How build an OOP system in Scheme?

- **Objects**: as procedures that take **messages**
  - **Instances have Identity**: in sense of eq?
    - Object instances are unique Scheme procedures
  - **Local State**: gives each object (each instance of a class) the ability to perform differently
    - Each instance procedure has own local environment

- **Classes**: Scheme **make-<object>** procedures.
  - **Methods** returned in response to messages:
    - Scheme procedures (take method-dependent arguments)
  - **Inheritance Rule** telling what method to use
    - Conventions on messages & methods

Steps toward our Scheme OOPs:

1. **Basic Objects**
   A. messages and methods convention
   B. `self` variable to refer to oneself

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

3. **Multiple Inheritance**

Today's Example World: People, Professors, Arrogant-profs, and Students

PERSON
fname: lname:
SAY WHOAREYOU?

1. Method convention

- The response to every **message** is a **method**
- A **method** is a procedure that can be applied to actually do the work

```
(define (make-person fname lname)   ; specifies the person class
  (lambda (message)
    (cond ((eq? message 'WHOAREYOU?) (lambda () fname))
          ((eq? message 'CHANGE-MY-NAME)
           (lambda (new-name) (set! fname new-name)))
          ((eq? message 'SAY)
           (lambda (list-of-stuff)
             (display-message list-of-stuff)
             'NUF-SAID))
          (else (no-method)))))
```

Alternative *case* syntax for message match:

- `case` is more general than this (see Scheme manual), but our convention for message matching will be:

```
(case message
 ((<msg-1>) <method-1>)
 ((<msg-2>) <method-2>)
 ...
 ((<msg-n>) <method-n>)
 (else <expr>))))
```


Method convention – with case syntax

- The response to every **message** is a method
- A method is a procedure that can be applied to actually do the work

```
(define (make-person fname lname)
  (lambda (message)
    (case message
      ((WHOAREYOU?) (lambda () fname))
      ((CHANGE-NAME)
       (lambda (new-name) (set! fname new-name)) )
      ((SAY)
       (lambda (list-of-stuff)
         (display-message list-of-stuff)
         'NUF-SAID))
      (else (no-method)))))
```

p: ()
body: fname

p: (new-name)
body: (set! ...)

How make and use the object?

- Making an object instance

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

- Using the object instance (painful way)

```
((g 'WHOAREYOU?))
==>(#[proc p:() body:fname])   ;apply to no args
==> george
```

- Two things going on:
  - method lookup – `(g 'WHOAREYOU?) ==> <method>`
  - method application – `(<method>) ==> <result>`

Using the object – easier way

- method lookup:

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

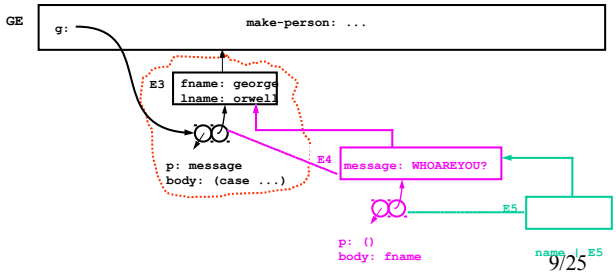
- "ask" an object to do something - combined method retrieval and application to args.

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

`(apply op args)` ➔ `(op arg1 arg2 … argn)`

Example

```
(define g (make-person 'george 'orwell))
(ask g 'WHOAREYOU?)
   ➔ (get-method 'WHOAREYOU? G) ➔ (g 'WHOAREYOU?)
                                ➔ ( (g 'WHOAREYOU?) )
```



Cleaning up some details of our implementation

- Dealing with missing methods
- The need for self-reference
- Dealing with “tags”

Detection of methods (or missing methods):

- Use `(no-method)` to indicate that there is no method

```
(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))
```

- Check if something is a method:

```
(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else
          (error "Object returned non-message" x))))
```


Limitation – self-reference

```
(ask g 'SAY '(the sky is blue))
the sky is blue
=> nuf-said

(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

Better Method Convention (1) -- self

```
(define (make-person 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-message list-of-stuff)
         'NUF-SAID))
      (else (no-method)))))
```


Better Method Convention (1) -- self

```
(define (make-person 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-message list-of-stuff)
         'NUF-SAID))
      (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
 nuf-said
```


Typing objects in an OOPS system

- 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

This means we need to identify **stud** as a student object, and **professor-1** as a professor object.

Adding a type method

```
(define (make-person 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-message list-of-stuff)
         'NUF-SAID))
      ((PERSON?)
       (lambda (self) #t))
      (else (no-method)))))
```


Adding a type method

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

(ask someone 'person?)
➔#t

(ask someone 'professor?)
;No method for professor? in bert
;Type D to debug error, Q to return back to REP loop

(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

- Basic objects
- Self reference
- Tagging object classes

- Using environments and procedures to capture and change local state

Steps toward our Scheme OOPs:

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

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

3. Multiple Inheritance

Why inheritance?

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.

Example World: Professor Subclass

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
==> nuf-said

(ask e 'LECTURE '(the sky is blue))
therefore the sky is blue
==> nuf-lectured
```

