CS301
Session 16

Agenda

- Introduction to μSmalltalk
- Pure object-oriented programming

Smalltalk

- Smalltalk: the original OO language
- *All values* in Smalltalk are objects, even numbers and booleans
- Other than message send (or method invocation) control flow mediated by boolean and block objects
- Blocks are closures and can be recursively defined at the top level

Object-oriented programming

- Language constructs: *objects* and *classes*
- Mechanisms: *inheritance* and *dynamic dispatch*
- Principles: *data encapsulation* and *code re-use*

Related languages

- Precursor: Simula
- Languages with OO features: CLOS, C++, OCaml, Eiffel, Python, Java, C#, even Visual Basic, many others
- OO is the language paradigm *du jour*

What is an object?

- An entity that responds to *messages* by changing its state and/or *answering* with a value
- An object is represented by a collection of
  - instance variables (private) that constitute its *state*
  - methods (public) that specify its response to messages
- Arguably, objects alone are enough for "pure" object-oriented programming

Adding classes

- Objects provide encapsulation and message handling
- Classes add *code re-use:* all members of the same class share the same methods
- Again, arguably we could stop there and have a meaningful OO language

Adding inheritance

- Inheritance creates a potentially complex web of code reuse
- Mechanisms: *subclassing* and *dynamic dispatch*
- Subclassing is *transitive*
- A subclass inherits the instance variables and methods of its superclass(es)
- A subclass may override (redefine) an inherited method

Dynamic dispatch

- How a message is handled is determined at runtime:
  - If there is a method defined in the receiver's class for the message, use it
  - Otherwise, search upward in the class hierarchy
- Consequence: the meaning of a message can't be determined statically
- *protocol* of an object: the messages it responds to - determined by its class and superclasses

self and super

- *Not* variables! `self` always refers to the receiver
- `super` always refers to the receiver, but dynamic dispatch is not used; instead:
  - Search upward in the class hierarchy for the method, starting in the superclass *of the class where* `super` *appears in the source.*
  - Result: method is known statically!

The method "new"

- `new` is *not* a keyword - a method in class Class responsible for creating instance variables
- Sometimes we override it, but it's not a good idea to omit "new super":

```
-> (class Bar Object (x)
    (classMethod new ())
    (method x () x))
<class Bar>
-> (val bar (new Bar))
nil
-> (x bar)
run-time error: UndefinedObject does not understand
message x
```


Variable names

- Familiar static scope rules; in order of precedence:
  - locals
  - method parameters
  - instance variables
  - globals

Example

- Recall random numbers from the midterm:

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

- Here it is in μSmalltalk:

```
(class Random ; class name
   Object ; superclass name
   (seed) ; instance variable
   ; a "constructor" with a parameter
   (classMethod new: (s) (initSeed: (new Random) s))
   (method initSeed: (s) (set seed s) self)
   ; the only "public" method
   (method next () (set seed (mod: (+ (* seed 9) 5) 1024))))
```


What was inherited?

- The class `Random` inherits the class method `new` (as well as `protocol` and `localProtocol`)

```
-> (localProtocol Random)
(classMethod new: (s) ...)
(method initSeed: (s) ...)
(method next () ...)
<class Random>
```

- Objects of class `Random` inherit methods from `Object`

Inherited methods

- Some examples:

```
-> (isMemberOf: r1 Random)
<True>
-> (isKindOf: r1 Object)
<True>
-> (isKindOf: r1 Number)
<False>
-> (= r1 r1)
<True>
-> (isNil r1)
<False>
```


Example(2)

- Remember infinite sequences in μScheme:

```
(define mk-seq (f n)
  (cons n (lambda () (mk-seq f (f n)))))
```

- A similar (but imperative!) μSmalltalk definition:

```
(class InfiniteSequence Object
 (generator ; generator block
  current) ; current element
 (classMethod new:from:by (first aBlock)
   (initInfiniteSeq:: (new self) first aBlock))
 (method initInfiniteSeq:: (first aBlock)  ; private
   (set generator aBlock) (set current first) self)
 (method current () current)
 (method next () (set current (value generator current))))
```

Using sequences

- The even numbers:

```
-> (val evens (new:from:by
InfiniteSequence 0 (block (n) (+ n 2))))
<InfiniteSequence>
-> (next evens)
2
-> (current evens)
2
-> (next evens)
4
```


Sequences for random numbers

- Define a subclass:

```
(class Random
       InfiniteSequence ; superclass
       () ; no extra instance variables
      (classMethod new: (s)
       (new:from:by super
         s
         (block (s) (mod: (+ (* s 9) 5) 1024)))))
```


