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.Collection;  
020import java.util.Iterator;  
021import java.util.LinkedList;  
022import java.util.Queue;  
023  
024/\*\*  
025 \* An IteratorChain is an Iterator that wraps a number of Iterators.  
026 \* <p>  
027 \* This class makes multiple iterators look like one to the caller. When any  
028 \* method from the Iterator interface is called, the IteratorChain will delegate  
029 \* to a single underlying Iterator. The IteratorChain will invoke the Iterators  
030 \* in sequence until all Iterators are exhausted.  
031 \* <p>  
032 \* Under many circumstances, linking Iterators together in this manner is more  
033 \* efficient (and convenient) than reading out the contents of each Iterator  
034 \* into a List and creating a new Iterator.  
035 \* <p>  
036 \* Calling a method that adds new Iterator <i>after a method in the Iterator  
037 \* interface has been called</i> will result in an UnsupportedOperationException.  
038 \* <p>  
039 \* NOTE: As from version 3.0, the IteratorChain may contain no iterators. In  
040 \* this case the class will function as an empty iterator.  
041 \* <p>  
042 \* NOTE: As from version 4.0, the IteratorChain stores the iterators in a queue  
043 \* and removes any reference to them as soon as they are not used anymore. Thus  
044 \* the methods {@code setIterator(Iterator)} and {@code getIterators()} have been  
045 \* removed and {@link #size()} will return the number of remaining iterators in  
046 \* the queue.  
047 \*  
048 \* @since 2.1  
049 \*/  
050public class IteratorChain<E> implements Iterator<E> {  
051  
052 /\*\* The chain of iterators \*/  
053 private final Queue<Iterator<? extends E>> iteratorChain = new LinkedList<>();  
054  
055 /\*\* The current iterator \*/  
056 private Iterator<? extends E> currentIterator = null;  
057  
058 /\*\*  
059 \* The "last used" Iterator is the Iterator upon which next() or hasNext()  
060 \* was most recently called used for the remove() operation only  
061 \*/  
062 private Iterator<? extends E> lastUsedIterator = null;  
063  
064 /\*\*  
065 \* ComparatorChain is "locked" after the first time compare(Object,Object)  
066 \* is called  
067 \*/  
068 private boolean isLocked = false;  
069  
070 //-----------------------------------------------------------------------  
071 /\*\*  
072 \* Construct an IteratorChain with no Iterators.  
073 \* <p>  
074 \* You will normally use {@link #addIterator(Iterator)} to add some  
075 \* iterators after using this constructor.  
076 \*/  
077 public IteratorChain() {  
078 super();  
079 }  
080  
081 /\*\*  
082 \* Construct an IteratorChain with a single Iterator.  
083 \* <p>  
084 \* This method takes one iterator. The newly constructed iterator will  
085 \* iterate through that iterator. Thus calling this constructor on its own  
086 \* will have no effect other than decorating the input iterator.  
087 \* <p>  
088 \* You will normally use {@link #addIterator(Iterator)} to add some more  
089 \* iterators after using this constructor.  
090 \*  
091 \* @param iterator the first child iterator in the IteratorChain, not null  
092 \* @throws NullPointerException if the iterator is null  
093 \*/  
094 public IteratorChain(final Iterator<? extends E> iterator) {  
095 super();  
096 addIterator(iterator);  
097 }  
098  
099 /\*\*  
100 \* Constructs a new <code>IteratorChain</code> over the two given iterators.  
101 \* <p>  
102 \* This method takes two iterators. The newly constructed iterator will  
103 \* iterate through each one of the input iterators in turn.  
104 \*  
105 \* @param first the first child iterator in the IteratorChain, not null  
106 \* @param second the second child iterator in the IteratorChain, not null  
107 \* @throws NullPointerException if either iterator is null  
108 \*/  
109 public IteratorChain(final Iterator<? extends E> first, final Iterator<? extends E> second) {  
110 super();  
111 addIterator(first);  
112 addIterator(second);  
113 }  
114  
115 /\*\*  
116 \* Constructs a new <code>IteratorChain</code> over the array of iterators.  
117 \* <p>  
118 \* This method takes an array of iterators. The newly constructed iterator  
119 \* will iterate through each one of the input iterators in turn.  
120 \*  
121 \* @param iteratorChain the array of iterators, not null  
122 \* @throws NullPointerException if iterators array is or contains null  
123 \*/  
124 public IteratorChain(final Iterator<? extends E>... iteratorChain) {  
125 super();  
126 for (final Iterator<? extends E> element : iteratorChain) {  
127 addIterator(element);  
128 }  
129 }  
130  
131 /\*\*  
132 \* Constructs a new <code>IteratorChain</code> over the collection of  
133 \* iterators.  
134 \* <p>  
135 \* This method takes a collection of iterators. The newly constructed  
136 \* iterator will iterate through each one of the input iterators in turn.  
