

CS230 Data Structures

The Java `LinkedList` class

The Java language provides a built-in class for creating and manipulating a linked list of objects, called `LinkedList`. Unlike the linked lists that you worked with in CS111, the Java `LinkedList` is implemented as a doubly-linked list of nodes that each contain data of type `Object`. Each node of the list contains three parts: the data, a pointer to the next node in the list, and a pointer to the previous node in the list. The list is also circular, so that the last node of the list contains a pointer to the first node of the list, and the first node contains a pointer to the last node. In class, we will illustrate the structure of a Java `LinkedList`. This handout describes some of the built-in methods defined for the `LinkedList` class, and provides some code examples that demonstrate their use. More information can be found at the Sun Java website: <http://java.sun.com/j2se/1.5/docs/api/>

The `LinkedList` class is defined in the `java.util` package. To use the `LinkedList` class, the following `import` statement must appear at the top of the code file:

```
import java.util.*;
```

Constructing a new list

An initial empty list can be created using the no-input `LinkedList` constructor:

```
public LinkedList ()
```

Adding an Object to a list

Every element of a `LinkedList` must be of type `Object`, which means that it can be an instance of any built-in Java class, or an instance of a new class that you define. It cannot be a simple type, such as `int` or `char` (although these simple types could be "wrapped" in a Java class such as `Integer` or `Character`).

There are four ways to add a new node to a list, using the methods `add()`, `addFirst()` and `addLast()`. All of these methods have an input that is the `Object` that will be stored in the data part of the new node.

```
public void add (int index, Object o)
```

```
public void addFirst (Object o)
```

```
public void addLast (Object o)
```

```
public boolean add (Object o)
```

Nodes of a `LinkedList` can be referenced by an index that specifies their position in the list. Similar to an array, indices begin with 0 for the first node. The two-input `add()` method has an input `index`, and inserts the new node at the specified index. As a result of this insertion, the indices of later nodes in the list are increased by one.

The single-input `add()` method and the `addLast()` method insert the new node at the end of the list, while `addFirst()` adds the node at the beginning of the list.

Accessing the Object stored in a node of a list

There are three methods available to access the contents of a node in a `LinkedList`:

```
public Object getFirst ()
public Object getLast ()
public Object get (int index)
```

The `getFirst()` and `getLast()` methods return the `Object` stored in the first and last node of the list, respectively, while the `get()` method returns the `Object` stored in the node located at a particular index. Because these methods return the contents of a node as type `Object`, this value may need to be *cast* to a more specific class.

Removing a node from a list

There are three methods that can be used to remove a node from a `LinkedList`:

```
public boolean remove (Object o)
public Object remove (int index)
public Object removeFirst ()
public Object removeLast ()
```

The first version of the `remove()` method removes the first occurrence of a node in the list that contains the specified `Object`. If such a node is found and removed successfully, this method returns `true`. If the input `Object` is not contained in the list, this method returns `false`. The second version of the `remove()` method removes the node that is located at the specified index, and returns the `Object` that is contained in this node. The `removeFirst()` and `removeLast()` methods remove the first and last node of a list, respectively, and return the `Object` contained in this node. When a node is removed from a list, the indices of later nodes of the list are decreased by one.

Changing the contents of a node in a list

The `Object` contained in a particular node of the list can be changed to a different `Object` using the `set()` method, whose inputs include the new `Object` and the index of the node to be changed:

```
public Object set (int index, Object o)
```

The new `Object` is also returned by this method.

Additional useful methods

The following methods will also come in handy when working with the `LinkedList` class:

```
public int size ()
```

```
public void clear ()
```

```
public boolean isEmpty ()
```

The `size()` method returns the number of nodes in the list. The `clear()` method removes all of the nodes of the list. The `isEmpty()` method returns `true` if there are no nodes in the list, and `false` otherwise.

Finally, the built-in `toString()` method returns a `String` that contains the contents of a list, surrounded by brackets and with the contents of each node separated by commas. Consider the following code statements:

```
LinkedList L = new LinkedList();  
L.add("one");  
L.add("two");  
L.add("three");  
System.out.println("contents of L: " + L);
```

The execution of these statements results in the following printout:

```
contents of L: [one, two, three]
```

When the final `println()` statement was executed, Java automatically invoked the method `L.toString()`, which returned the string "[one, two, three]".

Casting list elements

As you can see from the definitions of the `LinkedList` methods, a linked list contains and returns `Objects`. Therefore, to use the elements of a list you have to cast them correctly after you extract them from the list. Consider the following code fragment:

```
LinkedList L3 = new LinkedList();  
L3.add(new Country("chile"));  
L3.add(new Country("chile"));  
L3.add(new Country("peru"));
```

Even though you inserted `Countries`, you extract `Objects`. The following is the right way to compare the elements of a list to a given `Country` while searching for it:

```
chile = new Country("chile");  
for (i = 0; i < L3.size(); i++)  
    if (((Country)L3.get(i)).equals(chile))  
        System.out.println("chile found at index " + i);
```

A common mistake is to forget the `(Country)` casting. Java quietly will fail to find the element – it will not complain for type mismatch, it will just not find it.