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.lang.reflect.Array;  
023import java.util.AbstractList;  
024import java.util.Collection;  
025import java.util.ConcurrentModificationException;  
026import java.util.Iterator;  
027import java.util.List;  
028import java.util.ListIterator;  
029import java.util.NoSuchElementException;  
030  
031import org.apache.commons.collections4.OrderedIterator;  
032  
033/\*\*  
034 \* An abstract implementation of a linked list which provides numerous points for  
035 \* subclasses to override.  
036 \* <p>  
037 \* Overridable methods are provided to change the storage node and to change how  
038 \* nodes are added to and removed. Hopefully, all you need for unusual subclasses  
039 \* is here.  
040 \* </p>  
041 \*  
042 \* @since 3.0  
043 \*/  
044public abstract class AbstractLinkedList<E> implements List<E> {  
045  
046 /\*  
047 \* Implementation notes:  
048 \* - a standard circular doubly-linked list  
049 \* - a marker node is stored to mark the start and the end of the list  
050 \* - node creation and removal always occurs through createNode() and  
051 \* removeNode().  
052 \* - a modification count is kept, with the same semantics as  
053 \* {@link java.util.LinkedList}.  
054 \* - respects {@link AbstractList#modCount}  
055 \*/  
056  
057 /\*\*  
058 \* A {@link Node} which indicates the start and end of the list and does not  
059 \* hold a value. The value of <code>next</code> is the first item in the  
060 \* list. The value of of <code>previous</code> is the last item in the list.  
061 \*/  
062 transient Node<E> header;  
063  
064 /\*\* The size of the list \*/  
065 transient int size;  
066  
067 /\*\* Modification count for iterators \*/  
068 transient int modCount;  
069  
070 /\*\*  
071 \* Constructor that does nothing intended for deserialization.  
072 \* <p>  
073 \* If this constructor is used by a serializable subclass then the init()  
074 \* method must be called.  
075 \*/  
076 protected AbstractLinkedList() {  
077 super();  
078 }  
079  
080 /\*\*  
081 \* Constructs a list copying data from the specified collection.  
082 \*  
083 \* @param coll the collection to copy  
084 \*/  
085 protected AbstractLinkedList(final Collection<? extends E> coll) {  
086 super();  
087 init();  
088 addAll(coll);  
089 }  
090  
091 /\*\*  
092 \* The equivalent of a default constructor, broken out so it can be called  
093 \* by any constructor and by <code>readObject</code>.  
094 \* Subclasses which override this method should make sure they call super,  
095 \* so the list is initialised properly.  
096 \*/  
097 protected void init() {  
098 header = createHeaderNode();  
099 }  
100  
101 //-----------------------------------------------------------------------  
102  
103 @Override  
104 public int size() {  
105 return size;  
106 }  
107  
108 @Override  
109 public boolean isEmpty() {  
110 return size() == 0;  
111 }  
112  
113 @Override  
114 public E get(final int index) {  
115 final Node<E> node = getNode(index, false);  
116 return node.getValue();  
117 }  
118  
119 //-----------------------------------------------------------------------  
120  
121 @Override  
122 public Iterator<E> iterator() {  
123 return listIterator();  
124 }  
125  
126 @Override  
127 public ListIterator<E> listIterator() {  
128 return new LinkedListIterator<>(this, 0);  
129 }  
130  
131 @Override  
132 public ListIterator<E> listIterator(final int fromIndex) {  
133 return new LinkedListIterator<>(this, fromIndex);  
134 }  
135  
136 //-----------------------------------------------------------------------  
137  
138 @Override  
139 public int indexOf(final Object value) {  
140 int i = 0;  
141 for (Node<E> node = header.next; node != header; node = node.next) {  
142 if (isEqualValue(node.getValue(), value)) {  
143 return i;  
144 }  
145 i++;  
146 }  
147 return -1;  
148 }  
149  
150 @Override  
151 public int lastIndexOf(final Object value) {  
152 int i = size - 1;  
153 for (Node<E> node = header.previous; node != header; node = node.previous) {  
154 if (isEqualValue(node.getValue(), value)) {  
155 return i;  
156 }  
157 i--;  
158 }  
159 return -1;  
160 }  
161  
162 @Override  
163 public boolean contains(final Object value) {  
164 return indexOf(value) != -1;  
165 }  
166  
167 @Override  
168 public boolean containsAll(final Collection<?> coll) {  
169 for (final Object o : coll) {  
170 if (!contains(o)) {  
171 return false;  
172 }  
173 }  
174 return true;  
175 }  
176  
177 //-----------------------------------------------------------------------  
178  
179 @Override  
180 public Object[] toArray() {  
181 return toArray(new Object[size]);  
182 }  
183  
184 @Override  
185 @SuppressWarnings("unchecked")  
186 public <T> T[] toArray(T[] array) {  
187 // Extend the array if needed  
188 if (array.length < size) {  
189 final Class<?> componentType = array.getClass().getComponentType();  
190 array = (T[]) Array.newInstance(componentType, size);  
191 }  
192 // Copy the values into the array  
193 int i = 0;  
194 for (Node<E> node = header.next; node != header; node = node.next, i++) {  
195 array[i] = (T) node.getValue();  
196 }  
197 // Set the value after the last value to null  
198 if (array.length > size) {  
199 array[size] = null;  
200 }  
201 return array;  
202 }  
203  
204 /\*\*  
205 \* Gets a sublist of the main list.  
