Exercise 9.11

The *Device* class must have a definition of *getName* method, because *Device* class is the static type.

Exercise 9.12

At runtime, the *getName* that is defined in the dynamic type - *Printer* class - will be executed.

Exercise 9.13

All classes inherit from *Object* class, so if the *Student* class does not override the `toString` method, the one in `object.toString` will be used. This will print out the class name and a memory address. These lines will compile.

9.14

The following lines will compile. `System.out.println()` will search for `toString` method when attempting to get a representation of the object. This will return the same piece of information in *Exercise 9.13*, eg. `Student@43b6c732`.

Exercise 9.15

Since the *Object* class has a method `toString`, this code will compile. At runtime, the compiler will check if this method exists in the dynamic type, so if we have overridden it with another method in the *Student* class, the custom method will be executed, i.e. the one in the *Student* class.

Exercise 9.16

```
1 T x = new D();
```

Linked List

Time complexity

isHealthy = $O(n)$
LinkedList = $O(1)$
addFirst = $O(1)$
addLast = $O(1)$
getFirst = $O(1)$
getLast = $O(1)$
get = $O(n)$
removeFirst = $O(1)$
clear = $O(1)$
size = $O(1)$
isEmpty = $O(1)$
toString = $O(n)$

Source code

LinkedList class

```
1  package inda2v;
2
3  /**
4   * A singly linked list.
5   *
6   * @author Artem Los (artem@artemlos.net)
7   * @version
8   */
9  public class LinkedList<T> {
10     private ListElement<T> first;    // First element in list.
11     private ListElement<T> last;     // Last element in list.
12     private int size;                // Number of elements in list.
13
14     /**
15      * A list element.
16      */
17     private static class ListElement<T> {
18         public T data;
19         public ListElement<T> next;
20
21         public ListElement(T data) {
22             this.data = data;
23             this.next = null;
24         }
25     }
26
27     /**
28      * This TEST METHOD returns true if the following invariants hold:
29      * <ul>
30      *   <li> size equals the number of list elements, </li>
31      *   <li> if size == 0, first == null and last == null, </li>
32      *   <li> if size > 0, first != null and last != null, </li>
33      *   <li> if size == 1, first == last, </li>
34      *   <li> last.next == null. </li>
35      * </ul>
36      */
37     public boolean isHealthy() {
38
39         ListElement<T> current = first;
40         int counter = 0;
41
42         while(current != null && current.data != null)
43         {
44             current = current.next;
45             counter++;
46         }
47
48         if(size != counter)
49         {
50             return false;
51         }
52
```

```
53
54          if(last.next != null)
55              return false;
56          if(size == 0)
57              return (first == null && last == null);
58          if(size > 0)
59              return (first != null && last != null);
60          if(size == 1)
61              return  (first == last);
62
63          return false;
64      }
65
66      /**
67       * Creates an empty list.
68       */
69      public LinkedList() {
70          // TODO
71          last = new ListElement<T>(null);
72          first = last;
73          size = 0;
74      }
75
76      /**
77       * Inserts the given element at the beginning of this list.
78       */
79      public void addFirst(T element) {
80
81          ListElement<T> newElement = new ListElement<T>(element);
82
83
84          if(first.data == null)
85          {
86              first = newElement;
87              last = first; // switched from first=last (remember same in
                    addLast)
88          }
89          else
90          {
91              /*
92              newElement.next = new ListElement<T>(first.data);
93              newElement.next.next = first.next;
94
95              first = newElement;
96              */
97
98              newElement.next = first;
99              first = newElement;
100         }
101         size++;
102     }
103
104     /**
105      * Inserts the given element at the end of this list.
106      */
107     public void addLast(T element) {
```

```
108
109         ListElement<T> newElement = new ListElement<T>(element);
110
111         if(last.data == null)
112         {
113             last = newElement;
114             first=last; // changed from last to first. (same has to be
                    done in addLast)
115         }
116         else
117         {
118             last.next = newElement;
119             last = newElement;
120         }
121
122         size++;
123     }
124
125     /**
126      * Returns the first element of this list.
127      * Returns <code>null</code> if the list is empty.
128      */
129     public T getFirst() {
130         // TODO
131         if(first == null)
132             return null;
133
134         return first.data;
135     }
136
137     /**
138      * Returns the last element of this list.
139      * Returns <code>null</code> if the list is empty.
140      */
141     public T getLast() {
142         // TODO
143
144         if(last == null)
145             return null;
146
147         return last.data;
148     }
149
150     /**
151      * Returns the element at the specified position in this list.
152      * Returns <code>null</code> if <code>index</code> is out of bounds.
153      */
154     public T get(int index) {
155
156         ListElement<T> current;
157
158         current = first;
159
160         if(index >= size)
161         {
162             //not allowed. fail.
```

```
163             return null;
164         }
165 
166         for (int i = 0; i < index; i++) {
167             current = current.next;
168         }
169 
170         return current.data;
171     }
172 
173     /**
174      * Removes and returns the first element from this list.
175      * Returns <code>null</code> if the list is empty.
176      */
177     public T removeFirst() {
178         // TODO
179         if(size == 0 || first == null)
180             return null;
181 
182         ListElement<T> temp = first;
183 
184         first = first.next;
185         size--;
186 
187         return temp.data;
188     }
189 
190     /**
191      * Removes all of the elements from this list.
192      */
193     public void clear() {
194         last = new ListElement<T>(null);
195         first = last;
196         size = 0;
197     }
198 
199     /**
200      * Returns the number of elements in this list.
201      */
202     public int size() {
203         // TODO
204         return size;
205     }
206 
207     /**
208      * Returns <code>true</code> if this list contains no elements.
209      */
210     public boolean isEmpty() {
211         if(size == 0)
212             return true;
213         else
214             return false;
215     }
216 
217     /**
218      * Returns a string representation of this list. The string
```

```
219         * representation consists of a list of the elements enclosed in
220         * square brackets ("[]"). Adjacent elements are separated by the
221         * characters ", " (comma and space). Elements are converted to
222         * strings by the method toString() inherited from Object.
223         */
224        public String toString() {
225
226            String out = "[";
227
228            ListElement<T> current;
229
230            current = first;
231
232            if(current == null || current.data == null)
233                return "[]";
234
235            for (int i = 0; i < size-1; i++) {
236                out += current.data.toString() + ", ";
237                current = current.next;
238
239            }
240            out += current.data.toString() + "]";
241
242            return out;
243
244        }
245    }
```