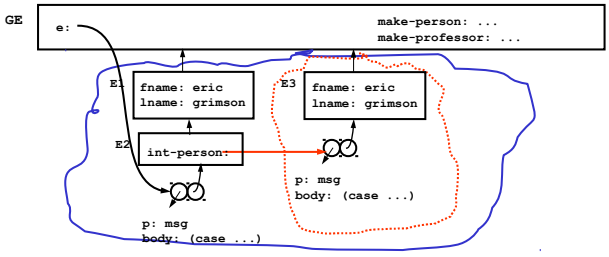

2. Approach: Inheriting Superclass Methods

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

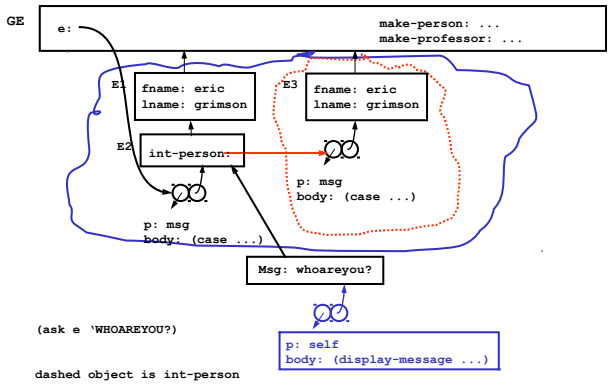
```
(ask e 'SAY '(the sky is blue))
the sky is blue
```


How the internal object works:




```
(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)
```

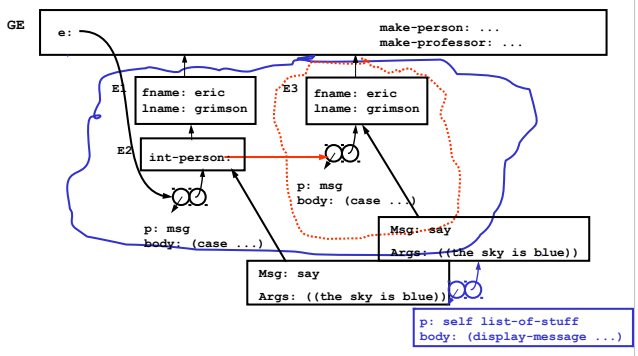
How the internal object works:



(ask e 'WHOAREYOU?)

dashed object is int-person
solid object is our new professor object (with an internal person inside)

How the internal object works:



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

dashed object is int-person
solid object is our new professor object (with an internal person inside)

2. Approach: Delegation to Superclass

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

```
(define (make-professor name)
  (let ((int-person (make-person name)))
    (lambda (message)
      (case message
        ((LECTURE)     ;now implement this...
         (lambda (self stuff)
           (display-message (cons 'therefore stuff))
           'nuf-said))
        (else (get-method message int-person))))))

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


2. Approach: Delegation to Superclass

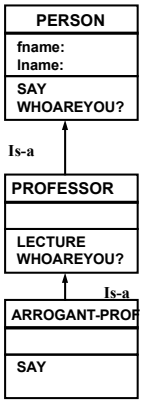
- Can change or specialize behavior of methods:
  - Internal object acts on behalf of the `professor` object by delegation

```
(define (make-professor name)
  (let ((int-person (make-person name)))
    (lambda (message)
      (case message
        ((LECTURE)     ;now implement this...
         (lambda (self stuff)
           (delegate int-person self 'SAY
                     (cons 'therefore stuff)))
        (else (get-method message int-person))))))

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

Delegate 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))))


(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))))
```


Example: An Arrogant-professor Subclass



An arrogant professor ends ***everything*** he/she says with "obviously"

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

(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 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 'gun))
  (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)!

Fixing the Bug: `ask` vs. `delegate`

```
(define (make-professor name)
  (let ((int-person (make-person name)))
    (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))
(ask e 'LECTURE '(the sky is blue))
 therefore the sky is blue obviously
```

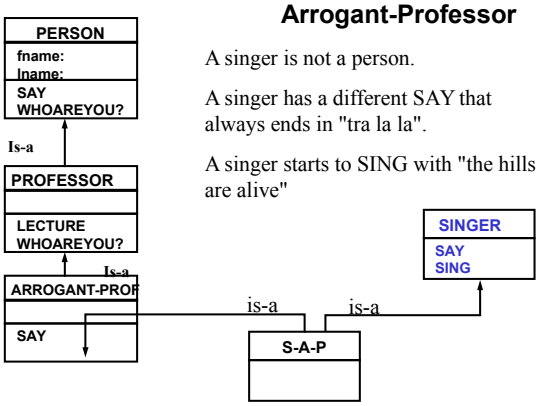

Steps toward our Scheme OOPs:

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

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

3. **Multiple Inheritance** ←

Example: A Singer, and A Singing Arrogant-Professor



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"

3. Multiple Inheritance

- The singer as a "base" class (no superclasses):

```
(define (make-singer)
  (lambda (message)
    (case message
      ((SAY)
       (lambda (self stuff)
         (display-message
            (append stuff '(tra la la))))
      ((SING)
       (lambda (self)
         (ask self 'SAY '(the hills are alive))))
      (else (no-method)))))
```

A Singing Arrogant Professor

- Now we'll create a singing arrogant professor:

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

(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

- Just 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

- We could build an OOPS with lots of flexibility - suppose we want to pass the message on to *multiple* internal objects?

```
(define (make-s-a-l 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)))))

(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)
  (foreach
    (lambda (to-whom)
      (apply delegate to-whom from message args))
  to-list)
```

Summary

- Basic objects
- Inheritance
- Delegation