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.ArrayDeque;  
020import java.util.Deque;  
021import java.util.Iterator;  
022import java.util.NoSuchElementException;  
023  
024import org.apache.commons.collections4.Transformer;  
025  
026/\*\*  
027 \* An Iterator that can traverse multiple iterators down an object graph.  
028 \* <p>  
029 \* This iterator can extract multiple objects from a complex tree-like object graph.  
030 \* The iteration starts from a single root object.  
031 \* It uses a <code>Transformer</code> to extract the iterators and elements.  
032 \* Its main benefit is that no intermediate <code>List</code> is created.  
033 \* <p>  
034 \* For example, consider an object graph:  
035 \* <pre>  
036 \* |- Branch -- Leaf  
037 \* | \- Leaf  
038 \* |- Tree | /- Leaf  
039 \* | |- Branch -- Leaf  
040 \* Forest | \- Leaf  
041 \* | |- Branch -- Leaf  
042 \* | | \- Leaf  
043 \* |- Tree | /- Leaf  
044 \* |- Branch -- Leaf  
045 \* |- Branch -- Leaf</pre>  
046 \* The following <code>Transformer</code>, used in this class, will extract all  
047 \* the Leaf objects without creating a combined intermediate list:  
048 \* <pre>  
049 \* public Object transform(Object input) {  
050 \* if (input instanceof Forest) {  
051 \* return ((Forest) input).treeIterator();  
052 \* }  
053 \* if (input instanceof Tree) {  
054 \* return ((Tree) input).branchIterator();  
055 \* }  
056 \* if (input instanceof Branch) {  
057 \* return ((Branch) input).leafIterator();  
058 \* }  
059 \* if (input instanceof Leaf) {  
060 \* return input;  
061 \* }  
062 \* throw new ClassCastException();  
063 \* }</pre>  
064 \* <p>  
065 \* Internally, iteration starts from the root object. When next is called,  
066 \* the transformer is called to examine the object. The transformer will return  
067 \* either an iterator or an object. If the object is an Iterator, the next element  
068 \* from that iterator is obtained and the process repeats. If the element is an object  
069 \* it is returned.  
070 \* <p>  
071 \* Under many circumstances, linking Iterators together in this manner is  
072 \* more efficient (and convenient) than using nested for loops to extract a list.  
073 \*  
074 \* @since 3.1  
075 \*/  
076public class ObjectGraphIterator<E> implements Iterator<E> {  
077  
078 /\*\* The stack of iterators \*/  
079 private final Deque<Iterator<? extends E>> stack = new ArrayDeque<>(8);  
080 /\*\* The root object in the tree \*/  
081 private E root;  
082 /\*\* The transformer to use \*/  
083 private final Transformer<? super E, ? extends E> transformer;  
084  
085 /\*\* Whether there is another element in the iteration \*/  
086 private boolean hasNext = false;  
087 /\*\* The current iterator \*/  
088 private Iterator<? extends E> currentIterator;  
089 /\*\* The current value \*/  
090 private E currentValue;  
091 /\*\* The last used iterator, needed for remove() \*/  
092 private Iterator<? extends E> lastUsedIterator;  
093  
094 //-----------------------------------------------------------------------  
095 /\*\*  
096 \* Constructs an ObjectGraphIterator using a root object and transformer.  
097 \* <p>  
098 \* The root object can be an iterator, in which case it will be immediately  
099 \* looped around.  
100 \*  
101 \* @param root the root object, null will result in an empty iterator  
102 \* @param transformer the transformer to use, null will use a no effect transformer  
103 \*/  
104 @SuppressWarnings("unchecked")  
105 public ObjectGraphIterator(final E root, final Transformer<? super E, ? extends E> transformer) {  
106 super();  
107 if (root instanceof Iterator) {  
108 this.currentIterator = (Iterator<? extends E>) root;  
109 } else {  
110 this.root = root;  
111 }  
112 this.transformer = transformer;  
113 }  
114  
115 /\*\*  
116 \* Constructs a ObjectGraphIterator that will handle an iterator of iterators.  
117 \* <p>  
118 \* This constructor exists for convenience to emphasise that this class can  
119 \* be used to iterate over nested iterators. That is to say that the iterator  
120 \* passed in here contains other iterators, which may in turn contain further  
121 \* iterators.  
