rex to Java

James Gosling, Inventor of Java

James Gosling received a BSc in Computer Science from the University of Calgary, Canada in 1977. He received a PhD in Computer Science from Carnegie-Mellon University in 1983. He is currently a Distinguished Engineer at Sun Microsystems. He has built satellite data acquisition systems, a multiprocessor version of Unix, several compilers, mail systems and window managers. He has also built a WYSIWYG text editor, a constraint based drawing editor and a version of a text editor called Emacs for Unix systems. More recently he has been the lead engineer for the Java/HotJava system.

http://java.sun.com/people/jag/

Java,
an Imperative Language

- Imperative languages often ***permit*** the use of functional programming.
- Sometimes just say "no" to side-effects.
- Otherwise use functions and side-effects articulately.
- **Best of both worlds!**

Java vs. rex

- The analog to ***function*** in rex is ***method*** in Java. Functions are applied as aFunction(x, y, z), while methods are applied like x.aMethod(y, z).

  principal argument (an object)

- Argument and return **types** must be ***declared*** in Java, not in rex.

- Both allow **recursion**.

- All of the underlying functionality in rex is implementable.

- Think of rex lists as your **abstraction**, use Java to implement it.

Java to rex

- Our initial foray into Java will be to implement the concepts of rex, particularly lists.
  - Think of rex lists as your **abstraction**, use Java to **implement** them.

Java brief review

The empty Java program

```
class Empty ☹
{

public static void main(String arg[ ])
   {
   }

}
```

The empty Java program

```
class Empty
{

public static void main(String arg[ ])
   {
   }

}
```

The one and only class of this program

Makes this method accessible from the outside.

The main method for this class (called at start-up).

External arguments for this method.

Result type of this method (none).

Says that this method depends only on the class, not any specific object.

The “Hello, world” program in Java

```
class Hello ☺
{
public static void main(String arg[])
   {
   System.out.println(”Hello, world!");
   }
}
```

The “Hello, world” program in Java

The empty program + one line.

```
class Hello
{
public static void main(String arg[])
   {
   System.out.println(”Hello, world!");
   }
}
```

The print-with-end-of-line method for object System.out.

The standard output stream object, pre-defined in the System class.

The “System” class.

Running Java on turing

- Current version is 1.4
- To compile:

UNIX convention for compiler, e.g. javac, cc

```
javac Hello.java
```

- To execute:

No "c" here.

No ".class" here.

```
java Hello
```

Running Java on turing

```
turing 101> ls Hello.*
Hello.java
```

Check what's there.

```
turing 102> javac Hello.java
```

Compile it.

```
turing 103> ls Hello.*
Hello.class   Hello.java
```

Check what's there now.

```
turing 104> java Hello
Hello, world!
```

Run it.

Be astounded by results.

HMC Shortcuts

Since java is a prefix of javac, this tends to confound using command completion (e.g. `!j` in the Cshell).

In your .cshrc should be the following command definitions:

```
alias jc 'javac \!$.java'                  #compile java
alias je 'java'                             #execute
alias jx 'javac \!$.java ; java \!$'        #compile and execute
```

Example usage:

```
jc Hello            # same as javac Hello.java
je Hello            # same as java Hello
jx Hello            # same as javac Hello.java; java Hello
```

Then use `!jc`, `!je`, or `!jx` to **re-do** previous commands of same type.

Java Data Items

- Java data items are either:
  - **Primitives**, such as
    - int, long, float, double, char
  - **Objects**, such as
    - String, Long, Double
    - Objects you define
    - Arrays are essentially Objects too.

Purposes of Objects

- **Aggregate** various data objects together
- Allow **mutation** of the **state** of data objects
- Control use and access of data according to specific **disciplines**
- and other good stuff, such as inheritance and delegation

Immutable Objects

- An Object is ***immutable*** if its state never changes once it is created.
- **Functional programming** deals with immutable objects *almost* exclusively
  - (exception: delayed evaluation)
- The aggregating and disciplined access properties of Objects are still very useful.

OpenList class

- This is a class you will construct and own.
- It will allow you to solve problems for which rex is suitable in Java, as well as problems for which rex is less suitable.
- Think of rex lists as the abstraction. Use Java to implement.

Object Creation

- Objects are created using **constructors.**
- For a given Class of Objects, there can be *multiple* types of constructors, each providing different types of parameters to define the creation of an object.
- In Java, constructors always take the same name as their Class.

