μSmalltalk extended example: Stack

First define the Node class

```
(class Node Object (data next)
        (method data ( ) data)
        (method next ( ) next)
        (method data: (x) (set data x))
        (method next: (x) (set next x))
)
```

Next define the Stack class

μSmalltalk Stack example (continued)

```
(class Stack Object (top)
        (method isEmpty ( ) (isNil top))
        (method push: (x) (locals z)
                (begin (set z (new Node))
                        (data: z x)
                        (next: z top)
                        (set top z)
                       x))
        (method pop ( ) (locals x)
                (if (isEmpty self) [nil]
                        [(set x (data top))
                         (set top (next top))
                         x]))
        (method display ( ) (locals z)
                (begin (set z top)
                        (while [(notNil z)]
                                [(println (data z))
                                 (set z (next z))]))))
)
```

μSmalltalk Stack example (continued)

```
-> (val S (new Stack))
<Stack>
-> (isEmpty S)
<True>
-> (push: S #a)
a
-> (push: S #b)
b
-> (push: S #c)
c
-> (isEmpty S)
<False>
-> (display S)
c
b
a
nil
```

μSmalltalk Stack example (continued)

```
-> (pop S)
c
-> (pop S)
b
-> (pop S)
a
-> (isEmpty S)
<True>
-> (pop S)
nil
```

μSmalltalk Stack example (continued)

Now define a subclass Multistack of superclass Stack

```
(class Multistack Stack ( )
        (method multi:push: (n x)
                (ifTrue: (> n 0)
                        [(push: self x)
                         (multi:push: self (- n 1) x)]))
        (method multi:pop (n)
                (ifTrue: (> n 0)
                        [(println (pop self))
                         (multi:pop self (- n 1))]))
)
```

μSmalltalk Stack example (continued)

```
-> (val M (new Multistack))
<Multistack>
-> (multi:push: M 4 #a)
nil
-> (multi:push: M 2 #b)
nil
-> (multi:pop M 4)
b
b
a
a
nil
-> (multi:pop M 4)
a
a
nil
nil
nil
```

Class Methods (and variables)

Can associate functions and state with class, not just instance

Class variable: allocated and associated once per class

- only in full Smalltalk (not μSmalltalk, but see exercise)
- available to every object of that class
- like C/C++/Java '`static`' variables, Algol `own` variables
- 'global variables' $\equiv$ class variables of root class `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 one argument uses mixfix syntax:

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

assigns to an element of the array.

Real Smalltalk Syntax (continued)

Default precedence avoids excessive parentheses (e.g., Booleans)

```
x <= y
  ifTrue:  [Transcript show: x]
  ifFalse: [Transcript show: y]
```

Three kinds of message names:

- **Unary (ordinary identifier, postfix) — highest precedence**
- **Binary (operator symbol, infix)**
- **Keyword (number of colons = number of args) — lowest precedence**

Another example:

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

Real Smalltalk extended example: Stack

Real Smalltalk code is written using the menu-driven “browser” with a point-and-click interface. This code has been tested using the Squeak system, which is based on Smalltalk-80.

```
Object subclass: #Node
    instanceVariableNames: 'data next'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'MyClasses'

data
    ^data
next
    ^next
data: x
    data := x
next: x
    next := x
```

Real Smalltalk Stack example (continued)

```
Object subclass: #Stack
    instanceVariableNames: 'top'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'MyClasses'

isEmpty
    ^top isNil
push: x
    | z |
    z := Node new.
    z data: x.
    z next: top.
    top := z.
    ^x
```

Real Smalltalk Stack example (continued)

```
pop
    | x |
    self isEmpty ifTrue: [^nil]
                 ifFalse: [x := top data. top := top next. ^x]
display
    | z |
    z := top.
    [z notNil] whileTrue:
        [Transcript show: z data. Transcript cr. z := z next]
```

Real Smalltalk Stack example (continued)

```
Smalltalk at: #S put: Stack new
S
    a Stack
S isEmpty
    true
S push: #a
    #a
S push: #b
    #b
S push: #c
    #c
S isEmpty
    false
S display
    c
    b
    a
```

Real Smalltalk Stack example (continued)

```
S pop
    #c
S pop
    #b
S pop
    #a
S isEmpty
    true
S pop
    nil
```

Real Smalltalk Stack example (continued)

```
Stack subclass: #Multistack
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'MyClasses'

multi: n push: x
    n>0 ifTrue:
        [self push: x. self multi: n-1 push: x]

multipop: n
    n>0 ifTrue:
        [Transcript show: self pop. Transcript cr.
         self multipop: n-1]
```

Real Smalltalk Stack example (continued)

```
Smalltalk at: #M put: Multistack new
M
    a Multistack
M isEmpty
    true
M multi: 4 push: #a
M multi: 2 push: #b
M multipop: 4
    b
    b
    a
    a
M multipop: 4
    a
    a
    nil
    nil
```

Side issue: objects and closures are equivalent

Recall random-number function in Scheme used a closure:

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

We can simulate the above using an object: see exercise

Objects and closures are equivalent (continued)

Simulate Smalltalk's Set class in μScheme using closure:

```
(define mkSet ( )
  (let* ((members (mkList))
         (super   (mkCollection)))
    (lambda (operation)
      (if (= operation 'do:)
        (lambda (block)
           ((members 'do) block))
      (if (= operation add:)
        (lambda (item)
           (if ((members 'includes:) item)
               ((members 'add:) item)
               item))
      ...
      (error 'unknown-message-to-set))))))
```

Objects and closures are equivalent (continued)

```
-> (val s (mkSet))
-> ((s 'add:) 99)
-> ((s 'add:) 33)
-> ((s 'member:) 8)
#f
-> ((s 'member:) 99)
#t
```

Example of *dynamic dispatch* on name of method

- **cleaner in full Scheme (Abelson and Sussman 1985)**

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 it

Smalltalk a significant commercial success

- **not a monster, but still selling systems**
- **resurgence with free “Squeak”: `www.squeak.org`**

(especially on LISP machines)

Smalltalk : Assessment (continued)

Remarkable results in

- **simulation**
- **prototyping**
- **teaching young people to program**

For big projects, often want less dynamic system:

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

O-O influence: Self, C++, Modula-3, Ada 95, Python, Java, C#