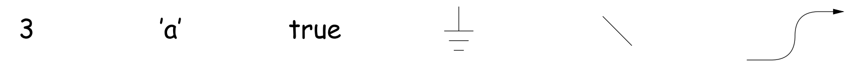
CS61B Lecture #3: Containers

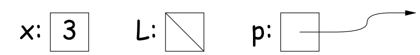
- **Readers Available:** from *Vick Copy* (corner of Euclid and Hearst) *not* Copy Central! Also online.
- Please read Chapter 2 of *Assorted Materials on Java* from the reader.
- **Room change:** Discussion 114 (3-4 Thurs.) is now in 289 Cory (used to be 3111 Etch.)
- **Midterm** is tentatively scheduled for the evening of 9 March (Thursday).
- **Project 1** will be due the preceding week (1 March).
- **Today.** Simple classes. Scheme-like lists. Destructive vs. non-destructive operations. Models of memory.

Values and Containers

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

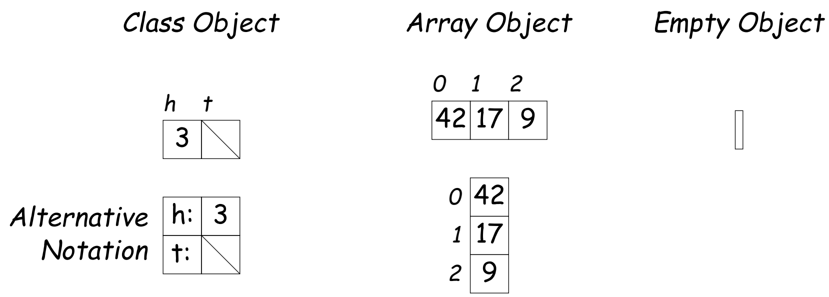


- *Simple containers* contain values:



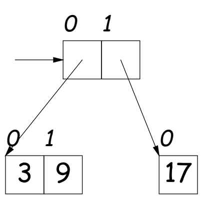
Examples: variables, fields, individual array elements, parameters.

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



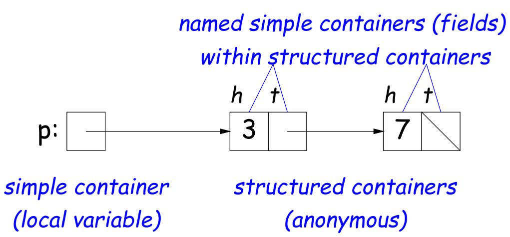
Pointers

- *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.



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!

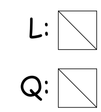
Defining New Types of Object

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

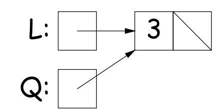
```
public class IntList {
  // Constructor function
  // (used to initialize new object)
  /** List cell containing (HEAD, TAIL). */
  public IntList (int head, IntList tail) {
    this.head = head; this.tail = tail;
  }

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


Primitive Operations

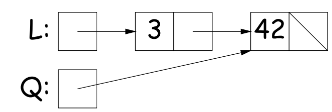
```
IntList Q, L;
```



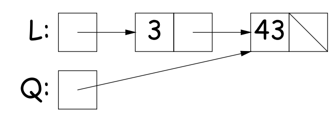

```
L = new IntList(3, null);
Q = L;
```




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




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



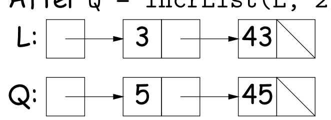
Destructive vs. Non-destructive

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.

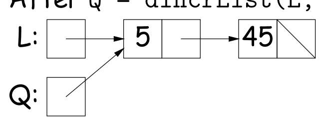
```
/** List of all items in P incremented by n. */
static IntList incrList (IntList P, int n) {
  if (P == null)
    return null;
  else return new IntList (P.head+n, incrList(P.tail, n));
}
```

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:

After `Q = incrList(L, 2)`:



After `Q = dincrList(L, 2)` (destructive):



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;
}
```