

Object-Oriented Programming in Scheme

Based on paper by J. Rees and N. Adams.
via notes for MIT 6.001, UML 91.301

Use Scheme how an OO language works –
including what it does “under the hood”

- Buggy inheritance
- Correct inheritance
- Multiple inheritance
- Structure similar to HW7

Example

```
> (get-method allyn 'say)
#<procedure:7:11>
> ((get-method allyn 'say) '(this talk is on OO programming))
(this talk is on oo programming)
> (get-method allyn 'give-me-an-A)
(no-method "SPEAKER")
```

No Method:

```
(define (no-method name)
  (list 'no-method name))
(define (no-method? x)
  (if (pair? x)
      (eq? (car x) 'no-method)
      false))
(define (method? x)
  (not (no-method? x)))
```

1

3

Implementation of Objects

An **object** is a procedure that, given a **message** as argument, returns another procedure called a **method**.

```
(define (get-method object message)
  (object message))
```

Running example:

```
(define (make-speaker)
  (define (self message)
    (cond ((eq? message 'say)
           (lambda (stuff) (display stuff)))
          (else (no-method "SPEAKER"))))
  self)
```

2

Some syntax

Get a method from an object and apply the method to arguments:

```
(define (ask object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method args)
        (error "No method" message (cadr method)))))
```

```
> (ask allyn 'say '(this talk is on OO programming))
(this talk is on oo programming)
> (ask allyn 'give-me-an-A)
No method give-me-an-a "SPEAKER"
```

4

Inheritance

A lecturer is a speaker that adds “You should be taking notes” to everything it says.

```
(define (make-lecturer)
  (let ((speaker (make-speaker)))
    (define (self message)
      (cond ((eq? message 'lecture)
             (lambda (stuff)
               (ask self 'say stuff)
               (ask self 'say '(you should be taking notes))))
            (else (get-method speaker message))))
      self))
```

5

Inheritance

```
> (define Gerry (make-lecturer))
> (ask Gerry 'say '(the sky is blue))
(the sky is blue)

> (ask Gerry 'lecture '(the sky is blue))
(the sky is blue)
(you should be taking notes)

> (ask Gerry 'give-me-an-A)
No method give-me-an-a "SPEAKER"
```

7

Inheritance

```
(define (make-lecturer)
  (let ((speaker (make-speaker)))
    (define (self message)
      ...
      (else (get-method speaker message))))
```

Lecturer can lecture or say

Lecturer inherits ability to say from speaker.

speaker is a superclass of lecturer.

make-lecturer defines a class, (make-lecturer) returns an object.

make-speaker defines a class, (make-speaker) returns an object.

6

So does Inheritance work?

```
(define (make-arrogant-lecturer)
  (let ((lecturer (make-lecturer)))
    (define (self message)
      (cond ((eq? message 'say)
             (lambda (stuff)
               (ask lecturer 'say (append '(it is obvious that)
                                           stuff))))
            (else (get-method lecturer message))))
      self))

> (define Albert (make-arrogant-lecturer))
```

Anything an arrogant-lecturer says is prefixed by “it is obvious that”.

8

Albert isn't arrogant when he lectures

```
> (define Albert (make-arrogant-lecturer))

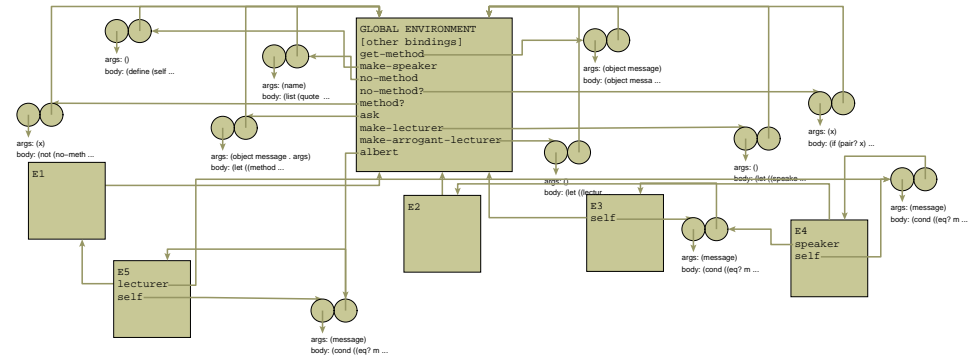
> (ask Albert 'say '(the sky is blue))
(it is obvious that the sky is blue)

> (ask Albert 'lecture '(the sky is blue))
(the sky is blue)
(you should be taking notes)
```

Make Albert an arrogant lecturer!

