

Object-Orientation

Basic ideas:

- **data abstraction** (may or may not hide information)
- **inheritance**
reuse of implementations
- **subtyping**
reuse of interfaces (“protocols”)
- (disciplined) **dynamic binding** of functions

Key goal is **reuse**

Languages:

Simula (1967)

Smalltalk (1980)

LISP Flavors, Loops, CLOS,...

C++, Objective C,...

Eiffel, OWL,...

PostScript in NeWS

Modula-3, Oberon, Ada 95,...

Python, Java

:

The language fad of the 90s.

Smalltalk

Pure object-oriented language:

- **Everything** is an object
- Even values like `true`, `false`, `3`, `92` are objects
- **Classes** are objects!
- There are **no functions**—only objects

Limited control structures:

- **message passing**
- **sequential execution**

(Full Smalltalk also has “nonlocal return.”)

Smalltalk does not have `if` or `while`

- they are simulated with objects

Abstract syntax

μ Smalltalk expressions:

```
datatype exp = VAR      of name
              | SET      of name * exp
              | BEGIN    of exp list
              | SEND      of name * exp * exp list
              | BLOCK    of name list * exp
              | LITERAL  of rep
```

VAR, SET, and BEGIN as in Impcore.

SEND is message passing:

- Method is specified by **name**
- Always sent to a **receiver**
- Optional **arguments**

BLOCK and LITERAL are special objects.

Protocol—the interface to an object

Object's **protocol** \equiv set of messages it can respond to

Protocol for all objects (defined on class `Object`):

<code>isKindOf: aClass</code>	Receiver inherits from arg?
<code>isMemberOf: aClass</code>	Receiver's class is arg?
<code>= anObject</code>	Equality
<code>!= anObject</code>	Inequality
<code>isNil</code>	Receiver is nil?
<code>notNil</code>	Receiver is not nil?
<code>print</code>	Print receiver.
<code>println</code>	Print receiver, then newline.
<code>error: aSymbol</code>	Error message
<code>subclassResponsibility</code>	Missing method

Simple examples

Every object inherits methods from Object

```
-> (isKindOf: 3 Object)
<True>
-> (isMemberOf: 3 Object)
<False>
-> (isNil 3)
<False>
-> (isNil nil)
<True>
-> (println nil)
nil
nil
-> true
<True>
-> True
<class True>
-> Object
<class Object>
```

Inheritance and Subtyping

“T inherits from Super”

“T is a subtype of Super”

“T extends Super”

Smalltalk, C++

Modula-3

Oberon

T is the

- subclass
- derived class
- subtype

Super is the

- superclass
- parent class
- supertype

Inheritance:

T inherits Super's

- state (instance variables)
- operations (methods)

and therefore, also inherits

- protocol

Supertype's methods can be

- “redefined” (Smalltalk)
- “overridden” (Modula-3)

Subtyping: may use a T wherever a Super is expected

Making subtyping precise

Subtyping is always *transitive*

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

Subtyping is frequently *reflexive* and *antisymmetric*.
(partial order)

Key rule is **subsumption rule**:

$$\frac{e : \tau \quad \tau <: \tau'}{e : \tau'}$$

In Smalltalk, $\tau <: \tau'$ iff protocol τ contains every message in protocol τ'
(fairly useless in dynamically typed system)

In C++, Modula-3, Java, $\tau <: \tau'$ iff τ is a subclass of
(inherits from) τ'

Decouple subtyping, inheritance? Experiment...

μ Smalltalk classes

Classes define **instance variables and methods**

```
(class A Object ; Object is root of hierarchy
  (x y) ; Instance variables
  (method f (a b) (begin ...))
  (method display () (begin ...))
)
```

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 () (begin ...)) ; redefined
  (method g (a) (begin ...)) ; new method
)
```

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

Methods:

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 (includes subclasses).
- Methods are **public**: any code can send any message to any object.
“Private methods” exist but are only a programming convention

In other object-oriented languages, programmers can control public/private

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) ...)
)
```

The **class**, not an **instance**, responds to `initialBalance:`

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
-> (totalSpentFor: myaccount #books)
100
```

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`: stands for receiver of a message

Implementing Financial History

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

Reusing superclass methods

Can often define methods in superclass
to be used in many subclasses

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

Using Deductibles

```
-> (val myaccount (mkDeductibleHistory 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
```

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!*)

Example:

```
(class A Object ()  
  (method isa () #A))  
(class B A ()  
  (method isa () #B)  
  (method whatis () (isa self))  
  (method bypass () (isa super))) ; always to 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

Purpose: access to overridden (redefined) methods

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

Control flow — blocks and Booleans

Blocks are **closures!**

- (block (*formals*) *expressions*)
- Special syntax for parameterless blocks:
[*expressions*]

Blocks are **objects**

- Cannot apply a block, but can send it the `value` message

```
-> (val twice (block (n) (+ n n)))
```

```
<Block>
```

```
-> (value twice 3)
```

```
6
```

Booleans work in continuation-passing style

- message to Boolean gives action for true, false

Example: minimum

```
-> (val x 10)
```

```
-> (val y 20)
```

```
-> (ifTrue:ifFalse: (<= x y) [x] [y])
```

```
10
```

Comparison produces Boolean, which gets message, 2 blocks

Protocol for the Booleans

<code>ifTrue:ifFalse: trueBlock falseBlock</code>	Full conditional
<code>ifTrue: trueBlock</code>	Part conditional (for side effect)
<code>ifFalse: falseBlock</code>	Part conditional (for side effect)
<code>& aBoolean</code>	Conjunction
<code> aBoolean</code>	Disjunction
<code>not</code>	Negation
<code>eqv: aBoolean</code>	Equality
<code>xor: aBoolean</code>	Difference
<code>and: altBlock</code>	Short-circuit conjunction
<code>or: altBlock</code>	Short-circuit disjunction

Implementation of the Booleans

Booleans work by having *classes* `True` and `False`, each with 1 value

Need 2 classes so we can have 2 method definitions:

```
(class True Boolean ()
  (method ifTrue:ifFalse: (trueBlock falseBlock)
    (value trueBlock))
)
(class False Boolean ()
  (method ifTrue:ifFalse: (trueBlock falseBlock)
    (value falseBlock))
)
```

All other methods implementable in terms of

`ifTrue:ifFalse:`

```
(class Boolean Object ()
  (method ifTrue:ifFalse: (trueBlock falseBlock)
    (subclassResponsibility self))
  (method ifTrue: (trueBlock)
    (ifTrue:ifFalse: self trueBlock []))
  (method not ()
    (ifTrue:ifFalse: self [false] [true]))
  ...
)
```

Blocks: more control flow

Can also send block a message requiring looping:

```
-> (val x 10)
-> (val y 20)
-> (whileTrue: [(<= x (* 10 y))]
    [(set x (* x 3))])
```

nil

```
-> x
```

270

Protocol for blocks:

value arguments

Evaluate, answer
value of body

`whileTrue: bodyBlock`

Send `value` to the
receiver, and if answer
is true, send `value` to
`bodyBlock` and
repeat.

`whileFalse: bodyBlock`

Repeat body while
(`value receiver`)
answers false

Class Hierarchies

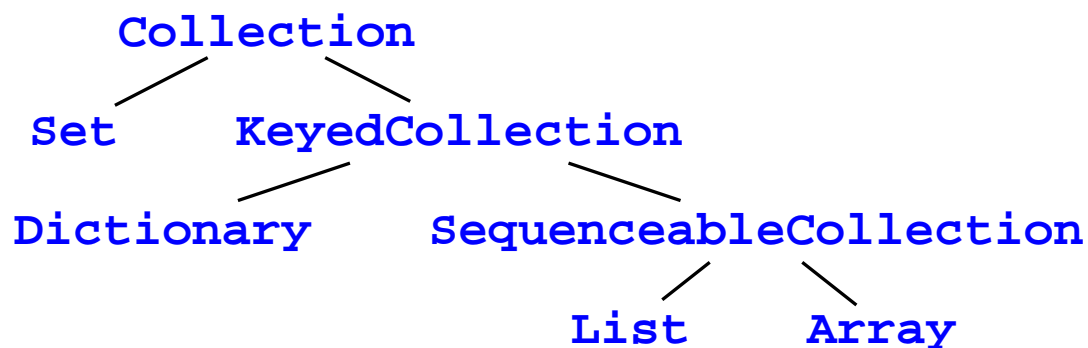
Key to successful object-oriented programming is **good class hierarchy**

Real Smalltalk book is 90pp on language,
300 pp on library

Famous Smalltalk **browser** helps navigate library

Many useful classes are abstract, to define ADTs

The “collection hierarchy”