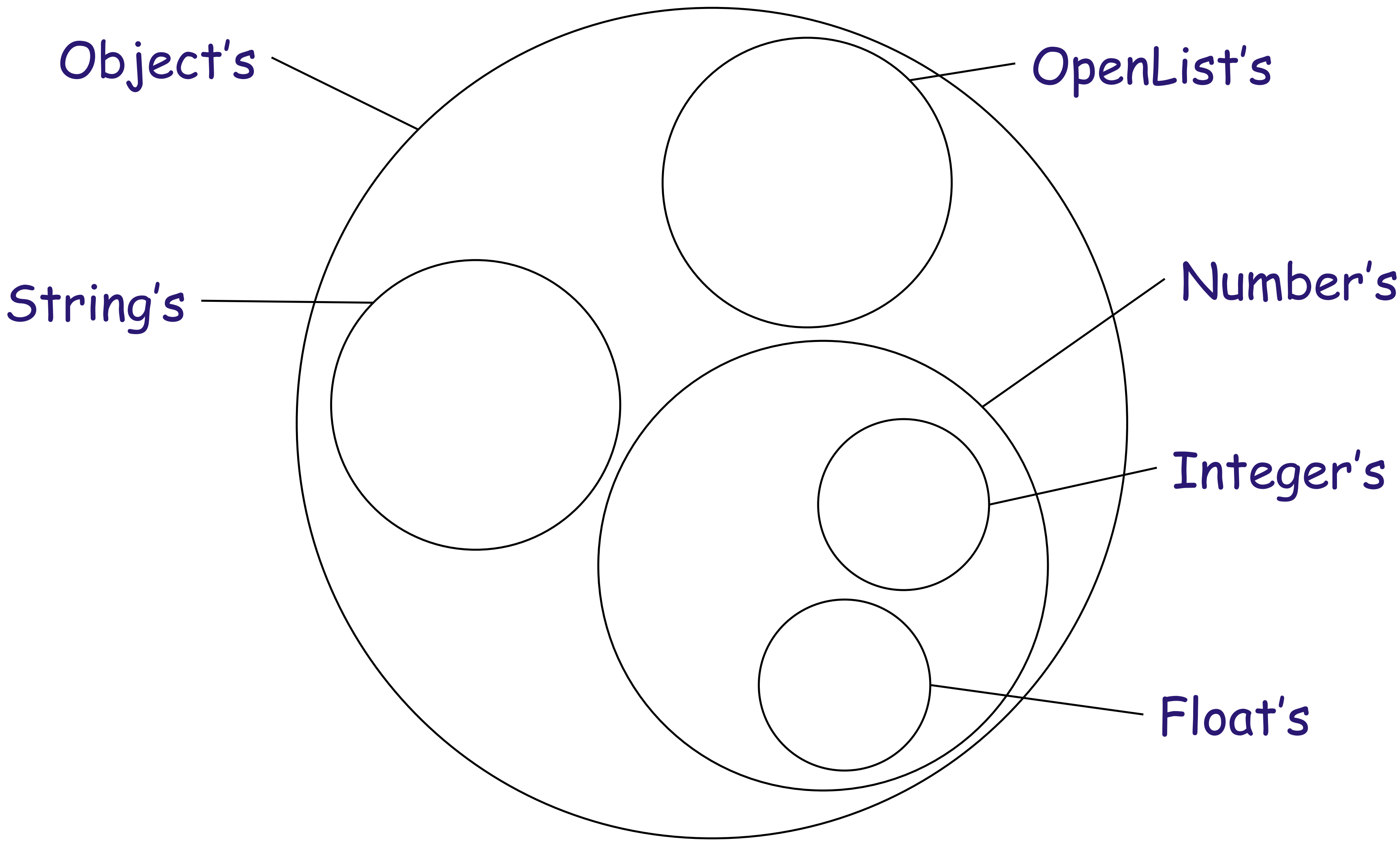
The Class "Object" in Java

- In Java, all objects are derived from an ancestral class called

  Object
- For example, if we define a class OpenList, an object of that class is still an Object as well.

