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 functional-style lists using classes

1 Extending our Set of Animals

As a warm-up exercise, we decided to add Fish, Sharks, and Fruit flies to the set of animals for our zoo. This illustrates the differences between interfaces and abstract classes, and shows another example of class extension.

2 Example: Adding FruitFlies

What if I wanted to expand my zoo to include Fruit Flies? Fruit flies are so small that we don't worry about tracking their length – they will always be considered normal size. If I make a Fruit fly class extend SizedAnimal, then I end up tracking a field (`length`) that I don't need. But if I have that class extend the `IAnimal` interface, I can still put FruitFlies in my zoo. Specifically:

```
public class FruitFly implements IAnimal {
 public FruitFly() {}

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

3 Class Extension without Abstraction

Abstract classes support field and method abstraction, but class extensions are also used to express hierarchy among data. For example, let's add two kinds of animals to our class hierarchy: Fish, which have a length and an optimal saline level for water in their tanks; and Sharks, which are fish for which we also record the number of teeth that they have. The new classes appear as follows:

```
public class Fish extends SizedAnimal {
 double salinity;

 public Fish (int length, double salinity) {
   super(length);
   this.salinity = salinity;
 }

 /**
  * check whether fish's length is considered normal
  */
 public boolean isNormalSize () {
   return isLenWithin(3, 15);
 }
}

public class Shark extends Fish {
 int teeth;

 public Shark (int length, int teeth) {
   super(length, 3.75);
   this.teeth = teeth;
 }
}
```

A few things to note here:

- The salinity data has type `double`; this is a common type to use for real numbers.
- `Shark` extends `Fish`, but `Fish` is not an abstract class. It still makes sense to create `Fish` that are not also `Sharks`.
- Constructors do not need to take all of the initial field data as parameters. For example, if we assume that all sharks have the same saline level, then the `Shark` constructor asks for only the length and number of attacks; it provides the fixed saline level to the `Fish` constructor on the call to `super`.

- `Shark` does not need to define `isLenWithin`, since it inherits the definition from `Fish`. If you wanted `Shark` to have its own definition (since it might have a different normal size), you could provide one in the `Shark` class. Java calls the most specific method for each object.

4 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.

5 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)

```
public interface IList {
  public boolean isEmpty();          // is the list empty?
  public IList addFirst(int elt);    // add element to front of the list
  public IList delete(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.

5.1 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;
  }
}
```

5.2 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.

5.3 delete

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 delete(int remelt) {
    if (remelt == this.first) {
      return this.rest;
    } else {
      return new NodeList(this.first, this.rest.delete(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.

5.4 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.

5.5 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.

```
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();
  }
}
```

Question: 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? We'll come back to that another day.

6 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`.