122 \*  
123 \* @param rootIterator the root iterator, null will result in an empty iterator  
124 \*/  
125 public ObjectGraphIterator(final Iterator<? extends E> rootIterator) {  
126 super();  
127 this.currentIterator = rootIterator;  
128 this.transformer = null;  
129 }  
130  
131 //-----------------------------------------------------------------------  
132 /\*\*  
133 \* Loops around the iterators to find the next value to return.  
134 \*/  
135 protected void updateCurrentIterator() {  
136 if (hasNext) {  
137 return;  
138 }  
139 if (currentIterator == null) {  
140 if (root == null) { // NOPMD  
141 // do nothing, hasNext will be false  
142 } else {  
143 if (transformer == null) {  
144 findNext(root);  
145 } else {  
146 findNext(transformer.transform(root));  
147 }  
148 root = null;  
149 }  
150 } else {  
151 findNextByIterator(currentIterator);  
152 }  
153 }  
154  
155 /\*\*  
156 \* Finds the next object in the iteration given any start object.  
157 \*  
158 \* @param value the value to start from  
159 \*/  
160 @SuppressWarnings("unchecked")  
161 protected void findNext(final E value) {  
162 if (value instanceof Iterator) {  
163 // need to examine this iterator  
164 findNextByIterator((Iterator<? extends E>) value);  
165 } else {  
166 // next value found  
167 currentValue = value;  
168 hasNext = true;  
169 }  
170 }  
171  
172 /\*\*  
173 \* Finds the next object in the iteration given an iterator.  
174 \*  
175 \* @param iterator the iterator to start from  
176 \*/  
177 protected void findNextByIterator(final Iterator<? extends E> iterator) {  
178 if (iterator != currentIterator) {  
179 // recurse a level  
180 if (currentIterator != null) {  
181 stack.push(currentIterator);  
182 }  
183 currentIterator = iterator;  
184 }  
185  
186 while (currentIterator.hasNext() && hasNext == false) {  
187 E next = currentIterator.next();  
188 if (transformer != null) {  
189 next = transformer.transform(next);  
190 }  
191 findNext(next);  
192 }  
193 // if we havn't found the next value and iterators are not yet exhausted  
194 if (!hasNext && !stack.isEmpty()) {  
195 // current iterator exhausted, go up a level  
196 currentIterator = stack.pop();  
197 findNextByIterator(currentIterator);  
198 }  
199 }  
200  
201 //-----------------------------------------------------------------------  
202 /\*\*  
203 \* Checks whether there are any more elements in the iteration to obtain.  
204 \*  
205 \* @return true if elements remain in the iteration  
206 \*/  
207 @Override  
208 public boolean hasNext() {  
209 updateCurrentIterator();  
210 return hasNext;  
211 }  
212  
213 /\*\*  
214 \* Gets the next element of the iteration.  
215 \*  
216 \* @return the next element from the iteration  
217 \* @throws NoSuchElementException if all the Iterators are exhausted  
218 \*/  
219 @Override  
220 public E next() {  
221 updateCurrentIterator();  
222 if (hasNext == false) {  
223 throw new NoSuchElementException("No more elements in the iteration");  
224 }  
225 lastUsedIterator = currentIterator;  
226 final E result = currentValue;  
227 currentValue = null;  
228 hasNext = false;  
229 return result;  
230 }  
231  
232 /\*\*  
233 \* Removes from the underlying collection the last element returned.  
234 \* <p>  
235 \* This method calls remove() on the underlying Iterator and it may  
236 \* throw an UnsupportedOperationException if the underlying Iterator  
237 \* does not support this method.  
238 \*  
239 \* @throws UnsupportedOperationException  
240 \* if the remove operator is not supported by the underlying Iterator  
241 \* @throws IllegalStateException  
242 \* if the next method has not yet been called, or the remove method has  
243 \* already been called after the last call to the next method.  
244 \*/  
245 @Override  
246 public void remove() {  
247 if (lastUsedIterator == null) {  
248 throw new IllegalStateException("Iterator remove() cannot be called at this time");  
249 }  
250 lastUsedIterator.remove();  
251 lastUsedIterator = null;  
252 }  
253  
254}