206 \*  
207 \* @param fromIndexInclusive the index to start from  
208 \* @param toIndexExclusive the index to end at  
209 \* @return the new sublist  
210 \*/  
211 @Override  
212 public List<E> subList(final int fromIndexInclusive, final int toIndexExclusive) {  
213 return new LinkedSubList<>(this, fromIndexInclusive, toIndexExclusive);  
214 }  
215  
216 //-----------------------------------------------------------------------  
217  
218 @Override  
219 public boolean add(final E value) {  
220 addLast(value);  
221 return true;  
222 }  
223  
224 @Override  
225 public void add(final int index, final E value) {  
226 final Node<E> node = getNode(index, true);  
227 addNodeBefore(node, value);  
228 }  
229  
230 @Override  
231 public boolean addAll(final Collection<? extends E> coll) {  
232 return addAll(size, coll);  
233 }  
234  
235 @Override  
236 public boolean addAll(final int index, final Collection<? extends E> coll) {  
237 final Node<E> node = getNode(index, true);  
238 for (final E e : coll) {  
239 addNodeBefore(node, e);  
240 }  
241 return true;  
242 }  
243  
244 //-----------------------------------------------------------------------  
245  
246 @Override  
247 public E remove(final int index) {  
248 final Node<E> node = getNode(index, false);  
249 final E oldValue = node.getValue();  
250 removeNode(node);  
251 return oldValue;  
252 }  
253  
254 @Override  
255 public boolean remove(final Object value) {  
256 for (Node<E> node = header.next; node != header; node = node.next) {  
257 if (isEqualValue(node.getValue(), value)) {  
258 removeNode(node);  
259 return true;  
260 }  
261 }  
262 return false;  
263 }  
264  
265 /\*\*  
266 \* {@inheritDoc}  
267 \* <p>  
268 \* This implementation iterates over the elements of this list, checking each element in  
269 \* turn to see if it's contained in <code>coll</code>. If it's contained, it's removed  
270 \* from this list. As a consequence, it is advised to use a collection type for  
271 \* <code>coll</code> that provides a fast (e.g. O(1)) implementation of  
272 \* {@link Collection#contains(Object)}.  
273 \*/  
274 @Override  
275 public boolean removeAll(final Collection<?> coll) {  
276 boolean modified = false;  
277 final Iterator<E> it = iterator();  
278 while (it.hasNext()) {  
279 if (coll.contains(it.next())) {  
280 it.remove();  
281 modified = true;  
282 }  
283 }  
284 return modified;  
285 }  
286  
287 //-----------------------------------------------------------------------  
288  
289 /\*\*  
290 \* {@inheritDoc}  
291 \* <p>  
292 \* This implementation iterates over the elements of this list, checking each element in  
293 \* turn to see if it's contained in <code>coll</code>. If it's not contained, it's removed  
294 \* from this list. As a consequence, it is advised to use a collection type for  
295 \* <code>coll</code> that provides a fast (e.g. O(1)) implementation of  
296 \* {@link Collection#contains(Object)}.  
