Values and Containers

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

- **Simple containers** contain values:

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

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

  ![Diagram of class object, array object, and empty object]

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).

- **Exactly like Scheme!**

**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]

**Values and Containers**

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

  ![Diagram of values]

- **Simple containers** contain values:

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

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

  ![Diagram of class object, array object, and empty object]

**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).

- **Exactly like Scheme!**

**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]
Class declarations introduce new types of objects.

Example: list of integers:

```java
class IntList {
    // Constructor function
    IntList (int head, IntList tail) {
        this.head = head; this.tail = tail;
    }
    // Names of simple containers (fields)
    public int head;
    public IntList tail;
}
```

Names of simple containers (fields):
public int head;
public IntList tail;

Primitive Operations

```java
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
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):  After Q = dincrList(L, 2) (destructive):
L: 3 43 56
Q: 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;
}
```