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

```
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

```
package inda2v;

/**
 * A singly linked list.
 *
 * @author Artem Los (artem@artemlos.net)
 * @version
 */
public class LinkedList<T> {
    private ListElement<T> first;   // First element in list.
    private ListElement<T> last;    // Last element in list.
    private int size;               // Number of elements in list.

    /**
     * A list element.
     */
    private static class ListElement<T> {
        public T data;
        public ListElement<T> next;

        public ListElement(T data) {
            this.data = data;
            this.next = null;
        }
    }

    /**
     * This TEST METHOD returns true if the following invariants hold:
     * <ul>
     *   <li> size equals the number of list elements, </li>
     *   <li> if size == 0, first == null and last == null, </li>
     *   <li> if size > 0, first != null and last != null, </li>
     *   <li> if size == 1, first == last, </li>
     *   <li> last.next == null. </li>
     * </ul>
     */
    public boolean isHealthy() {

        ListElement<T> current = first;
        int counter = 0;

        while(current != null && current.data != null)
        {
            current = current.next;
            counter++;
        }

        if(size != counter)
        {
            return false;
        }


        if(last.next != null)
            return false;
        if(size == 0)
            return (first == null && last == null);
        if(size > 0)
            return (first != null && last != null);
        if(size == 1)
            return  (first == last);

        return false;
    }

    /**
     * Creates an empty list.
     */
    public LinkedList() {
        // TODO
        last = new ListElement<T>(null);
        first = last;
        size = 0;
    }

    /**
     * Inserts the given element at the beginning of this list.
     */
    public void addFirst(T element) {

        ListElement<T> newElement = new ListElement<T>(element);


        if(first.data == null)
        {
            first = newElement;
            last = first; // switched from first=last (remember same in addLast)
        }
        else
        {
            /*
            newElement.next = new ListElement<T>(first.data);
            newElement.next.next = first.next;

            first = newElement;
            */

            newElement.next = first;
            first = newElement;
        }
        size++;
    }

    /**
     * Inserts the given element at the end of this list.
     */
    public void addLast(T element) {

        ListElement<T> newElement = new ListElement<T>(element);

        if(last.data == null)
        {
            last = newElement;
            first=last; // changed from last to first. (same has to be
                done in addLast)
        }
        else
        {
            last.next = newElement;
            last = newElement;
        }

        size++;
    }

    /**
     * Returns the first element of this list.
     * Returns <code>null</code> if the list is empty.
     */
    public T getFirst() {
        // TODO
        if(first == null)
            return null;

        return first.data;
    }

    /**
     * Returns the last element of this list.
     * Returns <code>null</code> if the list is empty.
     */
    public T getLast() {
        // TODO

        if(last == null)
            return null;

        return last.data;
    }

    /**
     * Returns the element at the specified position in this list.
     * Returns <code>null</code> if <code>index</code> is out of bounds.
     */
    public T get(int index) {

        ListElement<T> current;

        current = first;

        if(index >= size)
        {
            //not allowed. fail.
            return null;
        }

        for (int i = 0; i < index; i++) {
            current = current.next;
        }

        return current.data;
    }

    /**
     * Removes and returns the first element from this list.
     * Returns <code>null</code> if the list is empty.
     */
    public T removeFirst() {
        // TODO
        if(size == 0 || first == null)
            return null;

        ListElement<T> temp = first;

        first = first.next;
        size--;

        return temp.data;
    }

    /**
     * Removes all of the elements from this list.
     */
    public void clear() {
        last = new ListElement<T>(null);
        first = last;
        size = 0;
    }

    /**
     * Returns the number of elements in this list.
     */
    public int size() {
        // TODO
        return size;
    }

    /**
     * Returns <code>true</code> if this list contains no elements.
     */
    public boolean isEmpty() {
        if(size == 0)
            return true;
        else
            return false;
    }

    /**
     * Returns a string representation of this list. The string
     * representation consists of a list of the elements enclosed in
     * square brackets ("[]"). Adjacent elements are separated by the
     * characters ", " (comma and space). Elements are converted to
     * strings by the method toString() inherited from Object.
     */
    public String toString() {