297 \*/  
298 @Override  
299 public boolean retainAll(final Collection<?> coll) {  
300 boolean modified = false;  
301 final Iterator<E> it = iterator();  
302 while (it.hasNext()) {  
303 if (coll.contains(it.next()) == false) {  
304 it.remove();  
305 modified = true;  
306 }  
307 }  
308 return modified;  
309 }  
310  
311 @Override  
312 public E set(final int index, final E value) {  
313 final Node<E> node = getNode(index, false);  
314 final E oldValue = node.getValue();  
315 updateNode(node, value);  
316 return oldValue;  
317 }  
318  
319 @Override  
320 public void clear() {  
321 removeAllNodes();  
322 }  
323  
324 //-----------------------------------------------------------------------  
325  
326 public E getFirst() {  
327 final Node<E> node = header.next;  
328 if (node == header) {  
329 throw new NoSuchElementException();  
330 }  
331 return node.getValue();  
332 }  
333  
334 public E getLast() {  
335 final Node<E> node = header.previous;  
336 if (node == header) {  
337 throw new NoSuchElementException();  
338 }  
339 return node.getValue();  
340 }  
341  
342 public boolean addFirst(final E o) {  
343 addNodeAfter(header, o);  
344 return true;  
345 }  
346  
347 public boolean addLast(final E o) {  
348 addNodeBefore(header, o);  
349 return true;  
350 }  
351  
352 public E removeFirst() {  
353 final Node<E> node = header.next;  
354 if (node == header) {  
355 throw new NoSuchElementException();  
356 }  
357 final E oldValue = node.getValue();  
358 removeNode(node);  
359 return oldValue;  
360 }  
361  
362 public E removeLast() {  
363 final Node<E> node = header.previous;  
364 if (node == header) {  
365 throw new NoSuchElementException();  
366 }  
367 final E oldValue = node.getValue();  
368 removeNode(node);  
369 return oldValue;  
370 }  
371  
372 //-----------------------------------------------------------------------  
373 @Override  
374 public boolean equals(final Object obj) {  
375 if (obj == this) {  
376 return true;  
377 }  
378 if (obj instanceof List == false) {  
379 return false;  
380 }  
381 final List<?> other = (List<?>) obj;  
382 if (other.size() != size()) {  
383 return false;  
384 }  
385 final ListIterator<?> it1 = listIterator();  
386 final ListIterator<?> it2 = other.listIterator();  
387 while (it1.hasNext() && it2.hasNext()) {  
388 final Object o1 = it1.next();  
389 final Object o2 = it2.next();  
390 if (!(o1 == null ? o2 == null : o1.equals(o2))) {  
391 return false;  
392 }  
393 }  
394 return !(it1.hasNext() || it2.hasNext());  
395 }  
396  
397 @Override  
398 public int hashCode() {  
399 int hashCode = 1;  
400 for (final E e : this) {  
401 hashCode = 31 \* hashCode + (e == null ? 0 : e.hashCode());  
402 }  
403 return hashCode;  
404 }  
405  
406 @Override  
407 public String toString() {  
408 if (size() == 0) {  
409 return "[]";  
410 }  
411 final StringBuilder buf = new StringBuilder(16 \* size());  
412 buf.append('[');  
413  
414 final Iterator<E> it = iterator();  
415 boolean hasNext = it.hasNext();  
416 while (hasNext) {  
417 final Object value = it.next();  
418 buf.append(value == this ? "(this Collection)" : value);  
419 hasNext = it.hasNext();  
420 if (hasNext) {  
421 buf.append(", ");  
422 }  
423 }  
424 buf.append(']');  
425 return buf.toString();  
426 }  
427  
428 //-----------------------------------------------------------------------  
429 /\*\*  
430 \* Compares two values for equals.  
431 \* This implementation uses the equals method.  
432 \* Subclasses can override this to match differently.  
433 \*  
434 \* @param value1 the first value to compare, may be null  
435 \* @param value2 the second value to compare, may be null  
436 \* @return true if equal  
437 \*/  
438 protected boolean isEqualValue(final Object value1, final Object value2) {  
439 return value1 == value2 || (value1 != null && value1.equals(value2));  
440 }  
441  
442 /\*\*  
443 \* Updates the node with a new value.  
