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.BitSet;  
021import java.util.Collection;  
022import java.util.Comparator;  
023import java.util.Iterator;  
024import java.util.List;  
025import java.util.NoSuchElementException;  
026  
027import org.apache.commons.collections4.list.UnmodifiableList;  
028  
029  
030/\*\*  
031 \* Provides an ordered iteration over the elements contained in a collection of  
032 \* ordered Iterators.  
033 \* <p>  
034 \* Given two ordered {@link Iterator} instances <code>A</code> and  
035 \* <code>B</code>, the {@link #next} method on this iterator will return the  
036 \* lesser of <code>A.next()</code> and <code>B.next()</code>.  
037 \*  
038 \* @since 2.1  
039 \*/  
040public class CollatingIterator<E> implements Iterator<E> {  
041  
042 /\*\* The {@link Comparator} used to evaluate order. \*/  
043 private Comparator<? super E> comparator = null;  
044  
045 /\*\* The list of {@link Iterator}s to evaluate. \*/  
046 private List<Iterator<? extends E>> iterators = null;  
047  
048 /\*\* {@link Iterator#next Next} objects peeked from each iterator. \*/  
049 private List<E> values = null;  
050  
051 /\*\* Whether or not each {@link #values} element has been set. \*/  
052 private BitSet valueSet = null;  
053  
054 /\*\*  
055 \* Index of the {@link #iterators iterator} from whom the last returned  
056 \* value was obtained.  
057 \*/  
058 private int lastReturned = -1;  
059  
060 // Constructors  
061 // ----------------------------------------------------------------------  
062 /\*\*  
063 \* Constructs a new <code>CollatingIterator</code>. A comparator must be  
064 \* set by calling {@link #setComparator(Comparator)} before invoking  
065 \* {@link #hasNext()}, or {@link #next()} for the first time. Child  
066 \* iterators will have to be manually added using the  
067 \* {@link #addIterator(Iterator)} method.  
068 \*/  
069 public CollatingIterator() {  
070 this(null, 2);  
071 }  
072  
073 /\*\*  
074 \* Constructs a new <code>CollatingIterator</code> that will used the  
075 \* specified comparator for ordering. Child iterators will have to be  
076 \* manually added using the {@link #addIterator(Iterator)} method.  
077 \*  
078 \* @param comp the comparator to use to sort; must not be null,  
079 \* unless you'll be invoking {@link #setComparator(Comparator)} later on.  
080 \*/  
081 public CollatingIterator(final Comparator<? super E> comp) {  
082 this(comp, 2);  
083 }  
084  
085 /\*\*  
086 \* Constructs a new <code>CollatingIterator</code> that will used the  
087 \* specified comparator for ordering and have the specified initial  
088 \* capacity. Child iterators will have to be manually added using the  
089 \* {@link #addIterator(Iterator)} method.  
090 \*  
091 \* @param comp the comparator to use to sort; must not be null,  
092 \* unless you'll be invoking {@link #setComparator(Comparator)} later on.  
093 \* @param initIterCapacity the initial capacity for the internal list of  
094 \* child iterators  
095 \*/  
096 public CollatingIterator(final Comparator<? super E> comp, final int initIterCapacity) {  
097 iterators = new ArrayList<>(initIterCapacity);  
098 setComparator(comp);  
099 }  
100  
101 /\*\*  
102 \* Constructs a new <code>CollatingIterator</code> that will use the  
103 \* specified comparator to provide ordered iteration over the two given  
104 \* iterators.  
105 \*  
106 \* @param comp the comparator to use to sort; must not be null,  
107 \* unless you'll be invoking {@link #setComparator(Comparator)} later on.  
108 \* @param a the first child ordered iterator  
109 \* @param b the second child ordered iterator  
110 \* @throws NullPointerException if either iterator is null  
111 \*/  
112 public CollatingIterator(final Comparator<? super E> comp, final Iterator<? extends E> a,  
113 final Iterator<? extends E> b) {  
114 this(comp, 2);  
115 addIterator(a);  
116 addIterator(b);  
117 }  
118  
119 /\*\*  
120 \* Constructs a new <code>CollatingIterator</code> that will use the  
121 \* specified comparator to provide ordered iteration over the array of  
122 \* iterators.  
123 \*  
124 \* @param comp the comparator to use to sort; must not be null,  
125 \* unless you'll be invoking {@link #setComparator(Comparator)} later on.  
