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.iterators;  
018  
019import java.util.List;  
020import java.util.ListIterator;  
021import java.util.NoSuchElementException;  
022  
023import org.apache.commons.collections4.ResettableListIterator;  
024  
025/\*\*  
026 \* A ListIterator that restarts when it reaches the end or when it  
027 \* reaches the beginning.  
028 \* <p>  
029 \* The iterator will loop continuously around the provided list,  
030 \* unless there are no elements in the collection to begin with, or  
031 \* all of the elements have been {@link #remove removed}.  
032 \* <p>  
033 \* Concurrent modifications are not directly supported, and for most  
034 \* collection implementations will throw a  
035 \* ConcurrentModificationException.  
036 \*  
037 \* @since 3.2  
038 \*/  
039public class LoopingListIterator<E> implements ResettableListIterator<E> {  
040  
041 /\*\* The list to base the iterator on \*/  
042 private final List<E> list;  
043 /\*\* The current list iterator \*/  
044 private ListIterator<E> iterator;  
045  
046 /\*\*  
047 \* Constructor that wraps a list.  
048 \* <p>  
049 \* There is no way to reset a ListIterator instance without  
050 \* recreating it from the original source, so the List must be  
051 \* passed in and a reference to it held.  
052 \*  
053 \* @param list the list to wrap  
054 \* @throws NullPointerException if the list it null  
055 \*/  
056 public LoopingListIterator(final List<E> list) {  
057 if (list == null) {  
058 throw new NullPointerException("The list must not be null");  
059 }  
060 this.list = list;  
061 \_reset();  
062 }  
063  
064 /\*\*  
065 \* Returns whether this iterator has any more elements.  
066 \* <p>  
067 \* Returns false only if the list originally had zero elements, or  
068 \* all elements have been {@link #remove removed}.  
069 \*  
070 \* @return <code>true</code> if there are more elements  
071 \*/  
072 @Override  
073 public boolean hasNext() {  
074 return !list.isEmpty();  
075 }  
076  
077 /\*\*  
078 \* Returns the next object in the list.  
079 \* <p>  
080 \* If at the end of the list, returns the first element.  
081 \*  
082 \* @return the object after the last element returned  
083 \* @throws NoSuchElementException if there are no elements in the list  
084 \*/  
085 @Override  
086 public E next() {  
087 if (list.isEmpty()) {  
088 throw new NoSuchElementException(  
089 "There are no elements for this iterator to loop on");  
090 }  
091 if (iterator.hasNext() == false) {  
092 reset();  
093 }  
094 return iterator.next();  
095 }  
096  
097 /\*\*  
098 \* Returns the index of the element that would be returned by a  
099 \* subsequent call to {@link #next}.  
100 \* <p>  
101 \* As would be expected, if the iterator is at the physical end of  
102 \* the underlying list, 0 is returned, signifying the beginning of  
103 \* the list.  
104 \*  
105 \* @return the index of the element that would be returned if next() were called  
106 \* @throws NoSuchElementException if there are no elements in the list  
107 \*/  
108 @Override  
109 public int nextIndex() {  
110 if (list.isEmpty()) {  
111 throw new NoSuchElementException(  
112 "There are no elements for this iterator to loop on");  
113 }  
114 if (iterator.hasNext() == false) {  
115 return 0;  
116 }  
117 return iterator.nextIndex();  
118 }  
119  
120 /\*\*  
121 \* Returns whether this iterator has any more previous elements.  
122 \* <p>  
123 \* Returns false only if the list originally had zero elements, or  
124 \* all elements have been {@link #remove removed}.  
125 \*  
126 \* @return <code>true</code> if there are more elements  
127 \*/  
128 @Override  
129 public boolean hasPrevious() {  
130 return !list.isEmpty();  
131 }  
132  
133 /\*\*  
134 \* Returns the previous object in the list.  
135 \* <p>  
136 \* If at the beginning of the list, return the last element. Note  
137 \* that in this case, traversal to find that element takes linear time.  
