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.ListIterator;  
020import java.util.NoSuchElementException;  
021  
022import org.apache.commons.collections4.Predicate;  
023  
024/\*\*  
025 \* Decorates another {@link ListIterator} using a predicate to filter elements.  
026 \* <p>  
027 \* This iterator decorates the underlying iterator, only allowing through  
028 \* those elements that match the specified {@link Predicate Predicate}.  
029 \*  
030 \* @since 2.0  
031 \*/  
032public class FilterListIterator<E> implements ListIterator<E> {  
033  
034 /\*\* The iterator being used \*/  
035 private ListIterator<? extends E> iterator;  
036  
037 /\*\* The predicate being used \*/  
038 private Predicate<? super E> predicate;  
039  
040 /\*\*  
041 \* The value of the next (matching) object, when  
042 \* {@link #nextObjectSet} is true.  
043 \*/  
044 private E nextObject;  
045  
046 /\*\*  
047 \* Whether or not the {@link #nextObject} has been set  
048 \* (possibly to <code>null</code>).  
049 \*/  
050 private boolean nextObjectSet = false;  
051  
052 /\*\*  
053 \* The value of the previous (matching) object, when  
054 \* {@link #previousObjectSet} is true.  
055 \*/  
056 private E previousObject;  
057  
058 /\*\*  
059 \* Whether or not the {@link #previousObject} has been set  
060 \* (possibly to <code>null</code>).  
061 \*/  
062 private boolean previousObjectSet = false;  
063  
064 /\*\*  
065 \* The index of the element that would be returned by {@link #next}.  
066 \*/  
067 private int nextIndex = 0;  
068  
069 //-----------------------------------------------------------------------  
070 /\*\*  
071 \* Constructs a new <code>FilterListIterator</code> that will not function  
072 \* until {@link #setListIterator(ListIterator) setListIterator}  
073 \* and {@link #setPredicate(Predicate) setPredicate} are invoked.  
074 \*/  
075 public FilterListIterator() {  
076 super();  
077 }  
078  
079 /\*\*  
080 \* Constructs a new <code>FilterListIterator</code> that will not  
081 \* function until {@link #setPredicate(Predicate) setPredicate} is invoked.  
082 \*  
083 \* @param iterator the iterator to use  
084 \*/  
085 public FilterListIterator(final ListIterator<? extends E> iterator ) {  
086 super();  
087 this.iterator = iterator;  
088 }  
089  
090 /\*\*  
091 \* Constructs a new <code>FilterListIterator</code>.  
092 \*  
093 \* @param iterator the iterator to use  
094 \* @param predicate the predicate to use  
095 \*/  
096 public FilterListIterator(final ListIterator<? extends E> iterator, final Predicate<? super E> predicate) {  
097 super();  
098 this.iterator = iterator;  
099 this.predicate = predicate;  
100 }  
101  
102 /\*\*  
103 \* Constructs a new <code>FilterListIterator</code> that will not function  
104 \* until {@link #setListIterator(ListIterator) setListIterator} is invoked.  
105 \*  
106 \* @param predicate the predicate to use.  
107 \*/  
108 public FilterListIterator(final Predicate<? super E> predicate) {  
109 super();  
110 this.predicate = predicate;  
111 }  
112  
113 //-----------------------------------------------------------------------  
114 /\*\*  
115 \* Not supported.  