Class Hierarchy:

Abstraction, not Containment



Purpose of Hierarchical Abstraction

- Group classes with *similar* properties into super-classes.
- e.g. Number is a super-class of:
  - Integer
  - Float
  - Double
  - Long
  - BigInteger
  - BigDecimal
  - Byte
  - Short
- Objects in these classes share some properties, but are distinct in others.
- The bottom line: **intellectual economy**.

One OpenList Constructor

- To create an OpenList, we need to give values to two internal variables, First and Rest:

```
public class OpenList
{
private Object First;
private OpenList Rest;

public OpenList(Object _First, OpenList _Rest)
  {
  First = _First;
  Rest  = _Rest;
  }
```

... more stuff to come ...

```
 }
```

Getters

- Attributes of objects should usually not be accessed within an object simply by referring to them:

```
OpenList x = new OpenList(…);
…
System.out.println(x.First);
```

BAD

- **except** possibly for debugging purposes.
- Instead, use a **getter** method:

```
Object first()
{
return First;
}
...
System.out.println(x.first());
```

GOOD

Static?

- In Java, a method's action may or may not depend on the Object on which the method is called:
  - methods that do *not* depend on an Object should be annotated as

    `static`

Example of Static

- Within class OpenList:

```
public class OpenList
{

public static OpenList nil;

private Object First;
private OpenList Rest;

. . .
}
```

The One, True, Empty List

Static Methods can only call static

- A static method can only depend on
  - variables declared as static
  - other static methods
- A static method, therefore, ***cannot*** depend on:
  - variables **not** declared as static
  - other methods **not** declared as static unless those methods are applied to some static object
- Any other way just doesn't make sense.
- The compiler will tell you, but sometimes in a cryptic way.

Example

```
class myBad
{
int x;

myBad(int x)
   {
   this.x = x;
   }

int getX()
   {
   return x;
   }

static boolean test()
   {
   return getX() > 0;
   }
}
```

Illegal:
static depends on non-static

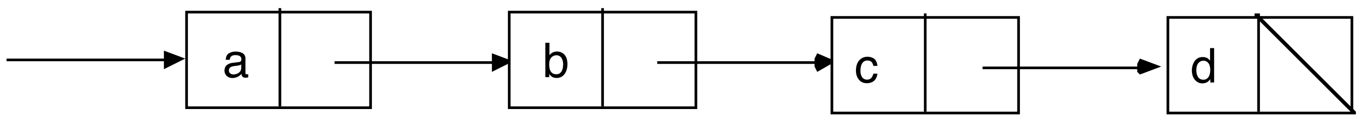
Example

```
class myGood
{
static MyOb myOb = new MyOb();


static boolean test()
  {
  return myOb.getX() > 0;
  }
}
```

