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.list;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.lang.ref.WeakReference;  
024import java.util.ArrayList;  
025import java.util.Collection;  
026import java.util.ConcurrentModificationException;  
027import java.util.Iterator;  
028import java.util.List;  
029import java.util.ListIterator;  
030  
031/\*\*  
032 \* A <code>List</code> implementation with a <code>ListIterator</code> that  
033 \* allows concurrent modifications to the underlying list.  
034 \* <p>  
035 \* This implementation supports all of the optional {@link List} operations.  
036 \* It extends <code>AbstractLinkedList</code> and thus provides the  
037 \* stack/queue/dequeue operations available in {@link java.util.LinkedList}.  
038 \* </p>  
039 \* <p>  
040 \* The main feature of this class is the ability to modify the list and the  
041 \* iterator at the same time. Both the {@link #listIterator()} and {@link #cursor()}  
042 \* methods provides access to a <code>Cursor</code> instance which extends  
043 \* <code>ListIterator</code>. The cursor allows changes to the list concurrent  
044 \* with changes to the iterator. Note that the {@link #iterator()} method and  
045 \* sublists do <b>not</b> provide this cursor behaviour.  
046 \* </p>  
047 \* <p>  
048 \* The <code>Cursor</code> class is provided partly for backwards compatibility  
049 \* and partly because it allows the cursor to be directly closed. Closing the  
050 \* cursor is optional because references are held via a <code>WeakReference</code>.  
051 \* For most purposes, simply modify the iterator and list at will, and then let  
052 \* the garbage collector to the rest.  
053 \* </p>  
054 \* <p>  
055 \* <b>Note that this implementation is not synchronized.</b>  
056 \* </p>  
057 \*  
058 \* @see java.util.LinkedList  
059 \* @since 1.0  
060 \*/  
061public class CursorableLinkedList<E> extends AbstractLinkedList<E> implements Serializable {  
062  
063 /\*\* Ensure serialization compatibility \*/  
064 private static final long serialVersionUID = 8836393098519411393L;  
065  
066 /\*\* A list of the cursor currently open on this list \*/  
067 private transient List<WeakReference<Cursor<E>>> cursors;  
068  
069 //-----------------------------------------------------------------------  
070 /\*\*  
071 \* Constructor that creates.  
072 \*/  
073 public CursorableLinkedList() {  
074 super();  
075 init(); // must call init() as use super();  
076 }  
077  
078 /\*\*  
079 \* Constructor that copies the specified collection  
080 \*  
081 \* @param coll the collection to copy  
082 \*/  
083 public CursorableLinkedList(final Collection<? extends E> coll) {  
084 super(coll);  
085 }  
086  
087 /\*\*  
088 \* The equivalent of a default constructor called  
089 \* by any constructor and by <code>readObject</code>.  
090 \*/  
091 @Override  
092 protected void init() {  
093 super.init();  
094 cursors = new ArrayList<>();  
095 }  
096  
097 //-----------------------------------------------------------------------  
098 /\*\*  
099 \* Returns an iterator that does <b>not</b> support concurrent modification.  
100 \* <p>  
101 \* If the underlying list is modified while iterating using this iterator  
102 \* a ConcurrentModificationException will occur.  
103 \* The cursor behaviour is available via {@link #listIterator()}.  
104 \*  
105 \* @return a new iterator that does <b>not</b> support concurrent modification  
106 \*/  
107 @Override  
108 public Iterator<E> iterator() {  
109 return super.listIterator(0);  
110 }  
111  
112 /\*\*  
113 \* Returns a cursor iterator that allows changes to the underlying list in parallel.  
114 \* <p>  
115 \* The cursor enables iteration and list changes to occur in any order without  
116 \* invalidating the iterator (from one thread). When elements are added to the  
117 \* list, an event is fired to all active cursors enabling them to adjust to the  
118 \* change in the list.  
119 \* <p>  
120 \* When the "current" (i.e., last returned by {@link ListIterator#next}  
121 \* or {@link ListIterator#previous}) element of the list is removed,  
122 \* the cursor automatically adjusts to the change (invalidating the  
123 \* last returned value such that it cannot be removed).  
124 \*  
125 \* @return a new cursor iterator  
126 \*/  
127 @Override  
128 public ListIterator<E> listIterator() {  
129 return cursor(0);  
130 }  
131  
132 /\*\*  
133 \* Returns a cursor iterator that allows changes to the underlying list in parallel.  