116 \* @param o the element to insert  
117 \*/  
118 @Override  
119 public void add(final E o) {  
120 throw new UnsupportedOperationException("FilterListIterator.add(Object) is not supported.");  
121 }  
122  
123 @Override  
124 public boolean hasNext() {  
125 return nextObjectSet || setNextObject();  
126 }  
127  
128 @Override  
129 public boolean hasPrevious() {  
130 return previousObjectSet || setPreviousObject();  
131 }  
132  
133 @Override  
134 public E next() {  
135 if (!nextObjectSet && !setNextObject()) {  
136 throw new NoSuchElementException();  
137 }  
138 nextIndex++;  
139 final E temp = nextObject;  
140 clearNextObject();  
141 return temp;  
142 }  
143  
144 @Override  
145 public int nextIndex() {  
146 return nextIndex;  
147 }  
148  
149 @Override  
150 public E previous() {  
151 if (!previousObjectSet && !setPreviousObject()) {  
152 throw new NoSuchElementException();  
153 }  
154 nextIndex--;  
155 final E temp = previousObject;  
156 clearPreviousObject();  
157 return temp;  
158 }  
159  
160 @Override  
161 public int previousIndex() {  
162 return nextIndex-1;  
163 }  
164  
165 /\*\* Not supported. \*/  
166 @Override  
167 public void remove() {  
168 throw new UnsupportedOperationException("FilterListIterator.remove() is not supported.");  
169 }  
170  
171 /\*\*  
172 \* Not supported.  
173 \* @param o the element with which to replace the last element returned by  
174 \* {@code next} or {@code previous}  
175 \*/  
176 @Override  
177 public void set(final E o) {  
178 throw new UnsupportedOperationException("FilterListIterator.set(Object) is not supported.");  
179 }  
180  
181 //-----------------------------------------------------------------------  
182 /\*\*  
183 \* Gets the iterator this iterator is using.  
184 \*  
185 \* @return the iterator.  
186 \*/  
187 public ListIterator<? extends E> getListIterator() {  
188 return iterator;  
189 }  
190  
191 /\*\*  
192 \* Sets the iterator for this iterator to use.  
193 \* If iteration has started, this effectively resets the iterator.  
194 \*  
195 \* @param iterator the iterator to use  
196 \*/  
197 public void setListIterator(final ListIterator<? extends E> iterator) {  
198 this.iterator = iterator;  
199 }  
200  
201 //-----------------------------------------------------------------------  
202 /\*\*  
203 \* Gets the predicate this iterator is using.  
204 \*  
205 \* @return the predicate.  
206 \*/  
207 public Predicate<? super E> getPredicate() {  
208 return predicate;  
209 }  
210  
211 /\*\*  
212 \* Sets the predicate this the iterator to use.  
213 \*  
214 \* @param predicate the transformer to use  
215 \*/  
216 public void setPredicate(final Predicate<? super E> predicate) {  
217 this.predicate = predicate;  
218 }  
219  
220 //-----------------------------------------------------------------------  
221 private void clearNextObject() {  
222 nextObject = null;  
223 nextObjectSet = false;  
224 }  
225  
226 private boolean setNextObject() {  
227 // if previousObjectSet,  
228 // then we've walked back one step in the  
229 // underlying list (due to a hasPrevious() call)  
230 // so skip ahead one matching object  
231 if (previousObjectSet) {  
232 clearPreviousObject();  
233 if (!setNextObject()) {  
234 return false;  
235 }  
236 clearNextObject();  
237 }  
238  
239 if (iterator == null) {  
240 return false;  
241 }  
242 while (iterator.hasNext()) {  
243 final E object = iterator.next();  
244 if (predicate.evaluate(object)) {  
245 nextObject = object;  
246 nextObjectSet = true;  
247 return true;  
248 }  
249 }  
250 return false;  
251 }  
252  
253 private void clearPreviousObject() {  
254 previousObject = null;  
255 previousObjectSet = false;  
256 }  
257  
258 private boolean setPreviousObject() {  
259 // if nextObjectSet,  
260 // then we've walked back one step in the  
261 // underlying list (due to a hasNext() call)  
262 // so skip ahead one matching object  
263 if (nextObjectSet) {  
264 clearNextObject();  
265 if (!setPreviousObject()) {  
266 return false;  
267 }  
268 clearPreviousObject();  
269 }  
270  
271 if (iterator == null) {  
272 return false;  
273 }  
274 while (iterator.hasPrevious()) {  
275 final E object = iterator.previous();  
276 if (predicate.evaluate(object)) {  
277 previousObject = object;  
278 previousObjectSet = true;  
279 return true;  
280 }  
281 }  
282 return false;  
283 }  
284  
285}