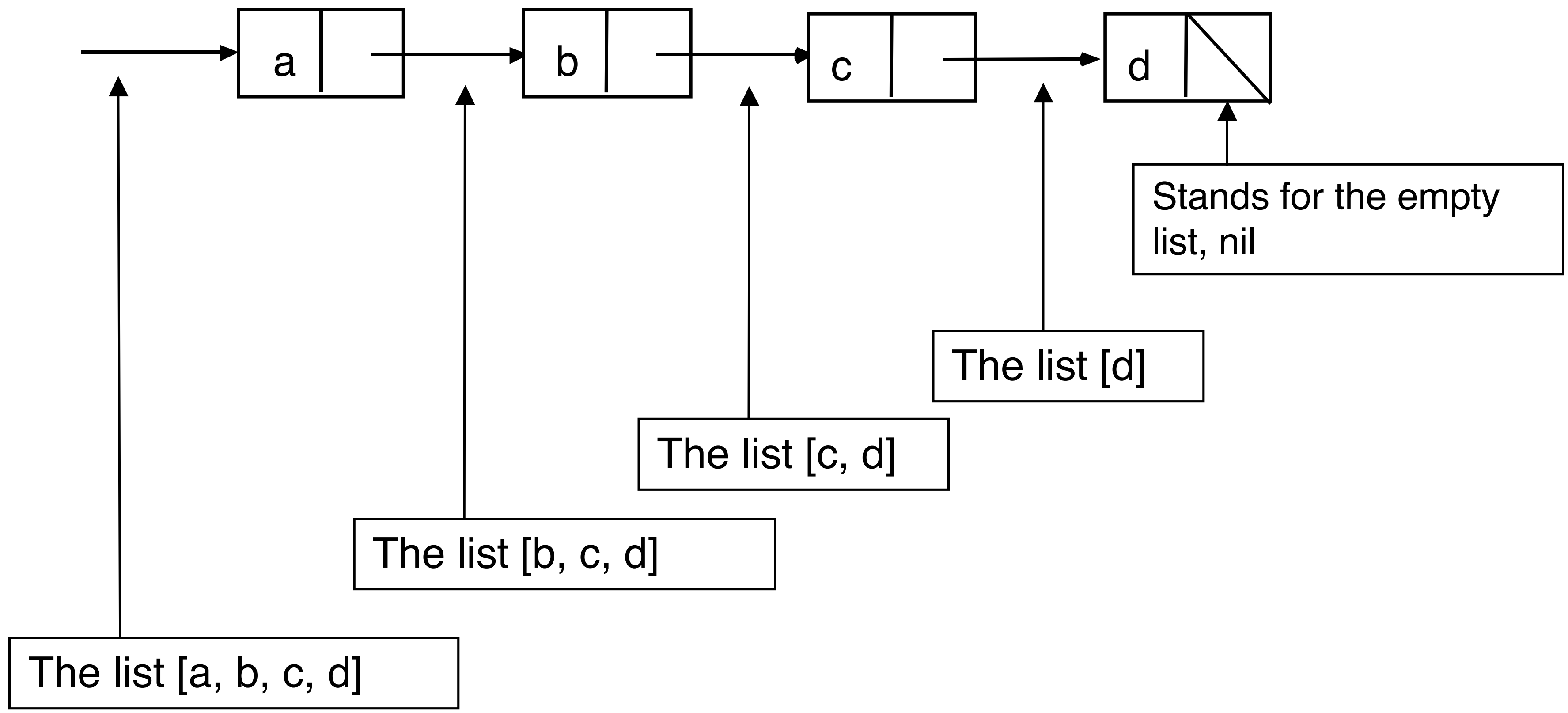
OK, depends on an object

An Open List



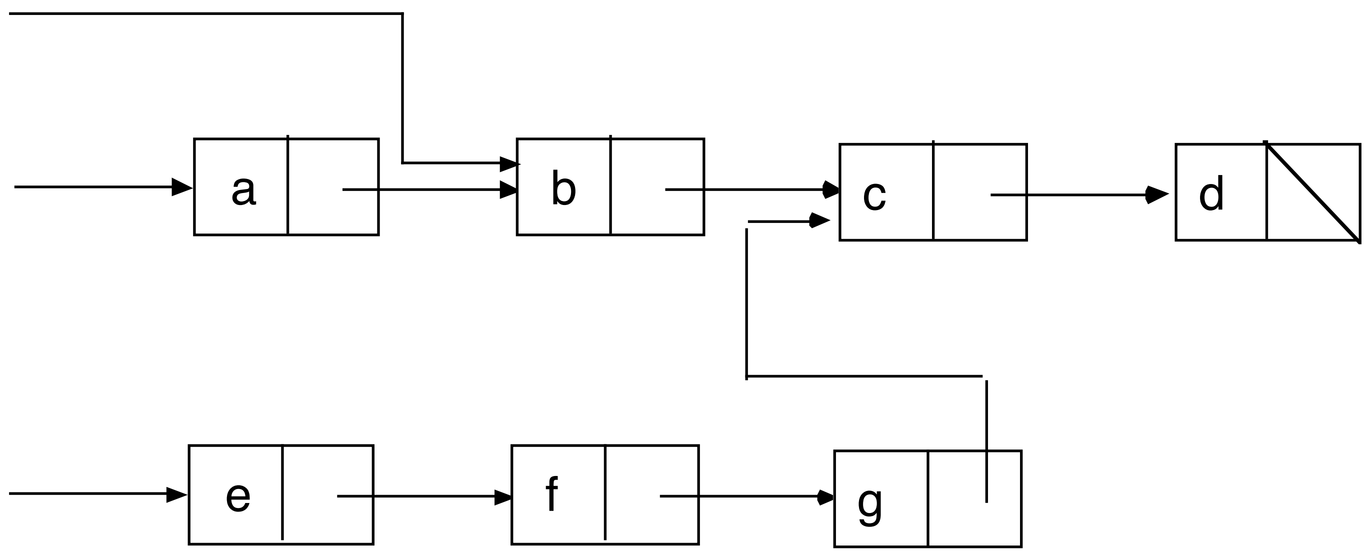
- Each list element begins a list in its own right.
- A list is **identified with** a reference to its first element.
- The empty list is identified with a special value, which we are calling **nil**.
- In the text, we used null for the empty list, but we're changing this now!

