CS61B Lecture 3

Monday, January 27, 2020

Announcements

- Labs are normally due at midnight Friday. Last week's is due tonight.

Values and Containers

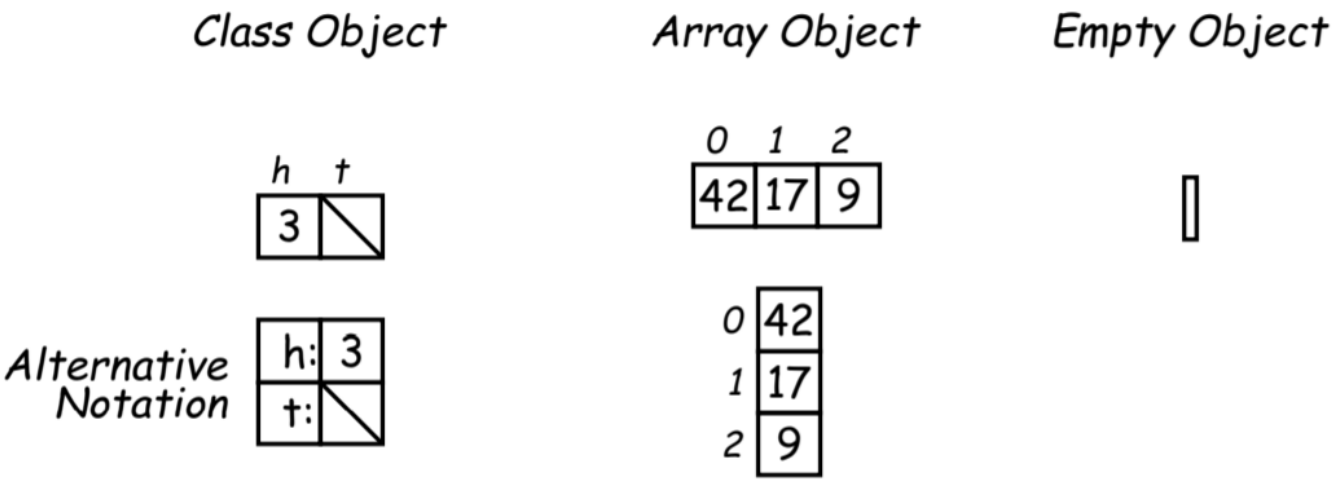
Values are numbers, booleans and pointers. **Values never change.**

Simple containers contain values. Examples include variables, fields, individual array elements, parameters.

Structured containers

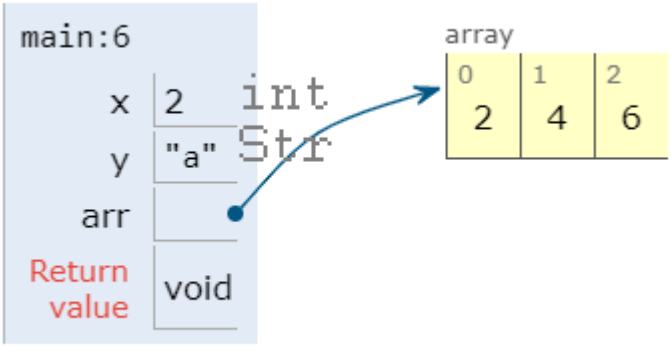
Structured containers contain (0 or more) simple containers:

- Class Object
- Array Object
- Empty Object



In 61B, when you draw environment diagrams, you should indicate what type of object they are.

```
int x = 2;
Str y = a;
int[] arr = new int[]{2,4,6};
```



Pointers

Pointers (or references) are values that reference (point to) containers.

One particular pointer, called null, points to nothing.

In Java, structured containers contain only simple containers, but
pointers allow us to build arbitrarily big or complex structures anyway.

Compared to Python, Java arrays are of a fixed size. Once you define an array of some size, for example 3 in the above code, you cannot add another element to the end.

It would be really nice if we had some data structure that we could keep adding to the end of... turns out we have. Linked lists are the data structure of choice here!

Containers in Java

Containers may be named or anonymous.

In Java, all simple containers are named, all structured containers are anonymous, and pointers point only to structured containers.
(Therefore, structured containers contain only simple containers).

In Java, assignment copies values into simple containers, exactly like Scheme and Python!
• (Python also has slice assignment, as in `x[3:7] =...`, which is shorthand for something else entirely.)