134 \* <p>  
135 \* The cursor enables iteration and list changes to occur in any order without  
136 \* invalidating the iterator (from one thread). When elements are added to the  
137 \* list, an event is fired to all active cursors enabling them to adjust to the  
138 \* change in the list.  
139 \* <p>  
140 \* When the "current" (i.e., last returned by {@link ListIterator#next}  
141 \* or {@link ListIterator#previous}) element of the list is removed,  
142 \* the cursor automatically adjusts to the change (invalidating the  
143 \* last returned value such that it cannot be removed).  
144 \*  
145 \* @param fromIndex the index to start from  
146 \* @return a new cursor iterator  
147 \*/  
148 @Override  
149 public ListIterator<E> listIterator(final int fromIndex) {  
150 return cursor(fromIndex);  
151 }  
152  
153 /\*\*  
154 \* Returns a {@link Cursor} for iterating through the elements of this list.  
155 \* <p>  
156 \* A <code>Cursor</code> is a <code>ListIterator</code> with an additional  
157 \* <code>close()</code> method. Calling this method immediately discards the  
158 \* references to the cursor. If it is not called, then the garbage collector  
159 \* will still remove the reference as it is held via a <code>WeakReference</code>.  
160 \* <p>  
161 \* The cursor enables iteration and list changes to occur in any order without  
162 \* invalidating the iterator (from one thread). When elements are added to the  
163 \* list, an event is fired to all active cursors enabling them to adjust to the  
164 \* change in the list.  
165 \* <p>  
166 \* When the "current" (i.e., last returned by {@link ListIterator#next}  
167 \* or {@link ListIterator#previous}) element of the list is removed,  
168 \* the cursor automatically adjusts to the change (invalidating the  
169 \* last returned value such that it cannot be removed).  
170 \* <p>  
171 \* The {@link #listIterator()} method returns the same as this method, and can  
172 \* be cast to a <code>Cursor</code> if the <code>close</code> method is required.  
173 \*  
174 \* @return a new cursor iterator  
175 \*/  
176 public CursorableLinkedList.Cursor<E> cursor() {  
177 return cursor(0);  
178 }  
179  
180 /\*\*  
181 \* Returns a {@link Cursor} for iterating through the elements of this list  
182 \* starting from a specified index.  
183 \* <p>  
184 \* A <code>Cursor</code> is a <code>ListIterator</code> with an additional  
185 \* <code>close()</code> method. Calling this method immediately discards the  
186 \* references to the cursor. If it is not called, then the garbage collector  
187 \* will still remove the reference as it is held via a <code>WeakReference</code>.  
188 \* <p>  
189 \* The cursor enables iteration and list changes to occur in any order without  
190 \* invalidating the iterator (from one thread). When elements are added to the  
191 \* list, an event is fired to all active cursors enabling them to adjust to the  
192 \* change in the list.  
193 \* <p>  
194 \* When the "current" (i.e., last returned by {@link ListIterator#next}  
195 \* or {@link ListIterator#previous}) element of the list is removed,  
196 \* the cursor automatically adjusts to the change (invalidating the  
197 \* last returned value such that it cannot be removed).  
198 \* <p>  
199 \* The {@link #listIterator(int)} method returns the same as this method, and can  
200 \* be cast to a <code>Cursor</code> if the <code>close</code> method is required.  
201 \*  
202 \* @param fromIndex the index to start from  
203 \* @return a new cursor iterator  
204 \* @throws IndexOutOfBoundsException if the index is out of range  
205 \* (index < 0 || index > size()).  
206 \*/  
207 public CursorableLinkedList.Cursor<E> cursor(final int fromIndex) {  
208 final Cursor<E> cursor = new Cursor<>(this, fromIndex);  
209 registerCursor(cursor);  
210 return cursor;  
211 }  
212  
213 //-----------------------------------------------------------------------  
214 /\*\*  
215 \* Updates the node with a new value.  
216 \* This implementation sets the value on the node.  
217 \* Subclasses can override this to record the change.  
218 \*  
219 \* @param node node to update  
220 \* @param value new value of the node  
221 \*/  
222 @Override  
223 protected void updateNode(final Node<E> node, final E value) {  
224 super.updateNode(node, value);  
225 broadcastNodeChanged(node);  
226 }  
227  
228 /\*\*  
229 \* Inserts a new node into the list.  
