Programming Language Theory

ICS313

Nancy E. Reed
nreed@hawaii.edu

Common Lisp and Java
- Selected Java ⇔ Common Lisp feature comparisons

How Java Works
- Java Code
  - Compiler
  - Java Bytecode
  - Java Virtual Machine (Bytecode Interpreter)
  - Just-In-Time Compiler
  - Machine Language
  - CPU

How Lisp Works
- Lisp starts by loading an interpreter
  - The interpreter is a loop consisting of:
    1. read a lisp expression
    2. call EVAL on the expression read
    3. print the result
- Load – reading a file into Lisp executes all the statements in the file (definitions, variables, loading other files, …)
- Compilation
  - File at-a-time
  - Implementation dependent fast load/binary files
- Saving Lisp images
  - Binary of environment with code loaded

What You Keep from Java
- Garbage Collection
- Exception Handling
- Packages
- Dynamic Code Loading
- Runtime Binding/Typing
- Speed
- Optimizations

What You Gain with Lisp
- On-Line Programming
- Macros
- Programming Style
- Rapid Prototyping
- Object Oriented Style
- OO Inheritance
- JIT compiler

Java
- Yes
- Better
- Yes
- Yes
- Pokey

Lisp
- Yes
- Good
- Yes
- Yes
- Fast compiled
- compilers

Java
- No
- No
- Declarative
- Not Great
- Classes & Methods
- Single

Lisp/CLOS
- Yes
- Yes
- Functional+
- Great
- General Functions
- Multiple
Hello, World! Comparison

...in the file HelloWorld.java:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
javac HelloWorld.java
java HelloWorld
Hello, World!
```

Hello, World! Explained

```lisp
(defun hello-world ()
    "print a greeting"
    (princ "Hello, World!"))
(values)

(hello-world)
Hello World!
```

Syntax Basics

- Lisp and Java both treat whitespace the same except inside strings
- Lisp Comments use one or more semicolons ;
- `"Hello World! "` :function Hello world
- `// Hello World!` : No separator at the end of or between statements

Lisp: Function call

```lisp
(defun func(a s)
    (let (x)
        (declare (type double x))
        (setq x (cos a))
        x))
```

Java:

```java
public double func(double a, PrintStream s) {
    double x;
    s.println("Hello!");
    x = Math.cos(a);
    return x;
}
```

Syntax Basics

- Parentheses and a few keywords
  - Functions are usually used to return something
  - By default, last single item evaluated is returned
  - Prevent return using `(values)` [no arguments]
  - Return multiple with `(values A B C...)` [and multiple-value-bind A B C ...]
- No separator at the end of or between statements
- Item returned from function is the last one evaluated within the body of the function

### Syntax Basics

- Functions vs. expressions return values

```lisp
(setq a
(cos (+ (* pi b)
(/ a c))))
```

```java
System.out.println("My name is: "+
myName);
```

```java
double x = 40.234;
```

### Example Calculation

- The “Miraculous” Bailey-Borwein-Plouffe (BBP) Pi Algorithm

  - Can find the $n$th hexadecimal digit of \( \pi \) without knowing digits $0...n-1$!
  - Recently computed 10 billionth hexadecimal digit of \( \pi \) (it’s 9)

\[
\pi = \sum_{n=0}^\infty \frac{1}{16^n} \left( \frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right)
\]

### Java Is Strongly Typed

- In Lisp, variables, arguments, and return values need not be explicitly typed
- Values have types, not variables (unless optionally declared)

```java
public static int doSomething(float x, int[] y, myObject z)
{
...
}
```

### Data Types

- Variables, arguments, and return values

```java
{ int x;
float y = 3.14159;
int [] q = { 1, 2, 3 };
String[][] s = { {"A", "B"}, {"C", "D"} };
x = (int) y; // a cast
}
```

```java
(let* 
(x (y 3.14159)
(q #\{1 23\})
(s #\{"A" "B" \#"C" "D"\}))
(setq x (truncate y))
(y (defn doSomething (x y z))
)
```
Using Types

```plaintext
x = 4;
x++;
y = (float) x + 3.0;
z = y * (y + 4.2);
q = s + "World!";
s = "My age is " + 43;
c *= a == b ? x[a] : ++b % 3;
```

Lisp Has (but Java doesn't) ...

- Global functions
  - Available at the top level
- Global variables
  - Available at the top level
- Closures
  - Encapsulations of variables/state
- Function pointers and mapping
  - Pass functions as parameters
  - Return function as value
  - Built-in iteration functions

Control Structures

Java

```java
if...else, ? :
switch...case...
while
   do...while
   for
   break, continue
```

Lisp

```lisp
if, when
cond
(loop
   while...do...) 
   (loop 
   do...while...)
   dotimes, do, etc.
   return
tagbody
go
```

Java has labels.... ...but no goto

While

```java
while(x != 3)
{
   System.out.println("Guess again!");
   x = Integer.parseInt(reader.readLine());
}
```

```lisp
(loop while (/= x 3) do
   (terpri) (princ "Guess Again!")
   (setq x (read)))
```

For

```java
for(x=0 ; x<100 ; x++)
   System.out.println("Number: " + x);
```

```lisp
(dotimes (x 100)
   (format t "~a" x))
```

If

```java
if(x== 0)
   System.out.println("It's now 0");
```

```lisp
when (= x 0)
   (terpri) (princ "It's now 0")
```

```java
for(x0 , y = 99 ; x<100 ; x++, y--)
   if (a[x] == b[y])
      break; // found it!
```

