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.text.MessageFormat;  
020import java.util.ArrayList;  
021import java.util.Iterator;  
022import java.util.List;  
023import java.util.ListIterator;  
024import java.util.NoSuchElementException;  
025  
026import org.apache.commons.collections4.ResettableListIterator;  
027  
028/\*\*  
029 \* Converts an {@link Iterator} into a {@link ResettableListIterator}.  
030 \* For plain <code>Iterator</code>s this is accomplished by caching the returned  
031 \* elements. This class can also be used to simply add  
032 \* {@link org.apache.commons.collections4.ResettableIterator ResettableIterator}  
033 \* functionality to a given {@link ListIterator}.  
034 \* <p>  
035 \* The <code>ListIterator</code> interface has additional useful methods  
036 \* for navigation - <code>previous()</code> and the index methods.  
037 \* This class allows a regular <code>Iterator</code> to behave as a  
038 \* <code>ListIterator</code>. It achieves this by building a list internally  
039 \* of as the underlying iterator is traversed.  
040 \* <p>  
041 \* The optional operations of <code>ListIterator</code> are not supported for plain <code>Iterator</code>s.  
042 \* <p>  
043 \* This class implements ResettableListIterator from Commons Collections 3.2.  
044 \*  
045 \* @since 2.1  
046 \*/  
047public class ListIteratorWrapper<E> implements ResettableListIterator<E> {  
048  
049 /\*\* Message used when set or add are called. \*/  
050 private static final String UNSUPPORTED\_OPERATION\_MESSAGE =  
051 "ListIteratorWrapper does not support optional operations of ListIterator.";  
052  
053 /\*\* Message used when set or add are called. \*/  
054 private static final String CANNOT\_REMOVE\_MESSAGE = "Cannot remove element at index {0}.";  
055  
056 /\*\* The underlying iterator being decorated. \*/  
057 private final Iterator<? extends E> iterator;  
058 /\*\* The list being used to cache the iterator. \*/  
059 private final List<E> list = new ArrayList<>();  
060  
061 /\*\* The current index of this iterator. \*/  
062 private int currentIndex = 0;  
063 /\*\* The current index of the wrapped iterator. \*/  
064 private int wrappedIteratorIndex = 0;  
065 /\*\* recall whether the wrapped iterator's "cursor" is in such a state as to allow remove() to be called \*/  
066 private boolean removeState;  
067  
068 // Constructor  
069 //-------------------------------------------------------------------------  
070 /\*\*  
071 \* Constructs a new <code>ListIteratorWrapper</code> that will wrap  
072 \* the given iterator.  
073 \*  
074 \* @param iterator the iterator to wrap  
075 \* @throws NullPointerException if the iterator is null  
076 \*/  
077 public ListIteratorWrapper(final Iterator<? extends E> iterator) {  
078 super();  
079 if (iterator == null) {  
080 throw new NullPointerException("Iterator must not be null");  
081 }  
082 this.iterator = iterator;  
083 }  
084  
085 // ListIterator interface  
086 //-------------------------------------------------------------------------  
087 /\*\*  
088 \* Throws {@link UnsupportedOperationException}  
089 \* unless the underlying <code>Iterator</code> is a <code>ListIterator</code>.  
090 \*  
091 \* @param obj the object to add  
092 \* @throws UnsupportedOperationException if the underlying iterator is not of  
093 \* type {@link ListIterator}  
094 \*/  
095 @Override  
096 public void add(final E obj) throws UnsupportedOperationException {  
097 if (iterator instanceof ListIterator) {  
098 @SuppressWarnings("unchecked")  
099 final ListIterator<E> li = (ListIterator<E>) iterator;  
100 li.add(obj);  
101 return;  
102 }  
103 throw new UnsupportedOperationException(UNSUPPORTED\_OPERATION\_MESSAGE);  
104 }  
105  
106 /\*\*  
107 \* Returns true if there are more elements in the iterator.  
108 \*  
109 \* @return true if there are more elements  
110 \*/  
111 @Override  
112 public boolean hasNext() {  
113 if (currentIndex == wrappedIteratorIndex || iterator instanceof ListIterator) {  
114 return iterator.hasNext();  
115 }  
116 return true;  
117 }  
118  
119 /\*\*  
120 \* Returns true if there are previous elements in the iterator.  
121 \*  
122 \* @return true if there are previous elements  
123 \*/  
124 @Override  
125 public boolean hasPrevious() {  
126 if (iterator instanceof ListIterator) {  
127 final ListIterator<?> li = (ListIterator<?>) iterator;  
128 return li.hasPrevious();  
129 }  
130 return currentIndex > 0;  
131 }  
132  
133 /\*\*  
134 \* Returns the next element from the iterator.  