LinkedListTest class

```
1   package inda2v;
2
3   import static org.junit.Assert.*;
4   import inda2v.*;
5
6   import org.junit.After;
7   import org.junit.Before;
8   import org.junit.Test;
9
10  public class LinkedListTest {
11
12
13      LinkedList<Object> newList;
14
15      @Before
16      public void Initialization() {
17          newList = new LinkedList<Object>();
18      }
19
20      @Test
21      public void LinkedListTest() {
22          //test for the constructor
23
24          LinkedList<Object> aList = new LinkedList<Object>();
25
26          //relies on that we declared size() correctly.
```

```
27          assertEquals(0, aList.size());
28
29          assertEquals(null, aList.getFirst());
30          assertEquals(null, aList.getLast());
31      }
32
33      @Test
34      public void AddFirstTest()  {
35          //
36          int currentSize = newList.size();
37
38          Object cat = "A Cat stored as an object";
39          newList.addFirst(cat);
40
41          assertEquals(currentSize +1, newList.size());
42          assertEquals(cat , newList.getFirst());
43
44
45          // some additional tests for getFirst and getLast
46          LinkedList<String> bList = new LinkedList<String>();
47
48          bList.addFirst("hi");
49          bList.addFirst("there");
50          bList.addFirst("test");
51
52          assertEquals("hi", bList.getLast());
53          assertEquals("test", bList.getFirst());
54
55          assertTrue(bList.isHealthy());
56
57      }
58
59      @Test
60      public void AddLastTest() {
61          int currentSize = newList.size();
62
63          Object cat = "The last cat stored as an object";
64
65          newList.addLast(cat);
66
67          assertEquals(currentSize +1, newList.size());
68          assertEquals(cat, newList.getLast());
69
70
71          // some additional tests for getFirst and getLast
72          LinkedList<String> bList = new LinkedList<String>();
73
74          bList.addFirst("hi");
75          bList.addLast("matrix");
76          bList.addLast("determinant");
77
78          assertEquals("determinant", bList.getLast());
79          assertEquals("hi", bList.getFirst());
80
81          assertTrue(bList.isHealthy());
82          assertTrue(newList.isHealthy());
```

```
83
84      }
85
86      @Test
87      public void getTest()
88      {
89
90          //assertTrue(newList.isHealthy());
91
92          Object first = newList.getFirst();
93
94          Object atIndexTwo = newList.get(2);
95
96          assertEquals(first, newList.getFirst()); // pass -> nothing weird
                occurred because of change in reference during search.
97
98          Object objBefore  = "The cat gets into a List.";
99          Object objAfter = "The cat survived.";
100
101
102         newList.addFirst(objBefore);
103         newList.addLast(objAfter);
104
105         assertEquals(objBefore, newList.get(0));
106         assertEquals(objAfter, newList.get(1));
107
108         LinkedList<String> ls = new LinkedList<String>();
109         ls.addFirst("hello");
110         ls.addFirst("hi");
111         ls.addLast("see you");
112         ls.addLast("ciao");
113
114
115     }
116
117     @Test
118     public void toStringTest() {
119
120         LinkedList<String> test = new LinkedList<String>();
121
122         test.addFirst("hi");
123         test.addFirst("there");
124         test.addFirst("test");
125
126
127         assertEquals("[test, there, hi]", test.toString());
128     }
129
130     @Test
131     public void emptyArrayTest() {
132         LinkedList<String> test = new LinkedList<String>();
133
134         assertTrue(test.isEmpty());
135
136         test.addFirst("hi");
137
```

```
138             assertEquals("hi", test.getFirst());
139             assertEquals("hi", test.getLast());
140 
141         }
142 
143 
144         @After
145         public void removeFirstTest()
146         {
147             Object firstItem = newList.getFirst();
148             int currentSize = newList.size();
149 
150             Object removedItem = newList.removeFirst();
151 
152             if(newList.size() != 0)
153             {
154                 assertTrue(newList.isHealthy());
155                 assertEquals(currentSize-1, newList.size());
156                 assertEquals(firstItem, removedItem);
157                 assertTrue(newList.isHealthy());
158             }
159 
160         }
161 
162         @After
163         public void ClearingTest()
164         {
165             newList.clear();
166             assertTrue(newList.isEmpty());
167         }
168 
169 
170 
171 }
```