```lisp
(dot ((x 0) (y 99 -1)) (= x 100)
   (setf (aref a x) (aref b y))
   (when (= (aref a x) 15)
      (return)))
```

```java
if(al> b[x])
   System.out.println("Fixing...");
   a[x] = b[x];
else if (a[x] < b[x])
   System.out.println("It's Less");
else
   System.out.println("It's Equal");
```

```lisp
if (> (aref a x) (aref b x))
   (terpri) (princ "Fixing")
   (setf (aref a x) (aref b x))
```

```java
if (< (aref a x) (aref b x))
   (format t "~a" x)
   System.out.println("It's Equal");
```

```lisp
if (< (aref a x) (aref b x))
   (format t "~a" x)
   System.out.println("It's Equal");
```
Multiple Selection

```
switch(x)
{
    case 1:
        System.out.println("It's a 1");
        break;
    case 2: case 3:
        System.out.println("It's a 2");
        System.out.println("...or a 3?");
        break;
    case 4:
        System.out.println("It's a 4");
        break;
    default:
        System.out.println("Yuck!");
        break;
}
```
Constructors and Finalizers

```java
public class Account {
    public int funds;
    ...

    public Account(int x) {
        funds = x; // open account with $x
    }

    public Account() {
        funds = 0; // open account with $0
    }

    protected void finalize() throws Throwable {
        System.out.println("Well, Goodbye!");
    }
}
```

Lisp/CLOS equivalent to finalize:

```lisp
(defclass account ()
    ((funds :type 'integer :accessor funds :initarg funds))
    (:default-initargs :funds 0))
```

Creating an Instance

- New instances are created using `make-instance`
- `Account a = new Account();` (make-instance 'account)
- `Account b = new Account(30);` (make-instance 'account :funds 30)

Using Instances Externally

- The methods and variables (slots) are accessed with their slot name and instance name

```java
Account a = new Account();
// open A with $0
System.out.println("A: $" + a.funds);
System.out.println("Add 100 to A: $" + a.add(100));
a.funds = a.funds * 5;
```

Passing Parameters

- Parameters are passed by copy to methods and returned from methods in both Lisp and Java
  - The value of an atomic type is simply copied.
  - A reference (pointer) to an object type is copied.

Subclasses

- An essential concept in OOP is inheritance
  - One object derives features from another object
  - Java has only single inheritance
  - Lisp/CLOS has multiple inheritance

Lisp/CLOS:

```lisp
(defclass square ()
    (height :type 'float :accessor height :initform 1.0))
(defmethod area ((s square)) (* (width s) (height s)))
```

Java:

```java
public class Square {
    public float height;
    public Square() { height=1.0; }
    public float area() { return height * height; }
}
```

```java
public class Triangle extends Square {
    public float base;
    public Triangle() { base = 1.0; }
    public float area() { return base * height * 0.5; }
}
```
The Object Class

- Because classes can only inherit from one superclass, all Java classes form a hierarchy.
- All Java classes ultimately inherit from a root class, Object.
  - If a class doesn’t say what it inherits from (using `extends`), it inherits directly from Object.

Arrays

- Single dimension only in Java.
- Constructing a preset array of integers
  ```java
  int i[] = new int[100]; // initially 0
  Account a[] = new Account[100]; // initially nil
  for(int x=0;x<100;x++)
      a[x]= new Account();
  ```

Strings

- Strings are objects in Java (the class `String`)
- Strings are sequences of Unicode characters
- Strings are not mutable
  - The `StringBuffer` class is for mutable strings

Making and Throwing Exceptions

```java
public class NoZeroException extends Exception
{
    //... elsewhere, in a different class...
    public int firstZeroInArray(int[] a) throws NoZeroException
    {
        // (dots (x (length a)) (when (zero? (aref a x)) (return x)))
        ; Uh oh... throw new NoZeroException(); // Uh oh...
    }
}
```

Strings

```java
String s = "Hello World";
(String x = s.substring(6);)
"Yo Yo Yo!".length();
```

Exception Handling

- Lisp has conditions
- Java has exceptions

```java
public void printFirstZero(int[] a)
{
    try
    {
        int firstZero = firstZeroInArray(a);
        System.out.println("First Zero At: "+firstZero);
    }
    catch (NoZeroException e)
    {
        System.out.println("No 1st zero");
    }
}
```

Strings

```java
String s = "Hello World";
(String x = s.substring(6);)
"Yo Yo Yo!".length();
```

Catching Exceptions

```java
(defn print-first-zero (a)
  (handler-case (format t "First Zero At ~s" (first-zero-in-array a))
    (no-zero-exception (e) (terpri) (princ "No 1st zero")))
  ;; or alternately...
  (defn print-first-zero (a)
    (unless (catch 'no-zero
      (format t "First Zero At ~s" (first-zero-in-array a))
    (terpri) (princ "No 1st zero")))
  )
```
### Packages

- Define namespaces in Java and Lisp
  - Very important: Java is a *really big* language
- Lisp packages are not necessarily hierarchical as Java packages are
  - Package == directory
    - Subpackage == subdirectory
    - Class/interface in package == .java file in that directory
- Lisp has both import and export (export not in Java)

### Summary

- Java ⇔ Common Lisp
- Acknowledgements: based on a presentation at AAAI by Sean Luke, University of Maryland