126 \* @param iterators the array of iterators  
127 \* @throws NullPointerException if iterators array is or contains null  
128 \*/  
129 public CollatingIterator(final Comparator<? super E> comp, final Iterator<? extends E>[] iterators) {  
130 this(comp, iterators.length);  
131 for (final Iterator<? extends E> iterator : iterators) {  
132 addIterator(iterator);  
133 }  
134 }  
135  
136 /\*\*  
137 \* Constructs a new <code>CollatingIterator</code> that will use the  
138 \* specified comparator to provide ordered iteration over the collection of  
139 \* iterators.  
140 \*  
141 \* @param comp the comparator to use to sort; must not be null,  
142 \* unless you'll be invoking {@link #setComparator(Comparator)} later on.  
143 \* @param iterators the collection of iterators  
144 \* @throws NullPointerException if the iterators collection is or contains null  
145 \* @throws ClassCastException if the iterators collection contains an  
146 \* element that's not an {@link Iterator}  
147 \*/  
148 public CollatingIterator(final Comparator<? super E> comp, final Collection<Iterator<? extends E>> iterators) {  
149 this(comp, iterators.size());  
150 for (final Iterator<? extends E> iterator : iterators) {  
151 addIterator(iterator);  
152 }  
153 }  
154  
155 // Public Methods  
156 // ----------------------------------------------------------------------  
157 /\*\*  
158 \* Adds the given {@link Iterator} to the iterators being collated.  
159 \*  
160 \* @param iterator the iterator to add to the collation, must not be null  
161 \* @throws IllegalStateException if iteration has started  
162 \* @throws NullPointerException if the iterator is null  
163 \*/  
164 public void addIterator(final Iterator<? extends E> iterator) {  
165 checkNotStarted();  
166 if (iterator == null) {  
167 throw new NullPointerException("Iterator must not be null");  
168 }  
169 iterators.add(iterator);  
170 }  
171  
172 /\*\*  
173 \* Sets the iterator at the given index.  
174 \*  
175 \* @param index index of the Iterator to replace  
176 \* @param iterator Iterator to place at the given index  
177 \* @throws IndexOutOfBoundsException if index < 0 or index > size()  
178 \* @throws IllegalStateException if iteration has started  
179 \* @throws NullPointerException if the iterator is null  
180 \*/  
181 public void setIterator(final int index, final Iterator<? extends E> iterator) {  
182 checkNotStarted();  
183 if (iterator == null) {  
184 throw new NullPointerException("Iterator must not be null");  
185 }  
186 iterators.set(index, iterator);  
187 }  
188  
189 /\*\*  
190 \* Gets the list of Iterators (unmodifiable).  
191 \*  
192 \* @return the unmodifiable list of iterators added  
193 \*/  
194 public List<Iterator<? extends E>> getIterators() {  
195 return UnmodifiableList.unmodifiableList(iterators);  
196 }  
197  
198 /\*\*  
199 \* Gets the {@link Comparator} by which collatation occurs.  
200 \*  
201 \* @return the {@link Comparator}  
202 \*/  
203 public Comparator<? super E> getComparator() {  
204 return comparator;  
205 }  
206  
207 /\*\*  
208 \* Sets the {@link Comparator} by which collation occurs. If you  
209 \* would like to use the natural sort order (or, in other words,  
210 \* if the elements in the iterators are implementing the  
211 \* {@link java.lang.Comparable} interface), then use the  
212 \* {@link org.apache.commons.collections4.comparators.ComparableComparator}.  
213 \*  
214 \* @param comp the {@link Comparator} to set  
215 \* @throws IllegalStateException if iteration has started  
216 \*/  
217 public void setComparator(final Comparator<? super E> comp) {  
218 checkNotStarted();  
219 comparator = comp;  
220 }  
221  
222 // Iterator Methods  
223 // -------------------------------------------------------------------  
224 /\*\*  
225 \* Returns <code>true</code> if any child iterator has remaining elements.  
226 \*  
227 \* @return true if this iterator has remaining elements  
228 \*/  
229 @Override  
230 public boolean hasNext() {  
231 start();  
232 return anyValueSet(valueSet) || anyHasNext(iterators);  
233 }  
234  
235 /\*\*  
236 \* Returns the next ordered element from a child iterator.  
237 \*  
238 \* @return the next ordered element  
239 \* @throws NoSuchElementException if no child iterator has any more elements  
240 \*/  
241 @Override  
242 public E next() throws NoSuchElementException {  
243 if (hasNext() == false) {  
244 throw new NoSuchElementException();  
245 }  
246 final int leastIndex = least();  
247 if (leastIndex == -1) {  
248 throw new NoSuchElementException();  
249 }  
250 final E val = values.get(leastIndex);  
251 clear(leastIndex);  
252 lastReturned = leastIndex;  
253 return val;  
254 }  
255  
256 /\*\*  
257 \* Removes the last returned element from the child iterator that produced it.  
