Recreation

hat $\lfloor(2+\sqrt{3})^n\rfloor$ is odd for all integer $n \geq 0$.

larsky, N. N. Chentzov, I. M. Yaglom, *The USSR Olympiad Problem* 93), from the W. H. Freeman edition, 1962.]

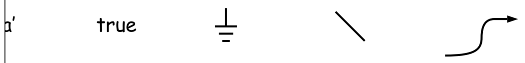
3 Lecture #3: Values and Containers

mally due at midnight Friday. Last week's lab, however, oming Friday at midnight.

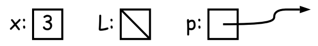
ple classes. Scheme-like lists. Destructive vs. non-operations. Models of memory.

Values and Containers

umbers, booleans, and pointers. Values never change. mple, the assignment 3 = 2 would be invalid.)



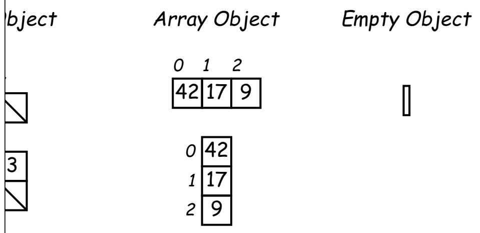
iners contain values:



riables, fields, individual array elements, parameters.
s of containers can change.

Structured Containers

tainers contain (0 or more) other containers:

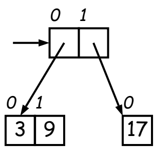


Pointers

references) are values that *reference* (point to) con-

ar pointer, called **null**, points to nothing.

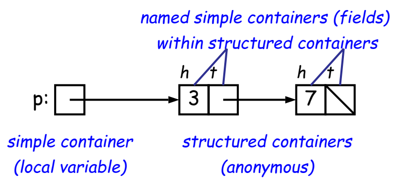
uctured containers contain only simple containers, but w us to build arbitrarily big or complex structures any-



Containers in Java

ay be *named* or *anonymous*.

simple containers are named, *all* structured contain-ymous, and pointers point only to structured containers. structured containers contain only simple containers).



gnment copies values into simple containers.

Scheme and Python!

has slice assignment, as in `x[3:7]=...`, which is short-ething else entirely.)

Defining New Types of Object

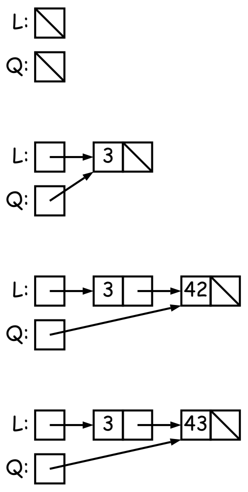
ations introduce new types of objects.

of integers:

```
s IntList {
uctor function (used to initialize new object)
cell containing (HEAD, TAIL). */
tList(int head, IntList tail) {
ad = head; this.tail = tail;


of simple containers (fields)
G: public instance variables usually bad style!
t head;
tList tail;
```

Primitive Operations

```
st(3, null);

st(42, null);

 1;
== 43
head == 43
```



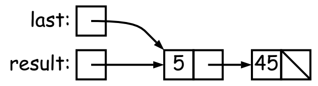
cursion: Another Way to View Pointers

ind the idea of "copying an arrow" somewhat odd.

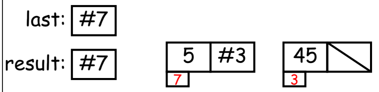
iew: think of a pointer as a *label*, like a street address.

has a permanent label on it, like the address plaque on

ble containing a pointer is like a scrap of paper with a ss written on it.



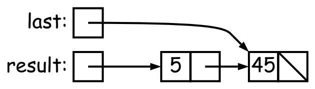
iew:



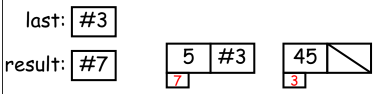
nother Way to View Pointers (II)

ointer to a variable looks just like assigning an integer

ecuting "last = last.tail;" we have



iew:



ative view, you might be less inclined to think that as- uld change object #7 itself, rather than just "last".

nternally, pointers really are just numbers, but Java as more than that: they have *types*, and you can't just ers into pointers.

