Object-Oriented Design & Implementation

- Focus on classes
  - Relationships between classes
  - Kinds of interactions that need to be supported between instances of classes

- Careful attention to behavior desired
  - Inheritance of methods
  - Explicit use of superclass methods
  - Shadowing of methods to over-ride default behaviors

- An extended example to illustrate class design and implementation

WH(OOPS)!

- Class diagrams in previous lecture appear simpler than they really are!
  - Additional clutter from environments such as those created by `(create-named-object ...), (make-handler ...), etc.

Person class

```scheme
(define p1 (create-person 'joe))
(ask p1 'whoareyou?) ⇒ joe
(ask p1 'say '(the sky is blue)) ⇒ (the sky is blue)
```

Person class implementation

```scheme
(define (create-person name)
  (create-instance person name))
(define (person self name)
  (let ((root-part (root-object self)))
    (make-handler
     'person
     (make-methods
      'WHOAREYOU? (lambda () name)
      'SAY (lambda (stuff) stuff))
     root-part)))
```

Professor class implementation

```scheme
(define (create-professor name)
  (create-instance professor name))
(define (professor self name)
  (let ((person-part (person self name)))
    (make-handler
     'professor
     (make-methods
      'WHOAREYOU? (lambda () (list 'prof (ask person-part 'WHOAREYOU?)))
      'LECTURE (lambda (notes)
                 (cons 'therefore
                        (ask person-part 'SAY notes)))
                 person-part)))
```

Professor class behavior

```scheme
(define prof1 (create-professor 'fred))
(ask prof1 'whoareyou?) ⇒ (prof fred)
(ask prof1 'lecture '(the sky is blue)) ⇒ (therefore the sky is blue)
```

A professor’s lecture method will use the person say method.
Arrogant-Prof implementation

```scheme
(define (create-arrogant-prof name)
  (create-instance arrogant-prof name))

(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (make-handler
     'arrogant-prof
     (make-methods
      'SAY
      (lambda (stuff)
       (append (ask prof-part 'say stuff)
               (list 'obviously)))
      prof-part))))
```

Arrogant-Prof behavior

```scheme
(define ap1 (create-arrogant-prof 'perfect))
(ask ap1 'whoareyou?)
⇒ (prof perfect)
(ask ap1 'say (the sky is blue))
⇒ (the sky is blue obviously)
```

Arrogant-Prof oddity

```scheme
(ask ap1 say (the sky is blue))
⇒ PROF:say skyblue + obviously
⇒ PER:say skyblue + obviously
⇒ skyblue + obviously
```

Arrogant-Prof oddity resolved

```scheme
(ask ap1 'lecture '(the sky is blue))
⇒ no method; PROF:lect skyblue
⇒ therefore + PER:say skyblue
⇒ therefore + skyblue
```

Professor class – revised implementation

```scheme
(define (create-professor name)
  (create-instance professor name))

(define (professor self name)
  (let ((person-part (person self name)))
    (make-handler
     'professor
     (make-methods
      'WHOAREYOU?
      (lambda () (list 'prof (ask person-part 'WHOAREYOU?)))
      'LECTURE
      (lambda (notes)
       (cons 'therefore + SELF:say
              (ask person-part 'SAY notes)))
       person-part)))))
```

Key question: What is SELF here?
When to ask self vs. ask a part?

• No problem when you completely over-ride a method
  – E.g., if spy is-a person and defines a new WHOAREYOU? method, there is no interaction between them

<table>
<thead>
<tr>
<th>PERSON</th>
<th>spy</th>
</tr>
</thead>
<tbody>
<tr>
<td>name:</td>
<td></td>
</tr>
<tr>
<td>WHOAREYOU? name:</td>
<td>stuff</td>
</tr>
</tbody>
</table>

When to ask self vs. ask a part?

• If a method on a specialized class needs to use the same method on one of its superclasses
  – Then it’s appropriate to call (ask <part> ...) within that method

```
(define (thing self name location)
  (let ((named-part (named-object self name)))
    (make-handler 'THING
      (make-methods 'INSTALL (lambda ()
        (ask named-part 'INSTALL)
        (ask (ask self 'LOCATION) 'ADD-THING self))
      )
    )
  )
)
```

When to ask self vs. ask a part?

