

CS61B Lecture #4: Values and Containers

- Labs are normally due at midnight Friday. This week, we're not fussy, but do be sure to submit the lab.
- Readings for today: Chapter 4 from *A Java Reference*. See also, *Head First Java*, Chapter 3, Chapter 5.
- Looking ahead: *Head First Java*, Chapters 2 and 4.
- **Today.** Simple classes. Scheme-like lists. Destructive vs. non-destructive operations. Models of memory.
- Project #0 is out (actually, has been since start of class). Due Sept. 27.

Last modified: Wed Sep 11 14:29:11 2013

CS61B: Lecture #4 1

Values and Containers

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

3 'a' true $\frac{1}{\infty}$ \

- *Simple containers* contain values:

x:

3

 L:

/

 p:

→

Examples: variables, fields, individual array elements, parameters.

- *Structured containers* contain (0 or more) other containers:

Class Object

Array Object

Empty Object

h t

3	/
---	---

0 1 2

42	17	9
----	----	---

--

Alternative Notation

h: 3
t:

/

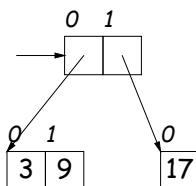
0 42
1 17
2 9

Last modified: Wed Sep 11 14:29:11 2013

CS61B: Lecture #4 2

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.

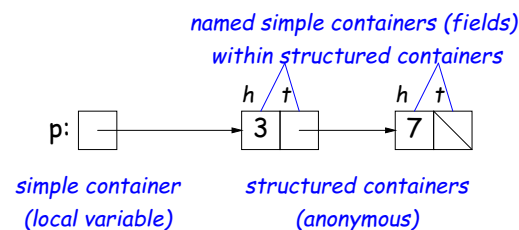


Last modified: Wed Sep 11 14:29:11 2013

CS61B: Lecture #4 3

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!

Last modified: Wed Sep 11 14:29:11 2013

CS61B: Lecture #4 4

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;
    }

    // Names of simple containers (fields)
    public int head;
    public IntList tail;
}
```

Last modified: Wed Sep 11 14:29:11 2013

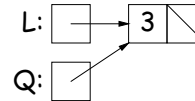
CS61B: Lecture #4 5

Primitive Operations

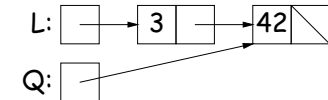
```
IntList Q, L;
```



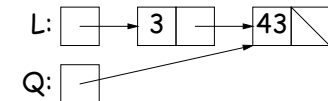
```
L = new IntList(3, null);
Q = L;
```



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



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

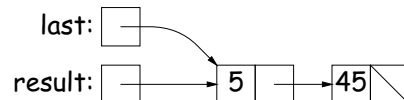


Last modified: Wed Sep 11 14:29:11 2013

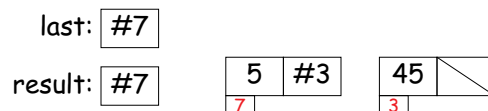
CS61B: Lecture #4 6

Side Excursion: Another Way to View 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:

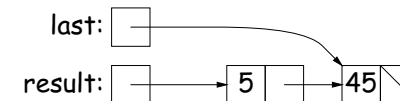


Last modified: Wed Sep 11 14:29:11 2013

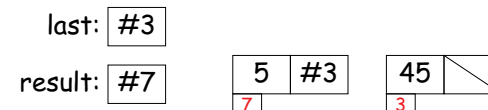
CS61B: Lecture #4 7

Another Way to View Pointers (II)

- 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.

Last modified: Wed Sep 11 14:29:11 2013

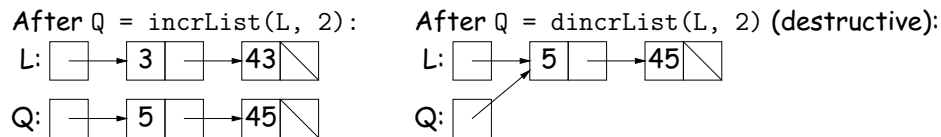
CS61B: Lecture #4 8

Destructive vs. Non-destructive

Problem: 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 .

```
/** List of all items in P incremented by n. Does not modify
 * existing IntLists. */
static IntList incrList(IntList P, int n) {
    return /*( P, with each element incremented by n )*/
}
```

We say `incrList` is *non-destructive*, because it leaves the input objects unchanged, as shown on the left. A *destructive* method may modify the input objects, so that the original data is no longer available, as shown on the right:



Nondestructive IncrList: Recursive

```
/** List of all items in P incremented by n. */
static IntList incrList(IntList P, int n) {
    if (P == null)
        return null;
    else return new IntList(P.head+n, incrList(P.tail, n));
}
```

- Why does `incrList` have to return its result, rather than just setting `P`?
- In the call `incrList(P, 2)`, where `P` contains 3 and 43, which `IntList` object gets created first?

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive. Easier to build things first-to-last, unlike recursive version:

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

