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: \[
  \begin{array}{c}
  3 \\
  \end{array}
  \]

  L: \[
  \begin{array}{c}
  \quad \\
  \end{array}
  \]

  p: \[
  \begin{array}{c}
  \quad \\
  \end{array}
  \]

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

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

  \[
  \begin{array}{c}
  \text{Class Object} \\
  h \quad t \\
  \begin{array}{c}
  3 \\
  \end{array}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{Array Object} \\
  0 \quad 1 \quad 2 \\
  \begin{array}{c}
  42 \quad 17 \quad 9 \\
  \end{array}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{Empty Object} \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{Alternative} \\
  \text{Notation} \\
  h: \begin{array}{c}
  3 \\
  \end{array} \\
  t: \begin{array}{c}
  \quad \\
  \end{array}
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{} \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{} \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{} \\
  \end{array}
  \]
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).

```
<table>
<thead>
<tr>
<th>named simple containers (fields)</th>
</tr>
</thead>
<tbody>
<tr>
<td>within structured containers</td>
</tr>
</tbody>
</table>

p: ─────> 3 ─────> 7

<table>
<thead>
<tr>
<th>simple container (local variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>structured containers (anonymous)</td>
</tr>
</tbody>
</table>

- 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
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
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:

  ```plaintext
  last:      result:  
             5       → 45
  ```

- Alternative view:

  ```plaintext
  last: #7    result: #7 5 #3 45
         7      3
  ```
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

  last: [Diagram showing a pointer]
  result: 5 ———> 45

• Alternative view:

  last: #3
  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 \( n \).

```c
/** 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:

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

\[ L: \quad 3 \quad \rightarrow \quad 43 \]

\[ Q: \quad 5 \quad \rightarrow \quad 45 \]

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

\[ L: \quad 5 \quad \rightarrow \quad 45 \]

\[ Q: \quad \text{} \]
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:

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

Last modified: Fri Aug 31 14:51:39 2012
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;
}
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;
}
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;
}
```
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.tail
            = new IntList (P.head+n, null);
        last = last.tail; /**/<br />
    }<br />
    return result;
}
```