444 \* This implementation sets the value on the node.  
445 \* Subclasses can override this to record the change.  
446 \*  
447 \* @param node node to update  
448 \* @param value new value of the node  
449 \*/  
450 protected void updateNode(final Node<E> node, final E value) {  
451 node.setValue(value);  
452 }  
453  
454 /\*\*  
455 \* Creates a new node with previous, next and element all set to null.  
456 \* This implementation creates a new empty Node.  
457 \* Subclasses can override this to create a different class.  
458 \*  
459 \* @return newly created node  
460 \*/  
461 protected Node<E> createHeaderNode() {  
462 return new Node<>();  
463 }  
464  
465 /\*\*  
466 \* Creates a new node with the specified properties.  
467 \* This implementation creates a new Node with data.  
468 \* Subclasses can override this to create a different class.  
469 \*  
470 \* @param value value of the new node  
471 \* @return a new node containing the value  
472 \*/  
473 protected Node<E> createNode(final E value) {  
474 return new Node<>(value);  
475 }  
476  
477 /\*\*  
478 \* Creates a new node with the specified object as its  
479 \* <code>value</code> and inserts it before <code>node</code>.  
480 \* <p>  
481 \* This implementation uses {@link #createNode(Object)} and  
482 \* {@link #addNode(AbstractLinkedList.Node,AbstractLinkedList.Node)}.  
483 \*  
484 \* @param node node to insert before  
485 \* @param value value of the newly added node  
486 \* @throws NullPointerException if <code>node</code> is null  
487 \*/  
488 protected void addNodeBefore(final Node<E> node, final E value) {  
489 final Node<E> newNode = createNode(value);  
490 addNode(newNode, node);  
491 }  
492  
493 /\*\*  
494 \* Creates a new node with the specified object as its  
495 \* <code>value</code> and inserts it after <code>node</code>.  
496 \* <p>  
497 \* This implementation uses {@link #createNode(Object)} and  
498 \* {@link #addNode(AbstractLinkedList.Node,AbstractLinkedList.Node)}.  
499 \*  
500 \* @param node node to insert after  
501 \* @param value value of the newly added node  
502 \* @throws NullPointerException if <code>node</code> is null  
503 \*/  
504 protected void addNodeAfter(final Node<E> node, final E value) {  
505 final Node<E> newNode = createNode(value);  
506 addNode(newNode, node.next);  
507 }  
508  
509 /\*\*  
510 \* Inserts a new node into the list.  
511 \*  
512 \* @param nodeToInsert new node to insert  
513 \* @param insertBeforeNode node to insert before  
514 \* @throws NullPointerException if either node is null  
515 \*/  
516 protected void addNode(final Node<E> nodeToInsert, final Node<E> insertBeforeNode) {  
517 nodeToInsert.next = insertBeforeNode;  
518 nodeToInsert.previous = insertBeforeNode.previous;  
519 insertBeforeNode.previous.next = nodeToInsert;  
520 insertBeforeNode.previous = nodeToInsert;  
521 size++;  
522 modCount++;  
523 }  
524  
525 /\*\*  
526 \* Removes the specified node from the list.  
527 \*  
528 \* @param node the node to remove  
529 \* @throws NullPointerException if <code>node</code> is null  
530 \*/  
531 protected void removeNode(final Node<E> node) {  
532 node.previous.next = node.next;  
533 node.next.previous = node.previous;  
534 size--;  
535 modCount++;  
536 }  
537  
538 /\*\*  
539 \* Removes all nodes by resetting the circular list marker.  
540 \*/  
541 protected void removeAllNodes() {  
542 header.next = header;  
543 header.previous = header;  
544 size = 0;  
545 modCount++;  
546 }  
547  
548 /\*\*  
549 \* Gets the node at a particular index.  