258 \*  
259 \* @throws IllegalStateException if there is no last returned element, or if  
260 \* the last returned element has already been removed  
261 \*/  
262 @Override  
263 public void remove() {  
264 if (lastReturned == -1) {  
265 throw new IllegalStateException("No value can be removed at present");  
266 }  
267 iterators.get(lastReturned).remove();  
268 }  
269  
270 /\*\*  
271 \* Returns the index of the iterator that returned the last element.  
272 \*  
273 \* @return the index of the iterator that returned the last element  
274 \* @throws IllegalStateException if there is no last returned element  
275 \*/  
276 public int getIteratorIndex() {  
277 if (lastReturned == -1) {  
278 throw new IllegalStateException("No value has been returned yet");  
279 }  
280  
281 return lastReturned;  
282 }  
283  
284 // Private Methods  
285 // -------------------------------------------------------------------  
286 /\*\*  
287 \* Initializes the collating state if it hasn't been already.  
288 \*/  
289 private void start() {  
290 if (values == null) {  
291 values = new ArrayList<>(iterators.size());  
292 valueSet = new BitSet(iterators.size());  
293 for (int i = 0; i < iterators.size(); i++) {  
294 values.add(null);  
295 valueSet.clear(i);  
296 }  
297 }  
298 }  
299  
300 /\*\*  
301 \* Sets the {@link #values} and {@link #valueSet} attributes at position  
302 \* <i>i</i> to the next value of the {@link #iterators iterator} at position  
303 \* <i>i</i>, or clear them if the <i>i</i><sup>th</sup> iterator has no next  
304 \* value.  
305 \*  
306 \* @return {@code false} iff there was no value to set  
307 \*/  
308 private boolean set(final int i) {  
309 final Iterator<? extends E> it = iterators.get(i);  
310 if (it.hasNext()) {  
311 values.set(i, it.next());  
312 valueSet.set(i);  
313 return true;  
314 }  
315 values.set(i, null);  
316 valueSet.clear(i);  
317 return false;  
318 }  
319  
320 /\*\*  
321 \* Clears the {@link #values} and {@link #valueSet} attributes at position  
322 \* <i>i</i>.  
323 \*/  
324 private void clear(final int i) {  
325 values.set(i, null);  
326 valueSet.clear(i);  
327 }  
328  
329 /\*\*  
330 \* Throws {@link IllegalStateException} if iteration has started via  
331 \* {@link #start}.  
332 \*  
333 \* @throws IllegalStateException if iteration started  
334 \*/  
335 private void checkNotStarted() throws IllegalStateException {  
336 if (values != null) {  
337 throw new IllegalStateException("Can't do that after next or hasNext has been called.");  
338 }  
339 }  
340  
341 /\*\*  
342 \* Returns the index of the least element in {@link #values},  
343 \* {@link #set(int) setting} any uninitialized values.  
344 \*  
345 \* @throws NullPointerException if no comparator is set  
346 \*/  
347 private int least() {  
348 int leastIndex = -1;  
349 E leastObject = null;  
350 for (int i = 0; i < values.size(); i++) {  
351 if (valueSet.get(i) == false) {  
352 set(i);  
353 }  
354 if (valueSet.get(i)) {  
355 if (leastIndex == -1) {  
356 leastIndex = i;  
357 leastObject = values.get(i);  
358 } else {  
359 final E curObject = values.get(i);  
360 if (comparator == null) {  
361 throw new NullPointerException("You must invoke setComparator() to set a comparator first.");  
362 }  
363 if (comparator.compare(curObject, leastObject) < 0) {  
364 leastObject = curObject;  
365 leastIndex = i;  
366 }  
367 }  
368 }  
369 }  
370 return leastIndex;  
371 }  
372  
373 /\*\*  
374 \* Returns <code>true</code> iff any bit in the given set is  
375 \* <code>true</code>.  
376 \*/  
377 private boolean anyValueSet(final BitSet set) {  
378 for (int i = 0; i < set.size(); i++) {  
379 if (set.get(i)) {  
380 return true;  
381 }  
382 }  
383 return false;  
384 }  
385  
386 /\*\*  
387 \* Returns <code>true</code> iff any {@link Iterator} in the given list has  
388 \* a next value.  
389 \*/  
390 private boolean anyHasNext(final List<Iterator<? extends E>> iters) {  
391 for (final Iterator<? extends E> iterator : iters) {  
392 if (iterator.hasNext()) {  
393 return true;  
394 }  
395 }  
396 return false;  
397 }  
398  
399}