135 \*  
136 \* @return the next element from the iterator  
137 \* @throws NoSuchElementException if there are no more elements  
138 \*/  
139 @Override  
140 public E next() throws NoSuchElementException {  
141 if (iterator instanceof ListIterator) {  
142 return iterator.next();  
143 }  
144  
145 if (currentIndex < wrappedIteratorIndex) {  
146 ++currentIndex;  
147 return list.get(currentIndex - 1);  
148 }  
149  
150 final E retval = iterator.next();  
151 list.add(retval);  
152 ++currentIndex;  
153 ++wrappedIteratorIndex;  
154 removeState = true;  
155 return retval;  
156 }  
157  
158 /\*\*  
159 \* Returns the index of the next element.  
160 \*  
161 \* @return the index of the next element  
162 \*/  
163 @Override  
164 public int nextIndex() {  
165 if (iterator instanceof ListIterator) {  
166 final ListIterator<?> li = (ListIterator<?>) iterator;  
167 return li.nextIndex();  
168 }  
169 return currentIndex;  
170 }  
171  
172 /\*\*  
173 \* Returns the previous element.  
174 \*  
175 \* @return the previous element  
176 \* @throws NoSuchElementException if there are no previous elements  
177 \*/  
178 @Override  
179 public E previous() throws NoSuchElementException {  
180 if (iterator instanceof ListIterator) {  
181 @SuppressWarnings("unchecked")  
182 final ListIterator<E> li = (ListIterator<E>) iterator;  
183 return li.previous();  
184 }  
185  
186 if (currentIndex == 0) {  
187 throw new NoSuchElementException();  
188 }  
189 removeState = wrappedIteratorIndex == currentIndex;  
190 return list.get(--currentIndex);  
191 }  
192  
193 /\*\*  
194 \* Returns the index of the previous element.  
195 \*  
196 \* @return the index of the previous element  
197 \*/  
198 @Override  
199 public int previousIndex() {  
200 if (iterator instanceof ListIterator) {  
201 final ListIterator<?> li = (ListIterator<?>) iterator;  
202 return li.previousIndex();  
203 }  
204 return currentIndex - 1;  
205 }  
206  
207 /\*\*  
208 \* Throws {@link UnsupportedOperationException} if {@link #previous()} has ever been called.  
209 \*  
210 \* @throws UnsupportedOperationException always  
211 \*/  
212 @Override  
213 public void remove() throws UnsupportedOperationException {  
214 if (iterator instanceof ListIterator) {  
215 iterator.remove();  
216 return;  
217 }  
218 int removeIndex = currentIndex;  
219 if (currentIndex == wrappedIteratorIndex) {  
220 --removeIndex;  
221 }  
222 if (!removeState || wrappedIteratorIndex - currentIndex > 1) {  
223 throw new IllegalStateException(MessageFormat.format(CANNOT\_REMOVE\_MESSAGE, Integer.valueOf(removeIndex)));  
224 }  
225 iterator.remove();  
226 list.remove(removeIndex);  
227 currentIndex = removeIndex;  
228 wrappedIteratorIndex--;  
229 removeState = false;  
230 }  
231  
232 /\*\*  
233 \* Throws {@link UnsupportedOperationException}  
234 \* unless the underlying <code>Iterator</code> is a <code>ListIterator</code>.  
235 \*  
236 \* @param obj the object to set  
237 \* @throws UnsupportedOperationException if the underlying iterator is not of  
238 \* type {@link ListIterator}  
239 \*/  
240 @Override  
241 public void set(final E obj) throws UnsupportedOperationException {  
242 if (iterator instanceof ListIterator) {  
243 @SuppressWarnings("unchecked")  
244 final ListIterator<E> li = (ListIterator<E>) iterator;  
245 li.set(obj);  
246 return;  
247 }  
248 throw new UnsupportedOperationException(UNSUPPORTED\_OPERATION\_MESSAGE);  
249 }  
250  
251 // ResettableIterator interface  
252 //-------------------------------------------------------------------------  
253 /\*\*  
254 \* Resets this iterator back to the position at which the iterator  
255 \* was created.  
256 \*  
257 \* @since 3.2  
258 \*/  
259 @Override  
260 public void reset() {  
261 if (iterator instanceof ListIterator) {  
262 final ListIterator<?> li = (ListIterator<?>) iterator;  
263 while (li.previousIndex() >= 0) {  
264 li.previous();  
265 }  
266 return;  
267 }  
268 currentIndex = 0;  
269 }  
270  
271}