138 \*  
139 \* @return the object before the last element returned  
140 \* @throws NoSuchElementException if there are no elements in the list  
141 \*/  
142 @Override  
143 public E previous() {  
144 if (list.isEmpty()) {  
145 throw new NoSuchElementException(  
146 "There are no elements for this iterator to loop on");  
147 }  
148 if (iterator.hasPrevious() == false) {  
149 E result = null;  
150 while (iterator.hasNext()) {  
151 result = iterator.next();  
152 }  
153 iterator.previous();  
154 return result;  
155 }  
156 return iterator.previous();  
157 }  
158  
159 /\*\*  
160 \* Returns the index of the element that would be returned by a  
161 \* subsequent call to {@link #previous}.  
162 \* <p>  
163 \* As would be expected, if at the iterator is at the physical  
164 \* beginning of the underlying list, the list's size minus one is  
165 \* returned, signifying the end of the list.  
166 \*  
167 \* @return the index of the element that would be returned if previous() were called  
168 \* @throws NoSuchElementException if there are no elements in the list  
169 \*/  
170 @Override  
171 public int previousIndex() {  
172 if (list.isEmpty()) {  
173 throw new NoSuchElementException(  
174 "There are no elements for this iterator to loop on");  
175 }  
176 if (iterator.hasPrevious() == false) {  
177 return list.size() - 1;  
178 }  
179 return iterator.previousIndex();  
180 }  
181  
182 /\*\*  
183 \* Removes the previously retrieved item from the underlying list.  
184 \* <p>  
185 \* This feature is only supported if the underlying list's  
186 \* {@link List#iterator iterator} method returns an implementation  
187 \* that supports it.  
188 \* <p>  
189 \* This method can only be called after at least one {@link #next}  
190 \* or {@link #previous} method call. After a removal, the remove  
191 \* method may not be called again until another {@link #next} or  
192 \* {@link #previous} has been performed. If the {@link #reset} is  
193 \* called, then remove may not be called until {@link #next} or  
194 \* {@link #previous} is called again.  
195 \*  
196 \* @throws UnsupportedOperationException if the remove method is  
197 \* not supported by the iterator implementation of the underlying  
198 \* list  
199 \*/  
200 @Override  
201 public void remove() {  
202 iterator.remove();  
203 }  
204  
205 /\*\*  
206 \* Inserts the specified element into the underlying list.  
207 \* <p>  
208 \* The element is inserted before the next element that would be  
209 \* returned by {@link #next}, if any, and after the next element  
210 \* that would be returned by {@link #previous}, if any.  
211 \* <p>  
212 \* This feature is only supported if the underlying list's  
213 \* {@link List#listIterator} method returns an implementation  
214 \* that supports it.  
215 \*  
216 \* @param obj the element to insert  
217 \* @throws UnsupportedOperationException if the add method is not  
218 \* supported by the iterator implementation of the underlying list  
219 \*/  
220 @Override  
221 public void add(final E obj) {  
222 iterator.add(obj);  
223 }  
224  
225 /\*\*  
226 \* Replaces the last element that was returned by {@link #next} or  
227 \* {@link #previous}.  
228 \* <p>  
229 \* This feature is only supported if the underlying list's  
230 \* {@link List#listIterator} method returns an implementation  
231 \* that supports it.  
232 \*  
233 \* @param obj the element with which to replace the last element returned  
234 \* @throws UnsupportedOperationException if the set method is not  
235 \* supported by the iterator implementation of the underlying list  
236 \*/  
237 @Override  
238 public void set(final E obj) {  
239 iterator.set(obj);  
240 }  
241  
242 /\*\*  
243 \* Resets the iterator back to the start of the list.  
244 \*/  
245 @Override  
246 public void reset() {  
247 \_reset();  
248 }  
249  
250 private void \_reset() {  
251 iterator = list.listIterator();  
252 }  
253  
254 /\*\*  
255 \* Gets the size of the list underlying the iterator.  
256 \*  
257 \* @return the current list size  
258 \*/  
259 public int size() {  
260 return list.size();  
261 }  
262  
263}