Open Lists Identified with References



Sharing in Open Lists

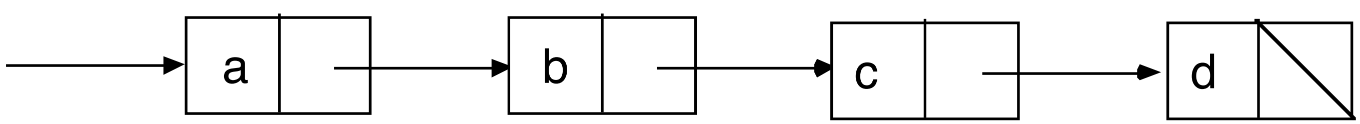
Display the list identified with each reference.



Why is list mutation discouraged?

Passing an Open List as an Argument to a Function

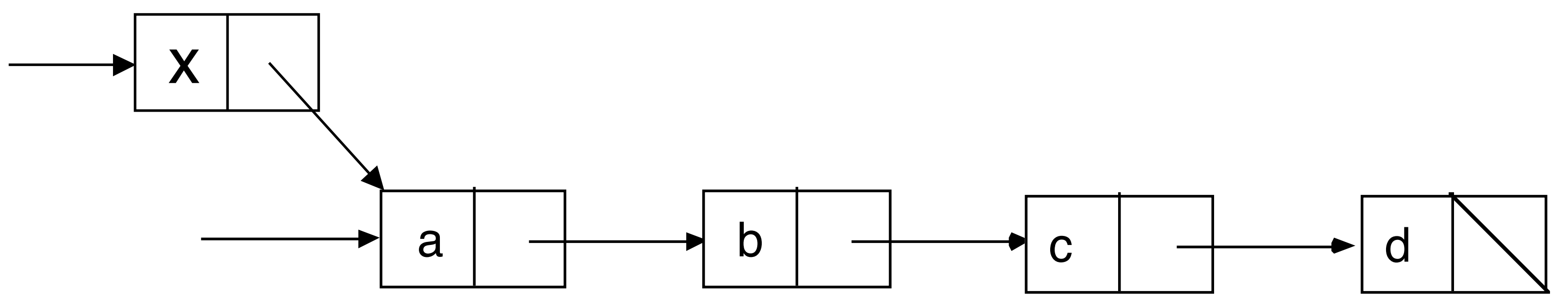
- To pass an open list as an argument in, we simply pass its *reference*.
- The list is **not** literally copied.



Open List Consing

- To "cons" an element to an open list, we simply put the element in a new cell and hook the cell to the original list:
- consing x to the front [x | [a, b, c, d]] yields

caution: rex, not Java, notation!



cons, the pseudo-constructor

- cons constructs OpenLists, but is not technically a **Constructor** in the sense of Java.
- I call it a pseudo-constructor, as it really is a method.
- It can also be called a "factory method" because it produces new objects.

The static cons

- The typical use of cons is static:

```
public class OpenList
{
public static OpenList nil;

private Object First;
private OpenList Rest;

public static OpenList cons(_First, _Rest)
  {
  return new OpenList(_First, _Rest);
  }
. . .
}
```

The definition of nil;

- nil is defined to be some OpenList.

```
public class OpenList
{
public static OpenList nil = cons(null, null);

private Object First;
private OpenList Rest;

public static OpenList cons(_First, _Rest)
  {
  return new OpenList(_First, _Rest);
  }
}
```

What is null?

- null is built into Java.
- It is the reference that refers to NO object.
- It has a value, but we cannot apply a method to it:

```
Object A = null;
A.myMethod();
```

**Bad:
syntactically OK**

Why cons(null, null) for nil?

- Actually, cons(anything, anything) would work.
- cons(null, null) is just the simplest OpenList we can create
- We agree never to look at the First and Rest of nil.

Reference Comparison

- References are values.
- They can be compared.
- Checking whether an OpenList is nil will be done by comparing its reference to the reference to nil.
- As long as there is only one nil, we are safe.