550 \*  
551 \* @param index the index, starting from 0  
552 \* @param endMarkerAllowed whether or not the end marker can be returned if  
553 \* startIndex is set to the list's size  
554 \* @return the node at the given index  
555 \* @throws IndexOutOfBoundsException if the index is less than 0; equal to  
556 \* the size of the list and endMakerAllowed is false; or greater than the  
557 \* size of the list  
558 \*/  
559 protected Node<E> getNode(final int index, final boolean endMarkerAllowed) throws IndexOutOfBoundsException {  
560 // Check the index is within the bounds  
561 if (index < 0) {  
562 throw new IndexOutOfBoundsException("Couldn't get the node: " +  
563 "index (" + index + ") less than zero.");  
564 }  
565 if (!endMarkerAllowed && index == size) {  
566 throw new IndexOutOfBoundsException("Couldn't get the node: " +  
567 "index (" + index + ") is the size of the list.");  
568 }  
569 if (index > size) {  
570 throw new IndexOutOfBoundsException("Couldn't get the node: " +  
571 "index (" + index + ") greater than the size of the " +  
572 "list (" + size + ").");  
573 }  
574 // Search the list and get the node  
575 Node<E> node;  
576 if (index < size / 2) {  
577 // Search forwards  
578 node = header.next;  
579 for (int currentIndex = 0; currentIndex < index; currentIndex++) {  
580 node = node.next;  
581 }  
582 } else {  
583 // Search backwards  
584 node = header;  
585 for (int currentIndex = size; currentIndex > index; currentIndex--) {  
586 node = node.previous;  
587 }  
588 }  
589 return node;  
590 }  
591  
592 //-----------------------------------------------------------------------  
593 /\*\*  
594 \* Creates an iterator for the sublist.  
595 \*  
596 \* @param subList the sublist to get an iterator for  
597 \* @return a new iterator on the given sublist  
598 \*/  
599 protected Iterator<E> createSubListIterator(final LinkedSubList<E> subList) {  
600 return createSubListListIterator(subList, 0);  
601 }  
602  
603 /\*\*  
604 \* Creates a list iterator for the sublist.  
605 \*  
606 \* @param subList the sublist to get an iterator for  
607 \* @param fromIndex the index to start from, relative to the sublist  
608 \* @return a new list iterator on the given sublist  
609 \*/  
610 protected ListIterator<E> createSubListListIterator(final LinkedSubList<E> subList, final int fromIndex) {  
611 return new LinkedSubListIterator<>(subList, fromIndex);  
612 }  
613  
614 //-----------------------------------------------------------------------  
615 /\*\*  
616 \* Serializes the data held in this object to the stream specified.  
617 \* <p>  
618 \* The first serializable subclass must call this method from  
619 \* <code>writeObject</code>.  
620 \*  
621 \* @param outputStream the stream to write the object to  
622 \* @throws IOException if anything goes wrong  
623 \*/  
624 protected void doWriteObject(final ObjectOutputStream outputStream) throws IOException {  
625 // Write the size so we know how many nodes to read back  
626 outputStream.writeInt(size());  
627 for (final E e : this) {  
628 outputStream.writeObject(e);  
629 }  
630 }  
631  
632 /\*\*  
633 \* Deserializes the data held in this object to the stream specified.  
634 \* <p>  
635 \* The first serializable subclass must call this method from  
636 \* <code>readObject</code>.  
637 \*  
638 \* @param inputStream the stream to read the object from  
639 \* @throws IOException if any error occurs while reading from the stream  
640 \* @throws ClassNotFoundException if a class read from the stream can not be loaded  
641 \*/  
642 @SuppressWarnings("unchecked")  
643 protected void doReadObject(final ObjectInputStream inputStream) throws IOException, ClassNotFoundException {  
644 init();  
645 final int size = inputStream.readInt();  
646 for (int i = 0; i < size; i++) {  
647 add((E) inputStream.readObject());  
648 }  
649 }  
650  
651 //-----------------------------------------------------------------------  
652 /\*\*  
653 \* A node within the linked list.  
654 \* <p>  
655 \* From Commons Collections 3.1, all access to the <code>value</code> property  
656 \* is via the methods on this class.  
657 \*/  
658 protected static class Node<E> {  
659  
660 /\*\* A pointer to the node before this node \*/  
661 protected Node<E> previous;  
662 /\*\* A pointer to the node after this node \*/  
663 protected Node<E> next;  
664 /\*\* The object contained within this node \*/  
665 protected E value;  
666  
667 /\*\*  
668 \* Constructs a new header node.  
669 \*/  
670 protected Node() {  
671 super();  
672 previous = this;  
673 next = this;  
674 }  
675  
676 /\*\*  
677 \* Constructs a new node.  