A lecturer 'say's stuff plus '(you should be taking notes). A speaker understands a 'say message. A lecturer gets a method for 'say from its speaker. An arrogant-lecturer understands a 'say message.

Where is self?



9

10

Ask passes object to method

Old

```
(define (ask object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method args)
        (error "No method" message (cadr method)))))
```

New

```
(define (ask object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method (cons object args))
        (error "No method" message (cadr method)))))
```

Methods take self

```
(define (make-speaker)
  (define (self message)
    (cond ((eq? message 'say)
          (lambda (stuff) (display stuff)))
          (else (no-method "SPEAKER"))))
  self)

(define (make-lecturer)
  (define (self message)
    (cond ((eq? message 'say)
          (lambda (self stuff) (display stuff)))
          (else (no-method message))))
  self)
```

11

12

Methods take self 2

```
(define (make-lecturer)
  (let ((speaker (make-speaker)))
    (define (self message)
      (cond ((eq? message 'lecture)
             (lambda (stuff)
               (ask self 'say stuff)
               (ask self 'say '(you should be taking notes))))
            (else (get-method speaker message))))
    self))

(define (make-lecturer)
  (let ((speaker (make-speaker)))
    (define (self message)
      (cond ((eq? message 'lecture)
             (lambda (self stuff)
               (ask self 'say stuff)
               (ask self 'say '(you should be taking notes))))
            (else (get-method speaker message))))
    self))
```

13

Methods take self 3

```
(define (make-arrogant-lecturer)
  (let ((lecturer (make-lecturer)))
    (define (self message)
      (cond ((eq? message 'say)
             (lambda (stuff)
               (ask lecturer 'say (append '(it is obvious that)
                                           stuff))))
            (else (get-method lecturer message))))
    self))

(define (make-arrogant-lecturer)
  (let ((lecturer (make-lecturer)))
    (define (self message)
      (cond ((eq? message 'say)
             (lambda (self stuff)
               (ask lecturer 'say (append '(it is obvious that)
                                           stuff))))
            (else (get-method lecturer message))))
    self))
```

14

How does this cure the problem?

```
(ask Albert 'lecture '(the sky is blue))

(lambda (object message .args)
  (let ((method (get-method alberts-self 'lecture)))
    (if (method? method)
        (apply method (cons alberts-self '((the sky is blue))))
        (error ...))))

... some steps ...

(apply (lambda (self stuff)
         (ask self 'say stuff)
         (ask self 'say '(you should be taking notes)))
       (cons alberts-self '((the sky is blue))))

...
(lambda (self stuff)
  (ask alberts-self 'say '(the sky is blue))
  (ask alberts-self 'say '(you should be taking notes)))
```

15

But I wanted the old Albert!

Sending a message to a superclass.

Recall ask

```
(define (ask object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method (cons object args))
        (error "No method" message (cadr method)))))
```

New behavior:

```
(apply (get-method lecturer message) (cons self args))
```

Old behavior:

```
(apply (get-method lecturer message) (cons lecturer args))
```

16

Multiple inheritance

```
(define (make-singer)
  (lambda (message)
    (cond ((eq? message 'say)
           (lambda (self stuff)
             (display (append '(tra-la-la --) stuff))))
          ((eq? message 'sing)
           (lambda (self)
             (display '(tra-la-la))))
          (else (no-method "SINGER")))))

(define anna (make-singer))

(ask anna 'sing)
(TRA-LA-LA)

(ask anna 'say '(the sky is blue))
(TRA-LA-LA -- THE SKY IS BLUE)
```