Reference Comparison (2)

- Two distinct objects can have identical “content”, but
- Their references will compare !=
- Two references only compare == if they are the **same object**.

Implementing isEmpty()

```
public class OpenList
{
public static OpenList nil = cons(null, null);

. . .

public static boolean isEmpty(OpenList L)
  {
  return L == nil;
  }

public boolean isEmpty()
  {
  return isEmpty(this);
  }
}
```

Keyword this means “The current object”

Using isEmpty()

Inside class OpenList:

```
OpenList L = . . .;

isEmpty(L);

L.isEmpty();
```

Inside a *different* class:

```
OpenList L = . . .;

OpenList.isEmpty(L);

L.isEmpty();
```

Less Convenient

.equals() method

- The customary way to compare for content is to use

  ```
  Ob1.equals(Ob2)
  ```

  not

  ```
  Ob1 == Ob2.
  ```

- Our list **nil** is not based on content, but rather absolute reference.

Non-static cons?

- Non-static cons would have to add a new Object to an OpenList.
- Outside the OpenList class it would be applied as:

```
OpenList L = . . .;
OpenList M;

M = L.cons(Ob2).cons(Ob1);
```

Reverse order

or

```
M = OpenList.cons(Ob1, OpenList.cons(Ob2, M));
```

list methods

- Define methods named **list** of various numbers of arguments, e.g.

```
M = OpenList.list(Ob1, Ob2, Ob3, . . ., ObN);
```

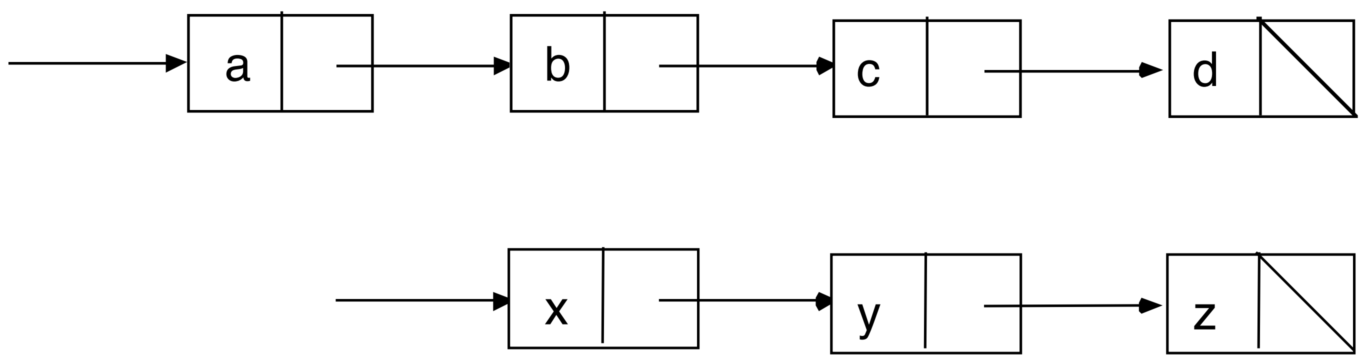
- More convenient that using a bunch of OpenList.cons

- Alas, need a different definition for each number of arguments.

Appending Open Lists

- What happens when we append one open list to another, as in

`append(L, M)`?



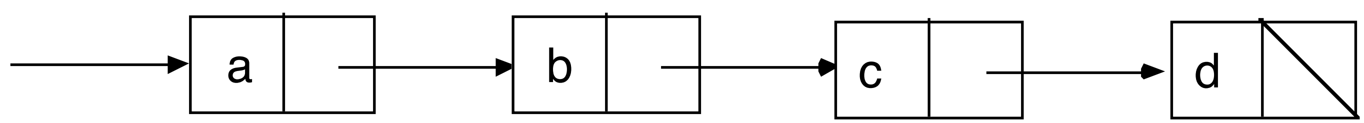
Recall definition of append

- append([], M) => M;
- append([A | L], M) => [A | append(L, M)];

Reversing an Open List

- What happens when we reverse an open list?

