C:\Users\Rich\Documents\NetBeansProjects\Lab07\src\LinkedPositionalList.java

```
1
2 import java.util.NoSuchElementException;
3
4 /**
5  * Data Structures & Algorithms 6th Edition
6  * Goodrick, Tamassia, Goldwasser
7  * Code Fragements 7.9, 7.10, 7.11, 7.12 & 7.14
8  *
9  * toString method added by Latimer
10  */
11
12 /** Implementation of a positional list stored as a doubly linked list. */
13 public class LinkedPositionalList<E> implements PositionalList<E> {
14
15  //---------- nested Node class ----------------------------------
16
17    private static class Node<E> implements Position<E> {
18
19        private E element;      // reference to the element stored at this node
20        private Node<E> prev;   // reference to the prevous node in the list
21        private Node<E> next;   // reference to the subsequent node in the list
22
23        public Node( E e, Node<E> p, Node<E> n ){
24            element = e;
25            prev = p;
26            next = n;
27        }
28
29        @Override
30        public E getElement( ) throws IllegalStateException
31        {
32            if ( next == null )
33                throw new IllegalStateException( "Position no longer valid." );
34            return element;
35        }
36
37        public Node<E> getPrev( )
38        {
39            return prev;
40        }
41
42        public Node<E> getNext( )
43        {
44            return next;
45        }
46
47        public void setElemetn( E e )
48        {
49            element = e;
50        }
51
52        public void setPrev( Node<E> p )
53        {
54            prev = p;
55        }
56
```

Remember a position has a Node so position getElement= element at that node/position

```
57      public void setNext( Node<E> n )
58      {
59          next = n;
60      }
61  }//------------ end of nested Node class ---------------------
62
63  /**
64   * Data Structures & Algorithms 6th Edition
65   * Goodrick, Tamassia, Goldwasser
66   * Code Fragement 7.14
67   */
68
69  //----- nested PositionIterator class -----
70  private class PositionIterator implements Iterator<Position<E>>{
71      private Position<E> cursor = first();  // position of the next element to report
72      private Position<E> recent = null;     // position of last reported element
73      /** Tests whether the iterator has a next object. */
74      @Override
75      public boolean hasNext( ) { return ( cursor != null ); }
76      /** Returns the next position in the iterator. */
77      @Override
78      public Position<E> next( ) throws NoSuchElementException {
79          if ( cursor == null ) throw new NoSuchElementException( "nothing left " );
80          recent = cursor;
81          cursor = after( cursor );
82          return recent;
83      }
84      /** Removes the element returned by most recent call to next. */
85      @Override
86      public void remove( ) throws IllegalStateException {
87          if ( recent == null ) throw new IllegalStateException( "nothing to remove" );
88          LinkedPositionalList.this.remove( recent );        // remove from outer list
89          recent = null;            // do not allow remove again until next is called
90      }
91  } //----- end of nested PositionIterator class -----
92
93  //----- nested PositionIterable class -----
94  private class PositionIterable implements Iterable<Position<E>>{
95      @Override
96      public Iterator<Position<E>> iterator( ) { return new PositionIterator( ); }
97  } //----- end of nested PositionIterable class -----
98
99  /** Returns an iterable representation of the list's positions.
100  * @return */
101 public Iterable<Position<E>> positions( ) {
102     return new PositionIterable( );  // create a new instance of the inner class
103 }
104
105 //----- nested ElementIterator class -----
106 /* This class adapts the iteration produced by positions( ) to return elements. */
107 private class ElementIterator implements Iterator<E> {
108     Iterator<Position<E>> posIterator = new PositionIterator( );
109     @Override
110     public boolean hasNext( ) { return posIterator.hasNext( ); }
111     @Override
112     public E next( ) { return posIterator.next( ).getElement( ); } // return element
113     @Override
114     public void remove( ) { posIterator.remove( ); }
115 }
```

if a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface.

the Node class implements position

```
LuckyNumberList aList = new LuckyNumberList();

Iterator<Position<LuckyNumber>> defaultListIterator = aList.positions().iterator();
```

PositionIterable which in turn will create a new instance of iterator if called

Iterable iterator()