230 \*  
231 \* @param nodeToInsert new node to insert  
232 \* @param insertBeforeNode node to insert before  
233 \* @throws NullPointerException if either node is null  
234 \*/  
235 @Override  
236 protected void addNode(final Node<E> nodeToInsert, final Node<E> insertBeforeNode) {  
237 super.addNode(nodeToInsert, insertBeforeNode);  
238 broadcastNodeInserted(nodeToInsert);  
239 }  
240  
241 /\*\*  
242 \* Removes the specified node from the list.  
243 \*  
244 \* @param node the node to remove  
245 \* @throws NullPointerException if <code>node</code> is null  
246 \*/  
247 @Override  
248 protected void removeNode(final Node<E> node) {  
249 super.removeNode(node);  
250 broadcastNodeRemoved(node);  
251 }  
252  
253 /\*\*  
254 \* Removes all nodes by iteration.  
255 \*/  
256 @Override  
257 protected void removeAllNodes() {  
258 if (size() > 0) {  
259 // superclass implementation would break all the iterators  
260 final Iterator<E> it = iterator();  
261 while (it.hasNext()) {  
262 it.next();  
263 it.remove();  
264 }  
265 }  
266 }  
267  
268 //-----------------------------------------------------------------------  
269 /\*\*  
270 \* Registers a cursor to be notified of changes to this list.  
271 \*  
272 \* @param cursor the cursor to register  
273 \*/  
274 protected void registerCursor(final Cursor<E> cursor) {  
275 // We take this opportunity to clean the cursors list  
276 // of WeakReference objects to garbage-collected cursors.  
277 for (final Iterator<WeakReference<Cursor<E>>> it = cursors.iterator(); it.hasNext();) {  
278 final WeakReference<Cursor<E>> ref = it.next();  
279 if (ref.get() == null) {  
280 it.remove();  
281 }  
282 }  
283 cursors.add(new WeakReference<>(cursor));  
284 }  
285  
286 /\*\*  
287 \* Deregisters a cursor from the list to be notified of changes.  
288 \*  
289 \* @param cursor the cursor to deregister  
290 \*/  
291 protected void unregisterCursor(final Cursor<E> cursor) {  
292 for (final Iterator<WeakReference<Cursor<E>>> it = cursors.iterator(); it.hasNext();) {  
293 final WeakReference<Cursor<E>> ref = it.next();  
294 final Cursor<E> cur = ref.get();  
295 if (cur == null) {  
296 // some other unrelated cursor object has been  
297 // garbage-collected; let's take the opportunity to  
298 // clean up the cursors list anyway..  
299 it.remove();  
300 } else if (cur == cursor) {  
301 ref.clear();  
302 it.remove();  
303 break;  
304 }  
305 }  
306 }  
307  
308 //-----------------------------------------------------------------------  
309 /\*\*  
310 \* Informs all of my registered cursors that the specified  
311 \* element was changed.  
312 \*  
313 \* @param node the node that was changed  
314 \*/  
315 protected void broadcastNodeChanged(final Node<E> node) {  
316 final Iterator<WeakReference<Cursor<E>>> it = cursors.iterator();  
317 while (it.hasNext()) {  
318 final WeakReference<Cursor<E>> ref = it.next();  
319 final Cursor<E> cursor = ref.get();  
320 if (cursor == null) {  
321 it.remove(); // clean up list  
322 } else {  
323 cursor.nodeChanged(node);  
324 }  
325 }  
326 }  
327  
328 /\*\*  
329 \* Informs all of my registered cursors that the specified  
330 \* element was just removed from my list.  
331 \*  
332 \* @param node the node that was changed  
333 \*/  
334 protected void broadcastNodeRemoved(final Node<E> node) {  
335 final Iterator<WeakReference<Cursor<E>>> it = cursors.iterator();  
336 while (it.hasNext()) {  
337 final WeakReference<Cursor<E>> ref = it.next();  
338 final Cursor<E> cursor = ref.get();  
339 if (cursor == null) {  
340 it.remove(); // clean up list  
341 } else {  
342 cursor.nodeRemoved(node);  
343 }  
344 }  
345 }  
346  
347 /\*\*  
348 \* Informs all of my registered cursors that the specified  
349 \* element was just added to my list.  