Collection

Set

KeyedCollection

Dictionary

SequenceableCollection

List

Array

contain things

objects in no particular order

objects accessible by keys

any key

keys are consecutive integers

can grow and shrink

fixed size, fast access

Collection protocol

Mutators:

`add: newObject` **Add argument**

`addAll: aCollection` **Add every element of arg**

`remove: oldObject` **Remove arg, error if absent**

`remove:ifAbsent: oldObject exnBlock` **Remove the argument, evaluate exnBlock if absent**

`removeAll: aCollection` **Rm every element of arg**

Observers:

`isEmpty` **Is it empty?**

`size` **How many elements?**

`includes: anObject` **Does receiver contain arg?**

`occurrencesOf: anObject` **How many times?**

`detect: aBlock` **Find and answer element satisfying aBlock (cf μ Scheme exists?)**

`detect:ifNone: aBlock exnBlock` **Detect, recover if none**

`asSet` **Set of receiver's elements**

More collection protocol

Iterators:

do: aBlock For each element **x**, evaluate
(value aBlock **x**).

inject:into: thisValue binaryBlock
Essentially μ Scheme foldl

select: aBlock **Essentially μ Scheme filter**

reject: aBlock **Filter for *not* satisfying** aBlock

collect: aBlock **Essentially μ Scheme map**

Implementing Collections

Subclass need only provide `do:`, `add:`,

`remove:ifAbsent:` **and private method** `species`

```
(class Collection Object
  () ; abstract
  (method do: (aBlock)
    (subclassResponsibility self))
  (method add: (newObject)
    (subclassResponsibility self))
  (method remove:ifAbsent (oldObject exnBlock)
    (subclassResponsibility self))
  (method species ()
    (subclassResponsibility self))
  ⟨other methods of class Collection⟩
)
```

All other methods given in terms of these:

```
⟨other methods of class Collection⟩=
(method addAll: (aCollection)
  (begin
    (do: aCollection (block (x) (add: self x)))
    aCollection))
(method size () (temp)
  (begin (set temp 0)
    (do: self (block (_) (set temp (+ temp 1))))
    temp))
```

These methods always work

Subclasses can override (redefine) with more efficient versions

Implementing Collections, continued

Use `species` to create “collection like the reciever.”

Example: filtering

```
<other methods of class Collection>=  
(method select: (aBlock) (temp)  
  (begin  
    (set temp (new (species self)))  
    (do: self (block (x)  
      (ifTrue: (value aBlock x)  
        [(add: temp x)])))  
    temp))
```

Sets (by delegation to Lists)

```
(class Set Collection
  (members) ; list of elements
  (classMethod new () (initSet (new super)))
  (method initSet () ; private method
    (begin
      (set members (new List))
      self))
  (method do: (aBlock) (do: members aBlock))
  (method remove:ifAbsent: (item exnBlock)
    (remove:ifAbsent: members item exnBlock))
  (method add: (item)
    (begin
      (ifFalse: (includes: members item)
        [(add: members item)])
      item))
  (method species () Set)
  (method asSet () self) ; extra efficient
)
```

Most subclass methods work by delegating all or part of work to list members

N.B. Set is a *client* of List, not a subclass!

Double Dispatch

Typical object-orientation:

Code you execute depends on class of the **receiver** (first argument)

What if you need to choose code based on **both receiver and argument**?