        String out = "[";

        ListElement<T> current;

        current = first;

        if(current == null || current.data == null)
            return "[]";

        for (int i = 0; i < size-1; i++) {
            out += current.data.toString() + ", ";
            current = current.next;

        }
        out += current.data.toString() + "]";

        return out;

    }
}
```

LinkedListTest class

```
package inda2v;

import static org.junit.Assert.*;
import inda2v.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class LinkedListTest {


    LinkedList<Object> newList;

    @Before
    public void Initialization() {
        newList = new LinkedList<Object>();
    }

    @Test
    public void LinkedListTest() {
        //test for the constructor

        LinkedList<Object> aList = new LinkedList<Object>();

        //relies on that we declared size() correctly.
        assertEquals(0, aList.size());

        assertEquals(null, aList.getFirst());
        assertEquals(null, aList.getLast());
    }

    @Test
    public void AddFirstTest()  {
        //
        int currentSize = newList.size();

        Object cat = ”A Cat stored as an object”;
        newList.addFirst(cat);

        assertEquals(currentSize +1, newList.size());
        assertEquals(cat , newList.getFirst());


        // some additional tests for getFirst and getLast
        LinkedList<String> bList = new LinkedList<String >();

        bList.addFirst(”hi”);
        bList.addFirst(”there”);
        bList.addFirst(”test”);

        assertEquals(”hi”, bList.getLast());
        assertEquals(”test”, bList.getFirst());

        assertTrue(bList.isHealthy());

    }

    @Test
    public void AddLastTest() {
        int currentSize = newList.size();

        Object cat = ”The last cat stored as an object”;

        newList.addLast(cat);

        assertEquals(currentSize +1, newList.size());
        assertEquals(cat, newList.getLast());


        // some additional tests for getFirst and getLast
        LinkedList<String> bList = new LinkedList<String >();

        bList.addFirst(”hi”);
        bList.addLast(”matrix”);
        bList.addLast(”determinant”);

        assertEquals(”determinant”, bList.getLast());
        assertEquals(”hi”, bList.getFirst());

        assertTrue(bList.isHealthy());
        assertTrue(newList.isHealthy());

    }

    @Test
    public void getTest()
    {

        //assertTrue(newList.isHealthy());

        Object first = newList.getFirst();

        Object atIndexTwo = newList.get(2);

        assertEquals(first, newList.getFirst()); // pass -> nothing weird occurred because of change in reference during search.

        Object objBefore  = "The cat gets into a List.";
        Object objAfter = "The cat survived.";


        newList.addFirst(objBefore);
        newList.addLast(objAfter);

        assertEquals(objBefore, newList.get(0));
        assertEquals(objAfter, newList.get(1));

        LinkedList<String> ls = new LinkedList<String>();
        ls.addFirst("hello");
        ls.addFirst("hi");
        ls.addLast("see you");
        ls.addLast("ciao");


    }

    @Test
    public void toStringTest() {

        LinkedList<String> test = new LinkedList<String>();

        test.addFirst("hi");
        test.addFirst("there");
        test.addFirst("test");


        assertEquals("[test, there, hi]", test.toString());
    }

    @Test
    public void emptyArrayTest() {
        LinkedList<String> test = new LinkedList<String>();

        assertTrue(test.isEmpty());

        test.addFirst("hi");

        assertEquals("hi", test.getFirst());
        assertEquals("hi", test.getLast());

    }


    @After
    public void removeFirstTest()
    {
        Object firstItem = newList.getFirst();
        int currentSize = newList.size();

        Object removedItem = newList.removeFirst();

        if(newList.size() != 0)
        {
            assertTrue(newList.isHealthy());
            assertEquals(currentSize-1, newList.size());
            assertEquals(firstItem , removedItem);
            assertTrue(newList.isHealthy());
        }

    }

    @After
    public void ClearingTest()
    {
        newList.clear();
        assertTrue(newList.isEmpty());
    }



}
```