  
206  
207 @Override  
208 public boolean addAll(final Collection<? extends E> coll) {  
209 boolean result = false;  
210 for (final E e : coll) {  
211 result |= add(e);  
212 }  
213 return result;  
214 }  
215  
216 @Override  
217 public boolean remove(final Object object) {  
218 final boolean result = decorated().remove(object);  
219 if (result) {  
220 setOrder.remove(object);  
221 }  
222 return result;  
223 }  
224  
225 /\*\*  
226 \* @since 4.4  
227 \*/  
228 @Override  
229 public boolean removeIf(final Predicate<? super E> filter) {  
230 if (Objects.isNull(filter)) {  
231 return false;  
232 }  
233 final boolean result = decorated().removeIf(filter);  
234 if (result) {  
235 setOrder.removeIf(filter);  
236 }  
237 return result;  
238 }  
239  
240 @Override  
241 public boolean removeAll(final Collection<?> coll) {  
242 boolean result = false;  
243 for (final Object name : coll) {  
244 result |= remove(name);  
245 }  
246 return result;  
247 }  
248  
249 /\*\*  
250 \* {@inheritDoc}  
251 \* <p>  
252 \* This implementation iterates over the elements of this set, checking  
253 \* each element in turn to see if it's contained in <code>coll</code>.  
254 \* If it's not contained, it's removed from this set. As a consequence,  
255 \* it is advised to use a collection type for <code>coll</code> that provides  
256 \* a fast (e.g. O(1)) implementation of {@link Collection#contains(Object)}.  
257 \*/  
258 @Override  
259 public boolean retainAll(final Collection<?> coll) {  
260 final boolean result = decorated().retainAll(coll);  
261 if (result == false) {  
262 return false;  
263 }  
264 if (decorated().size() == 0) {  
265 setOrder.clear();  
266 } else {  
267 for (final Iterator<E> it = setOrder.iterator(); it.hasNext();) {  
268 if (!decorated().contains(it.next())) {  
269 it.remove();  
270 }  
271 }  
272 }  
273 return result;  
274 }  
275  
276 @Override  
277 public Object[] toArray() {  
278 return setOrder.toArray();  
279 }  
280  
281 @Override  
282 public <T> T[] toArray(final T a[]) {  
283 return setOrder.toArray(a);  
284 }  
285  
286 // -----------------------------------------------------------------------  
287 // Additional methods that comply to the {@link List} interface  
288 // -----------------------------------------------------------------------  
289  
290 /\*\*  
291 \* Returns the element at the specified position in this ordered set.  
292 \*  
293 \* @param index the position of the element in the ordered {@link Set}.  
294 \* @return the element at position {@code index}  
295 \* @see List#get(int)  
296 \*/  
297 public E get(final int index) {  
298 return setOrder.get(index);  
299 }  
300  
301 /\*\*  
302 \* Returns the index of the first occurrence of the specified element in  
303 \* ordered set.  
304 \*  
305 \* @param object the element to search for  
306 \* @return the index of the first occurrence of the object, or {@code -1} if  
307 \* this ordered set does not contain this object  
308 \* @see List#indexOf(Object)  
309 \*/  
310 public int indexOf(final Object object) {  
311 return setOrder.indexOf(object);  
312 }  
313  
314 /\*\*  
315 \* Inserts the specified element at the specified position if it is not yet  
316 \* contained in this ordered set (optional operation). Shifts the element  
317 \* currently at this position and any subsequent elements to the right.  
318 \*  
319 \* @param index the index at which the element is to be inserted  
320 \* @param object the element to be inserted  
321 \* @see List#add(int, Object)  
322 \*/  
323 public void add(final int index, final E object) {  
324 if (!contains(object)) {  
325 decorated().add(object);  
326 setOrder.add(index, object);  
327 }  
328 }  
329  
330 /\*\*  
331 \* Inserts all elements in the specified collection not yet contained in the  
332 \* ordered set at the specified position (optional operation). Shifts the  
333 \* element currently at the position and all subsequent elements to the  
334 \* right.  
335 \*  
336 \* @param index the position to insert the elements  
337 \* @param coll the collection containing the elements to be inserted  
338 \* @return {@code true} if this ordered set changed as a result of the call  
339 \* @see List#addAll(int, Collection)  
340 \*/  
341 public boolean addAll(final int index, final Collection<? extends E> coll) {  
342 boolean changed = false;  
343 // collect all elements to be added for performance reasons  
344 final List<E> toAdd = new ArrayList<>();  
345 for (final E e : coll) {  
346 if (contains(e)) {  
347 continue;  
348 }  
349 decorated().add(e);  
350 toAdd.add(e);  
351 changed = true;  
352 }  
353  
354 if (changed) {  
355 setOrder.addAll(index, toAdd);  
356 }  
357  
358 return changed;  
359 }  
360  
361 /\*\*  
362 \* Removes the element at the specified position from the ordered set.  
363 \* Shifts any subsequent elements to the left.  
364 \*  
365 \* @param index the index of the element to be removed  
366 \* @return the element that has been remove from the ordered set  
367 \* @see List#remove(int)  
368 \*/  
369 public E remove(final int index) {  
370 final E obj = setOrder.remove(index);  
371 remove(obj);  
372 return obj;  
373 }  
374  
375 /\*\*  
376 \* Uses the underlying List's toString so that order is achieved. This means  
377 \* that the decorated Set's toString is not used, so any custom toStrings  
378 \* will be ignored.  
379 \*  
380 \* @return a string representation of the ordered set  
381 \*/  
382 // Fortunately List.toString and Set.toString look the same  
383 @Override  
384 public String toString() {  
385 return setOrder.toString();  
386 }  
387  
388 // -----------------------------------------------------------------------  
389 /\*\*  
390 \* Internal iterator handle remove.  
391 \*/  
392 static class OrderedSetIterator<E>  
393 extends AbstractIteratorDecorator<E>  
394 implements OrderedIterator<E> {  
395  
396 /\*\* Object we iterate on \*/  
397 private final Collection<E> set;  
398  
399 /\*\* Last object retrieved \*/  
400 private E last;  
401  
402 private OrderedSetIterator(final ListIterator<E> iterator, final Collection<E> set) {  
403 super(iterator);  
404 this.set = set;  
405 }  
406  
407 @Override  
408 public E next() {  
409 last = getIterator().next();  
410 return last;  
411 }  
412  
413 @Override  
414 public void remove() {  
415 set.remove(last);  
416 getIterator().remove();  
417 last = null;  
418 }  
419  
420 @Override  
421 public boolean hasPrevious() {  
422 return ((ListIterator<E>) getIterator()).hasPrevious();  
423 }  
424  
425 @Override  
426 public E previous() {  
427 last = ((ListIterator<E>) getIterator()).previous();  
428 return last;  
429 }  
430 }  
431  
432}