`reverse(L)`



Recall definition of reverse

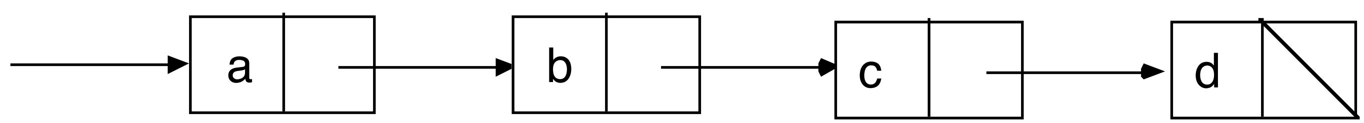
- reverse(L) = reverse(L, []);

- reverse([], M) => M;
- reverse([A | L], M) => reverse(L, [A | M]);

Mapping an Open List

- What happens when we map over an open list?

```
L.map(fun)
```



Wrappers for Primitives

- Items in a OpenList must be Objects.
- Primitives (ints, longs, floats, doubles, chars …) are **not** Objects in Java.
- The constructor **`Long( )`** makes an Object for any long by creating a "wrapper" which is an object.
- Other wrappers: **`Integer()`**, **`Float()`**, **`Double()`**, **`Boolean()`**, Snoop, Queen Latiffa, Ice-T, …

Strings

- In contrast to `long`, `int`, `float`, etc. `String`s are already objects.
- Consequently, `String`s do not need extra wrappers.
- `OpenList`s are also Objects.

Getters for Wrappers

- These can be applied to any Object derived from class **`Number`**, which includes **`Long`**, **`Integer`**, …:

  `longValue(), intValue(), …`

- Use the on-line javadoc pages on the web to find info:

  http://java.sun.com/j2se/1.3/docs/api/

Conversion to String

- Class `String` includes the following static methods (not constructors):

```
valueof(double d)
valueof(long x)
     ...
```

- Each returns a `String`.

Cheap Conversion to String

- "Adding" a number to a string will convert the number to a string, then ***concatenate*** it:

```
String s = "" + 31415;
```

Conversion **from** String

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

Type Discrimination

- The type of an Object can be discriminated using the `instanceof` operator:

```
Object ob = L.first();

if( ob instanceof Long ) ...

if( ob instanceof OpenList ) ...
```

Equality Checking

- To check whether two Objects are *equal*, **DO NOT USE ==**. This only checks whether the **references** to those objects are identical. The Objects could be equal, but be different Objects. This applies for `String`s, for example.

- **DO** USE `equals`:

```
if( ob1.equals(ob2) )
```

A Recursive List Pattern (without using map)

- ad-hoc map-like operations, build list **outside-in**, using recursion:

```
static OpenList scale(long factor, OpenList L)
  {
  if( L.isEmpty() )
    return OpenList.nil;

  long first = ((Long)L.first()).longValue();

  Long result = new Long(factor*first);

  return cons(result, scale(factor, L.rest()));
  }
```

unwrap

wrap

recurse

An Iterative List Pattern

- build list **inside-out**, using ordinary iteration and an accumulator

```
static OpenList scaleAndReverse(long factor, OpenList L)
  {
  OpenList result = OpenList.nil;

  for( ; L.nonEmpty() ; L = L.rest() )
    {
    long first = ((Long)L.first()).longValue();

    result = cons(new Long(factor*first), result);
    }

  return result;
  }
```

unwrap

wrap

An Iterative Reduce Pattern

- collapse list into a value using ordinary iteration

```
static long sum(OpenList L)
  {
  long result = 0;

  for( ; L.nonEmpty() ; L = L.rest() )
    {
    long first = ((Long)L.first()).longValue();

    result += first;
    }

  return result;
  }
```

unwrap

An Recursive Merge Pattern

- merge two lists of Longs in increasing order

```
static OpenList merge(OpenList L, OpenList 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());
  }
```

unwrap

Try this

- determine whether an Object occurs in an OpenList

```
static boolean member(Object Ob, OpenList L)
  {












  }
```

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

- determine whether an Object occurs in a OpenList

```
static boolean member(Object Ob, OpenList L)
  {









  }
```

Give a non-static method

- determine whether an Object occurs in *this* OpenList

```
boolean member(Object Ob)
  {




  }
```

Open vs. Closed Lists

- Two list models are described in the text:
  - Open lists:
    - Elements and sublists can be shared
    - Mutation of lists is discouraged
    - Mathematically elegant
  - Closed lists:
    - Sharing generally not done
    - Mutation of lists is ok, because they are encapsulated
    - Mathematically less attractive
  - Closed lists can be built by wrapping open lists