17

Multiple inheritance 2

We'll make Ben be both a singer and a lecturer:

```
(define ben
  (let ((singer (make-singer))
        (lecturer (make-lecturer)))
    (lambda (message)
      (let ((sing (get-method singer message))
            (lect (get-method lecturer message)))
        (if (method? sing)
            sing
            lect))))))

(ask ben 'sing)
(TRA-LA-LA)

(ask ben 'lecture '(the sky is blue))
(TRA-LA-LA -- THE SKY IS BLUE)
(TRA-LA-LA -- YOU SHOULD BE TAKING NOTES)
```

18

Multiple inheritance 3

Alyssa prefers lecturing to singing, given a choice:

```
(define alyssa
  (let ((singer (make-singer))
        (lecturer (make-lecturer)))
    (lambda (message)
      (let ((sing (get-method singer message))
            (lect (get-method lecturer message)))
        (if (method? lect)
            lect
            sing))))))

(ask alyssa 'sing)
(TRA-LA-LA)

(ask alyssa 'lecture '(the sky is blue))
(THE SKY IS BLUE)
(YOU SHOULD BE TAKING NOTES)
```

19

Simulating actions over time

Example: the SICP Adventure Game each person object gets a 'move message every clock tick.

```
(define *clock-list* '())
(define *the-time* 0)

(define (add-to-clock-list person)
  (set! *clock-list* (cons person *clock-list*))
  'added)

(define (clock)
  (newline)
  (display "----Tick---")
  (set! *the-time* (+ *the-time* 1))
  (for-each (lambda (person) (ask person 'move))
            *clock-list*)
  'tick-tock)
```

20

Inheritance in SICP Adventure

SICP Adventure named-object

```
(define (make-named-object name)
  (lambda (message)
    (cond ((eq? message 'name) (lambda (self) name))
          (else (no-method name)))))
```

21

22

SICP Adventure mobile-object

```
(define (make-mobile-object name location)
  (let ((named-obj (make-named-object name)))
    (lambda (message)
      (cond
        ((eq? message 'place) (lambda (self) location))
        ((eq? message 'install)
         (lambda (self)
          ; Synchronize thing and place
          (ask location 'add-thing self)))
        ;; Private! use CHANGE-PLACE instead
        ((eq? message 'set-place)
         (lambda (self new-place)
          (set! location new-place)
          'place-set))
        (else (get-method named-obj message))))))
```

23

SICP Adventure person

```
(define (make-person name birthplace threshold)
  (let ((possessions '())
        (mobile-obj (make-mobile-object name birthplace)))
    (lambda (message)
      (cond
        ((eq? message 'person?) (lambda (self) true))
        ((eq? message 'possessions) (lambda (self) possessions))
        ...
        ((eq? message 'move)
         (lambda (self)
          (cond ((= (random threshold) 0) (ask self 'act)) true)
                (else
                 (display-message (list name "chose to stay put")
                                   true))))))
        ...
        ((eq? message 'install)
         (lambda (self)
          (add-to-clock-list self)
          ((get-method mobile-obj 'install) self))))
```

24

Example: moving the players yourself

```
(define you (make&install-person 'you computer-lab 100))
(define late-homework (make&install-thing 'late-homework computer-lab))

; Transcript...

; You have late homework to give to Holly

(ask you 'take late-homework)
; At computer-lab : you says -- I take late-homework
;Value: #t

(ask you 'go 'south)
; you moves from computer-lab to elevator-lobby
;Value: #t

(ask you 'go 'west)
; you moves from elevator-lobby to west-hall
;Value: #t
```

25

Example continued

```
(ask you 'go 'north)
; you moves from west-hall to robot-lab
; At robot-lab : you says -- Hi holly
;Value: #t

(ask you 'lose late-homework)
; At robot-lab : you says -- I lose late-homework
;Value: #t

(ask holly 'take late-homework)
; At robot-lab : holly says -- I take late-homework
;Value: #t
```

26

Summary

```
(define (<class> <construction-parameters>)
  (let ((<superclass-name> <superclass-object>)
        ...
        (<instance-variable> <initial-value>)
        ...))
  (define (<method> self <args>) <method-body>)
  ...
  (define (self message)
    (cond ((eq? message <selector>) <method-for-selector>)
          ...
          (else <get-method-from-superclass-or-error>))))
```

In Scheme

- self is explicit argument to method
- method dispatch by object is explicit
- method is a procedure (self is higher-order proc)

27