

# Introduction

**Last day to log in.** Be sure to do the first lab. Get an account form from me, if needed.

**Concurrent enrollment.** I will sign forms next week.

**Trying to get in?** You must be concurrently enrolled or wait-listed to get in. Be sure you are one of those two!

**Reminder about readers.** Two are available at Vick Copy. Be sure you have them.

**Today.** Simple classes. Scheme-like lists. Destructive vs. non-destructive operations. Models of memory.

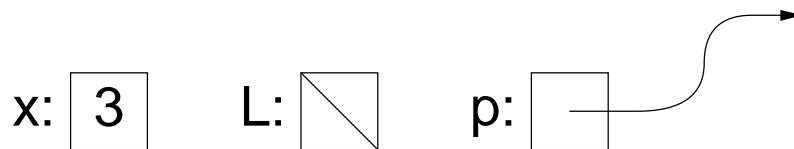
**Reading.** Please read 5.6.4 and 5.6.5 of *Programming Into Java*.

# Values and Containers

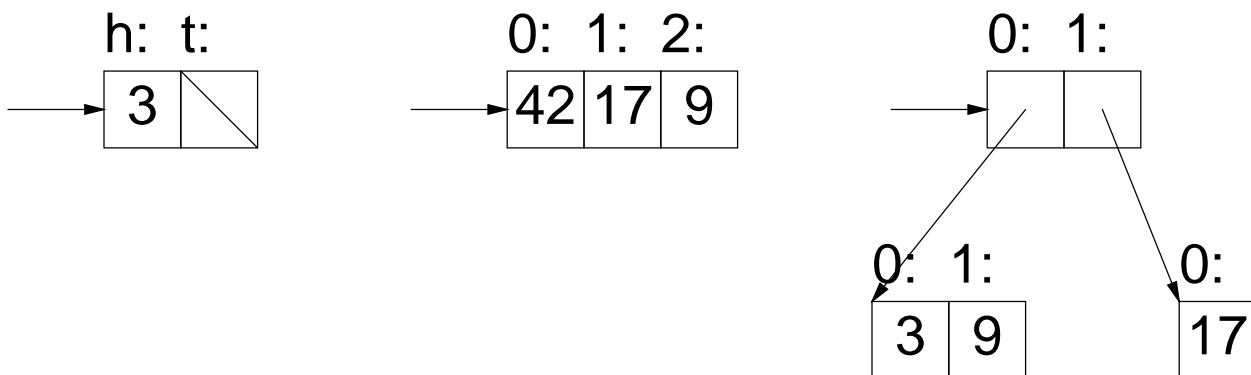
- *Values* are numbers, booleans, and pointers. Values never change.
- *Containers* contain values (variables, fields, array elements, parameters), or other containers.
- *Pointers* are values that *reference* (point to) containers.
- One particular pointer, called **null**, points to nothing.

# Varieties of Container

- The values in a container (its *state*) may change over time.
- Containers may be *named* or *anonymous*.
- *Simple containers* contain values:



- *Structured containers* (or *objects*) contain 0 or more named containers:



- 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 (no exceptions).

# Defining Objects

- Class declarations introduce new types of objects.
- Example: integer lists (simpler than last time):

```
public class IntList {
    // Constructor function
    // (used to initialize new object)
    /** List cell with given head and 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);
```

