```
 1: import java.util.Iterator;
 2: import java.util.NoSuchElementException;
 3:
 4:
 5: /**
 6:  * Stores a list of <T> elements
 7:  *
 8:  * @author 1828799
 9:  *
10:  * @param <T> unspecified objects type T
11:  */
12: public class MyLinkedList<T> implements HansenCollection<T>,
13: Comparable<MyLinkedList<T>>, Iterable<T>
14: {
15:    // Instance Variables
16:  private  ListNode  _head  =  null;
17:  private  ListNode  _tail  =  null;
18:    private int _size = 0;
19:
20:
21:    /**
22:     * Creates a new MyLinkedList
23:     */
24:    public MyLinkedList(){}
25:
26:
27:    /**
28:     * Appends an element to the end of the list.
29:     *
30:     * @param newElement the specified element to be added
31:     *
32:     * @throws CollectionFullException (never thrown since list can never fill)
33:     * @throws NullPointerException if element to be added is null
34:     */
35:    @Override
36:    public void addElement(T newElement) throws CollectionFullException,
37:          NullPointerException
38:    {
39:       // Verify that the input isn't null
40:       if (newElement == null)
41:       {
42:          throw new NullPointerException();
43:       }
44:
45:       // Add element using prependElement if the list is empty
46:       if (size() == 0)
47:       {
48:          prependElement(newElement);
49:       }
50:       else // When the list isn't empty
51:       {
52:          // Set newElement to be in the node which is the tail's next
53:          _tail.setNext(new ListNode<T>(newElement));
54:
55:          // Update the tail
56:          _tail = _tail.getNext();
57:
58:          // Increment the size
59:          _size++;
60:
```

```
61:         }
62:     }
63:
64:
65:     /**
```

```
66:      * Prepends an element to the list
67:      *
68:      * @param newElement the element to prepend to list
69:      *
70:      * @throws NullPointerException if element to be prepended is null
71:      */
72:     public void prependElement(T newElement) throws NullPointerException
73:     {
74:        // Create new node to hold newElement
75:        ListNode toPrepend = new ListNode<T>(newElement);
76:
77:        // Verify that the input isn't null
78:        if (newElement == null)
79:        {
80:           throw new NullPointerException();
81:        }
82:
83:        // Set the added nodes next to be the head
84:        toPrepend.setNext(_head);
85:
86:        // Update the head to be the node holding newElement
87:        _head = toPrepend;
88:
89:        // If size is zero then also set the new the new node to be the tail
90:        if (size() ==  0)
91:        {
92:           _tail = toPrepend;
93:        }
94:
95:        // Increment the size
96:        _size++;
97:     }
98:
99:
100:    /**
101:     * Inserts the new element after the first instance of the specified element
102:     * in the list, if the specified element is in list.
103:     *
104:     * @param existingElement the element to insert the new element after
105:     * @param newObject the element to be added
106:     *
107:     * @throws NullPointerException if element to be inserted is null
108:     * @throws NoSuchElementException specified element is not in list
109:     * @throws CollectionFullException never thrown since never full
110:     */
111:    public void insertAfter(T existingObject, T newObject) throws
112:    NoSuchElementException, NullPointerException, CollectionFullException
113:    {
114:       ListNode nodeBefore = null;
115:       ListNode nodeToInsert = null;
116:
117:       // Verify that the input isn't null
118:       if (existingObject == null || newObject == null)
119:       {
120:          throw new NullPointerException();
121:       }
122:
123:       // Find node containing the existing object by identity (if it exists)
124:       nodeBefore = findNode(existingObject, false);
125:
```

```
126:       // If the nodeBefore isn't in the list throw exception
127:       if (nodeBefore == null)
128:       {
129:          throw new NoSuchElementException();
130:       }
```

```
131:
132:        // Special Case: nodeBefore is the tail
133:        if (nodeBefore == _tail)
134:        {
135:           // Use addElement if nodeBefore is tail
136:           addElement(newObject);
137:        }
138:        else // Regular Case: nodeBefore is not the tail
139:        {
140:           // Create new node holding the newObject
141:           nodeToInsert = new ListNode<T>(newObject);
142:
143:           // Set new node's next to nodeBefore's next
144:           nodeToInsert.setNext(nodeBefore.getNext());
145:
146:           // Set nodeBefore's next to new node
147:           nodeBefore.setNext(nodeToInsert);
148:
149:           // Increment the size of list
150:           _size++;
151:        }
152:     }
153:
154:
155:     /**
156:      * Removes the first occurrence of the specified element from this list,
157:      * if it is present.
