• Dealing with missing methods
• The need for self-reference
• Dealing with “tags”
DETECTION OF METHODS

or Missing Methods

• Use \textbf{(no-method)} to indicate that there is no method

\begin{verbatim}
(define no-method
  (let (((tag (list 'NO-METHOD)))
    (lambda () tag)))))
\end{verbatim}

• Check if something is a method:

\begin{verbatim}
(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else
         (error "Object returned non-message" x)))))
\end{verbatim}
We want a person to call its own method, but …

Problem: no access to the "object" from inside itself!

Solution: add explicit self argument to all methods

limitation

self-reference

(define g (make-person 'george 'orwell))

(ask g 'SAY '(the sky is blue))
 ;==> the sky is blue

(ask g 'CHANGE-NAME 'ishmael)

==> want g to “SAY” his new name whenever it changes

• We want a person to call its own method, but ...

• Problem: no access to the "object" from inside itself!

• Solution: add explicit self argument to all methods
;; specifies the person class with self
(define (make-person-self fname lname)
  (lambda (message)
    (case message
      ((WHOAREYOU?) (lambda (self) fname))
      ((CHANGE-NAME)
        (lambda (self new-name) (set! fname new-name)))
      ((SAY)
        (lambda (self list-of-stuff)
          (display-msg list-of-stuff)))
      (else (no-method))))
;;; specifies the person class with self and ask
(define (make-person-self-ask fname lname)
  (lambda (message)
    (case message
      (((WHOAREYOU?) (lambda (self) fname))
        ((CHANGE-NAME)
          (lambda (self new-name)
            (set! fname new-name)
            (ask self 'SAY (list 'call 'me fname)))
        ((SAY)
          (lambda (self list-of-stuff)
            (display-msg list-of-stuff)))
        (else (no-method))))))
BETTER METHOD CONVENTION - 2

ask

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

(ask g 'CHANGE-NAME 'ishmael)

==> (apply #[proc p: self, new-name body:...] <g-object> 'ishmael)

==> (ask <g-object> 'say ...)

;==> (call me ishmael)
OBJECT TYPES IN OOP

Why?

• We want a method that acts differently depending on object type

(ask stud 'question ap-1 '(why does this code work))
;=> this should be obvious to you

(ask professor-1 'question ap-1 '(why does this code work))
;=> Why are you asking me about why does this code work
; I thought you published a paper on that topic

=> identify **stud** as a student object,
and **professor-1** as a professor object.
OBJECT TYPES IN OOP

Adding a Type Method

;; specifies the person class with type
(define (make-person-type fname lname)
  (lambda (message)
    (case message
      ((WHOAREYOU?) (lambda (self) fname))
      ((CHANGE-NAME)
        (lambda (self new-name)
          (set! fname new-name)
          (ask self 'SAY (list 'call 'me fname))))
      ((SAY)
        (lambda (self list-of-stuff)
          (display-msg list-of-stuff)))
      ((PERSON?)
        (lambda (self) #T))
      (else (no-method))))
OBJECT TYPES IN OOP

Adding a Type Method

(define someone (make-person-type 'bert 'sesame))

(ask someone 'person?)
; ==> #T

(ask someone 'professor?)
; No method for professor? in bert

(define (is-a object type-pred)
  (if (not (procedure? object))
    #F
    (let ((method (get-method type-pred object)))
      (if (method? method)
        (ask object type-pred)
        #F))))
SUMMARY

What have we learnt so far?

• Basic objects
• Self reference
• Tagging object classes
• Using environments and procedures to capture and change local state
1. Basic Objects
   • messages and methods convention
   • `self` variable to refer to oneself

2. Inheritance
   • internal superclass instances
   • match method directly in object, or get-method from internal instance if needed
   • *delegation*: explicitly use methods from internal objects

3. Multiple Inheritance
INHERITANCE

Why?

Isolation of shared values in single variable maintains consistency.

Classes are basic units of programming. By allowing inheritance of methods, we isolate changes in behavior in a modular fashion.
A professor prefaces all lecture material with “Therefore”.