Boolean objects

- Example:

```
-> (if (= 0 1) [#bad!] [#good!])
good!
-> (if (= 0 1) #bad! #good!)
run-time error: Symbol does not understand message
value
Method-stack traceback:
  Sent 'value' in initial basis, line 25
  Sent 'ifTrue:ifFalse:' in initial basis, line 19
  Sent 'if' in standard input, line 38
-> (if (= 0 1) #bad! [#good!])
good!
```


Block objects

- Methods are "value" and "while"

```
-> (set n 2)
2
-> (begin (while [(< n 50)] [(set n (* 2 n))]) n)
64
-> (set n 2)
2
-> (begin (while (< n 50) [(set n (* 2 n))]) n)
run-time error: True does not understand message while
Method-stack traceback:
  Sent 'while' in standard input, line 58
-> (begin (while [(< n 50)] (set n (* 2 n))) n)
run-time error: SmallInteger does not understand message
value
Method-stack traceback:
  Sent 'value' in initial basis, line 32
```

Blocks are closures!

- And we can define recursive blocks at top level:

```
-> (define exp (base e)
     (begin
       (if (= e 0) [1]
           [(* base (value exp base (- e 1)))])))
<Block>
-> (exp 3 2)
syntax error: standard input, line 63: in message
send, message exp expects 0 arguments, but gets 1
argument
-> (value exp 3 2)
9
```

CS301
Session 17

Agenda

- Example: discrete-event simulation
- The predefined objects and object-oriented programming techniques

The computer lab simulation: a quaint scenario

Basic mechanism

- Event queue: a *priority queue* prioritized by event time
- To *schedule* an event, place it on the queue
- Each step in the simulation dequeues the highest priority event, updates a global clock, and sends the *takeAction* message to the event

Main classes

- *Simulation* (abstract) and *LabSimulation:* drive the simulation and report results
- *Lab:* manage the computer terminals
- *Queue:* represent the students waiting in line
- *EventQueue:* represent events waiting to happen
- *PriorityQueue:* a priority queue
- *WaitTimeList* and *ServiceTimeList:* the schedule of arrivals and needs for terminal time

The *Student* class

- The active agent, placed on event queue in two cases: for arrival in lab and for leaving a terminal
- State: scheduled to arrive, waiting in line, using a terminal, or done
- Method *takeAction* either "arrives in the lab" or "leaves the terminal"
- Method *arrive* either grabs a terminal or waits in line; in any case it also schedules a new student arrival

Scheduling events

- *scheduleNewArrival* sent to new *Student*
  - gets arrival and service times
  - adds self to event queue
- *scheduleLeaveTerminal* sent to existing *Student*
  - computes leave time as minimum of time needed to finish and time limit, and adds self to event queue
  - updates time still needed accordingly

Method *leaveTerminal*

- If done, releases terminal, updates stats, sends *grabTerminal* to any waiting student
- Otherwise,
  - If no one waiting, sends *scheduleLeaveTerminal* to self
  - Otherwise, releases terminal, joins waiting line, sends *grabTerminal* to first waiting student

Method *grabTerminal*

- Gets a terminal from the *Lab*
- Sends *scheduleLeaveTerminal* to self

The predefined objects

A "standard basis"

- Smalltalk itself is **small!** The predefined objects contain most of the "magic".
- Most are implemented in Smalltalk. Exceptions:
  - *Object* (no superclass)
  - *Class* (metaclasses inherit its methods)
  - *UndefinedObject* for technical reasons

Primitive methods

- Defined by the uSmalltalk interpreter:

```
eqObject print + - * div < > ...
```

- Programmer can add them to classes, e.g.

```
(class Foo Object ()
  (method foo: primitive eqObject))
<class Foo>
-> (val foo (new Foo))
<Foo>
-> (foo: foo foo)
<True>
-> (foo: foo 1)
<False>
```

Defining built-ins

- Class Object:

```
val objectClass =
  CLASS { name = "Object", super = NONE, ivars = ["self"], id = 1
   , methods = methods
    [ primMethod "print"   (unaryPrim defaultPrint)
    , userMethod "println" [] [] "(begin (print self)
                                         (print newline) self)"
    , primMethod "isNil"   (unaryPrim (fn _ => mkBoolean false))
    , primMethod "notNil"  (unaryPrim (fn _ => mkBoolean true))
    , primMethod "error:"  (binaryPrim error)
    , primMethod "="       (binaryPrim (mkBoolean o eqRep))
    , userMethod "!=" ["x"] [] "(not (= self x))"
    , primMethod "isKindOf:"    (binaryPrim kindOf)
    , primMethod "isMemberOf:"  (binaryPrim memberOf)
    , primMethod "subclassResponsibility"
       (unaryPrim
         (fn _ => raise RuntimeError "..."]}
```