Defining New Types of Object

Class declarations introduce new types of objects. Example: list of integers:

```
public class IntList {
    // Constructor function (used to initialize new object)
    /** List cell containing (HEAD, TAIL). */

    public IntList(int head, IntList tail){
        this.head = head; this.tail = tail;

        //Java uses this instead of self
    }
```

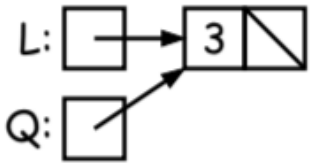
```
    // Names of simple containers (fields)
    // WARNING: public instance variables usually bad style!

    public int head;
    public IntList tail;
}
```

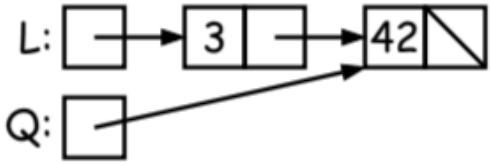
Primitive operations

```
IntList Q, L;
```

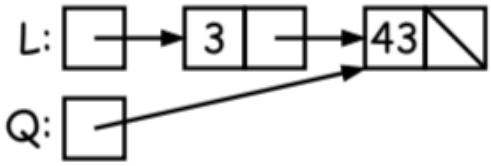

```
L = new IntList(3, null);
Q = L;
```




```
Q = new IntList(42, null);
L.tail = Q;
```

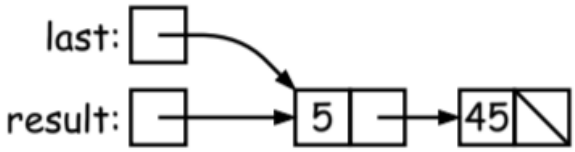



```
L.tail.head += 1;
// Now Q.head == 43
// and L.tail.head == 43
```

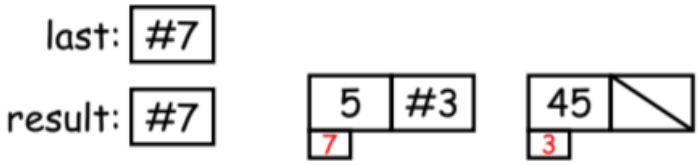


Different ways of viewing pointers

- Some folks find the idea of "copying an arrow" somewhat odd.
- Alternative view: think of a pointer as a *label*, like a street address.
- Each object has a permanent label on it, like the address plaque on a house.
- Then a variable containing a pointer is like a scrap of paper with a street address written on it.
- One view:

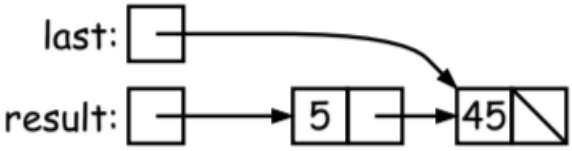


- Alternative view:

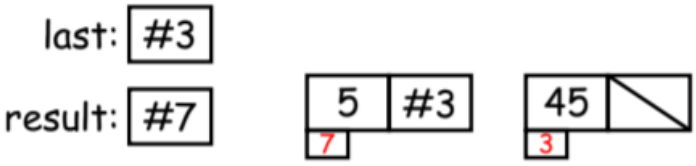


And another:

- Assigning a pointer to a variable looks just like assigning an integer to a variable.
- So, after executing "last = last.tail;" we have



- Alternative view:



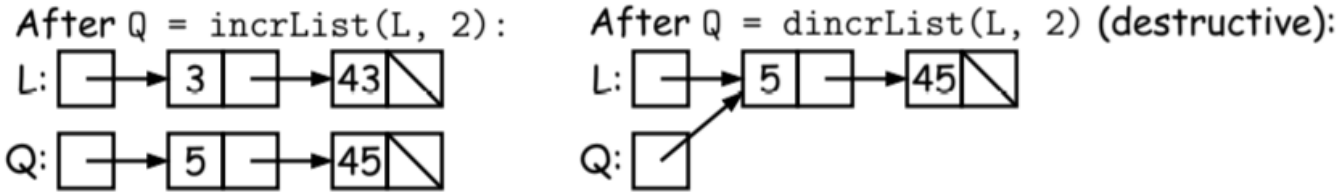
- Under alternative view, you might be less inclined to think that assignment would change object #7 itself, rather than just "last".
- BEWARE! Internally, pointers really are just numbers, but Java treats them as more than that: they have *types*, and you can't just change integers into pointers.

Destructive vs. Non-Destructive Methods

In 61A, we learnt about mutation operations. We now call them destructive and non-destructive methods. A destructive method mutates the original input, while a non-destructive method does not.

For example, Given a (pointer to a) list of integers, L, and an integer increment n, return a list created by incrementing all elements of the list by n.

```
static IntList incrList(IntList P, int n){
    return /*( P, with each element incremented by n )*/
}
```



So how does one actually write `incrList` ? We can use recursion:

```
static IntList incrList(IntList P, int n){
    if (P == null) {
        return null;
    } else {
        return new IntList(P.head+n, incrList(P.tail, n));
    }
}
```

Iterative IncrList

The difference with this version of `incrList` is that it builds the list from front to back. You can see the code is much longer, and it is also not tail recursive.

```
static IntList incrList(IntList P, int n) {
    if (P == null) {
        return null;
    } else {
        IntList result, last;
        //This initializes the variables.

        result = new IntList(P.head+n, null);
        last = result;
    }
    while (P.tail != null) {
        P = P.tail;
        last.tail = new IntList(P.head+n, null);
        last = last.tail;
    }
    return result;
}
```