(define e (make-professor 'eric 'grimson))

(ask e 'WHOAREYOU?)
;===> Professor grimson

(ask e 'SAY '(the sky is blue))
;===> the sky is blue

(ask e 'LECTURE '(the sky is blue))
;===> therefore the sky is blue
SECOND APPROACH
Inheriting Superclass Methods

• Subclass will inherit superclass behavior by adding an "internal" instance of the superclass
  • e.g. professor will have an internal person object
  • If message is not recognized, pass the buck

```
(define (make-professor fname lname) ;subclass
  (let ((int-person (make-person fname lname))) ;superclass
    (lambda (message)
      (case message
        ((LECTURE) ...) ;new method
          (define e (make-professor 'eric 'grimson))
        ((WHOAREYOU?)
          (lambda (self)
            (display-msg (list 'Professor lname))
            (ask e 'SAY '(the sky is blue)))
        (else (get-method message int-person))))))
```
(define e (make-professor 'eric 'grimson))

e:   
     make-person: ...
     make-professor: ...
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)

dashed object is int-person
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)

dashed object is int-person
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)

dashed object is int-person
solid object is our new professor object (with an internal person inside)
(ask e 'WHOAREYOU?)

E1
  fname: eric
  lname: grimson

E2
  int-person: p2
  p: msg
  b: (case ...)

E3
  fname: eric
  lname: grimson
  P2
  p: msg
  b: (case ...)

E4
  msg: whoareyou?

P1
  p: self
  b: (display-msg ...)

P2

P3

make-person: ...
make-professor: ...

p1
(ask e 'SAY '(the sky is blue))

E1

fname: eric
lname: grimson

E2

int-person: p2

E3

fname: eric
lname: grimson

E4

msg: say
args: ((the sky is blue))
(ask e 'SAY '(the sky is blue))
SECOND APPROACH
Delegation to Superclass

- Can change or specialize behavior of methods:
  - Internal object acts on behalf of the professor object by delegation

```scheme
(define (make-professor-lect fname lname)
  (let ((int-person (make-person-type fname lname)))
    (lambda (message)
      (case message
        ((LECTURE) ; now implement this...
          (lambda (self stuff)
            (display-msg (cons 'therefore stuff))))
        ((WHOAREYOU?)
          (lambda (self)
            (display-msg (list 'Professor lname))
            lname))
        (else (get-method message int-person))))))
```

(ask e 'LECTURE '(the sky is blue))

; ==> therefore the sky is blue
SECOND APPROACH

Delegation vs. Ask

(define (delegate to from message . args)
  (let ((method (get-method message to)))
    (if (method? method)
      (apply method from args) ; from becomes self
      (error "No method" message) )))
SECONd approach

Delegation to Superclass

• Can change or specialize behavior of methods:
  • Internal object acts on behalf of the professor object by delegation

;; make-professor with delegate
(define (make-professor-delegate fname lname)

(ask e 'LECTURE '(the sky is blue))
;;=> therefore the sky is blue
SECOND APPROACH
Delegation to Superclass

• Can change or specialize behavior of methods:
  • **Internal object** acts on behalf of the **professor** object by delegation

```scheme
;; make-professor with delegate
(define (make-professor-delegate fname lname)
  (let ((int-person (make-person-type fname lname)))
    (lambda (message)
      (case message
        ((LECTURE) ;now implement this...
          (lambda (self stuff)
            (delegate int-person self 'SAY
              (cons 'therefore stuff)))))))

((WHOAREYOU?)
  (lambda (self)
    (display-msg (list 'Professor lname))
    lname))
(else (get-method message int-person))))
```

(ask e 'LECTURE '(the sky is blue))
;;===> therefore the sky is blue
SECOND APPROACH

Delegation vs. Ask

(define (ask object message . args)
  (let ((method (get-method message object))))
  (if (method? method)
      (apply method object args) ;object becomes self
      (error "No method for message" message))))

(define (delegate to from message . args)
  (let ((method (get-method message to)))
    (if (method? method)
      (apply method from args) ;from becomes self
      (error "No method" message))))
An arrogant professor ends *everything* he/she says with “obviously”.

```
(define e (make-arrogant-professor 'eric 'grimson))
(ask e 'SAY '(the sky is blue))
;;===> the sky is blue obviously

(ask e 'LECTURE '(the sky is blue))
;;===> therefore the sky is blue obviously
```
EXAMPLE

An Arrogant Professor Subclass

(define (make-arrogant-professor fname lname) ;subclass
  (let ((int-prof (make-professor-delegate fname lname))) ;superclass
    (lambda (message)
      (case message
        ((SAY)
         (lambda (self stuff)
           (delegate int-prof self
                     'SAY (append stuff '(obviously))))
        (else (get-method message int-prof)))))))

(define e (make-arrogant-professor 'big 'shot))
(ask e 'LECTURE '(the sky is blue))

;==> therefore the sky is blue ; BUG! (obviously)
WHERE IS THE BUG?

