Last Comments on static

- Denotes code or variables not connected with individual objects.
  - Can be accessed even if you don’t have any objects of the class around.
  - Instance variables declared static have one value associated with the class; otherwise, each object has its own copy of the value.
  ```java
going { x; // (every object has an x component) 
static int x; // (one x variable for the entire class.)
```

Uses of static

- Single variables relevant to the entire class
  ```java
  static int numQuantitiesCreatedSoFar;
  ```
- Constants
  ```java
  Long.MIN_VALUE
  Math.PI
  ```
- "Ordinary Functions"
  ```java
  PolyList.list
  Math.sqrt
  ```

The final Modifier

- Instance variables declared final cannot be changed (assigned to) once initialized.
- Methods declared final cannot be replaced (overridden) in subclasses.
- Examples relevant to the homework:
  ```java
  PolyList.nil is static final
  Factor, Num, and Denom are final (as of yesterday)
  ```
Essential Polylist Operations

- Polylist.list(Ob₀, Ob₁, ..., Obₙ) creates a Polylist from objects (up N = 15 or so).
- L.first() returns the first element.
- L.rest() returns the rest of the elements as a Polylist.
- L.isEmpty() returns true if the list is empty.
- L.nonEmpty() returns true if non-empty.
- L.cons(Ob₀) creates a new list with Ob₀ first.
- Polylist.nil is the empty Polylist.

Other Polylist Operations

- L.append(M) creates a new list with elements of M following those of L.
- L.reverse() creates a new list reversing L.
- L.nth(N) returns the Nth element of L (N = 0, 1, 2, ...) [use sparingly! Why??]
- L.second(), L.third(), L.fourth(), L.fifth(), L.sixth() ... do the obvious
- L.toString() converts the whole list to a single String

More Polylist Operations

- Polylist.explode(S) explodes string S into a list of its characters (a static method); L.implode() turns the list back into a string.
- L.array() returns a Java array of the Objects in list L.
- L.prefix(N) returns the length N prefix of L.
- range(M, N) returns a list (M M+1 ... N).
- range(M, N, S) returns a list (M M+S M+2S ... N).

Putting Values into Lists

- These lists can contain any combination of objects
  - of any class, including other Polylist objects and string contents (which are objects of class String).
  - Primitives (ints, longs, floats, doubles, chars ...) are not objects in Java; need to be “wrapped” within an object before they can be added to a list.
  - The constructor Long(...) makes an object (of class Long) out of a primitive long; need the new keyword, of course.
    - Other wrappers: Integer(), Float(), Double(), Boolean(), Snoop, Ice-T ...

Polylist mylist =
Polylist.list("hello", new Long(3), new Float(3.14));
Getting Values out of Lists

- Polylists contain values of class Object.
- If you get a value out of a list (e.g., via the first or nth methods) all the typechecker knows is that it's some Object.

```java
Polylist p1 = Polylist.list("a");
Polylist p2 = Polylist.list(new Long(0));
Object o1 = p1.first(); // OK
Object o2 = p2.first(); // OK
String s1 = p1.first(); // REJECTED
Long l2 = p2.first(); // REJECTED
```

Getting Objects out of Lists

- If you expect a particular sort of object (because you know how the list was created) you can use a cast.
  - Does a run-time check of the object's class.

```java
Polylist p1 = Polylist.list("a");
Polylist p2 = Polylist.list(new Long(0));
Object o1 = p1.first(); // OK
Object o2 = p2.first(); // OK
String s1 = (String)p1.first(); // OK
Long l2 = (Long)p2.first(); // OK
String s2 = (String)p2.first(); // ?
```

Type Discrimination

- If you don't know which sort of object to expect, you can do a run-time check to see what casts are appropriate

```java
if ( ob instanceof Long )
    ...
else if ( ob instanceof Polylist )
    ...
```

Getting Primitives out of Objects

- Objects of classes Long, Integer, etc. (any subclass of Numeric) have extraction methods longValue(), intValue(), etc.
- See [http://java.sun.com/j2se/1.3/docs/api/](http://java.sun.com/j2se/1.3/docs/api/)

```java
Polylist p2 = Polylist.list(new Long(0));
Long l2 = (Long)p2.first(); // OK
long n = l2.longValue(); // OK
long m = ((Long)p2.first).longValue(); // OK
```
Conversion to String

- Class `String` includes the following static methods (not constructors):
  
  ```java
  valueof(double d)
  valueof(long x)
  ...
  ```
  
  - Each returns a `String`.

Conversion from String

- Use the appropriate static method in the class to which you wish to convert, e.g.
  ```java
  Long.parseLong(S)
  Double.parseDouble(S)
  ```
  (Don't use `getLong`, which has a different meaning entirely.)

Equality Checking

- To check whether two Objects are equal, DO NOT USE `==`.
  - This only checks whether the addresses of those objects are identical.
  - Arguments could be semantically equal, but still be distinct objects. E.g., strings or lists of strings

- DO use the `equals` method:
  ```java
  if ( ob1.equals(ob2) )
  ```

Higher-Order Functions in Polya

- Java does not have the notion of Higher-Order methods.
  - However, it is possible to achieve the effect in a slightly round-about way.
  - To represent a function, say of one argument, we will use objects of classes implement the `Function1` interface.

- This interface requires a method `apply` with takes an `Object` argument and returns an `Object` result.
Example: An argument to map

```java
class Scaler implements Function1 {
    long factor;
    Scaler(long factor) {
        this.factor = factor;
    }
    public Object apply(Object Arg) {
        long arg = ((Long)Arg).longValue();
        return new Long(factor*arg);
    }
}
```

Example: map usage

```
Polylist L1 = Polylist.list(new Long(1),
    new Long(2),
    new Long(3));
Polylist L2 = L1.map(new Scaler(100));
```

A Recursive List Pattern (without using map)
- ad-hoc map-like operations, build list **outside-in**, using recursion:

```
static Polylist scale(long factor, Polylist L) {
    if (L.isEmpty())
        return Polylist.nil;
    long first = ((Long)L.first()).longValue();
    Long result = new Long(factor*first);
    return Polylist.cons(result, scale(factor, L.rest()));
}
```

An Iterative List Pattern
- build list **inside-out**, using ordinary iteration and an accumulator:

```
static Polylist scaleAndReverse(long factor, Polylist L) {
    Polylist result = Polylist.nil;
    for (; L.nonEmpty() ; L = L.rest() ) {
        long first = ((Long)L.first()).longValue();
        result = Polylist.cons(new Long(factor*first), result);
    }
    return result;
}
An Iterative Reduce Pattern

- collapse list into a value using ordinary iteration

```java
static long sum(Polylist L)
{
    long result = 0;
    for(; L.nonEmpty(); L = L.rest())
    {
        long first = ((Long)L.first()).longValue();
        result += first;
    }
    return result;
}
```

An Recursive Merge Pattern

- merge two lists of Longs in increasing order

```java
static Polylist merge(Polylist L, Polylist M)
{
    if( L.isEmpty() ) return M;
    if( M.isEmpty() ) return L;
    long firstL = ((Long)L.first()).longValue();
    long firstM = ((Long)M.first()).longValue();
    if( firstL <= firstM )
        return merge(L.rest(), M).cons(L.first());
    else
        return merge(L, M.rest()).cons(M.first());
}
```

Try this

- determine whether an Object occurs in a Polylist

```java
static boolean member(Object Ob, Polylist L)
{
}
```

If you used recursion, try it with iteration, and vice-versa

- determine whether an Object occurs in a Polylist

```java
static boolean member(Object Ob, Polylist L)
{
}
```