678 \*  
679 \* @param value the value to store  
680 \*/  
681 protected Node(final E value) {  
682 super();  
683 this.value = value;  
684 }  
685  
686 /\*\*  
687 \* Constructs a new node.  
688 \*  
689 \* @param previous the previous node in the list  
690 \* @param next the next node in the list  
691 \* @param value the value to store  
692 \*/  
693 protected Node(final Node<E> previous, final Node<E> next, final E value) {  
694 super();  
695 this.previous = previous;  
696 this.next = next;  
697 this.value = value;  
698 }  
699  
700 /\*\*  
701 \* Gets the value of the node.  
702 \*  
703 \* @return the value  
704 \* @since 3.1  
705 \*/  
706 protected E getValue() {  
707 return value;  
708 }  
709  
710 /\*\*  
711 \* Sets the value of the node.  
712 \*  
713 \* @param value the value  
714 \* @since 3.1  
715 \*/  
716 protected void setValue(final E value) {  
717 this.value = value;  
718 }  
719  
720 /\*\*  
721 \* Gets the previous node.  
722 \*  
723 \* @return the previous node  
724 \* @since 3.1  
725 \*/  
726 protected Node<E> getPreviousNode() {  
727 return previous;  
728 }  
729  
730 /\*\*  
731 \* Sets the previous node.  
732 \*  
733 \* @param previous the previous node  
734 \* @since 3.1  
735 \*/  
736 protected void setPreviousNode(final Node<E> previous) {  
737 this.previous = previous;  
738 }  
739  
740 /\*\*  
741 \* Gets the next node.  
742 \*  
743 \* @return the next node  
744 \* @since 3.1  
745 \*/  
746 protected Node<E> getNextNode() {  
747 return next;  
748 }  
749  
750 /\*\*  
751 \* Sets the next node.  
752 \*  
753 \* @param next the next node  
754 \* @since 3.1  
755 \*/  
756 protected void setNextNode(final Node<E> next) {  
757 this.next = next;  
758 }  
759 }  
760  
761 //-----------------------------------------------------------------------  
762 /\*\*  
763 \* A list iterator over the linked list.  
764 \*/  
765 protected static class LinkedListIterator<E> implements ListIterator<E>, OrderedIterator<E> {  
766  
767 /\*\* The parent list \*/  
768 protected final AbstractLinkedList<E> parent;  
769  
770 /\*\*  
771 \* The node that will be returned by {@link #next()}. If this is equal  
772 \* to {@link AbstractLinkedList#header} then there are no more values to return.  
773 \*/  
774 protected Node<E> next;  
775  
776 /\*\*  
777 \* The index of {@link #next}.  
778 \*/  
779 protected int nextIndex;  
780  
781 /\*\*  
782 \* The last node that was returned by {@link #next()} or {@link  
783 \* #previous()}. Set to <code>null</code> if {@link #next()} or {@link  
784 \* #previous()} haven't been called, or if the node has been removed  
785 \* with {@link #remove()} or a new node added with {@link #add(Object)}.  
786 \* Should be accessed through {@link #getLastNodeReturned()} to enforce  
787 \* this behaviour.  
788 \*/  
789 protected Node<E> current;  
790  
791 /\*\*  
792 \* The modification count that the list is expected to have. If the list  
793 \* doesn't have this count, then a  
794 \* {@link java.util.ConcurrentModificationException} may be thrown by  
795 \* the operations.  
796 \*/  
797 protected int expectedModCount;  
798  
799 /\*\*  
800 \* Create a ListIterator for a list.  
801 \*  
802 \* @param parent the parent list  
803 \* @param fromIndex the index to start at  
804 \* @throws IndexOutOfBoundsException if fromIndex is less than 0 or greater than the size of the list  
805 \*/  
806 protected LinkedListIterator(final AbstractLinkedList<E> parent, final int fromIndex)  
807 throws IndexOutOfBoundsException {  
808 super();  
809 this.parent = parent;  
810 this.expectedModCount = parent.modCount;  
811 this.next = parent.getNode(fromIndex, true);  
812 this.nextIndex = fromIndex;  
813 }  
814  
815 /\*\*  
816 \* Checks the modification count of the list is the value that this  
817 \* object expects.  
