CS61B Lecture #3: Containers

• Today: Chapter 2 from the Assorted Materials on Java reader, and (for lab) the section on "Using Subversion" from the CS61B Tools Documentation reader.

• Please read (again) Chapters 3 and 4 of Head First Java for Wednesday.

• Midterm is scheduled for the evening of 15 October (Monday).

• Project 1 will be due the preceding week (3 October or so).

• 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.
  
  \[
  3 \quad 'a' \quad \text{true} \quad \frac{1}{\frac{1}{\frac{1}{}}}
  \]

• **Simple containers** contain values:
  
  \[
  x: 3 \quad L: \quad p:
  \]

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

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

  Class Object

  \[
  \begin{array}{cc}
  h & t \\
  3 &
  \end{array}
  \]

  Alternative Notation

  \[
  h: 3 \\
  t:
  \]

  Array Object

  \[
  0 \quad 1 \quad 2 \\
  42 \quad 17 \quad 9
  \]

  Empty Object

  \[
  \]

  0

  42

  1

  17

  2

  9
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.

![Diagram of pointers and containers]
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>simple container (local variable)</th>
<th>named simple containers (fields)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>within structured containers</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
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 \( n \).

```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));
}
```

We say \text{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 \( Q = \text{incrList}(L, 2) \):
\[
\begin{array}{c}
L: \quad 3 \rightarrow 43 \\
Q: \quad 5 \rightarrow 45 \\
\end{array}
\]

After \( Q = \text{dincrList}(L, 2) \) (destructive):
\[
\begin{array}{c}
L: \quad 5 \rightarrow 45 \\
Q: \quad \\
\end{array}
\]
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;
}
```

P: 3 43 56
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: Mon Oct 22 15:34:04 2007
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: Mon Oct 22 15:34:04 2007
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:

```
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;
}
```

P: 

```
3 ——> 43 ——> 56
```

last:

```

```

result:

```
5 ——> 45
```
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: Mon Oct 22 15:34:04 2007