CS 200 fall 2022 Lecture 3 (from OO): recursion

A way to define lists using records

How do we define a list using records?

A list: A) has an arbitrary number of elements (possibly 0); B) typically has elements all of the same type (e.g. all Strings, all ints…; and C) orders elements sequentially (one after the other).

The following won't work, because it violates principles A and B (in this example, we can only hold up to three elements in this list)

```
public record SimpleList(String elt1, String elt2, String elt3) { }
```

To allow for an arbitrary number of elements, we will instead use this definition, which will allow us to chain an arbitrary number of SimpleLists (if you've seen how Linked Lists work before, this is a similar principle):

```
public record SimpleList<T>(T first, SimpleList<T> rest) { }
```

The way we would instantiate a list with the elements ["sponge", "squid", "starfish"] using this definition would be:

```
SimpleList<String> strLst =
       new SimpleList<String>("sponge",
               new SimpleList<String>("squid",
                       new SimpleList<String>("starfish", null)));
```

You'll notice that in your lecture handout, we gave you a toString method in the SimpleList record. **You do not have to worry about how this method works, you can just use it to print SimpleLists in a concise way**.

```
public record SimpleList<T>(T first, SimpleList<T> rest) {
   @Override
   public String toString() {
       String builder = "SimpleList[" + first.toString();
       if (rest != null) {
           builder += rest.stringHelper();
       }
       builder += ']';
       return builder;
   }
   private String stringHelper() {
```

```
        String builder = ", " + first.toString();
        if (rest != null) {
            builder += rest.stringHelper();
        }
        return builder;
    }
};
```

This way, we can print SimpleLists, but we can also see that the structure is preserved by calling .first() and .rest():

```
System.out.println(strLst); // prints SimpleList[sponge, squid, starfish]
System.out.println(strLst.first()); // prints sponge
System.out.println(strLst.rest()); // prints SimpleList[squid, starfish]
System.out.println(strLst.rest().first()); // prints squid
```

Writing examples with SimpleLists

To write examples easily, we have given you a toSimpleList (which works similarly to the Arrays.asList that we saw in the previous lecture) to easily initialize SimpleLists without constantly typing “new SimpleList” repeatedly. **Again, you don’t need to know how this method works (it contains syntax you haven’t seen before); you can just use it for convenience.**

```
private static SimpleList toSimpleList(Object... args) {
    SimpleList simpleListBuilder = null;
    for (int i = args.length - 1; i >= 0; i--) {
        simpleListBuilder = new SimpleList<>(args[i], simpleListBuilder);
    }
    return simpleListBuilder;
}
```

In some method of your test file:

```
SimpleList<String> strLst2 = toSimpleList("squirrel", "lobster", "crab");
System.out.println(strLst2); // prints SimpleList[squirrel, lobster, crab]
System.out.println(strLst2.first()); // prints squirrel
```

To write examples on SimpleLists, we do them in a systematic way, by removing one element from the input list each time. For example, if the method doubl in Lec03 should return a list with every element in the input list doubled, we would write:

```
Assert.assertEquals(toSimpleList(0, -6, 8, 16), Lec03.doubl(toSimpleList(0, -3, 4, 8)));

Assert.assertEquals(toSimpleList(   -6, 8, 16), Lec03.doubl(   toSimpleList(-3, 4, 8)));

Assert.assertEquals(toSimpleList(       8, 16), Lec03.doubl(       toSimpleList(4, 8)));

Assert.assertEquals(toSimpleList(          16), Lec03.doubl(          toSimpleList(8)));

Assert.assertEquals(null,                                          Lec03.doubl(null));
```

Note the alignment here – this is done purposefully. If we annotate the pattern we see in the input lists by trying to write out what the input looks like in terms of the input in the line below it, we get:

```
Assert.assertEquals(toSimpleList(0, -6, 8, 16), Lec03.doubl(toSimpleList(0, -3, 4, 8)));
//                                                  SimpleList(0, SimpleList[-3, 4, 8])
Assert.assertEquals(toSimpleList(   -6, 8, 16), Lec03.doubl(   toSimpleList(-3, 4, 8)));
//                                                      SimpleList(-3, SimpleList[4, 8])
Assert.assertEquals(toSimpleList(       8, 16), Lec03.doubl(       toSimpleList(4, 8)));
//                                                          SimpleList(4, SimpleList[8])
Assert.assertEquals(toSimpleList(          16), Lec03.doubl(          toSimpleList(8)));
//                                                                 SimpleList(8, null)
Assert.assertEquals(null,                                          Lec03.doubl(null));
```

We can also do that for the expected lists:

```
Assert.assertEquals(toSimpleList(0, -6, 8, 16), Lec03.doubl(toSimpleList(0, -3, 4, 8)));
//                                                  SimpleList(0, SimpleList[-3, 4, 8])
//        SimpleList(0, SimpleList[-6, 8, 16])
Assert.assertEquals(toSimpleList(   -6, 8, 16), Lec03.doubl(   toSimpleList(-3, 4, 8)));
//                                                      SimpleList(-3, SimpleList[4, 8])
//            SimpleList(-6, SimpleList[8, 16])
Assert.assertEquals(toSimpleList(       8, 16), Lec03.doubl(       toSimpleList(4, 8)));
//                                                          SimpleList(4, SimpleList[8])
//                SimpleList(8, SimpleList[16])
Assert.assertEquals(toSimpleList(          16), Lec03.doubl(          toSimpleList(8)));
//                                                                 SimpleList(8, null)
//                        SimpleList(16, null)
Assert.assertEquals(null,                                          Lec03.doubl(null));
```