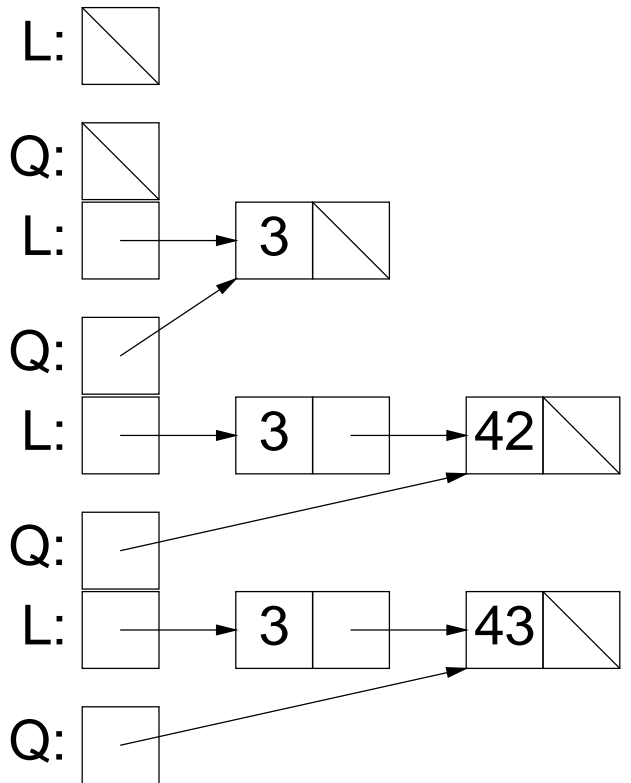
```
Q = L;
```

```
Q = new IntList(42, null);
```

```
L.tail = Q;
```

```
L.tail.head += 1;
```

```
// Now Q.head == 43
```



# Destructive vs. Non-destructive

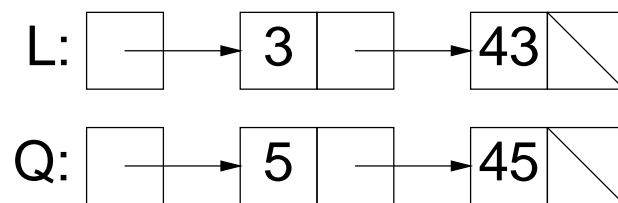
Here's `incrList` again, on `IntLists`:

```
/** 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));
}
```

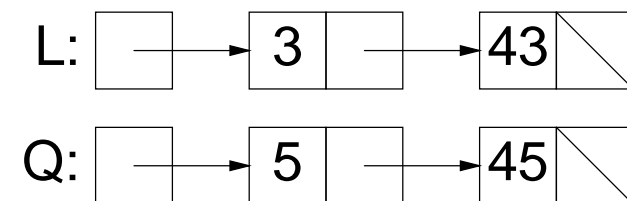
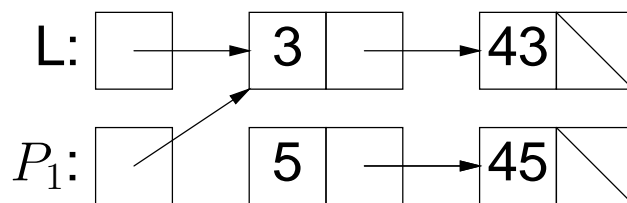
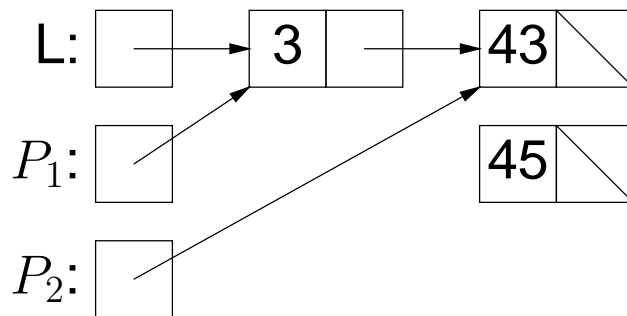
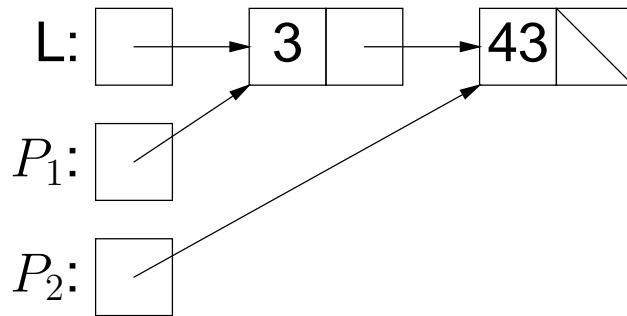
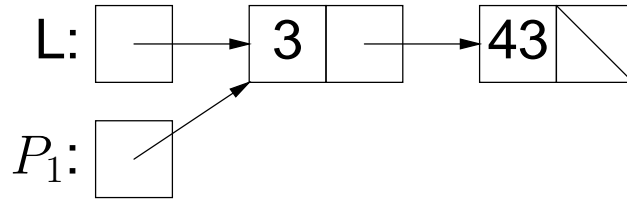
This is *non-destructive*: computing

```
Q = incrList(L, 2);
```

