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

```scheme
(define prof1 (create-professor 'fred))
(ask prof1 'say '(the sky is blue))
  => (the sky is blue)
```

Person instance

```scheme
(define p1 (create-person 'joe))
(show p1) ; show is an ad hoc display procedure for debugging
```

Professor instance

```
(define prof1 (create-professor 'fred))
```

Person

- `WHOAREYOU?`
- `SAY`

Professor

- `WHOAREYOU?`
- `SAY`
Professor class – with own methods

\[
\text{(define prof1 (create-professor 'fred))}
\]
\[
\text{(ask prof1 'whoareyou?)}
\rightarrow \text{(prof fred)}
\]
\[
\text{(ask prof1 'lecture '(the sky is blue))}
\rightarrow \text{(therefore the sky is blue)}
\]

A professor’s lecture method will use the person say method.

Professor class implementation

\[
\text{(define (create-professor name))}
\]
\[
\text{(define (professor self name))}
\]
\[
\text{(let ((person-part (person self name)))}
\]
\[
\text{(make-handler 'professor}
\]
\[
\text{(make-methods}
\]
\[
\text{'WHOAREYOU?}
\]
\[
\text{(lambda () (list 'prof (ask person-part 'WHOAREYOU?)}})
\]
\[
\text{'LECTURE}
\]
\[
\text{(lambda (notes) (cons 'therefore (ask person-part 'SAY notes) person-part)))}
\]

Arrogant-Prof class

\[
\text{(define ap1 (create-arrogant-prof 'perfect))}
\]
\[
\text{(ask ap1 'whoareyou?)}
\rightarrow \text{(prof perfect)}
\]
\[
\text{(ask ap1 'say '(the sky is blue))}
\rightarrow \text{(the sky is blue obviously)}
\]

Arrogant-Prof implementation

\[
\text{(define (create-arrogant-prof name))}
\]
\[
\text{(define (arrogant-prof self name))}
\]
\[
\text{(let ((prof-part (professor self name)))}
\]
\[
\text{(make-handler 'arrogant-prof}
\]
\[
\text{(make-methods}
\]
\[
\text{'SAY}
\]
\[
\text{(lambda (stuff) (append (ask prof-part 'say stuff) (list 'obviously)))}
\]
\[
\text{prof-part)))}
\]

Arrogant-Prof oddity

- Why didn’t arrogant-prof add "obviously" at the end?
  - Actual source of oddity is in the LECTURE method of the professor class, which used SAY method of person-part
  - So the arrogant-profs’ SAY method never got used

Arrogant-Prof oddity corrected

\[
\text{(define ap1 (create-arrogant-prof 'perfect))}
\]
\[
\text{(ask ap1 'lecture '(the sky is blue))}
\rightarrow \text{(therefore the sky is blue obviously)}
\]
Professor class – revised implementation

```
(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)))
```

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: (lambda () 'nobody) then there is no interaction between them
- 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
  - Note: (ask self ...) would lead to infinite loop!
- If a method on a specialized class needs to use a different method, it can do so on itself!

Student class

```
(define s1 (create-student 'bert))
(ask s1 'whoareyou?)
```

```
(bert)
(ask s1 'say '(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

```
(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)))
```

"callback" methods
Arrogant-Prof – specialized “answer”

```
(define s1 (create-student 'bert))
(define prof1 (create-professor 'fred))
(define ap1 (create-arrogant-prof 'perfect))
(ask s1 'question ap1 '(why is the sky blue))
  => (this should be obvious to you obviously)
(ask prof1 'question ap1 '(why is the sky blue))
  => (but you wrote a paper about why is the sky blue obviously)
```

Arrogant-Prof: revised implementation

```
(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

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

Singer implementation

```
(define (create-singer)
  (create-instance singer))
(define (singer self)
  (let ((root-part (root-object self)))
    (make-handler 'singer
      (make-methods 'SAY
        '(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

(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

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

(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))))))

Summary

• Classes: capture common behavior
• Instances: unique identity with own local state
• Hierarchy of classes
  – Inheritance of state and behavior from superclass
  – Multiple inheritance: rules for finding methods
• Object-Oriented Programming Systems (OOPS)
  – Abstract view: class and instance diagrams
  – User view: how to define classes, create instances
  – Implementation view: how we layer notion of object classes, instances, and inheritance on top of standard Scheme