But if we look at what the “rest” field of eact *expected* list is, we can do a substitution to the method call below:

```
Assert.assertEquals(toSimpleList(0, -6, 8, 16), Lec03.doubl(toSimpleList(0, -3, 4, 8)));
//                                                SimpleList(0, SimpleList[-3, 4, 8])
// SimpleList(0, Lec03.doubl(SimpleList[-3, 4, 8]))
Assert.assertEquals(toSimpleList(   -6, 8, 16), Lec03.doubl(   toSimpleList(-3, 4, 8)));
//                                                   SimpleList(-3, SimpleList[4, 8])
// SimpleList(3 * 2, Lec03.doubl(SimpleList[4, 8]))
Assert.assertEquals(toSimpleList(       8, 16), Lec03.doubl(       toSimpleList(4, 8)));
//                                                      SimpleList(4, SimpleList[8])
// SimpleList(4 * 2, Lec03.doubl(SimpleList[8]))
Assert.assertEquals(toSimpleList(          16), Lec03.doubl(          toSimpleList(8)));
//                                                         SimpleList(8, null)
// SimpleList(8 * 2, Lec03.doubl(null))
Assert.assertEquals(null,                                   Lec03.doubl(null));
```

Turning the examples into code

This hopefully shows a pattern: it looks like the expected output can be written like

```
SimpleList(input.first() * 2, Lec03.doubl(input.rest())
```

This indicates that we should write the doubl function in Lec03 as follows (remembering to account for the null case).

```
public static SimpleList<Integer> doubl(SimpleList<Integer> lst) {
   if (lst == null) {
       return null;
   } else {
       return new SimpleList<Integer>(lst.first() * 2,
               doubl(lst.rest()));
   }
}
```

What does this look like when the code is called on a SimpleList containing [4,8]? Well, first, we'll call

```
doubl([4, 8])
```

(here we are using [4,8] as shorthand for the SimpleList containing 4 and 8). Since lst is [4,8], it is not null, so we will enter the else block. The code will want to return

```
new SimpleList<Integer>(lst.first() * 2, doubl(lst.rest()));
```

when lst is [4,8], first is 4 and rest is [8] (remember your types here – rest is of type SimpleList, so it is the *SimpleList containing 8*, not just 8!), so the code will actually try to return

```
new SimpleList<Integer>(4 * 2, doubl([8]));
```

In a way, this shows that the code is *delegating* some of the computation to the call doubl([8]). A method that calls itself is a *recursive* method. Usually, recursion works because we make the recursive calls on smaller and smaller inputs, and get down to the *base case* (a case that returns some answer without making a recursive call). The list [8] is, indeed, a smaller input. Let's see how we can get down to the base case. Running doubl[8] will, again, enter the else block, and try to return

```
new SimpleList<Integer>(lst.first() * 2, doubl(lst.rest()));
```

In this case, first is 8 and rest is null, so it will try to return

```
new SimpleList<Integer>(8 * 2, doubl(null));
```

Before it can return this, it needs to compute doubl(null). But doubl(null) is the base case, because the guard (the question being asked) in the if-expression evaluates to true. So, the order in which the code runs is:

1. Call doubl([4,8])
2. Compute that we need to return `new SimpleList<Integer>(4 * 2, doubl([8]));` which we can't do until we compute doubl([8])
3. Call doubl([8])
4. Compute that we need to return `new SimpleList<Integer>(8 * 2, doubl(null));` which we can't do until we compute doubl(null)
5. Call doubl(null)
6. doubl(null) returns null
7. Now, the computation from step 4 has the answer to doubl(null) and can return `new SimpleList<Integer>(16, null)`
8. Now, the computation from step 2 has the answer to doubl([8]) and can return `new SimpleList<Integer>(8, new SimpleList<Integer>(16, null))`

Another example

We can write similar examples to compute the length of a list:

```
public void testLen() {
   // assume we have a static method in Lec03.java called "double" which takes
   // in a SimpleList of integers and returns the length of the list (as an int)

   Assert.assertEquals(4, Lec03.len(toSimpleList("squid", "crab", "whale", "snail")));
   //                       SimpleList("squid", SimpleList["crab", "whale", "snail"])
   //                1 + 3
   //  = 1 + Lec03.len(SimpleList["crab", "whale", "snail"])
   Assert.assertEquals(3, Lec03.len(toSimpleList(         "crab", "whale", "snail")));
   //                                SimpleList("crab", SimpleList["whale", "snail"])
   //                1 + 2
   //  = 1 + Lec03.len(SimpleList["whale", "snail"])
```

```
    Assert.assertEquals(2, Lec03.len(toSimpleList(                "whale", "snail")));
    //                                        SimpleList("whale", SimpleList["snail"])
    //                  1 + 1
    //   = 1 + Lec03.len(SimpleList["snail"])
    Assert.assertEquals(1, Lec03.len(toSimpleList(                       "snail")));
    //                                               SimpleList("snail", null)
    //                  1 + 0
    //   = 1 + Lec03.len(null)
    Assert.assertEquals(0,                                          Lec03.len(null));
}
```

This indicates that the pattern is:

```
1 + Lec03.len(input.rest())
```

And the code we want to write (accounting for the answer of 0 when the input is null) is:

```
public static int len(SimpleList lst) {
    if (lst == null) {
        return 0;
    } else {
        return 1 + len(lst.rest());
    }
}
```