

CS200: Implementing Immutable Lists

Kathi Fisler

February 9, 2022

Motivating Question

How do we build immutable lists through Java classes and objects?

1 Mutable vs Immutable Lists

Everyone has now worked with both mutable and immutable lists. Let's step back and contrast their behaviors:

```
1 // pyret/immutable
2 Limm = link(3, link(9, empty))
3 Limm2 = link(8, Limm)
4
5 // java/mutable
6 Lmut = new LinkedList<Integer>();
7 Lmut.addFirst(9);
8 Lmut.addFirst(3);
9 Lmut.addFirst(8);
```

As a reminder, with the immutable version, `Limm2` refers to a new list and `Limm` is unchanged by the `link` that creates `Limm2`. In the mutable version, we can't even define `Lmut2` since `addFirst` doesn't return a list; furthermore, every addition modifies the list.

1.1 Why Does this Difference Matter in Practice?

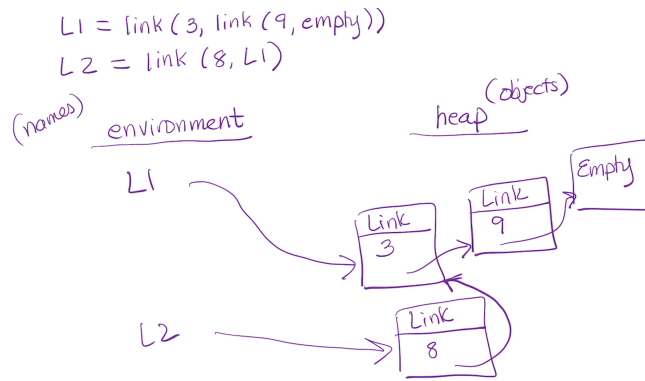
Mutable lists are good when you want to make a permanent change to data. Immutable lists are good when you want to make a temporary change, such as when you are exploring a possible scenario. We'll see both arise this semester.

2 The Layout of Immutable Lists in Memory

The next page has a picture of what the immutable list looks like in memory. The environment stores the variable names in the program. The heap stores the objects.

What classes do you need to capture this picture? We have two classes, `Empty` and `Link`. The `Link` class has two fields, corresponding to the `first` and `rest` components of the list.

```
1 public interface IList { }
2
3 public class Empty implements IList {
4     public Empty() {}
5 }
6
7 public class Link implements IList {
```



```

8  int first;
9  IList rest;
10
11 public Link(int first, IList rest) {
12     this.first = first;
13     this.rest = rest;
14 }
15 }

```

The `IList` interface creates a type name that represents “Empty or Link”. Without that, we can’t put a type on the `rest` field that allows the list to end (at `Empty`).

3 Core Methods on Lists

Let’s expand our interface to require some methods:

```

1 public interface IList {
2     public boolean isEmpty();
3     public Link addFirst(int newItem);
4     public int size();
5 }

```

Here are the `Empty` and `Link` classes with these methods implemented.

```

1 public class Empty implements IList {
2     public Empty() {}
3
4     public boolean isEmpty() {return true; }
5     public int size() { return 0; }
6     public Link addFirst(int newItem) { return new Link(newItem, this);}
7 }
8
9 public class Link implements IList {
10     int first;
11     IList rest;
12
13     public Link(int first, IList rest) {
14         this.first = first;
15         this.rest = rest;
16     }
17
18     public boolean isEmpty() {return false;}

```

```

19     public int size() { return 1 + this.rest.size(); }
20     public Link addFirst(int newItem) { return new Link(newItem, this); }
21 }

```

We can see these methods are small functions, with `size` being recursive.

We also observe that the two `addFirst` methods are identical. This suggests that we also need an abstract class:

```

1     public abstract class AbsLinkedList implements IList {
2         public Link addFirst(int newItem) { return new Link(newItem, this); }
3     }
4
5     public class Empty extends AbsLinkedList { ... }
6
7     public class Link extends AbsLinkedList { ... }

```

Do we still need the `IList` interface once we have the abstract class? Yes. We may want different implementations of `IList` (we'll do another one in a couple of days). The interface lets these different versions provide methods with the same name, but different implementation approaches under the hood.

But why are we even doing this when Java has `LinkedList` built in? Java has mutable lists built in. Here, we are building immutable lists. In addition, there are educational benefits to knowing how lists are implemented, as there will be ideas here that you will use in later projects.