350 \*  
351 \* @param node the node that was changed  
352 \*/  
353 protected void broadcastNodeInserted(final Node<E> node) {  
354 final Iterator<WeakReference<Cursor<E>>> it = cursors.iterator();  
355 while (it.hasNext()) {  
356 final WeakReference<Cursor<E>> ref = it.next();  
357 final Cursor<E> cursor = ref.get();  
358 if (cursor == null) {  
359 it.remove(); // clean up list  
360 } else {  
361 cursor.nodeInserted(node);  
362 }  
363 }  
364 }  
365  
366 //-----------------------------------------------------------------------  
367 /\*\*  
368 \* Serializes the data held in this object to the stream specified.  
369 \*  
370 \* @param out the output stream  
371 \* @throws IOException if an error occurs while writing to the stream  
372 \*/  
373 private void writeObject(final ObjectOutputStream out) throws IOException {  
374 out.defaultWriteObject();  
375 doWriteObject(out);  
376 }  
377  
378 /\*\*  
379 \* Deserializes the data held in this object to the stream specified.  
380 \*  
381 \* @param in the input stream  
382 \* @throws IOException if an error occurs while reading from the stream  
383 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
384 \*/  
385 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
386 in.defaultReadObject();  
387 doReadObject(in);  
388 }  
389  
390 //-----------------------------------------------------------------------  
391 /\*\*  
392 \* Creates a list iterator for the sublist.  
393 \*  
394 \* @param subList the sublist to get an iterator for  
395 \* @param fromIndex the index to start from, relative to the sublist  
396 \* @return the list iterator for the sublist  
397 \*/  
398 @Override  
399 protected ListIterator<E> createSubListListIterator(final LinkedSubList<E> subList, final int fromIndex) {  
400 final SubCursor<E> cursor = new SubCursor<>(subList, fromIndex);  
401 registerCursor(cursor);  
402 return cursor;  
403 }  
404  
405 //-----------------------------------------------------------------------  
406 /\*\*  
407 \* An extended <code>ListIterator</code> that allows concurrent changes to  
408 \* the underlying list.  
409 \*/  
410 public static class Cursor<E> extends AbstractLinkedList.LinkedListIterator<E> {  
411 /\*\* Is the cursor valid (not closed) \*/  
412 boolean valid = true;  
413 /\*\* Is the next index valid \*/  
414 boolean nextIndexValid = true;  
415 /\*\* Flag to indicate if the current element was removed by another object. \*/  
416 boolean currentRemovedByAnother = false;  
417  
418 /\*\*  
419 \* Constructs a new cursor.  
420 \*  
421 \* @param parent the parent list  
422 \* @param index the index to start from  
423 \*/  
424 protected Cursor(final CursorableLinkedList<E> parent, final int index) {  
425 super(parent, index);  
426 valid = true;  
427 }  
428  
429 /\*\*  
430 \* Removes the item last returned by this iterator.  
431 \* <p>  
432 \* There may have been subsequent alterations to the list  
433 \* since you obtained this item, however you can still remove it.  
434 \* You can even remove it if the item is no longer in the main list.  
435 \* However, you can't call this method on the same iterator more  
436 \* than once without calling next() or previous().  
437 \*  
438 \* @throws IllegalStateException if there is no item to remove  
439 \*/  
440 @Override  
441 public void remove() {  
442 // overridden, as the nodeRemoved() method updates the iterator  
443 // state in the parent.removeNode() call below  
444 if (current == null && currentRemovedByAnother) { // NOPMD  
445 // quietly ignore, as the last returned node was removed  
446 // by the list or some other iterator  
447 // by ignoring it, we keep this iterator independent from  
448 // other changes as much as possible  
449 } else {  
450 checkModCount();  
451 parent.removeNode(getLastNodeReturned());  
452 }  
453 currentRemovedByAnother = false;  
454 }  
455  
456 /\*\*  
457 \* Adds an object to the list.  
458 \* The object added here will be the new 'previous' in the iterator.  
459 \*  
460 \* @param obj the object to add  
461 \*/  
462 @Override  
463 public void add(final E obj) {  
464 // overridden, as the nodeInserted() method updates the iterator state  
465 super.add(obj);  
466 // matches the (next.previous == node) clause in nodeInserted()  
467 // thus next gets changed - reset it again here  
468 next = next.next;  
469 }  
470  
471 // set is not overridden, as it works ok  
472 // note that we want it to throw an exception if the element being  
473 // set has been removed from the real list (compare this with the  
474 // remove method where we silently ignore this case)  
475  
476 /\*\*  
477 \* Gets the index of the next element to be returned.  
