Java Jive
Java Jive
as sung by The Ink Spots

I love coffee, I love tea
I love the java jive and it loves me
Coffee and tea and the jivin' and me
A cup, a cup, a cup, a cup, a cup!

I love java, sweet and hot
Whoops! Mr. Moto, I'm a coffee pot
Shoot me the pot and I'll pour me a shot
A cup, a cup, a cup, a cup, a cup!

Oh, slip me a slug from the wonderful mug
And I cut a rug till I'm snug in a jug
A slice of onion and a raw one, draw one.
Waiter, waiter, percolator!

etc.
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).

- 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.
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.

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

Result type of this method (none).
class Hello
{
    public static void main(String arg[])
    {
        System.out.println("Hello, world!");
    }
}
The "Hello, world" program in Java

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

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

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

The empty program + one line.
Running Java on turing

• Current version is 1.3.1

• To compile: UNIX convention for compiler, e.g. javac, cc
  javac Hello.java

• To execute: No “c” here.
  java Hello
  No “.class” here.
Running Java on turing

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

Check what's there.
```

```bash
turing 102> javac Hello.java

Compile it.
```

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

Check what's there now.
```

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

Run it.
```

Be astounded by results.
Hacky 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 Objects

- 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
Immutable Objects

- An Object is immutable if it 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 the Unicalc problem in Java, as well as other things.

- Think of rex lists as the abstraction. Use Java to implement.
Using Your OpenList to Implement Unicalc Functionality

- We want to represent Unicalc Quantities.
- In Java, instead of using a list of 3 things, we will add a little more structure:

  ```java
  public class Quantity {
  private double Factor;
  private OpenList Num;
  private OpenList Denom;
  ... more stuff to come ...
  }
  ```
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.
Constructors for Quantities

- Constructors always take the same name as their Class.

- Therefore, all constructors for class `Quantity` will be called (you guessed it) `Quantity`.

- Constructors will differ depending on types.

- One constructor of a class can call another.
To define a quantity, we need to give values to all three internal variables:

```java
public class Quantity {
    // ... stuff you already saw ...

    public Quantity(double Factor, OpenList Num, OpenList Denom) {
        this.Factor = Factor;
        this.Num = Num;
        this.Denom = Denom;
    }

    // ... more stuff to come ...
}
```

Variables in **red** represent values in this Quantity. Variables in **green** represent values local to this constructor. The latter go away when the constructor is left.
Convenience Quantity Constructor
where the denominator is empty

```java
public class Quantity {
    … stuff you already saw …

    public Quantity(double Factor, OpenList Num) {
        this(Factor, Num, OpenList.nil);
    }

    … more stuff to come …
}
```

Variables in green represent values local to this constructor.

```java
this(...) means “call the constructor of this class with
the indicated arguments.
```
public class Quantity
{
    ... stuff you already saw ...

    Quantity(double Factor)
    {
        this(Factor, OpenList.nil, OpenList.nil);
    }

    ... more stuff to come ...

}

Variables in green represent values local to this constructor.

this(...) means “call the constructor of this class with
the indicated arguments.”
Convenience Quantity Constructor
where the numerator is a single unit and denominator is empty

```java
public class Quantity {
    ... stuff you already saw ...

    Quantity(double Factor, String NumUnit) {
        this(Factor, OpenList.list(NumUnit), OpenList.nil);
    }

    ... more stuff to come ...
}
```

Variables in **green** represent values local to this constructor. 
**this(...)** means “call the constructor of this class with the indicated arguments.”
Attributes of objects should never be accessed within an object simply by referring to them:

```
Quantity x = new Quantity(...);
...
System.out.println(x.Num);
```

*except* possibly for debugging purposes.

Instead, use a *getter* method:

```
int getNum()
{
    return Num;
}
...
System.out.println(x.getNum());
```
Reasons?
Static?

- What is *Static* all about?
- In Java, a method may or may not depend on a specific Object:
  - methods that do *not* depend on this state should be annotated as *static*
Static 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

- The compiler will tell you, but maybe in a cryptic way.
```java
class myBad
{
    int x;

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

    int getX()
    {
        return x;
    }

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

*Illegal: static depends on non-static*
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.
Open Lists Identified with References

Stands for the empty list, which is not == null (Spring 2002)

The list [d]

The list [c, d]

The list [b, c, d]

The list [a, b, c, d]
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, 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
Appending Open Lists

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

\[ L \text{. append}(M) \]
Reversing an Open List

- What happens when we reverse an open list?

```python
L.reverse()
```

![Diagram of a list with nodes a, b, c, and d in reverse order]
Mapping an Open List

- What happens when we map over an open list?

\[
L.\text{map}(\text{fun})
\]
OpenList Basics

- OpenList(Object First, OpenList Rest)
- L.first()
- L.rest()
- L.isEmpty()
- L.cons(Object First)
public class OpenList
{
    private Object First;
    private OpenList Rest;

    // The unique empty list
    public static OpenList nil = new OpenList();

    // Empty-list constructor (use only once)
    private OpenList() { }

// Constructor

public OpenList(Object First, Object Rest)
{
    ...
}

// Get first element of a non-empty list.

public Object first()
{
    ...
}
// Get rest of a non-empty list.

public OpenList rest()
{
    ...
}

// pseudo-constructor or "factory" for
// non-empty list

public OpenList cons(Object First)
{
    ...
}
Java Code for OpenList (4)

// emptiness test

public boolean isEmpty()
{
    
    ...

}
// pseudo-constructor or “factory” for non-empty list

public OpenList cons(Object First)
{
    ...
}

Static Methods
(closer to rex-style)

- **public static OpenList**
  `cons(Object First, OpenList First)`
  ```java
  {
    ...
  }
  ```
- **public static Object**
  `first(OpenList L)`
- **public static OpenList**
  `rest(OpenList L)`
- **public static boolean**
  `isEmpty(OpenList L)`
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, Ice-T, ...
Strings

- In contrast to long, int, float, etc., strings are already objects.

- Consequently, strings do not need extra wrappers.

- OpenLists 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:

```java
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 nm)`
  - `Double.parseDouble(String nm)`

- (Don’t use `getLong`, which has a different meaning entirely.)
Type Discrimination

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

  ```java
  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 strings, for example.

- **DO USE** `equals`:
  ```java
  if( ob1.equals(ob2) )
  ```
A Recursive List Pattern
(without using map)

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

```java
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()));
}
```
An Iterative List Pattern

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

```java
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;
}
```
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;
}
```
An Recursive Merge Pattern

- **merge two lists of Longs in increasing order**

```java
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());
}
```
Try this

- determine whether an Object occurs in a OpenList

```java
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

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