# Lecture 5: Implementing Lists, Version 1

## Contents

### Motivating Question

How can we build lists in Java?

### Objectives

By the end of this lecture, you will know:

- How to implement lists using classes
- How to trade space for time in implementing a recursive method
- How to do work other than initializing fields in constructors

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

- Use classes to implement recursively-defined data structures
- Terminate a program in an error situation

## 1 Implementing Lists

Now that we have covered the basics of OO, let's start applying OO to implement familiar data structures. Today, we start by implementing lists. We'll implement lists of numbers for now, for simplicity.

In CS17/19/111, you saw a definition of lists along the lines of the following:

```
type List:
 | empty
 | cons/link(int, List)
```

Using this as a guide, we want to implement lists in Java.

> *Give it a try – can you turn this definition into a collection of classes, interfaces, abstract classes, etc.* **Note:** if you have had some Java before, use only features and constructs that we have covered in the course so far.

Note the similarity between this and our previous setup of Boas and Dillos as Animals: there, we needed an interface for the shared type name, and a class for each variant. Following that, here's what you should have in Java (perhaps with different class/interface/variable names):

```
public interface IList {}

public class EmptyList implements IList {
  public EmptyList() {}
}

public class NodeList implements IList {
  public int first;
  public IList rest;

  public NodeList(int fst, IList rst) {
    this.first = fst;
    this.rest = rst;
  }
}
```

**Side note for those with prior Java:** Likely, many of you wanted to use `null` in your solution. Hold that thought. It actually isn't good OO practice, even though it is common practice in Java. We'll discuss the null-based solution explicitly next week (and explain what's wrong with it from an OO perspective).

How could we create a list of three numbers with these classes?

```
  new NodeList(3, new NodeList(6, new NodeList(1, new EmptyList())))
```

With a class hierarchy and a sample of data in hand, we can turn to writing methods on our list class.

## 2 Methods

Here's the `IList` interface that we want to implement in this lecture. (Note: I'm intentionally not using Javadoc to make the notes more compact – the posted code uses Javadoc instead, as you should when writing programs for homework, etc)

```
public interface IList {
  public boolean isEmpty();          // is the list empty?
  public IList addFirst(int elt);    // add element to front of the list
  public IList remEltOnce(int elt);  // remove first occurrence of elt
  public int head();                 // get the head of the list
  public IList tail();               // get the tail of the list
  public int size();                 // the number of elements in the list
}
```

We'll work on these in order.

### 2.1 isEmpty

This one is straightforward: each list class returns a boolean constant indicating whether or not it is empty. For example:

```
public class EmptyList implements IList {

  public boolean isEmpty() {
    return true;
  }
}
```

## 2.2 addFirst

Here, we create a new `NodeList` with the given element as the first item and `this` as the rest of the list.

```
public class EmptyList implements IList {

  public IList addFirst(int elt) {
    return new NodeList(elt, this);
  }
}
```

Note that this code is identical between the two list classes, so arguably you would make an abstract class to share this code. We skipped this in lecture so we could get through the other methods, but it is shown in the posted code.

It would also be reasonable to have the return type of `addFirst` be `NodeList`. You'd have to change the interface and the methods to use that return type.

Now that we have `addFirst`, we can rewrite our example list as follows:

```
  new EmptyList().addFirst(1).addFirst(6).addFirst(3)
```

Note that the elements appear to be written in the opposite order from our original example. You should convince yourself that these two expressions will produce lists with the same elements in the same order.

## 2.3 remEltOnce

In the `EmptyList` case, there's nothing to remove, so the method just returns `this`. In the `NodeList` case, we check whether the node contains the element, then either remove it or pass the computation on to the rest of the list to perform:

```
public class NodeList implements IList {

  public IList remEltOnce(int remelt) {
    if (remelt == this.first) {
      return this.rest;
    } else {
      return new NodeList(this.first, this.rest.remEltOnce(remelt));
    }
  }
|
```

Note that this is the same recursive solution that you would have written in CS17. We don't lose recursion as a technique just because we have switched to Java.

### 2.4 head and tail

In `NodeList`, these methods simply return the values of their corresponding fields:

```
public class NodeList implements IList {

  public int head() {
    return this.first;
  }

  public IList tail() {
    return this.rest;
  }
}
```

What happens in `EmptyList`, however? There are no first or rest components to return, so what should happen? Both of these methods should raise errors, since it isn't meaningful to break down an empty list:

```
public class EmptyList implements IList {

  public int head() {
    throw new RuntimeException("Attempt to take head of empty list");
  }

  public IList tail() {
    throw new RuntimeException("Attempt to take tail of empty list");
  }
}
```

In Java, `RuntimeExceptions` are for errors that you can't really recover from gracefully (like division by 0). They result in the program stopping completely. Later in the course, we will see more sophisticated situations of handling errors. For now, these are the closest Java analog to the simple error raising that you did in CS17/111/19.

