Lecture 7: Mutable Lists, Conceptually

Contents

Motivating Question

How are Java's built-in lists different from the functional lists in Racket, Reason, and Pyret?

Objectives

By the end of this lecture, you will know:

- How to create a list with Java's LinkedList class
- How to traverse a list with a for loop

1 One = Sign, Two Impacts

In the last lecture, we looked at two ways to try to update the room associated with a course. Roughly speaking, we tried the following two approaches:

```
Course cs18 = new Course(''csci'', 18, ''Bert 130'');
// approach #1
cs18 =  new Course(''csci'', 18, ''Sayles'');
// approach #2
cs18.room = ''SAL001'';
```

Under the hood, the first version adds a new `Course` to the heap, then redirects the reference from the name `cs18` in the environment to the new `Course` object. The second version, however, does neither. It simply updates the value in the `room` field of the original `Course` object. In particular, the second version leaves the environment untouched.

How can one use of = change the environment but not the other? What matters is the term on the left side of the =: if the left side of = is just a name (like `cs18`), then the environment updates (recall that the environment tracks how names map to values). If the left side of = is of the form `<object>.<field>`, then the field value changes within the heap. Think of = as saying: update the value associated with the left hand side: fields are stored in the heap, but plain names are stored in the environment.

Why is this distinction important? Because as we saw last lecture, changing the value associated with a name/variable in the environment only affects *future* use of that variable, but changing the value in a field affects all *existing* references to the corresponding object. We'll work more with when to use each kind of update. For now, it is worth bearing in mind that a key question while designing programs is whether an update should affect past or only future references to a variable name.

2 Built-in Lists in Java

Last week, we wrote our own implementation of lists, based on the `cons` and `link` constructs in Racket and Pyret (respectively). Like all programming languages, Java also has lists built-in, but they behave somewhat differently. Let's look at a concrete example of how to create a list containing the numbers 5 and 3 (in that order):

```
import java.util.LinkedList;

public class JListExample {

        JListExample(){}

        public static void main(String[] args) {
                LinkedList<Integer> L = new LinkedList<Integer>();
                L.addFirst(3);
                L.addFirst(5);
                System.out.println(L.toString());

                // compute sum of items in L
                Integer total = 0;  // a variable to hold the running sum
                for (Integer num : L) {
                        total = total + num;
                }
                System.out.println(total);
        }
}
```

Things to notice about the code to set up the list:

- We need to import the `LinkedList` library in order to use lists in Java. (For those with prior Java experience, we are intentionally using `LinkedList` instead of `ArrayList` for now – we will talk about the difference in about a week.)
- When we use the `LinkedList` class, we have to specify the type of the list contents within the angle brackets. Here, we write `LinkedList<Integer>` to indicate a list containing integers. This is a situation when you have to use `Integer` instead of `int`: the type name for list elements has to be the name of a class, abstract class, or interface.
- As in our list implementation, `addFirst` is the method that adds an item to the front of a list.
- Unlike in our list implementation, however, `addFirst` *actually modifies the contents of the list.*

This last point is perhaps the most significant. Whereas in Racket/Reason/Pyret, the original list is kept intact when you add or delete elements, Java modifies the original list. Think of a situation like undoing an edit in a document processor: in one case, both versions of the document would exist, but in the other, there would only be the current version (we'll return to undoing edits at the end of the semester).

2.1 Traversing and Processing Lists in Java

In Java, we don't process built-in lists recursively: Java lists don't have a "rest of list" operation that we would use in a recursive computation (we'll see why in the next lecture). Instead, Java uses a construct called a `for` loop. The code sample above shows how to use a for-loop to compute the sum of a list of numbers.

Things to notice:

- We start by creating a variable to hold the result of the function (in this case, `total`). We initially set the variable to the result we would get on the empty list (the same value we'd return in the empty case of the corresponding recursive function).
- the for-loop construct walks down the list, calling each number in the list `num` in turn (`num` plays a similar role to the `first` of the list in a recursive version). As Java encounters each num, it adds it to the `total`. Java goes through the lines in the body (inside) of the `for` once for every number in the list `L`. The list items are taken in order from the front to the end of the list.
- After the for-loop is done, Java returns the `total` as the result of the function.
- The computation in the non-empty of a recursive version is the same one that we use to update the `total` inside the for loop.

You'll get more practice with loops in this week's lab.

3 Functional vs Java/Python Lists

As we go through CS18, we'll get more practical experience with when to use mutable lists and when not to. For now, we want to think about what these two styles of lists look like in terms of implementation. We've already written our own functional (non-mutating) list classes. What would a version look like that did mutate the list?

We recalled our example from the top of the notes (the two calls to `addFirst`) and sent everyone into breakout rooms to discuss what might be going on at the level of memory to make that work. We jotted down some ideas, but generally left this as a cliffhanger for the next class ...