Smalltalk: the language

• Core language is small* and elegant
• Highly dynamic, few artificial restrictions: much like Scheme
• Invented by Alan Kay et al. at Xerox PARC in the 70's.

* But environment and libraries are somewhat complex (though still elegant), and probably quite different from what you are used to--we'll discuss those separately.
Variable bindings

Variable bindings:
   \[ x := 'hi'. \]

Variable bindings are mutable:
   \[ x := 28. \]
   \[ x := 54. \]

*changes* the original binding

- unlike ML
- more like Scheme's define special form
- Note that Smalltalk is dynamically typed
Messages

- Everything is an object
- Objects communicate via messages
- "Message send" = "virtual function call"
- Message types:
  - `x negated.` "Unary message syntax"
  - `x + 5.` "Binary message syntax"
  - `x gcd: 21.` "Keyword message syntax"
- Keyword message with multiple arguments:
  - `'Hello, world' replaceFrom: 1 to: 6 with: 'byebye' startingAt: 1.`
Syntax gotchas

• **Periods** separate statements; **semicolons** separate messages sent to the same receiver.
  • $2 + 5; \text{ negated. }$ "Evaluates $2 + 5$, then $2$ negated."

• **Strings** are single quoted; **comments** are double quoted.
  • 'This is a string'. "This is a comment"

• **All** binary messages associate **left to right**. Normal arithmetic precedence rules don't apply.
  • $2 + 3 \times 4$ "Evaluates to 20."
Closures

- Smalltalk has lexically scoped anonymous functions (a.k.a. lambdas/closures).
- Lambdas are objects, so they are evaluated by sending one of the value messages.

"Smalltalk"

\[
\begin{align*}
\text{[ 3 ].} \\
\text{[ 3 ] value.} \\
\text{[ :x :y | x + y ].} \\
a := [ :x :y | x + y ]. \\
a \text{ value: 1 value: 2.}
\end{align*}
\]

(* Rough ML equiv. *)

\[
\begin{align*}
\text{fn () => 3; } \\
(f \text{n () => 3})(); \\
\text{fn (x, y) => x + y; } \\
\text{val a = fn (x, y) => x + y;} \\
\text{a(1, 2);} \\
\end{align*}
\]
Closures and scope

• Closures are lexically scoped
• However, they may have arbitrary side effects, including the effect of changing bindings in enclosing environments:

```plaintext
i := 5.  "i gets 5"
[i := 7] value.  "i in outer scope gets 7"
[:i | i := 9] value: 2.
  "i gets 2, then 9 in local scope;"
  "i remains 7 in outer scope"
```
Closures and control

- ML and Scheme have both closures and special forms like if/then/else for control structures
- Smalltalk uses closures to implement control structures

Transcript open. "Open a Transcript window"

5 timesRepeat: [ Transcript show: 'hi'; cr. ].

x = 0 ifTrue: [ Transcript show: 'Cannot divide by zero' ]
    ifFalse: [ Transcript show: (1.0 / x) asString. ].

i := 0.
[ i < 10 ] whileTrue: [ i := i + 1. ].
Closures with many arguments are evaluated using up to 4 value: keywords:

```plaintext
seal := [ :a :b :c :d | a + b * c + d ].
seal value: 1 value: 2 value: 3 value: 4.
```

Longer argument lists use valueWithArguments:, which takes an array:

```plaintext
walrus := [ :a :b :c :d :e | a + b * c + d * e ].
walrus valueWithArguments: #( 10 20 30 40 50 ).

"Note #() syntax for arrays"
Access protection?

- Smalltalk has no access protection for methods.
- However, all member variables are accessible only to the owning instance.
- Classes inherit superclass instance variables, and can access them.
- In C++ terminology
  - **All methods are public.**
  - **All member variables are protected,**
    - except that you cannot access member variables of other objects of the same class, as in C++.
    - Ownership is "instance-based", not "class-based".
Classes are objects

• Everything is an object.
• Every object has a class.
• Classes are objects.
• So, what is the class of a class?

"Smalltalk expression"  
x := 3.  
x class.  
x class class.  
x class class class.  
x class class class class.  
x class class class class class.  

Result of printIt  
3  
SmallInteger  
SmallInteger class  
Metaclass  
Metaclass class  
Metaclass  
Metaclass class