μSmalltalk class definitions

Classes define instance variables and methods

```
(class A Object  ; A inherits from Object (root class)
  (x y)          ; Instance variables
  (method f::   (a b) ...)
  (method display ( ) ...)
)
```

A's state is an `x` and a `y`

Subclass `B` is also an `A`, plus more:

```
(class B A
  (w z)    ; additional instance variables
  (method display ( ) ...) ; redefined
  (method g:      (a) ...) ; new method
)
```

B's state is an `x` and a `y`, and also a `w` and a `z`!

`A` and `B` are displayed differently, but both have methods `f::` and `display`

Method lookup

To invoke method M on object O using class C:

1. if M is defined as part of C, use that definition
2. otherwise, invoke M on object O using class C's superclass

Walks up the inheritance relation (object hierarchy)

Normally C is O's class

Example:

```
(val a (new A))
(val b (new B))
(f:: a 2 3)          ; calls A's f::
(display a)          ; calls A's display
(f:: b 4 5)          ; call A's f:: (inherited)
(display b)          ; calls B's display (redefined)
(g: a 6)             ; error: A has no g: method
```

Method lookup is a form of dynamic binding

Data Abstraction using Objects

Smalltalk hides state but not operations:

- **Instance variables are always private: available only to methods defined on the class or its subclasses. (Actually, this is what Java, C++ call protected)**
- **Methods are public: any code can send any message to any object. "Private methods" exist but are only a programming convention, not enforced by the system**

In other object-oriented languages, programmers can control public/private for both instance variables and methods

Protocols

Class without function bodies describes “protocol”

What is an array? What responds to the protocol?

This “abstract class” serves *only* to define protocol

```
(class Array Object
  ( ) ; instance variables defined by subclasses
  (method at: (index) #subclass)
  (method at:put: (index value) #subclass)
  ...
)
```

Implementations should inherit from abstract class:

```
  (class DenseArray Array ... )
  (class SparseArray Array ...)
```

Delegation

Want to start with sparse, and later move to dense?

Use *delegation* or *composition*: Obtain code reuse *not* by inheritance, but rather by storing another instance

```
(class MixedArray Array
  (base size elts sparseFlag)
  (method initArray (base size)
        (set elts (mkSparseArray base size))
        (set sparseFlag true))
  (method at: (index) (at: elts index))
   ...
)
```

Extended Example : Financial History

Without method bodies, can see protocol

```
(class FinancialHistory Object
  (cashOnHand incomes expenditures)
  (classMethod initialBalance: (amount) ...)
  (method setInitialBalance: (amount) ...)
  (method receive:from: (amount source) ...)
  (method spend:for: (amount reason) ...)
  (method cashOnHand ( ) ...)
  (method totalReceivedFrom: (source) ...)
  (method totalSpentFor: (reason) ...)
)
```

Financial History (continued)

The class, not an instance, responds to `initialBalance:` whose purpose is to act as a constructor

Example using financial history:

```
-> (val myaccount
      (initialBalance: FinancialHistory 1000))
-> (spend:for: myaccount 50 #insurance)
-> (receive:from: myaccount 200 #salary)
-> (cashOnHand myaccount)
1150
-> (spend:for: myaccount 100 #books)
-> (cashOnHand myaccount)
1050
-> (spend:for: myaccount 200 #books)
-> (cashOnHand myaccount)
850
-> (totalSpentFor: myaccount #books)
300
```

Class methods and object creation

Because every class is also an *object*
you can send initialization messages to the class

All classes inherit `new` from class `Class`

Example:

```
(class FinancialHistory Object
    ...
    (classMethod initialBalance: (amount)
        (setInitialBalance: (new self) amount))
)
```

First look at `self`, which stands for receiver of a message (same as "`this`" in C++/Java)