158:      * @param elementToRemove the specified element to be removed
159:      *
160:      * @throws NoSuchElementException if specified element is not in list
161:      */
162:     @Override
163:     public void removeElement(T elementToRemove) throws NoSuchElementException
164:     {
165:        // Set ListNode variable current to null
166:        ListNode current = _head;
167:
168:        // Check for null input or an empty list
169:        if (elementToRemove == null || size() == 0)
170:        {
171:           throw new NoSuchElementException();
172:        }
173:
174:        // Special Case: Element to remove is the head
175:        if (_head.getContents().equals(elementToRemove))
176:        {
177:           // Set head equal to head's next
178:           _head = _head.getNext();
179:        }
180:
181:        // Walk the list to find the node before one containing element to remove
182:        else // When element to remove isn't the head
183:        {
184:           while (current.getNext() != null
185:                 && !((current.getNext()).getContents().equals(elementToRemove)))
186:           {
187:              // Set current to current's next
188:              current = current.getNext();
189:           }
190:
```

```
191:          if (current.getNext() == null) // When element isn't in the list
192:          {
193:             throw new NoSuchElementException();
194:          }
195:          else // When element is in the list
```

```
196:          {
197:             // Remove the element by manipulated the node before's next
198:             current.setNext((current.getNext()).getNext());
199:          }
200:       }
201:
202:       // Decrement the size of the list
203:       _size--;
204:    }
205:
206:
207:    /**
208:     * Returns the element if element is in list
209:     *
210:     * @param elementSought the specified element to be found
211:     *
212:     * @return returns the element if found, returns null if element is not
213:     * found
214:     */
215:    @Override
216:    public T findElement(T elementSought)
217:    {
218:       // Set T object variable elementFound to null
219:       T elementFound = null;
220:       ListNode nodeFound = findNode(elementSought, true); // By equality
221:
222:       // If the call findNode(elementSought) doesn't return null
223:       if (nodeFound != null)
224:       {
225:          // Set elementFound to content's of the node returned from findNode
226:          elementFound = (T) nodeFound.getContents();
227:       }
228:
229:       return elementFound;
230:    }
231:
232:
233:    /**
234:     * Returns true if this list contains the specified element
235:     *
236:     * @param elementSought the specified element to be found in list
237:     *
238:     * @return true if specified element is in list
239:     */
240:    @Override
241:    public boolean containsElement(T elementSought)
242:    {
243:       // Use findElement to determine if elementSought is in list
244:       return (findElement(elementSought) != null );
245:    }
246:
247:
248:    /**
249:     * Compares the list to the specified list
250:     *
251:     * @param o the object to be compared
252:     *
253:     * @throws NullPointerException if the specified object is null
254:     *
255:     * @return Returns a negative integer, zero, or a positive integer as this
256:     * object is less than, equal to, or greater than specified object.
```

```
257:     */
258:    @Override
259:    public int compareTo(MyLinkedList<T> o)
260:    {
```

```
  261:
  262:        int compareValue = 0; // The compared value to be returned
  263:        ListNode myCurrent = _head; // Starting point of this list
  264:        ListNode otherCurrent = o._head; // Starting point of other list
  265:
  266:        // CHECK IF OBJECT TO BE COMPARED TO IS NULL
  267:        // if o is null
  268:        //    throw NullPointerException
  269:
  270:        // Walk the list and compare at each step
  271:        while (compareValue == 0 && myCurrent != null && otherCurrent != null)
  272:        {
  273:           // Set compareValue to return of comparing contents of two currents
  274:           compareValue = ((Comparable) myCurrent.getContents()).compareTo((Compar
able) otherCurrent.getContents());
  275:
  276:           // Move each list the the next in list
  277:           myCurrent = myCurrent.getNext();
  278:           otherCurrent = otherCurrent.getNext();
  279:        }
  280:
  281:        // Check for when the lists are the same but one is longer
  282:        if (compareValue == 0 && size() - o.size() != 0)
  283:        {
  284:           // When different sizes set return to be this size - other size
  285:           compareValue = size() - o.size();
  286:        }
  287:
  288:        return compareValue;
  289:     }
  290:
  291:
  292:     /**
  293:      * Returns false since list can never be full.
  294:      *
  295:      * @return always returns false since list can never be full
  296:      */
  297:     @Override
  298:     public boolean isFull()
  299:     {
  300:        // WILL ALWAYS BE FALSE
  301:        return false;
  302:     }
  303:
  304:
  305:     /**
  306:      * Returns true if list contains no elements.
