Java

Java, an Imperative Language

- Imperative languages often permit the use of functional programming.
- e.g. sometimes just say "no" to side-effects.
- Better yet, use functions and side-effects articulately and to best advantages of both.

James Gosling, Inventor of Java

Java Jive

(review)

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 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 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.
- A library ("package") Polya (http://www.cs.hmc.edu/~keller/polya) provides much of rex functionality in Java (and a similar package exists for C++).

The empty Java program

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

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

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

The empty program + one line.
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 1.3.1
- To compile:
  UNIX convention for compiler, e.g. javac, cc
  javac hello.java
- To execute:
  No "c" here.
  java hello

One-time setup for using Java on turing

In your .cshrc file, include these lines:

setenv JAVA_HOME /usr/local/jdk1.31
setenv CLASSPATH $JAVA_HOME/lib:/cs/cs60/java/:

Note: additions don’t take effect until next login or until you execute:

source ~/.cshrc

Running Java on turing

  Check what’s there.
  turing 101> ls hello.*
  hello.java
  Compile it.
  turing 102> javac hello.java
  Check what’s there now.
  turing 103> ls hello.*
  hello.class  hello.java
  Run it.
  turing 104> java hello
  hello, world
  Be astounded by results.
**Useful Hacky Shortcuts**

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

Consider putting in your .cshrc the following command definitions:

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

Example usage:

- `jc foo`  # same as javac foo.java
- `je foo`  # same as java foo
- `jx foo`  # same as javac foo.java; java foo

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

---

**Java Objects**

- Java data items are either:
  - primitive, such as int, long, float, double, char
  - Objects, such as String, Long, Double
  - Arrays are sort of like 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

- The aggregating and disciplined access properties of Objects are still very useful.
Introducing Polya

- Permits rex-like computation within Java
- Name derives from 2 things:
  - George Polya, famous mathematician who wrote about problem solving ("How to Solve It")
  - Polya package solves the problem of rapidly prototyping list-handling programs
  - Polymist (polymorphic list) data structures, as in rex

Trivial Program using Polya

```java
import polya.*;

class helloPolya {
    public static void main(String args[])
    {
        System.out.println(Polylist.list("Hello", "Polya"));
    }
}
```

A list constructor.

Output of Trivial Program

(Hello Polya)

Paren that Polya puts around lists.

Polylist.nil

```java
import polya.*;

Polylist.nil represents the empty list.
```
Using Polya to Implement Unicalc Functionality (a4)

- As in a3, we want to represent Quantities.
- In Java, instead of using a list of 3 things, we will use a little more structure:
  ```java
  public class Quantity
  {
  double Factor;
  Polylist Num;
  Polylist Denom;
  ...
  }
  ```

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 can call another.

Basic Quantity Constructor

- To define a quantity, we need to give values to all three internal variables:
  ```java
  public class Quantity
  {
  ...
  Quantity(double Factor, Polylist Num, Polylist Denom)
  {
  this.Factor = Factor;
  this.Num = Num;
  this.Denom = Denom;
  }
  ...
  }
  ```

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 ...
    Quantity(double Factor, Polylist Num) {
        this(Factor, Num, Polylist.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 both numerator and denominator are empty

```java
public class Quantity {
    // ... stuff you already saw ...
    Quantity(double Factor) {
        this(Factor, Polylist.nil, Polylist.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, Polylist.list(NumUnit), Polylist.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 Methods

- In the skeletal solution to a4, we avoid having to prefix with Polylist.
  by defining local versions of functions:

  ```java
  /**
   * local cons for convenience
   */
  static Polylist cons(Object F, Polylist R) { return Polylist.cons(F, R); }

  /**
   * local list of 1 argument, for convenience
   */
  static Polylist list(Object X1) { return Polylist.list(X1); }
  ```
Getters

- Attributes of objects should never be accessed within an object simply by referring to them:
  ```java
  Quantity x = new Quantity(...);
  System.out.println(x.Num);
  ```
  BAD
- except possibly for debugging purposes.
- Instead, use a getter method:
  ```java
  int getNum() {
    return Num;
  }
  ...
  System.out.println(x.getNum());
  ```
  GOOD

Reasons?

Static?

- What is static all about?
- In Java, a method may or may not depend on the state of 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 local to the method
  - 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.
Example

class myBad
{
    int x;

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

    getX()
    {
        return x;
    }

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

Polylist Operations

import polya.*;

class listOps
{
    public static void main(String arg[])
    {
        Polylist L = Polylist.list(new Long(1), new Long(2), new Long(3));
        Polylist M = Polylist.list("apple", "banana", "grape", "kiwi");
        System.out.println("L = " + L);
        System.out.println("L.first() = " + L.first());
        System.out.println("L.rest() = " + L.rest());
        System.out.println("L.reverse() = " + L.reverse());
        System.out.println("M = " + M);
        System.out.println("L.append(M) = " + L.append(M));
        System.out.println("L.cons(new Long(0)) = " + L.cons(new Long(0)));
        System.out.println("M.nth(2) = " + M.nth(2));
        System.out.println("Polylist.list(L, M) = " + Polylist.list(L, M));
    }
}

Polya List Operation Execution

reference: Polylist L = Polylist.list(new Long(1), new Long(2), new Long(3)); Polylist M = Polylist.list("apple", "banana", "grape", "kiwi");

L = (1 2 3)
L.first() = 1
L.rest() = (2 3)
L.reverse() = (3 2 1)
M = (apple banana grape kiwi)
L.append(M) = (1 2 3 apple banana grape kiwi)
L.cons(new Long(0)) = (0 1 2 3)
M.nth(2) = grape
Polylist.list(L, M) = ((1 2 3) (apple banana grape kiwi))