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.ArrayList;  
020import java.util.Iterator;  
021import java.util.List;  
022import java.util.NoSuchElementException;  
023  
024import org.apache.commons.collections4.FluentIterable;  
025  
026/\*\*  
027 \* Provides an interleaved iteration over the elements contained in a  
028 \* collection of Iterators.  
029 \* <p>  
030 \* Given two {@link Iterator} instances {@code A} and {@code B}, the  
031 \* {@link #next} method on this iterator will switch between {@code A.next()}  
032 \* and {@code B.next()} until both iterators are exhausted.  
033 \* </p>  
034 \*  
035 \* @since 4.1  
036 \*/  
037public class ZippingIterator<E> implements Iterator<E> {  
038  
039 /\*\* The {@link Iterator}s to evaluate. \*/  
040 private final Iterator<Iterator<? extends E>> iterators;  
041  
042 /\*\* The next iterator to use for next(). \*/  
043 private Iterator<? extends E> nextIterator = null;  
044  
045 /\*\* The last iterator which was used for next(). \*/  
046 private Iterator<? extends E> lastReturned = null;  
047  
048 // Constructors  
049 // ----------------------------------------------------------------------  
050  
051 /\*\*  
052 \* Constructs a new <code>ZippingIterator</code> that will provide  
053 \* interleaved iteration over the two given iterators.  
054 \*  
055 \* @param a the first child iterator  
056 \* @param b the second child iterator  
057 \* @throws NullPointerException if either iterator is null  
058 \*/  
059 @SuppressWarnings("unchecked")  
060 public ZippingIterator(final Iterator<? extends E> a, final Iterator<? extends E> b) {  
061 this(new Iterator[] {a, b});  
062 }  
063  
064 /\*\*  
065 \* Constructs a new <code>ZippingIterator</code> that will provide  
066 \* interleaved iteration over the three given iterators.  
067 \*  
068 \* @param a the first child iterator  
069 \* @param b the second child iterator  
070 \* @param c the third child iterator  
071 \* @throws NullPointerException if either iterator is null  
072 \*/  
073 @SuppressWarnings("unchecked")  
074 public ZippingIterator(final Iterator<? extends E> a,  
075 final Iterator<? extends E> b,  
076 final Iterator<? extends E> c) {  
077 this(new Iterator[] {a, b, c});  
078 }  
079  
080 /\*\*  
081 \* Constructs a new <code>ZippingIterator</code> that will provide  
082 \* interleaved iteration of the specified iterators.  
083 \*  
084 \* @param iterators the array of iterators  
085 \* @throws NullPointerException if any iterator is null  
086 \*/  
087 public ZippingIterator(final Iterator<? extends E>... iterators) {  
088 // create a mutable list to be able to remove exhausted iterators  
089 final List<Iterator<? extends E>> list = new ArrayList<>();  
090 for (final Iterator<? extends E> iterator : iterators) {  
091 if (iterator == null) {  
092 throw new NullPointerException("Iterator must not be null.");  
093 }  
094 list.add(iterator);  
095 }  
096 this.iterators = FluentIterable.of(list).loop().iterator();  
097 }  
098  
099 // Iterator Methods  
100 // -------------------------------------------------------------------  
101  
102 /\*\*  
103 \* Returns {@code true} if any child iterator has remaining elements.  
104 \*  
105 \* @return true if this iterator has remaining elements  
106 \*/  
107 @Override  
108 public boolean hasNext() {  
109 // the next iterator has already been determined  
110 // this might happen if hasNext() is called multiple  
111 if (nextIterator != null) {  
112 return true;  
113 }  
114  
115 while(iterators.hasNext()) {  
116 final Iterator<? extends E> childIterator = iterators.next();  
117 if (childIterator.hasNext()) {  
118 nextIterator = childIterator;  
119 return true;  
120 }  
121 // iterator is exhausted, remove it  
122 iterators.remove();  
123 }  
124 return false;  
125 }  
126  
127 /\*\*  
128 \* Returns the next element from a child iterator.  
129 \*  
130 \* @return the next interleaved element  
131 \* @throws NoSuchElementException if no child iterator has any more elements  
132 \*/  
133 @Override  
134 public E next() throws NoSuchElementException {  
135 if (!hasNext()) {  
136 throw new NoSuchElementException();  
137 }  
138  
139 final E val = nextIterator.next();  
140 lastReturned = nextIterator;  
141 nextIterator = null;  
142 return val;  
143 }  
144  
145 /\*\*  
146 \* Removes the last returned element from the child iterator that produced it.  
147 \*  
148 \* @throws IllegalStateException if there is no last returned element, or if  
149 \* the last returned element has already been removed  
150 \*/  
151 @Override  
152 public void remove() {  
153 if (lastReturned == null) {  
154 throw new IllegalStateException("No value can be removed at present");  
155 }  
156 lastReturned.remove();  
157 lastReturned = null;  
158 }  
159  
160}