Built-ins

- Class UndefinedObject

```
val nilClass =
  mkClass "UndefinedObject" objectClass []
    [ primMethod "isNil"
       (unaryPrim (fn _ => mkBoolean true))
    , primMethod "notNil"
       (unaryPrim (fn _ => mkBoolean false))
    , primMethod "print"
       (unaryPrim (fn x => (print "nil"; x)))
    ]
```


`isNil` without testing equality

- A non-OO programmer might be tempted to write

```
(method isNil () (= self nil)))
```

- Real programmers know to use inheritance and method override instead:

```
(class Object ...
  (method isNil () false) ...)
(class UndefinedObject Object ...
  (method isNil () true) ...)
```


Booleans

- Definable in Smalltalk
- Abstract class Boolean defines

```
ifFalse:ifTrue:
ifTrue:
ifFalse:
not eqv: xor: & | and: or: if
```

- Subclasses True and False define

```
ifTrue:ifFalse:
```

- Each is instantiated exactly once

Blocks

- Hybrid of internal and definable:

```
(class Block Object
    () ; internal representation
    (method value primitive value)
    (method whileTrue: (body)
        (ifTrue:ifFalse: (value self)
            [(value body)
             (whileTrue: self body)]
            [nil]))
    (method while (body) (whileTrue: self body))
    (method whileFalse: (body)
         (ifTrue:ifFalse: (value self)
             [nil]
             [(value body)
              (whileFalse: self body)]))
)
```


Collection classes

- Inheritance hierarchy:

```
Collection (abstract)
  Set
  KeyedCollection (abstract)
    Dictionary
    SequenceableCollection (abstract)
      List
      Array
```


Collection implementation

- Pervasive use of inheritance and method override
- Class *Collection* is abstract, requiring its subclasses to define

```
add: do: remove:ifAbsent species
```

- ...and defines all the other methods in terms of these

Set

- A simple Collection class using a List to represent its members
- Set is a *client* of List, not a subclass!
- Only interesting code is

```
(method add: (item)
  (ifFalse: (includes: members item)
            [(add: members item)])
  item)
```

- The rest is "delegated" (jargon alert!) to the list rep

Scheme comparison

- Recall sets-as-lists in uScheme

```
(val emptyset '())
(define member? (x s) ...)
(define add-element (x s)
  (if (member? x s) s (cons x s)))
(define size (s) (length s))
(define union (s1 s2)
  (if (null? s1)
      s2
    (add-element (car s1) (union (cdr s1) s2))))
```

- How is the Smalltalk definition different? Advantages? Disadvantages?

Lists

- Some interesting techniques needed because lists are *mutable*
  - Sentinel object eliminates need to test for empty
  - Circular representation
- List representation is a sentinel
  - ...of class ListSentinel
  - ...which is a subclass of Cons

Sentinels

- The list sentinel represents the list
  - car is nil
  - cdr refers to first "real" cons cell of the list, or to self for empty list
  - pred refers to last cell of the list, or to self for empty list

Cons cells

- Class Cons has rep

```
car cdr
```

- and local protocol

```
(method car () ...)
(method car: (anObject) ...)
(method cdr () ...)
(method cdr: (anObject) ...)
(method deleteAfter () ...)
(method do: (aBlock) ...)
(method insertAfter: (anObject) ...)
(method pred: (aCons) ...)
(method rejectOne:ifAbsent:withPred: (aBlock exnBlock pred) ...)
```

- But has no instance variable `pred`!

List sentinels

- Class ListSentinel is a subclass of Cons with rep

```
pred
```

- and local protocol

```
(classMethod new () ...)
(method do: (aBlock) ...)
(method pred () ...)
(method pred: (aCons) ...)
(method rejectOne:ifAbsent:withPred: (aBlock
exnBlock pred) ...)
```


Arrays

- Primitive methods:

```
new size at: at:put:
```

- Primitive rep as an ML array:

```
rep = ...
  | ARRAY    of value Array.array
```


Defined methods for arrays

- Unimplemented: add, remove, and friends, because arrays are fixed-size
- Interesting: do

```
(method do: (aBlock) (locals index)
    (set index (firstKey self))
    (timesRepeat: (size self)
       [(value aBlock (at: self index))
        (set index (+ index 1))]))
```

- Boring:

```
firstKey lastKey species printName
```


Numbers and the like

- Inheritance hierarchy
  - Magnitude (ordered things)
    - Number
      - Fraction
      - Float
      - Integer (only SmallInteger in uScheme implementation)