Values and Containers

- **Values** are numbers, booleans, and pointers. Values never change.
  
  
  \[
  3 \quad \text{‘a’} \quad \text{true} \quad \frac{1}{7} \quad \frac{\text{\ldots}}{\ldots}
  \]

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

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

- **Structured containers** contain (0 or more) other containers:
  
  \[
  \begin{array}{ccc}
  \text{Class Object} & \text{Array Object} & \text{Empty Object} \\
  h: 3 & 0 & \frac{1}{2} \frac{9}{17} 42 \frac{0}{3} \\
  t: & 1 & 17 \\
  \end{array}
  \]

  Alternative Notation:
  
  \[
  \begin{array}{ccc}
  \text{h: 3} & 0 & 42 \\
  \text{t:} & 17 & 9 \\
  \end{array}
  \]

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).

- In Java, assignment copies values into simple containers. (Therefore, structured containers contain only 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;
}
```

```
IntList Q, L;
L:
Q:
L = new IntList(3, null);
Q = L;
L:
Q:
L = 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 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 = incrList(L, 2):
L: 3 43
Q: 5 45

After Q = dincrList(L, 2) (destructive):
L: 5 45
Q: 5 43
```

An Iterative Version

An iterative incrList is tricky, because it is not tail recursive.

```java
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last = new IntList(P.head+n, incrList(P.tail, n));
    while (P.tail != null) {
        P = P.tail;
        last.tail = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

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