• Problem is not in the new arrogant-professor subclass!
  • Arrogant-professor changed its “SAY" method with the expectation that everything an arrogant-professor says will be modified.

• The bug is in the professor class!
  • delegated “SAY” to internal person
  • should have asked whole self to “SAY”
  • but.... the arrogant-lecture “SAY” method didn't get called when we asked arrogant-professor to “LECTURE”

• With ask it is possible for a superclass to invoke a subclasses's method (as we want in this case)!
;; make-professor fixing the bug
(define (make-professor-ask fname lname)
  (let ((int-person (make-person-type fname lname)))
    (lambda (message)
      (case message
        ((LECTURE)
          (lambda (self stuff)
            ;bug (delegate int-person self 'SAY
            ;bug (append '(therefore) stuff))
            (ask self 'SAY
              (append '(therefore) stuff)))
            (else (get-method message int-person))))))

(define e (make-arrogant-professor 'eric 'grimson))

(ask e 'LECTURE '(the sky is blue))
;==> therefore the sky is blue obviously
STEPS TOWARD OUR SCHEME OOPS

1. Basic Objects
   - messages and methods convention
   - `self` variable to refer to oneself

2. Inheritance
   - internal superclass instances
   - match method directly in object, or get-method from internal instance if needed
   - *delegation*: explicitly use methods from internal objects

3. Multiple Inheritance
A singer is not a person.

A singer has a different SAY that always ends in "tra la la".

A singer starts to SING with "the hills are alive"
MULTIPLE INHERITANCE

Singer Class as a Base Class
(no Superclass)

(define (make-singer)
  (lambda (message)
    (case message
      ((SAY)
        (lambda (self stuff)
          (display-msg
            (append stuff '(tra la la)))))
      ((SING)
        (lambda (self)
          (ask self 'SAY '(the hills are alive))))
      (else (no-method))))
MULTIPLE INHERITANCE
A Singing Arrogant Professor

(define (make-s-a-p fname lname)
  (let ((int-singer (make-singer))
        (int-arrogant (make-arrogant-professor fname lname))
        (lambda (message)
           (find-method message int-singer int-arrogant))))

(define zoe (make-s-a-p 'zoe 'zinger))

(ask zoe 'SING)
;==> the hills are alive tra la la

(ask zoe 'LECTURE '(the sky is blue))
;==> therefore the sky is blue tra la la
MULTIPLE INHERITANCE

Finding a Method

- Look through the supplied objects from left to right until the first matching method is found.

```
(define (find-method message . objects)
  (define (try objects)
    (if (null? objects)
      (no-method)
      (let ((method (get-method message
                      (car objects)))))
        (if (not (eq? method (no-method)))
            method
            (try (cdr objects)))))
  (try objects))
```
UNUSUAL MULTIPLE INHERITANCE

Delegate to All

• Pass the message on to *multiple* internal objects:

```scheme
(define (make-s-a-p-dta fname lname)
  (let ((int-singer (make-singer))
     (int-arrognt (make-arrogant-professor fname lname)))
    (lambda (message)
      (lambda (self . args)
        (apply delegate-to-all (list int-singer int-arrognt)
          self message args))))))

(define zoe (make-s-a-p-dta 'zoe 'zinger))

(ask zoe 'SAY '(the sky is blue))
; ==> the sky is blue tra la la
;    the sky is blue obviously
```
UNUSUAL MULTIPLE INHERITANCE
Delegate to All

(define (delegate-to-all to-list from message . args)
  (for-each
    (lambda (to-whom)
      (apply delegate to-whom from message args))
    to-list))
SUMMARY
What have we learnt?

• Basic objects
• Inheritance
• Delegation