Implementing Financial History

```
(class FinancialHistory Object
 (cashOnHand incomes expenditures)
 (method setInitialBalance: (amount)
   (set cashOnHand amount)
   (set incomes       (new Dictionary))
   (set expenditures (new Dictionary))
   self)
 (method receive:from: (amount source)
   (at:put: incomes source
     (+ (totalReceivedFrom: self source) amount))
   (set cashOnHand (+ cashOnHand amount)))
 (method spend:for: (amount reason)
   (at:put: expenditures reason
     (+ (totalSpentFor: self reason) amount))
   (set cashOnHand (- cashOnHand amount)))
```

Implementing Financial History

```
 (method cashOnHand ( ) cashOnHand)
 (method totalReceivedFrom: (source)
  (if (includesKey: incomes source)
      [(at: incomes source)]
      [0]))
 (method totalSpentFor: (reason)
  (if (includesKey: expenditures reason)
      [(at: expenditures reason)]
      [0]))
 (classMethod initialBalance: (amount)
  (setInitialBalance: (new self) amount))
)
```

Naming and Scope rules

Much implicit notation!

(strength, but also weakness, of Smalltalk/C++ approach)

Every method has *implicit argument* `self`

References to instance variables implicitly refer to `self`

Can be looked up, set

Scope for variables is static (as in Impcore, Scheme)

Scope for message sends (methods) is dynamic

- based on class hierarchy

Method lookup: `(f x1 x2 x3 ...)`

If the receiver `x1` is object of class C, and

`f` is a method of C, or

`f` is a method of an ancestor of C

then use the method with `self = x1`

Otherwise, run-time error

Method Lookup

Crucial: lookup begins with the class of the *receiver*

class of sender (caller) doesn't matter

```
-> (class A Object ( )
     (method whatis ( ) (isa self))
     (method isa ( ) #A))
-> (class B A ( )
     (method isa ( ) #B))
-> (val x (new A))
-> (val y (new B))
-> (whatis x) ; self is x, an A
A
-> (whatis y) ; self is y, a B
B
```

Works even though both times the A `whatis` method is called

Superclass invokes `isa` method defined in subclass

- **no recompilation required! (A defined before B)**

Reusing superclass methods

Methods defined in superclass can be used within subclasses

```
(class DeductibleHistory FinancialHistory
 (deductible)
 (classMethod initialBalance: (amount)
   (initDeductibleHistory
     (initialBalance: super amount)))
 (method initDeductibleHistory ( )
    (set deductible 0)
    self)
 (method spend:Deduct: (amount reason)
    (spend:for: self amount reason) ; call method from superclass
    (set deductible (+ deductible amount)))
 (method spend:for:deduct: (amount reason deduction)
    (spend:for: self amount reason)
    (set deductible (+ deductible deduction)))
 (method totalDeductions ( ) deductible))
```

Using Deductibles

```
-> (val myaccount
      (initialBalance: DeductibleHistory 1000))
-> (spend:for: myaccount 50 #insurance)
-> (receive:from: myaccount 200 #salary)
-> (cashOnHand myaccount)
1150
-> (spend:Deduct: myaccount 100 #mortgage)
-> (cashOnHand myaccount)
1050
-> (totalDeductions myaccount)
100
-> (receive:from: myaccount 200 #salary)
-> (spend:Deduct: myaccount 25 #donation)
-> (cashOnHand myaccount)
1225
-> (totalDeductions myaccount)
125
```

Messages to super

`(f super x2 x3 ... xn)`

- the message is sent to `self`,
- but method lookup starts in the superclass of class where `super` appears (a *static* binding!)

Purpose: access to overridden (redefined) methods

- especially useful for class methods, e.g., initialization

Example of messages to super

```
 (class A Object ( )
   (method isa ( ) #A))
 (class B A ( )
   (method isa ( ) #B)
   (method whatis ( ) (isa self))
   (method bypass ( ) (isa super))) ; always invoke method in A
 (class C B ( )
   (method isa ( ) #C))
 (val y (new B))
 (val z (new C))
 (whatis y)
B
 (whatis z)
C
 (bypass y)
A
 (bypass z)
A
```