137 \*  
138 \* @param iteratorChain the collection of iterators, not null  
139 \* @throws NullPointerException if iterators collection is or contains null  
140 \* @throws ClassCastException if iterators collection doesn't contain an  
141 \* iterator  
142 \*/  
143 public IteratorChain(final Collection<Iterator<? extends E>> iteratorChain) {  
144 super();  
145 for (final Iterator<? extends E> iterator : iteratorChain) {  
146 addIterator(iterator);  
147 }  
148 }  
149  
150 //-----------------------------------------------------------------------  
151 /\*\*  
152 \* Add an Iterator to the end of the chain  
153 \*  
154 \* @param iterator Iterator to add  
155 \* @throws IllegalStateException if I've already started iterating  
156 \* @throws NullPointerException if the iterator is null  
157 \*/  
158 public void addIterator(final Iterator<? extends E> iterator) {  
159 checkLocked();  
160 if (iterator == null) {  
161 throw new NullPointerException("Iterator must not be null");  
162 }  
163 iteratorChain.add(iterator);  
164 }  
165  
166 /\*\*  
167 \* Returns the remaining number of Iterators in the current IteratorChain.  
168 \*  
169 \* @return Iterator count  
170 \*/  
171 public int size() {  
172 return iteratorChain.size();  
173 }  
174  
175 /\*\*  
176 \* Determine if modifications can still be made to the IteratorChain.  
177 \* IteratorChains cannot be modified once they have executed a method from  
178 \* the Iterator interface.  
179 \*  
180 \* @return true if IteratorChain cannot be modified, false if it can  
181 \*/  
182 public boolean isLocked() {  
183 return isLocked;  
184 }  
185  
186 /\*\*  
187 \* Checks whether the iterator chain is now locked and in use.  
188 \*/  
189 private void checkLocked() {  
190 if (isLocked == true) {  
191 throw new UnsupportedOperationException(  
192 "IteratorChain cannot be changed after the first use of a method from the Iterator interface");  
193 }  
194 }  
195  
196 /\*\*  
197 \* Lock the chain so no more iterators can be added. This must be called  
198 \* from all Iterator interface methods.  
199 \*/  
200 private void lockChain() {  
201 if (isLocked == false) {  
202 isLocked = true;  
203 }  
204 }  
205  
206 /\*\*  
207 \* Updates the current iterator field to ensure that the current Iterator is  
208 \* not exhausted  
209 \*/  
210 protected void updateCurrentIterator() {  
211 if (currentIterator == null) {  
212 if (iteratorChain.isEmpty()) {  
213 currentIterator = EmptyIterator.<E> emptyIterator();  
214 } else {  
215 currentIterator = iteratorChain.remove();  
216 }  
217 // set last used iterator here, in case the user calls remove  
218 // before calling hasNext() or next() (although they shouldn't)  
219 lastUsedIterator = currentIterator;  
220 }  
221  
222 while (currentIterator.hasNext() == false && !iteratorChain.isEmpty()) {  
223 currentIterator = iteratorChain.remove();  
224 }  
225 }  
226  
227 //-----------------------------------------------------------------------  
228 /\*\*  
229 \* Return true if any Iterator in the IteratorChain has a remaining element.  
230 \*  
231 \* @return true if elements remain  
232 \*/  
233 @Override  
234 public boolean hasNext() {  
235 lockChain();  
236 updateCurrentIterator();  
237 lastUsedIterator = currentIterator;  
238  
239 return currentIterator.hasNext();  
240 }  
241  
242 /\*\*  
243 \* Returns the next Object of the current Iterator  
244 \*  
245 \* @return Object from the current Iterator  
246 \* @throws java.util.NoSuchElementException if all the Iterators are  
247 \* exhausted  
248 \*/  
249 @Override  
250 public E next() {  
251 lockChain();  
252 updateCurrentIterator();  
253 lastUsedIterator = currentIterator;  
254  
255 return currentIterator.next();  
256 }  
257  
258 /\*\*  
259 \* Removes from the underlying collection the last element returned by the  
260 \* Iterator. As with next() and hasNext(), this method calls remove() on the  
261 \* underlying Iterator. Therefore, this method may throw an  
262 \* UnsupportedOperationException if the underlying Iterator does not support  
263 \* this method.  
264 \*  
265 \* @throws UnsupportedOperationException if the remove operator is not  
266 \* supported by the underlying Iterator  
267 \* @throws IllegalStateException if the next method has not yet been called,  
268 \* or the remove method has already been called after the last call to the  
269 \* next method.  
270 \*/  
271 @Override  
272 public void remove() {  
273 lockChain();  
274 if (currentIterator == null) {  
275 updateCurrentIterator();  
276 }  
277 lastUsedIterator.remove();  
278 }  
279  
280}