has a type Iterator<Position<E>>

to implement an element iterator

```
Iterator< Position<E> elementIter = new ElementIterator()
```

```
116
117  /** Returns an iterator of the elements stored in the list */
118  public Iterator<E> iterator( ) { return new ElementIterator( ); }
119
120
121  // instance variables of the LinkedPositionalList
122
123  private Node<E> header;           // header sentinel
124  private Node<E> trailer;          // trailer sentinel
125  private int size = 0;             // number of elements in the list
126
127  public LinkedPositionalList( ){
128     header = new Node<>( null, null, null );     // create header
129     trailer = new Node<>( null, header, null );  // create trailer is preceded by header
130     header.setNext(trailer);                     // header is followed by trailer
131  }
132
133  // private utilities
134  /**
135   * @param p position to validate
136   * @return node if position is valid
137   * @throws IllegalArgumentException if p no longer in list or p is not a position
138   */
139  private Node<E> validate( Position<E> p ) throws IllegalArgumentException {
140
141     if( !(p instanceof Node )) throw new IllegalArgumentException( "Invalid p" );
142
143     Node<E> node = ( Node<E> ) p;   // safe cast
144
145     if ( node.getNext() == null )
146        throw new IllegalArgumentException( "p is no longer in the list" );
147
148     return node;
149  }
150
151  /**
152   * @param node to be returned as position if not header or trailer
153   * @return position of node
154   */
155  private Position<E> position( Node<E> node ){
156    if ( node == header || node == trailer )
157       return null;
158    return node;
159  }
160
161  // public accessor methods
162
163  /**
164   * @return number of elements in linked list
165   */
166  @Override
167  public int size( ){
168     return size;
169  }
170
171  /**
172   * @return true if list is empty, false other wise
173   */
174  @Override
```

xx continuation of linked positionalList

that p may be looking at the trailer

not to be confused with positions() of type Iterable<Position<E>>

```
175   public boolean isEmpty( ){
176      return ( size == 0 );
177   }
178
179   /**
180    * @return the first position in linked list (null if empty).
181    */
182   @Override
183   public Position<E> first( ){
184      return position( header.getNext( ) );
185   }
186
187   /**
188    * @return the last position in linked list (null if empty).
189    */
190   @Override
191   public Position<E> last( ){
192      return position( trailer.getPrev( ) );
193   }
194
195   /**
196    * @param p position to get position immediately before
197    * @return position before p
198    * @throws IllegalArgumentException if p not valid
199    */
200   @Override
201   public Position<E> before( Position<E> p ) throws IllegalArgumentException{
202     Node<E> node = validate( p );
203     return position( node.getPrev( ) );
204   }
205
206   /**
207    * @param p position to get immediately after
208    * @return position after p
209    * @throws IllegalArgumentException if p not valid
210    */
211   @Override
212   public Position<E> after( Position<E> p ) throws IllegalArgumentException{
213      Node<E> node = validate( p );
214      return position( node.getNext( ) );
215   }
216
217   // private utilities
218
219   /**
220    * @param e element to be added
221    * @param pred node to add element after
222    * @param succ node to add element before
223    * @return position of newly added element
224    */
225   private Position<E> addBetween(E e, Node<E> pred, Node<E> succ ){
226      Node<E> newest = new Node<>(e, pred, succ);  // create and link new node
227      pred.setNext(newest);
228      succ.setPrev(newest);
229      size++;
230      return newest;
231   }
232
233   // public update methods
```

position doesn't have a getNext method

```
234
235   /**
236    * @param e element to be added just after header
237    * @return  position of newly added element
238    */
239   @Override
240   public Position<E> addFirst(E e) {
241      return addBetween( e, header, header.getNext() );
242   }
243
244   /**
245    * @param e element to be added just before trailer
246    * @return position of newly added element
247    */
248   @Override
249   public Position<E> addLast( E e ){
250      return addBetween(e, trailer.getPrev( ), trailer );
251   }
252
253   /**
254    *
255    * @param p position to add element before
256    * @param e element to be added
257    * @return position of newly added element
258    * @throws IllegalArgumentException if p is not valid
259    */
260   @Override
261   public Position<E> addBefore( Position<E> p, E e ) throws IllegalArgumentException {
262      Node<E> node = validate( p );
263      return addBetween(e, node.getPrev( ), node );
264   }
265
266   /**
267    * @param p position to add element after
268    * @param e element to be added
269    * @return position of newly added element
270    * @throws IllegalArgumentException if p is not valid
271    */
272   @Override
273   public Position<E> addAfter( Position<E> p, E e ) throws IllegalArgumentException {
274      Node<E> node = validate( p );
275      return addBetween(e, node, node.getNext( ) );
276   }
277
278   /**
279    * @param p position of node to update
280    * @param e new element for node
281    * @return old element in node before update
282    * @throws IllegalArgumentException if p not valid
283    */
284   @Override
285   public E set( Position<E> p, E e ) throws IllegalArgumentException {
286      Node<E> node = validate( p );
287      E answer = node.getElement( );
288      node.setElemetn( e );
289      return answer;
290   }
291
292   /**
```

```
293     * @param p position to be removed
294     * @return element that was removed
295     * @throws IllegalArgumentException if p not valid
296     */
297    public E remove( Position<E> p ) throws IllegalArgumentException {
298       Node<E> node = validate( p );
299       Node<E> predecessor = node.getPrev();
300       Node<E> successor = node.getNext();
301       predecessor.setNext( successor );
302       successor.setPrev( predecessor );
303       size--;
304       E answer = node.getElement( );
305       node.setElemetn( null );
306       node.setNext( null );
307       node.setPrev( null );
308       return answer;
309    }
310 }
311
```