### 2.5 size

Finally, we turn to a method that returns the size of the list. You've written recursive length functions many times in CS17, and we could certainly do the same here (following a pattern similar to that we used for `remEltOnce`). Let's start with that version:

```
public class EmptyList implements IList {
  public int size() {
    return 0;
  }
}
```

```
public class NodeList implements IList {
  public int first;
  public IList rest;

  public int size() {
    return 1 + this.rest.size();
  }
}
```

This version works fine, but it does traverse the entire list every time someone calls the `size` method. This raises a question – could we somehow do that computation just once, save the computed result, and make the method take less time?

To save the result, we first add a field to the `NodeList` class to store the computed result. Our goal is to just return that value when someone calls the `size` method.

```
public class NodeList implements IList {
  public int first;
  public IList rest;
  public int eltCount;

  public int size() {
    return this.eltCount;
  }
}
```

Where do actually do that computation, though? Think about the constructor: how should the constructor set up the value of `eltCount`? As a reminder, here is the constructor before we add the `eltCount` field:

```
public class NodeList implements IList {
  public int first;
  public IList rest;

  public NodeList(int fst, IList rst) {
    this.first = fst;
    this.rest = rst;
  }
}
```

Should we just add a parameter to the constructor that asks for the size? That would be inconvenient for the programmer who uses our list classes. For example, the list-example code we wrote before would now have to change to

```
new NodeList(3, new NodeList(6, new NodeList(1, new EmptyList(), 1),  2), 3)
```

(where the "1), 2), 3" at the end of the line adds the sizes). Our definition of the `addFirst` method would also be affected. The `eltCount` field is our own implementation detail to make the `size` method more efficient, so it should not be visible to the user of our code.

### 2.5.1 Adding Computation to Constructors

Constructors are just methods that are used to create objects. Like all methods, they can do arbitrary computation to complete their tasks. In this case, we want the constructor to *compute* the size when we create the list, then store that value in the field. Here's the code:

```
public class NodeList implements IList {
  public int first;
  public IList rest;
  public int eltCount;

 public NodeList(int fst, IList rst) {
    this.first = fst;
    this.rest = rst;
    this.eltCount = 1 + rst.size();
  }
}
```

For the empty list, we can simply return 0 as before (there's no expensive computation to save). We could also put a similar field in the `EmptyList` class, though it would take a bit of space that isn't otherwise necessary.

## 3 Creating an Abstract Class

If we put the `eltCount` field in both classes, we hae another bit of shared code between the `EmptyList` and `NodeList` classes. This definitely points to an abstract class to share these details. Trying to write it for yourself is a terrific exercise:

> *Try creating this abstract class (write it on paper) before looking at the posted code for today's lecture (which shows the solution)*

If you look at the posted code, you'll see that the `implements IList` annotation has also elevated to the abstract class. Since the interface is common to all classes that extend `AbsList`, it makes sense to annotate the abstract class instead.

## 4 Printing Lists: toString methods

One last tidbit, which is not about list classes in particular, but does show you a practical aspect of working in Java.

Sometimes, we just want to print out the result of a computation withut going through a test case. For example, we might want to have the following code in the `main` method of our `TestList` class:

```
public class ListTest {

  public static void main(String[] args) {
     IList list1 = (new EmptyList()).addFirst(3).addFirst(6).addFirst(1);
```

```
    System.out.println(list1);
  }
}
```

If we do this, we see something like

```
lec05.NodeList@15db9742
```

What is this? Remember our memory maps? This is saying that there's an object at memory address `15db9742` that was created from the `NodeList` class in the `lec05` package.

Not very useful, right?

In Java, when you define classes, you also want to tell Java what to print out about objects in the class when needed. We do this by putting a method named `toString` in the `EmptyList` and `NodeList` classes.

```
public class EmptyList implements IList {

  @Override
  public String toString() {
    return "empty";
  }
}

public class NodeList implements IList {

  @Override
  public String toString() {
    return this.first + ", " + this.rest.toString();
  }
}
```

With these, our `main` method snippet now displays

```
1, 6, 3, empty
```

You don't need to mention `toString` in the interface: every Java class gets a default `toString` method that you can override with a custom defintion, as we did here.

The `@Override` annotation indicates that we are providing a refined definition of a method that otherwise already exists in the class. Nothing goes wrong if you leave it off, but it is good practice to include it.

---

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.