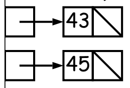
Destructive vs. Non-destructive

a (pointer to a) list of integers, L, and an integer in- rn a list created by incrementing all elements of the list

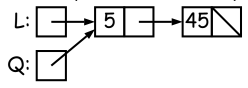
```
f all items in P incremented by n. Does not modify
ng IntLists. */
List incrList(IntList P, int n) {
 /*( P, with each element incremented by n )*/
```

t is *non-destructive,* because it leaves the input objects own on the left. A *destructive* method may modify the o that the original data is no longer available, as shown

ncrList(L, 2):



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



ondestructive IncrList: Recursive

```
f all items in P incremented by n. */
List incrList(IntList P, int n) {
 null)
 null;
urn new IntList(P.head+n, incrList(P.tail, n));
```

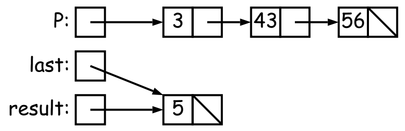
crList have to return its result, rather than just set-

crList(P, 2), where P contains 3 and 43, which IntList created first?

An Iterative Version

rList is tricky, because it is *not* tail recursive.
hings first-to-last, unlike recursive version:

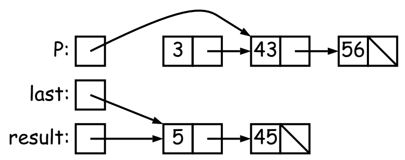
```
ncrList(IntList P, int n) {

, last;
      <<<
st(P.head+n, null);
!= null) {

ist(P.head+n, null);
tail;
```



An Iterative Version

rList is tricky, because it is *not* tail recursive.
hings first-to-last, unlike recursive version:

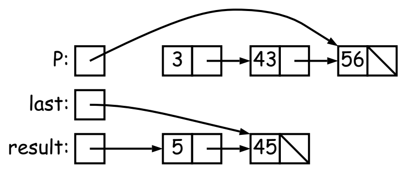
```
ncrList(IntList P, int n) {

, last;

st(P.head+n, null);
!= null) {
      <<<
ist(P.head+n, null);
tail;
```



An Iterative Version

rList is tricky, because it is *not* tail recursive.
hings first-to-last, unlike recursive version:

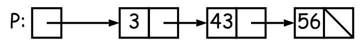
```
ncrList(IntList P, int n) {

, last;

st(P.head+n, null);
!= null) {
      <<<
ist(P.head+n, null);
tail;
```



An Iterative Version

rList is tricky, because it is *not* tail recursive.
hings first-to-last, unlike recursive version:

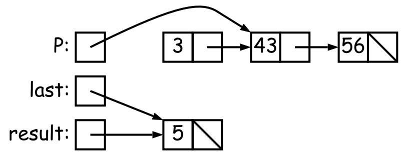
```
ncrList(IntList P, int n) {
      <<<

, last;

st(P.head+n, null);
!= null) {

ist(P.head+n, null);
tail;
```



An Iterative Version

rList is tricky, because it is *not* tail recursive.
hings first-to-last, unlike recursive version:

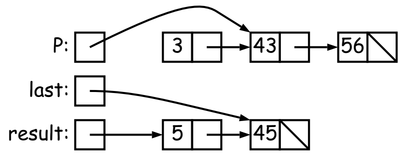
```
ncrList(IntList P, int n) {

, last;

st(P.head+n, null);
!= null) {
      <<<

ist(P.head+n, null);
tail;
```



An Iterative Version

rList is tricky, because it is *not* tail recursive.
hings first-to-last, unlike recursive version:

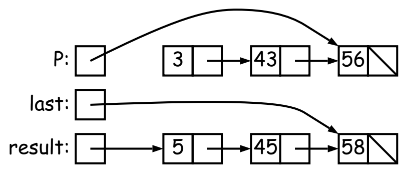
```
ncrList(IntList P, int n) {

, last;

st(P.head+n, null);
!= null) {


ist(P.head+n, null);
tail; <<<
```



An Iterative Version

rList is tricky, because it is *not* tail recursive.
hings first-to-last, unlike recursive version:

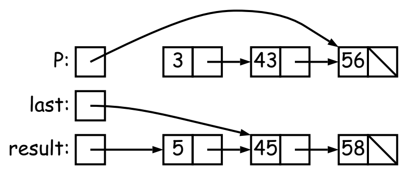
```
ncrList(IntList P, int n) {

, last;

st(P.head+n, null);
!= null) {


ist(P.head+n, null);
tail; <<<
```



An Iterative Version

rList is tricky, because it is *not* tail recursive.
hings first-to-last, unlike recursive version:

```
ncrList(IntList P, int n) {

, last;

st(P.head+n, null);
!= null) {
      <<<
ist(P.head+n, null);
tail;
```


P:
3
43
56
last:
result:
5
45
58