Lecture 8: Implementing LinkLists

Contents

Motivating Question

How can we build Java-style LinkedLists whose contents can be mutated?

Objectives

By the end of this lecture, you will know:

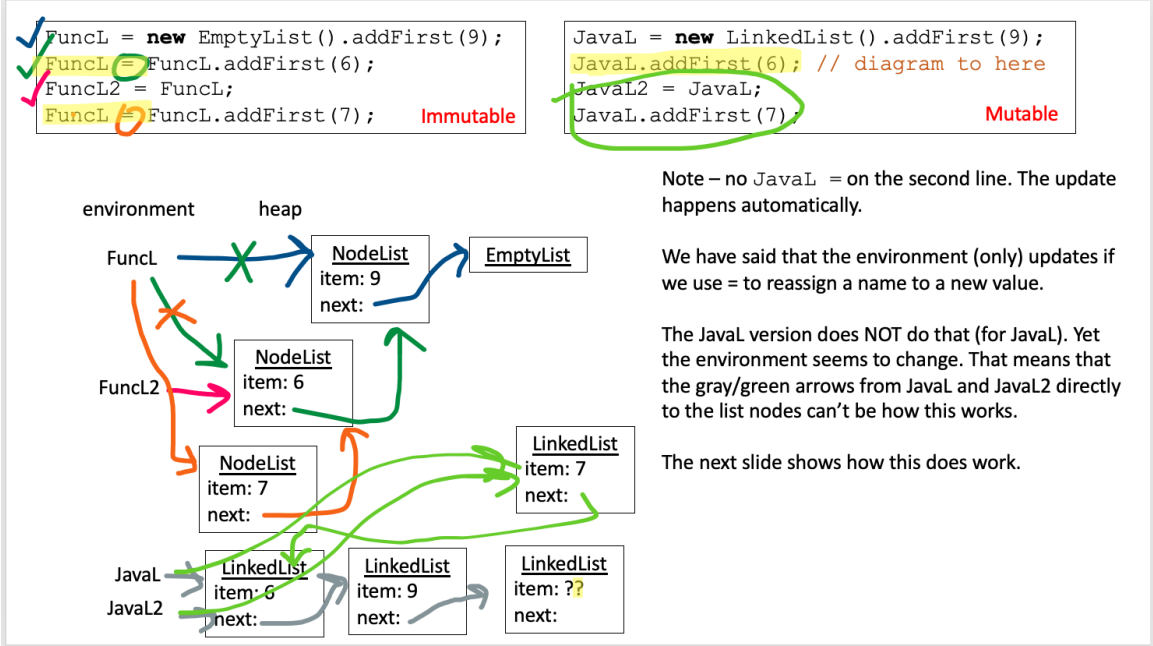
- How functional and mutating implementations of LinkedLists differ from one another under the hood

By the end of this lecture, you will be able to:

- add elements to the front of a mutable list

1 How Do Mutable Lists Work?

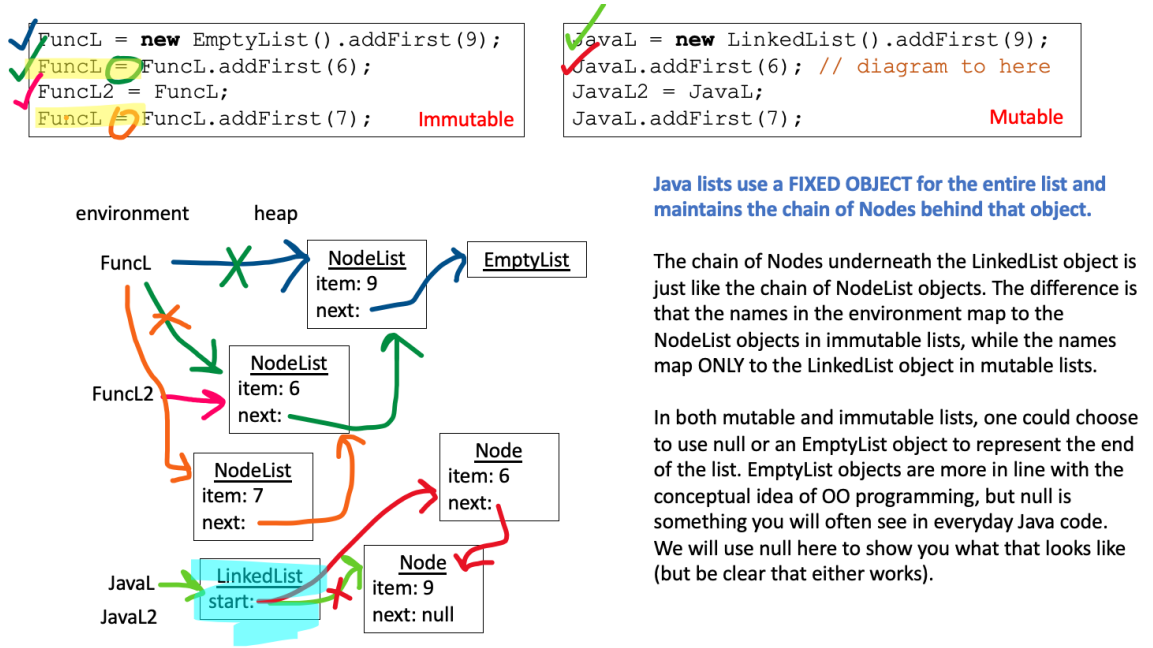
We began lecture reviewing the memory diagram for an immutable (functional) list, and for what we'd like to see happen for an immutable list. The video walks through the development, but here's the image of the first slide we developed while doing this.



