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.


Project #0 to be released later tonight. Due Sept. 20. Watch the Labs and Homework page.

---

**Values and Containers**

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

```
3 'a' true
```

- **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
```

<table>
<thead>
<tr>
<th>Simple container (local variable)</th>
<th>0 1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>h: 3</td>
<td>42 17 9</td>
</tr>
<tr>
<td>Alternative Notation</td>
<td></td>
</tr>
<tr>
<td>h: 3</td>
<td></td>
</tr>
<tr>
<td>t:</td>
<td></td>
</tr>
<tr>
<td>0 1 2</td>
<td></td>
</tr>
<tr>
<td>42 17 9</td>
<td></td>
</tr>
<tr>
<td>0 1 2</td>
<td></td>
</tr>
</tbody>
</table>

---

**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).

```

Named simple containers (fields) within structured containers
```

```

simple container (local variable)               structured containers (anonymous)
```

- 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:

```java
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;
}
```

**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:

```
last: [ ]
result: [5] 45
```

- Alternative view:

```
last: [#7]
result: [#7] 5 #3 45
```

**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

```
result: [5] 45
```

- Alternative view:

```
result: [#7] 5 #3 45
```

- 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

**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 \( n \).

```java
/** 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:

<table>
<thead>
<tr>
<th>L</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>43</td>
</tr>
</tbody>
</table>

After \( Q = \text{incrList}(L, 2) \):

<table>
<thead>
<tr>
<th>L</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>45</td>
</tr>
</tbody>
</table>

Nondestructive IncrList: Recursive

```java
/** 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:

```java
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 = last.tail
        last.tail = new IntList (P.head+n, null);
        last = last.tail;<<<
    }return result;
}
```