818 \*  
819 \* @throws ConcurrentModificationException If the list's modification  
820 \* count isn't the value that was expected.  
821 \*/  
822 protected void checkModCount() {  
823 if (parent.modCount != expectedModCount) {  
824 throw new ConcurrentModificationException();  
825 }  
826 }  
827  
828 /\*\*  
829 \* Gets the last node returned.  
830 \*  
831 \* @return the last node returned  
832 \* @throws IllegalStateException If {@link #next()} or {@link #previous()} haven't been called,  
833 \* or if the node has been removed with {@link #remove()} or a new node added with {@link #add(Object)}.  
834 \*/  
835 protected Node<E> getLastNodeReturned() throws IllegalStateException {  
836 if (current == null) {  
837 throw new IllegalStateException();  
838 }  
839 return current;  
840 }  
841  
842 @Override  
843 public boolean hasNext() {  
844 return next != parent.header;  
845 }  
846  
847 @Override  
848 public E next() {  
849 checkModCount();  
850 if (!hasNext()) {  
851 throw new NoSuchElementException("No element at index " + nextIndex + ".");  
852 }  
853 final E value = next.getValue();  
854 current = next;  
855 next = next.next;  
856 nextIndex++;  
857 return value;  
858 }  
859  
860 @Override  
861 public boolean hasPrevious() {  
862 return next.previous != parent.header;  
863 }  
864  
865 @Override  
866 public E previous() {  
867 checkModCount();  
868 if (!hasPrevious()) {  
869 throw new NoSuchElementException("Already at start of list.");  
870 }  
871 next = next.previous;  
872 final E value = next.getValue();  
873 current = next;  
874 nextIndex--;  
875 return value;  
876 }  
877  
878 @Override  
879 public int nextIndex() {  
880 return nextIndex;  
881 }  
882  
883 @Override  
884 public int previousIndex() {  
885 // not normally overridden, as relative to nextIndex()  
886 return nextIndex() - 1;  
887 }  
888  
889 @Override  
890 public void remove() {  
891 checkModCount();  
892 if (current == next) {  
893 // remove() following previous()  
894 next = next.next;  
895 parent.removeNode(getLastNodeReturned());  
896 } else {  
897 // remove() following next()  
898 parent.removeNode(getLastNodeReturned());  
899 nextIndex--;  
900 }  
901 current = null;  
902 expectedModCount++;  
903 }  
904  
905 @Override  
906 public void set(final E obj) {  
907 checkModCount();  
908 getLastNodeReturned().setValue(obj);  
909 }  
910  
911 @Override  
912 public void add(final E obj) {  
913 checkModCount();  
914 parent.addNodeBefore(next, obj);  
915 current = null;  
916 nextIndex++;  
917 expectedModCount++;  
918 }  
919  
920 }  
921  
922 //-----------------------------------------------------------------------  
923 /\*\*  
924 \* A list iterator over the linked sub list.  
925 \*/  
926 protected static class LinkedSubListIterator<E> extends LinkedListIterator<E> {  
927  
928 /\*\* The parent list \*/  
929 protected final LinkedSubList<E> sub;  
930  
931 protected LinkedSubListIterator(final LinkedSubList<E> sub, final int startIndex) {  
932 super(sub.parent, startIndex + sub.offset);  
933 this.sub = sub;  
934 }  
935  
936 @Override  
937 public boolean hasNext() {  
938 return nextIndex() < sub.size;  
939 }  
940  
941 @Override  
942 public boolean hasPrevious() {  
943 return previousIndex() >= 0;  
944 }  
945  
946 @Override  
947 public int nextIndex() {  
948 return super.nextIndex() - sub.offset;  
949 }  
950  
951 @Override  
952 public void add(final E obj) {  
953 super.add(obj);  
954 sub.expectedModCount = parent.modCount;  
955 sub.size++;  
956 }  
957  
958 @Override  
959 public void remove() {  
960 super.remove();  
961 sub.expectedModCount = parent.modCount;  
962 sub.size--;  
963 }  
964 }  
965  
966 //-----------------------------------------------------------------------  
967 /\*\*  
968 \* The sublist implementation for AbstractLinkedList.  