At the end of this slide, we concluded that this picture can't be how an immutable list works, because

- the name in the environment needs to be redirected to a new object in the heap, but
- the only way to redirect a name is to use `name = ...`, which is not in the code sample that works in Java

Here's the second slide that we developed to show how this does work.



The key is the `LinkedList` object that stays stable as elements are added to the chain of `Node` objects behind the scenes. The chain of node objects works exactly as it does in the immutable list. The difference is all in having a single fixed object between the name and the node chain.

(You can implement immutable lists with the fixed object as well. The point here is that you CANNOT implement mutable lists without the fixed object.)

The rest of these notes look at how to build all of this in code.

2 Implementing Mutable Lists

2.1 Creating a LinkList class

We started by creating a class `LinkList` (as opposed to `LinkedList` as shown in the memory diagram. We changed the name to avoid clashes with Java's built-in classes. Here's the initial class, with a field `start` to refer to the start of the list.

```
public class LinkList implements IList {
    _______ start;  // the actual list underneath
    LinkList(){}
}
```

What type might we insert into the blank space as the type for `start`? Here are several possibilities:

- `int`
- `Object`
- `IList`
- `AbsList`
- `EmptyList`
- `NodeList`

The `start` of the list should be an actual list, not the contents, which rules out `int`. This means it has to be a type that allows both `EmptyList` and `NodeList`. Each of `Object`, `AbsList`, and `IList` would do that. That said, `IList` is the best choice. `Object` is too general, and we should generally use interfaces as types when they are available (we'll see why that's a particularly good decision in this case in the next lecture).

Our class now looks like:

```
public class LinkList {
    IList start;  // the actual list underneath
    LinkList(){}
}
```

Is there any work for the constructor to do? Well, when we make a new list, it should start out as the empty list. There are two ways we could set that up: as part of the field or as part of the constructor:

```
public class LinkList {
    IList start;  // the actual list underneath
    LinkList(){
      this.start = null;
    }
}
```

```
public class LinkList {
    IList start = null;  // the actual list underneath
    LinkList(){}
}
```

Both approaches work, though the latter is more conventional since it doesn't depend on any input when the list is created.

As we saw in the slide example, we will also need a class for nodes:

```
public class Node {
    Integer item;
    Node next;

    public Node(Integer item, Node next) {
        this.item = item;
        this.next = next;
    }
}
```

2.2 Implementing AddFirst

Now, let's write the `addFirst` method in `LinkList`. Based on the picture, what needs to happen to add a node to the front of the list?

- We need a new node object for the new element
- The next reference in that node has to be the current first node (stored in `LinkList.start`)
- The `LinkList.start` field has to be redirected to the new Node.

Stop and Try It! Try to write this method for yourself, following these steps.

Here's the actual code:

```
public void addFirst(Integer newElt) {
   // mimic green version to red version on slides
   Node n = new Node(newElt, this.start);
   this.start = n;
   // mutable list COULD return this as a result
}
```

3 Implementing Contains

Now, let's write a method to check whether a specific item is in the list.

```
public boolean contains(Integer find) {
  ...
}
```

How might we do this? There are two approaches:

- Write a recursive function in the `Node` class and call that from the `LinkList.contains` method.
- Use a loop to implement `contains`.

Since you already have practice with recursive functions, let's do this with a loop (which will reinforce this week's lab):

```
public boolean contains(Integer find) {
   Node current = this.start;
   while (current != null) { // not at end of the list
       if (current.item.equals(find)) {
           return true;
       } else {
           current = current.next;
       }
   }
   // come here when reach null
   return false;
}
```

Study Question: Why did we do this with a `while` loop instead of a `for` loop?