Solution: use method name to encode **both operation and type of argument**

Example: mixed arithmetic

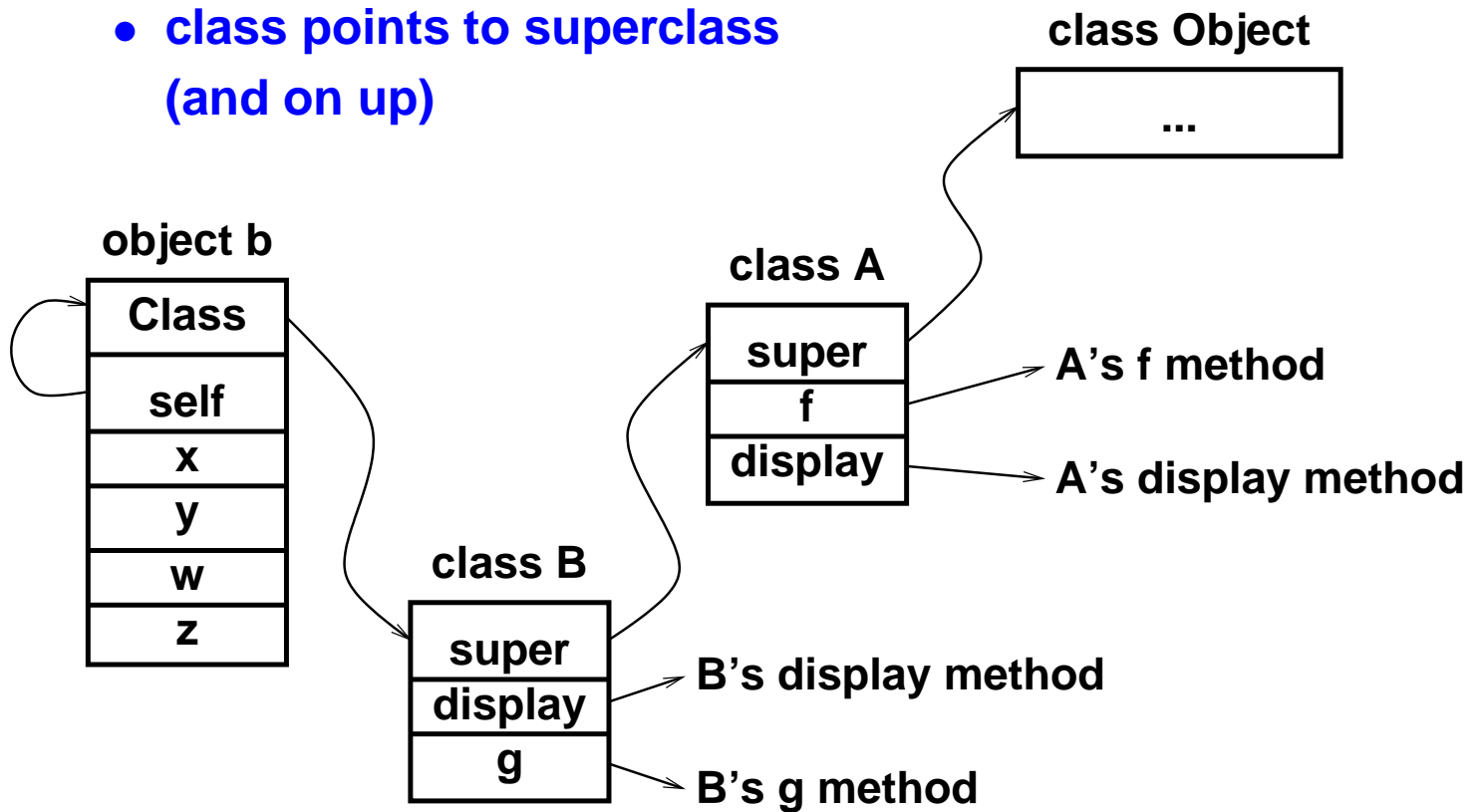
`addIntegerTo:`

`addFloatTo:`

Implementing Smalltalk

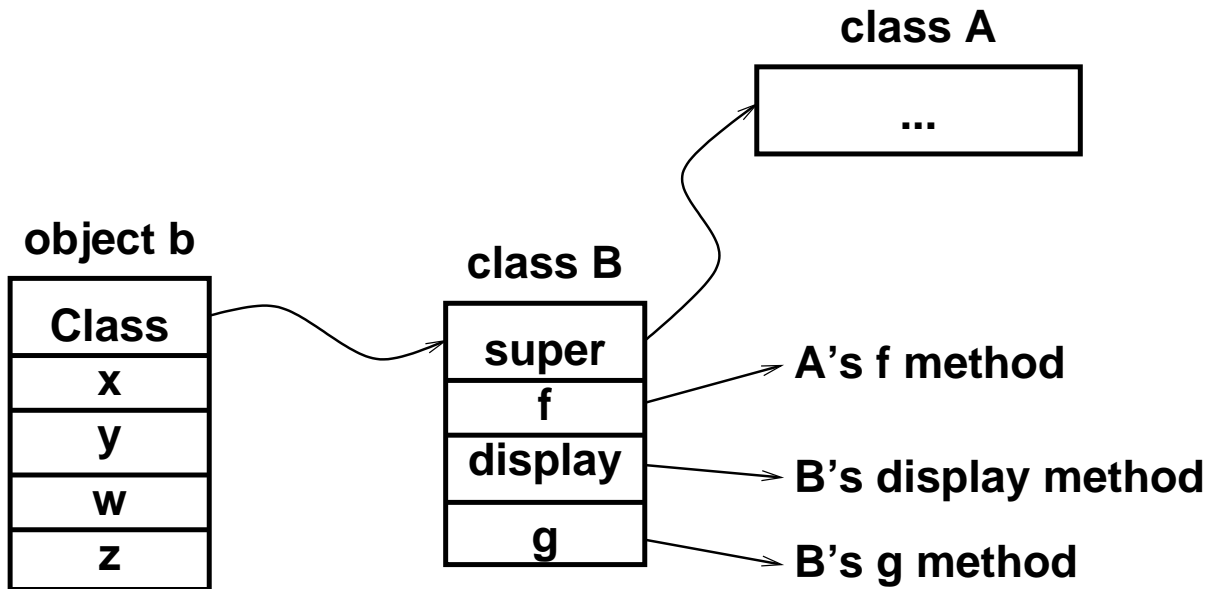
Basic idea:

- store **state with objects**
- store **methods with class**
- **object points to class**
- **class points to superclass**
(and on up)



Compiling objects

In compiled system (Modula-3, C++), precompute method lookup:



All method offsets known at link time (or compile time)

In Smalltalk, any object can respond to any message:
too many messages for tables

(“vtables” with hundreds or thousands of entries?!)

Classes defined and constructed at compile time:

```
datatype class
```

```
= CLASS of
```

```
{ name      : name
, super    : class option
, ivars    : name list      (* instance vars *)
, methods  : method env    (* exported & private *)
, id       : int           (* unique id *)
}
```

Methods can be primitive or user-defined

```
and method
```

```
= PRIM_METHOD of
```

```
name * (value * value list -> value)
```

```
| USER_METHOD of
```

```
{ name      : name,          formals : name list
, temps    : name list,     body    : exp
, superclass : class (* to send to super *)
}
```

μ Smalltalk implementation—Objects

Value is pair of (class, rep)

Rep is integer, symbol, or collection of instance variables.

and rep

```
= USER      of value ref env  (* instance vars *)
| ARRAY     of value Array.array
| NUM       of int
| SYM       of name
| CLASSREP  of class  (* classes are objects *)
| CLOSURE   of      (* blocks *)
```

```
      name list * exp * value ref env * class
withtype value = class * rep
```

Most objects are USER objects

Environments hold *mutable* cells: value ref.

μ Smalltalk implementation—instance creation

New instance variables are all nil.

```
local
  fun mkIvars (CLASS { ivars, super, ... }) =
    let val supervars =
        case super
        of NONE    => emptyEnv
         | SOME c  => mkIvars c
        in fun add (n, rho) = (* alloc new slot for n *)
            bind(n, ref nilValue, rho)
        end
    in foldl add supervars ivars
    end
in
  fun newUserObject (_, c) =
    let val ivars = mkIvars c
        val self = (c, USER ivars)
    in (find("self", ivars) := self; self)
    end
end
```

new assigns to self—makes it point to value itself!

Two environments: ρ and ξ (locals and globals)

superclass **governs messages to** super

```
fun eval(e, rho, superclass, xi) = let
  fun findMethod (name, class) = ...
  fun evalMethod (m, receiver, actuals) = ...
  fun evalClosure ((formals, body, rho,
                    superclass), actuals) = ...

  fun ev(VAR v) =
    !(find(v, rho)
      handle NotFound _ => find(v, xi))
  | ev(SET (n, e)) =
    let val v = ev e
        val cell = find(n, rho)
        handle NotFound _ => find(n, xi)
    in cell := v; v
    end
  | ev(VALUE v) = v
  | ev(LITERAL c) =
    (case c of NUM n => mkInteger n
             | SYM n => mkSymbol n
             | _ => ... error ...)

  ...
in ev e
end
```

More evaluation—method dispatch

```
fun ev(SEND (message, receiver, args)) =
  let val obj as (class, rep) = ev receiver
      val args = map ev args
      val dispatchingClass =
        case receiver of SUPER => superclass
                       | _      => class
      in case (message, rep)
          of ("value", CLOSURE clo) =>
              evalClosure(clo, args)
           | _ =>
              evalMethod (
                findMethod(message, dispatchingClass),
                obj, args))
          end
  end
```

