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.Iterator;  
020  
021/\*\*  
022 \* An LazyIteratorChain is an Iterator that wraps a number of Iterators in a lazy manner.  
023 \* <p>  
024 \* This class makes multiple iterators look like one to the caller. When any  
025 \* method from the Iterator interface is called, the LazyIteratorChain will delegate  
026 \* to a single underlying Iterator. The LazyIteratorChain will invoke the Iterators  
027 \* in sequence until all Iterators are exhausted.  
028 \* <p>  
029 \* The Iterators are provided by {@link #nextIterator(int)} which has to be overridden by  
030 \* sub-classes and allows to lazily create the Iterators as they are accessed:  
031 \* <pre>  
032 \* return new LazyIteratorChain<String>() {  
033 \* protected Iterator<String> nextIterator(int count) {  
034 \* return count == 1 ? Arrays.asList("foo", "bar").iterator() : null;  
035 \* }  
036 \* };  
037 \* </pre>  
038 \* <p>  
039 \* Once the inner Iterator's {@link Iterator#hasNext()} method returns false,  
040 \* {@link #nextIterator(int)} will be called to obtain another iterator, and so on  
041 \* until {@link #nextIterator(int)} returns null, indicating that the chain is exhausted.  
042 \* <p>  
043 \* NOTE: The LazyIteratorChain may contain no iterators. In this case the class will  
044 \* function as an empty iterator.  
045 \*  
046 \* @since 4.0  
047 \*/  
048public abstract class LazyIteratorChain<E> implements Iterator<E> {  
049  
050 /\*\* The number of times {@link #next()} was already called. \*/  
051 private int callCounter = 0;  
052  
053 /\*\* Indicates that the Iterator chain has been exhausted. \*/  
054 private boolean chainExhausted = false;  
055  
056 /\*\* The current iterator. \*/  
057 private Iterator<? extends E> currentIterator = null;  
058  
059 /\*\*  
060 \* The "last used" Iterator is the Iterator upon which next() or hasNext()  
061 \* was most recently called used for the remove() operation only.  
062 \*/  
063 private Iterator<? extends E> lastUsedIterator = null;  
064  
065 //-----------------------------------------------------------------------  
066  
067 /\*\*  
068 \* Gets the next iterator after the previous one has been exhausted.  
069 \* <p>  
070 \* This method <b>MUST</b> return null when there are no more iterators.  
071 \*  
072 \* @param count the number of time this method has been called (starts with 1)  
073 \* @return the next iterator, or null if there are no more.  
074 \*/  
075 protected abstract Iterator<? extends E> nextIterator(int count);  
076  
077 /\*\*  
078 \* Updates the current iterator field to ensure that the current Iterator  
079 \* is not exhausted.  
080 \*/  
081 private void updateCurrentIterator() {  
082 if (callCounter == 0) {  
083 currentIterator = nextIterator(++callCounter);  
084 if (currentIterator == null) {  
085 currentIterator = EmptyIterator.<E>emptyIterator();  
086 chainExhausted = true;  
087 }  
088 // set last used iterator here, in case the user calls remove  
089 // before calling hasNext() or next() (although they shouldn't)  
090 lastUsedIterator = currentIterator;  
091 }  
092  
093 while (currentIterator.hasNext() == false && !chainExhausted) {  
094 final Iterator<? extends E> nextIterator = nextIterator(++callCounter);  
095 if (nextIterator != null) {  
096 currentIterator = nextIterator;  
097 } else {  
098 chainExhausted = true;  
099 }  
100 }  
101 }  
102  
103 //-----------------------------------------------------------------------  
104  
105 /\*\*  
106 \* Return true if any Iterator in the chain has a remaining element.  
107 \*  
108 \* @return true if elements remain  
109 \*/  
110 @Override  
111 public boolean hasNext() {  
112 updateCurrentIterator();  
113 lastUsedIterator = currentIterator;  
114  
115 return currentIterator.hasNext();  
116 }  
117  
118 /\*\*  
119 \* Returns the next element of the current Iterator  
120 \*  
121 \* @return element from the current Iterator  
122 \* @throws java.util.NoSuchElementException if all the Iterators are exhausted  
123 \*/  
124 @Override  
125 public E next() {  
126 updateCurrentIterator();  
127 lastUsedIterator = currentIterator;  
128  
129 return currentIterator.next();  
130 }  
131  
132 /\*\*  
133 \* Removes from the underlying collection the last element returned by the Iterator.  
134 \* <p>  
135 \* As with next() and hasNext(), this method calls remove() on the underlying Iterator.  
136 \* Therefore, this method may throw an UnsupportedOperationException if the underlying  
137 \* Iterator does not support this method.  
138 \*  
139 \* @throws UnsupportedOperationException if the remove operator is not  
140 \* supported by the underlying Iterator  
141 \* @throws IllegalStateException if the next method has not yet been called,  
142 \* or the remove method has already been called after the last call to the next method.  
143 \*/  
144 @Override  
145 public void remove() {  
146 if (currentIterator == null) {  
147 updateCurrentIterator();  
148 }  
149 lastUsedIterator.remove();  
150 }  
151  
152}