478 \*  
479 \* @return the next index  
480 \*/  
481 @Override  
482 public int nextIndex() {  
483 if (nextIndexValid == false) {  
484 if (next == parent.header) {  
485 nextIndex = parent.size();  
486 } else {  
487 int pos = 0;  
488 Node<E> temp = parent.header.next;  
489 while (temp != next) {  
490 pos++;  
491 temp = temp.next;  
492 }  
493 nextIndex = pos;  
494 }  
495 nextIndexValid = true;  
496 }  
497 return nextIndex;  
498 }  
499  
500 /\*\*  
501 \* Handle event from the list when a node has changed.  
502 \*  
503 \* @param node the node that changed  
504 \*/  
505 protected void nodeChanged(final Node<E> node) {  
506 // do nothing  
507 }  
508  
509 /\*\*  
510 \* Handle event from the list when a node has been removed.  
511 \*  
512 \* @param node the node that was removed  
513 \*/  
514 protected void nodeRemoved(final Node<E> node) {  
515 if (node == next && node == current) {  
516 // state where next() followed by previous()  
517 next = node.next;  
518 current = null;  
519 currentRemovedByAnother = true;  
520 } else if (node == next) {  
521 // state where next() not followed by previous()  
522 // and we are matching next node  
523 next = node.next;  
524 currentRemovedByAnother = false;  
525 } else if (node == current) {  
526 // state where next() not followed by previous()  
527 // and we are matching current (last returned) node  
528 current = null;  
529 currentRemovedByAnother = true;  
530 nextIndex--;  
531 } else {  
532 nextIndexValid = false;  
533 currentRemovedByAnother = false;  
534 }  
535 }  
536  
537 /\*\*  
538 \* Handle event from the list when a node has been added.  
539 \*  
540 \* @param node the node that was added  
541 \*/  
542 protected void nodeInserted(final Node<E> node) {  
543 if (node.previous == current) {  
544 next = node;  
545 } else if (next.previous == node) {  
546 next = node;  
547 } else {  
548 nextIndexValid = false;  
549 }  
550 }  
551  
552 /\*\*  
553 \* Override superclass modCount check, and replace it with our valid flag.  
554 \*/  
555 @Override  
556 protected void checkModCount() {  
557 if (!valid) {  
558 throw new ConcurrentModificationException("Cursor closed");  
559 }  
560 }  
561  
562 /\*\*  
563 \* Mark this cursor as no longer being needed. Any resources  
564 \* associated with this cursor are immediately released.  
565 \* In previous versions of this class, it was mandatory to close  
566 \* all cursor objects to avoid memory leaks. It is <i>no longer</i>  
567 \* necessary to call this close method; an instance of this class  
568 \* can now be treated exactly like a normal iterator.  
569 \*/  
570 public void close() {  
571 if (valid) {  
572 ((CursorableLinkedList<E>) parent).unregisterCursor(this);  
573 valid = false;  
574 }  
575 }  
576 }  
577  
578 //-----------------------------------------------------------------------  
579 /\*\*  
580 \* A cursor for the sublist based on LinkedSubListIterator.  
581 \*  
582 \* @since 3.2  
583 \*/  
584 protected static class SubCursor<E> extends Cursor<E> {  
585  
586 /\*\* The parent list \*/  
587 protected final LinkedSubList<E> sub;  
588  
589 /\*\*  
590 \* Constructs a new cursor.  
591 \*  
592 \* @param sub the sub list  
593 \* @param index the index to start from  
594 \*/  
595 protected SubCursor(final LinkedSubList<E> sub, final int index) {  
596 super((CursorableLinkedList<E>) sub.parent, index + sub.offset);  
597 this.sub = sub;  
598 }  
599  
600 @Override  
601 public boolean hasNext() {  
602 return nextIndex() < sub.size;  
603 }  
604  
605 @Override  
606 public boolean hasPrevious() {  
607 return previousIndex() >= 0;  
608 }  
609  
610 @Override  
611 public int nextIndex() {  
612 return super.nextIndex() - sub.offset;  
613 }  
614  
615 @Override  
616 public void add(final E obj) {  
617 super.add(obj);  
618 sub.expectedModCount = parent.modCount;  
619 sub.size++;  
620 }  
621  
622 @Override  
623 public void remove() {  
624 super.remove();  
625 sub.expectedModCount = parent.modCount;  
626 sub.size--;  
627 }  
628 }  
629  
630}