Where

```
fun findMethod (name, class) =
  let fun fm (CLASS { methods, super, ...}) =
        find (name, methods)
        handle NotFound m =>
          case super
           of SOME c => fm c
            | NONE   => ... error ...
      in fm class
      end
```

Evaluating a method

Put together correct environment

```
fun evalMethod (PRIM_METHOD (name, f),
               receiver, actuals) =
  f (receiver, actuals)
| evalMethod (USER_METHOD { name, superclass,
                             formals, temps, body },
             receiver, actuals) =
  let val rho = instanceVars receiver
      val rho = bindList(formals, map ref actuals, rho)
      val rho = bindList(temps,
                        map (fn _ => ref nilValue) temps, rho)
  in eval(body, rho, superclass, xi)
  end
and instanceVars (_, USER rep) = rep
| instanceVars self = bind("self", ref self, emptyEnv)
```

Class Methods (and variables)

Can associate functions, state with **class**, not just instance

Class variable: allocated and associated once per class

- only in full Smalltalk (not μ Smalltalk)
- available to every object of that class
- like C 'static' variables, Algol `own` variables
- 'global variables' \equiv class variables of `Object`

Class method: defined on class

- send message to class
- typically used for object creation (`new`)
- also to initialize class variables (e.g., unique symbol)

In Smalltalk, classes are objects of their **metaclasses**

- class methods are methods of the metaclass
- class variables are instance vars of the metaclass
- `new` is a method of the metaclass

Try "meta-object protocol" and things get weird fast

- redefine your language on the fly!

Real Smalltalk Syntax

Receiver of message first, then message name, args

```
a at: i
```

for array or dictionary lookup

```
n + 1
```

sends message + to object n with argument 1!

More than 1 argument uses mixfix syntax:

```
a at: i put: x
```

assigns to an array.

Defaults avoid excessive parentheses (e.g., Booleans)

```
x <= y
```

```
ifTrue: [Transcript show: x]
```

```
ifFalse: [Transcript show: y]
```

```
[notNil item] whileTrue: [item := next D]
```

Two kinds of message names:

- “Symbolic” always binary (infix)
- “Unary/keyword,” number of colons = number of args

Side issue: objects and closures are equivalent

Recall random-number function used a closure:

```
(val init-rand (lambda (seed)
  (lambda ()
    (set seed (mod (+ (* seed 9) 5) 1024)))))
```

We can simulate it using an object:

```
-> (class Rand Object
  (seed)
  (classMethod new: (s) (seed: (new self) s))
  (method seed: (s) ; private
    (begin
      (set seed s)
      self))
  (method value ()
    (set seed (mod (+ (* seed 9) 5) 1024))))
```

```
-> (val r (new: Rand 1))
```

```
<Rand>
```

```
-> (value r)
```

```
14
```

```
-> (value r)
```

```
131
```

Objects and closures are equivalent, cont'd

Simulate array object using closure:

This is gross!!! needs to be based on a real example, perhaps Set???

```
(method mkArray (base size)
  (let ((elts (zerolist size)))
    (lambda (operation)
      (if (= operation 'at:)
          (lambda (i)
            (if (out-of-bounds size base i) 0
                (nth (- i base) elts)))
          (if (= operation 'at:put:)
              (lambda (i x)
                (if (out-of-bounds size base i) '()
                    (set elts
                        (changenth (- i base) x elts)))
                ))
              'error!))))))
```

```
-> (val A (mkArray 3 10))
-> ((A 'at:) 7 99)
-> ((A 'at:put:) 7)
99
-> ((A 'at:) 8)
0
```

Example of *dynamic dispatch* on name of method

- can be done more cleanly in full Scheme

- (see Abelson and Sussman)

Smalltalk — Assessment

A pathbreaker several ways:

- pure object-oriented language
- rich class hierarchy
- class browser
- rich programming environment on personal computer
- first serious generational garbage collector (GC)

Object-orientation, window systems quickly imitated
(especially on LISP machines)

Smalltalk a significant commercial success

- not a monster, but still selling systems

Remarkable results in

- simulation
- prototyping
- teaching young people to program

For big projects, want less dynamic system:

- some static checking
- hybrid languages (only some values are objects)

Object-oriented influence: Self, C++, Modula-3, Ada 95, Java