  307:      *
  308:      * @return true if list is empty
  309:      */
  310:     @Override
  311:     public boolean isEmpty()
  312:     {
  313:        // Use size() to determine if list is empty
  314:        return (size() == 0);
  315:     }
  316:
  317:
  318:     /**
  319:      * Empties the list
```

```
320:     */
321:    @Override
322:    public void makeEmpty()
323:    {
324:       // Set head to null
```

```
325:        _head = null;
326:
327:        // Set tail to null
328:        _tail = null;
329:
330:        // Set size to 0
331:        _size = 0;
332:     }
333:
334:
335:     /**
336:      * Returns the number of elements in list
337:      *
338:      * @return the number of objects in list
339:      */
340:     @Override
341:     public int size()
342:     {
343:        // Return the size that is tracked throughout
344:        return _size;
345:     }
346:
347:
348:     /**
349:      * Creates an iterator that can iterate over your list
350:      *
351:      * @return the new iterator
352:      */
353:     public Iterator<T> iterator()
354:     {
355:        return new Iterator<T>() {
356:
357:           ListNode current = _head;
358:
359:           /**
360:            * Returns true if the iteration has more elements
361:            *
362:            * @return  true if the iteration has more elements
363:            */
364:           public boolean hasNext()
365:           {
366:              return (current != null);
367:           }
368:
369:           /**
370:            * Returns the next element in the iteration
371:            *
372:            * @return the next element in the iteration
373:            *
374:            * @throws NoSuchElementException if the iteration has no more elements
375:            */
376:           public T next() throws NoSuchElementException
377:           {
378:              if (!hasNext())
379:              {
380:                 throw new NoSuchElementException();
381:              }
382:
383:              T contents = (T) current.getContents();
384:              current = current.getNext();
385:
```

```
386:             return contents;
387:         }
388:
389:         /**
```

```
390:             * Unsupported operation
391:             */
392:            public void remove(){throw new UnsupportedOperationException();}
393:        };
394:    }
395:
396:
397:    /**
398:     * Finds the node in the list that contains specified object, if such a node
399:     * is in the list.
400:     *
401:     * @param contentsOfNode the contents of the node to be found
402:     *
403:     * @return first node containing instance of specified object, null if no
404:     * node contains specified objects
405:     */
406:    private ListNode findNode(T contentsOfNode, boolean byEquality)
407:    {
408:       ListNode nodeFound = null; // Return value
409:       ListNode current = _head;
410:
411:       // Walk the list to find the node containing specified elements
412:       while (byEquality && current != null && nodeFound == null)
413:       {
414:          // If current's contents equal contentsOfNode
415:          if (current.getContents().equals(contentsOfNode))
416:          {
417:             // Set nodeFound to be current
418:             nodeFound = current;
419:          }
420:          // Set current to current's next
421:          current = current.getNext();
422:       }
423:
424:       while (!byEquality && current != null && nodeFound == null)
425:       {
426:          // If current's contents have same identity as contentsOfNode
427:          if (current.getContents() == contentsOfNode)
428:          {
429:             // Set nodeFound to be current
430:             nodeFound = current;
431:          }
432:          // Set current to current's next
433:          current = current.getNext();
434:       }
435:
436:       return nodeFound;
437:    }
438:
439:
440:    /**
441:     * An inner-class for use by a Linked List to hold the contents of the list.
442:     * Note that this class and its methods are not public nor private, their
443:     * visibility is within the "package" and that includes any other classes
444:     * defined in the same file.  This class definition can appear INSIDE the
445:     * definition of a Linked List class.
446:     *
447:     * @author David M. Hansen
448:     * @version 2.0
449:     * @param < T > type of object contained by this node
450:     */
```

```
451:     class ListNode < T >
452:     {
453:
454:        // Constructors
```

```
455:
456:        /**
457:         * Create a new ListNode holding the given object and
pointing to the
458:         * given node as the next node in the list
459:         * @param objectToHold the object to store in this node
460:         * @param nextNode the node this node should point to. Can
be null
461:         */
462:        ListNode(T objectToHold,
ListNode nextNode) 463:    {
464:           p_contents = objectToHold;
465:           p_next
= nextNode; 466:
           }
467:
468:        /**
469:         * Create a new ListNode holding the given object.  No next
node
470:         * @param objectToHold the object to store in this node
471:         */
472:        ListNode(T objectToHold)
473:        {
474:           // Use the more general constructor passing null as the
next
475:           // node
476:           this(objectToHold, null);
477:        }
478:
479:
480:        // Accessors
481:
482:        /**
483:         * @returns Object stored within this node
484:         */
485:        T getContents()
486:        {
487:           return p_contents;
488:        }
489:
490:        /**
491:         * @return the next node
492:         */
493:        ListNode getNext()
494:        {
495:           return p_next;
496:        }
497:
498:
499:
500:        // Mutators
501:
502:        /**
503:         * Set the node this node is linked to
504:         * @param nextNode the node to point to as our next node.
Can be null.
505:         */
506:        void
```

```
setNext(ListNode nextNode)
507:        {
508:           p_next
= nextNode; 509:
            }
510:
511:
512:        // Private attributes
513:        private T p_contents; // The object held by
this node 514: private ListNode p_next; // A reference
to the next node 515:
516:     } // ListNode
517:
518:
519: }
```