969 \*/  
970 protected static class LinkedSubList<E> extends AbstractList<E> {  
971 /\*\* The main list \*/  
972 AbstractLinkedList<E> parent;  
973 /\*\* Offset from the main list \*/  
974 int offset;  
975 /\*\* Sublist size \*/  
976 int size;  
977 /\*\* Sublist modCount \*/  
978 int expectedModCount;  
979  
980 protected LinkedSubList(final AbstractLinkedList<E> parent, final int fromIndex, final int toIndex) {  
981 if (fromIndex < 0) {  
982 throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);  
983 }  
984 if (toIndex > parent.size()) {  
985 throw new IndexOutOfBoundsException("toIndex = " + toIndex);  
986 }  
987 if (fromIndex > toIndex) {  
988 throw new IllegalArgumentException("fromIndex(" + fromIndex + ") > toIndex(" + toIndex + ")");  
989 }  
990 this.parent = parent;  
991 this.offset = fromIndex;  
992 this.size = toIndex - fromIndex;  
993 this.expectedModCount = parent.modCount;  
994 }  
995  
996 @Override  
997 public int size() {  
998 checkModCount();  
999 return size;  
1000 }  
1001  
1002 @Override  
1003 public E get(final int index) {  
1004 rangeCheck(index, size);  
1005 checkModCount();  
1006 return parent.get(index + offset);  
1007 }  
1008  
1009 @Override  
1010 public void add(final int index, final E obj) {  
1011 rangeCheck(index, size + 1);  
1012 checkModCount();  
1013 parent.add(index + offset, obj);  
1014 expectedModCount = parent.modCount;  
1015 size++;  
1016 LinkedSubList.this.modCount++;  
1017 }  
1018  
1019 @Override  
1020 public E remove(final int index) {  
1021 rangeCheck(index, size);  
1022 checkModCount();  
1023 final E result = parent.remove(index + offset);  
1024 expectedModCount = parent.modCount;  
1025 size--;  
1026 LinkedSubList.this.modCount++;  
1027 return result;  
1028 }  
1029  
1030 @Override  
1031 public boolean addAll(final Collection<? extends E> coll) {  
1032 return addAll(size, coll);  
1033 }  
1034  
1035 @Override  
1036 public boolean addAll(final int index, final Collection<? extends E> coll) {  
1037 rangeCheck(index, size + 1);  
1038 final int cSize = coll.size();  
1039 if (cSize == 0) {  
1040 return false;  
1041 }  
1042  
1043 checkModCount();  
1044 parent.addAll(offset + index, coll);  
1045 expectedModCount = parent.modCount;  
1046 size += cSize;  
1047 LinkedSubList.this.modCount++;  
1048 return true;  
1049 }  
1050  
1051 @Override  
1052 public E set(final int index, final E obj) {  
1053 rangeCheck(index, size);  
1054 checkModCount();  
1055 return parent.set(index + offset, obj);  
1056 }  
1057  
1058 @Override  
1059 public void clear() {  
1060 checkModCount();  
1061 final Iterator<E> it = iterator();  
1062 while (it.hasNext()) {  
1063 it.next();  
1064 it.remove();  
1065 }  
1066 }  
1067  
1068 @Override  
1069 public Iterator<E> iterator() {  
1070 checkModCount();  
1071 return parent.createSubListIterator(this);  
1072 }  
1073  
1074 @Override  
1075 public ListIterator<E> listIterator(final int index) {  
1076 rangeCheck(index, size + 1);  
1077 checkModCount();  
1078 return parent.createSubListListIterator(this, index);  
1079 }  
1080  
1081 @Override  
1082 public List<E> subList(final int fromIndexInclusive, final int toIndexExclusive) {  
1083 return new LinkedSubList<>(parent, fromIndexInclusive + offset, toIndexExclusive + offset);  
1084 }  
1085  
1086 protected void rangeCheck(final int index, final int beyond) {  
1087 if (index < 0 || index >= beyond) {  
1088 throw new IndexOutOfBoundsException("Index '" + index + "' out of bounds for size '" + size + "'");  
1089 }  
1090 }  
1091  
1092 protected void checkModCount() {  
1093 if (parent.modCount != expectedModCount) {  
1094 throw new ConcurrentModificationException();  
1095 }  
1096 }  
1097 }  
1098  
1099}