• If a method on a specialized class needs to use a different method, it can do so on itself!

```
(define (professor self name)
  (let ((person-part (person self name)))
    (make-handler
      'professor
      (make-methods
        'WHOAREYOU?
        (lambda () (list 'prof (ask person-part 'WHOAREYOU?)))
        'LECTURE
        (lambda (notes)
          (cons 'therefore
                (ask self 'SAY notes)))
      person-part))
  )
)
```

Student class

```
(define s1 (create-student 'bert))
(ask s1 'whoareyou?)
⇒ bert
(ask s1 'say '(i do not understand))
⇒ (excuse me but i do not understand)
```

Student implementation

```
(define (create-student name)
  (create-instance student name))

(define (student self name)
  (let ((person-part (person self name)))
    (make-handler
      'student
      (make-methods
        'SAY
        (lambda (stuff)
          (append '(excuse me but)
                  (ask person-part 'say stuff)))
      person-part))
  )
)
```

Question and Answer

```
(define p1 (create-person 'joe))
(define s1 (create-student 'bert))
(ask s1 'question p1 '(why is the sky blue))
⇒ bert i do not know about why is the sky blue
```
Person class – added methods

```scheme
(define (person self name)
  (let ((root-part (root-object self)))
    (make-handler
     'person
     (make-methods
      'WHOAREYOU? (lambda () name)
      'SAY (lambda (stuff) stuff)
      'QUESTION (lambda (of-whom query) ; person,list->list
                   (ask of-whom 'answer self query))
      'ANSWER (lambda (whom query) ; person,list->list
                (ask self 'say
                 (cons (ask whom 'WHOAREYOU?)
                        (append '(i do not know about)
                                query))))
     root-part)))
)
```

Arrogant-Prof – specialized “answer”

```scheme
(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (make-handler
     'arrogant-prof
     (make-methods
      'SAY (lambda (stuff)
             (append (ask prof-part 'say stuff)
                     (list 'obviously)))
      'ANSWER (lambda (whom query)
                (cond ((ask whom 'is-a 'student)
                       (ask self 'say
                        '(this should be obvious to you)))
                      ((ask whom 'is-a 'professor)
                       (ask self 'say
                        '(but you wrote a paper about)
                        query))
                      (else (ask prof-part 'answer whom query))))
     prof-part)))
)
```

Arrogant-Prof: revised implementation

```scheme
(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (make-handler
     'arrogant-prof
     (make-methods
      'SAY (lambda (stuff)
             (append (ask prof-part 'say stuff)
                     (list 'obviously)))
      'ANSWER (lambda (whom query)
                (cond ((ask whom 'is-a 'student)
                       (ask self 'say
                        '(this should be obvious to you)))
                      ((ask whom 'is-a 'professor)
                       (ask self 'say
                        '(but you wrote a paper about)
                        query))
                      (else (ask prof-part 'answer whom query))))
     prof-part)))
)
```

Lessons from our example class hierarchy

- Specifying class hierarchies
  - Convention on the structure of a class definition
  - to inherit structure and methods from superclasses
- Control over behavior
  - Can “ask” a sub-part to do something
  - Can “ask” self to do something
- Use of TYPE information for additional control

Steps toward our Scheme OOPS:

- Basic Objects
  - messages and methods convention
  - self variable to refer to oneself
- Inheritance
  - internal parts to inherit superclass behaviors
  - in local methods, can “ask” internal parts to do something
  - use get-method on superclass parts to find method if needed
- Multiple Inheritance

A Singer, and a Singing-Arrogant-Prof

- 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"
Singer implementation

```scheme
(define (create-singer)
  (create-instance singer))

(define (singer self)
  (let ((root-part (root-object self)))
    (make-handler
     'singer
     (make-methods
      'SAY
      (lambda (stuff) (append stuff '(tra la la)))
      'SING
      (lambda () (ask self 'SAY '(the hills are alive))))
    root-part)))
```

- The singer is a "base" class (its only superclass is root)

Singing-Arrogant-Prof implementation

```scheme
(define (create-singing-arrogant-prof name)
  (create-instance singing-arrogant-prof name))

(define (singing-arrogant-prof self name)
  (let ((singer-part (singer self))
        (arr-prof-part (arrogant-prof self name)))
    (make-handler
     'singing-arrogant-prof
     (make-methods)
     singer-part
     arr-prof-part)))
```

Example: A Singing Arrogant Professor

```scheme
(define sap1 (create-singing-arrogant-prof 'zoe))
(ask sap1 'whoareyou?)
⇒ (prof zoe)
(ask sap1 'say '(the hills are alive))
⇒ (the hills are alive tra la la)
(ask sap1 'lecture '(the sky is blue))
⇒ (the sky is blue tra la la)
⇒ (therefore the sky is blue tra la la)
```

- See that arrogant-prof’s SAY method is never used in sap1 (no “obviously” at end)
- Our get-method passes the SAY message along to the singer class first, so the singer’s SAY method is found
- If we needed finer control (e.g. some combination of SAYing)
  - Then we could implement a SAY method in singing-arrogant-prof class to specialize this behavior

Implementation View: Multiple Inheritance

- Our OOPS already has multiple inheritance:
  - Just look through the supplied objects (parts that correspond to superclasses) from left to right until the first matching method is found.

```scheme
(define (get-method message . objects)
  (find-method-from-handler-list message (map ->handler objects)))

(define (find-method-from-handler-list message objects)
  (if (null? objects)
      (no-method)
      (let ((method ((car objects) message)))
        (if (not (eq? method (no-method)))
            method
            (find-method-from-handler-list message (cdr objects)))))
```