keeps the original list intact, giving



# Tracing the Non-Destructive Method



# Destructive Operation

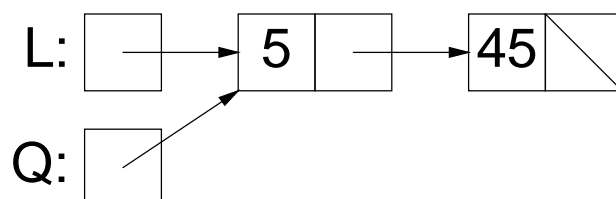
*Destructive* solutions may modify the original list. E.g.:

```
/** List L destructively incremented by n. */
static IntList dincrList (IntList L, int n) {
    if (L == null)
        return null;
    else {
        L.head += n;
        dincrList (L.tail, n);
        return L;
    }
}
```

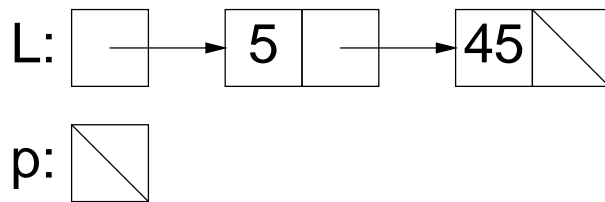
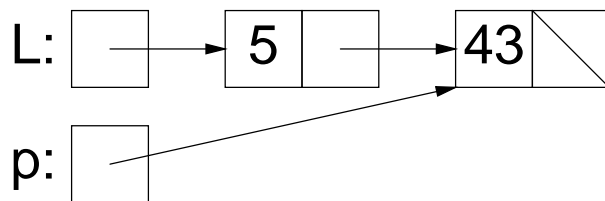
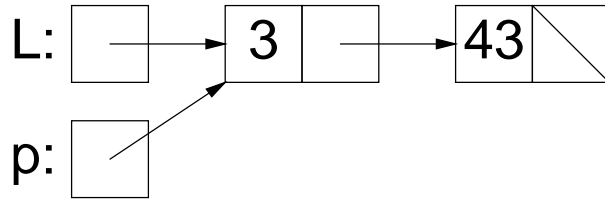
Or iteratively:

```
/** List L destructively incremented by n. */
static IntList dincrList (IntList L, int n) {
    for (IntList p = L; p != null; p = p.tail)
        p.head += n;
    return L;
}
```

Now,  $Q = \text{dincrList}(L, 2)$  gives



# Trace of Destructive Iterative Program



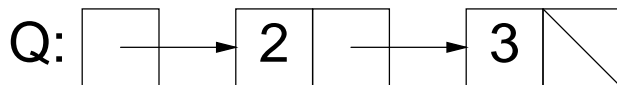
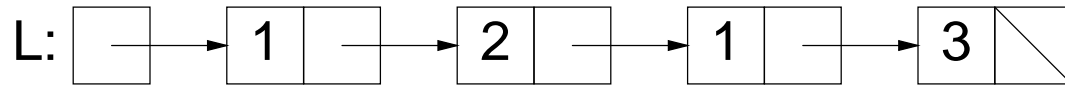
# Destructive vs. Non-destructive Removal

```
/** The list of all items in L unequal to X */
static IntList removeAll (IntList L, int x)
{
    if (L == null)
        return null;
    else if (L.head == x)
        return removeAll (L.tail, x);
    else
        return new IntList (L.head,
                             removeAll (L.tail, x));
}
```

```
/** Destructive version */
static IntList dremoveAll (IntList L, int x) {
    if (L == null)
        return null;
    else if (L.head == x)
        return dremoveAll (L.tail, x);
    else {
        L.tail = dremoveAll (L.tail, x);
        return L;
    }
}
```

# Destructive vs. Non-destructive Removal II

Consider result of  $Q = \text{removeAll}(L, 1)$ :



versus  $Q = \text{dremoveAll}(L, 1)$ :

