

## 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

4/11/05

1/28

## 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.

4/11/05

2/28

## Person class

PERSON
name:
WHOAREYOU?
SAY

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

4/11/05

3/28

## Person class implementation

```
(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)))
```

PERSON
name:
WHOAREYOU?
SAY

4/11/05

4/28

## Person instance

```
(define pl (create-person 'joe))

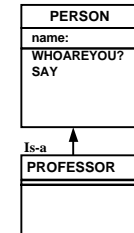
(show pl) ; show is an ad hoc display procedure for debugging
INSTANCE (instance #[compound-procedure 61 handler])
TYPE: (person root)
HANDLER: #[compound-procedure 61 handler]
TYPE: person
(methods (whoareyou? #[compound-procedure 65])
         (say #[compound-procedure 64]))
Parent frame: #[environment 66]
root-part: #[compound-procedure 67 handler]
Parent frame: global-environment
self: (instance #[compound-procedure 61 handler])
name: joe
;Value: instance
```

4/11/05

5/28

## Professor class

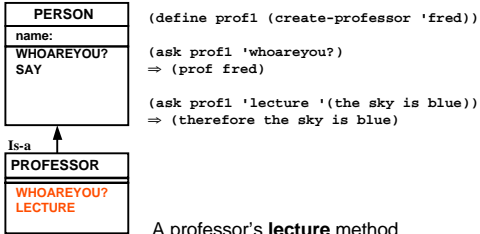
```
(define prof1 (create-professor 'fred))
(ask prof1 'say '(the sky is blue))
⇒ (the sky is blue)
```



4/11/05

6/28

### Professor class – with own methods



A professor's **lecture** method will use the person **say** method.

4/11/05

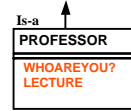
7/28

### Professor class implementation

```

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

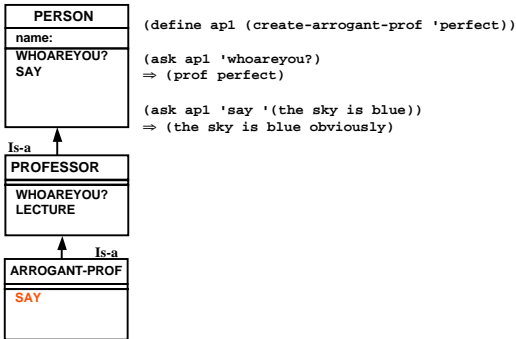
(define (professor self name)
  (let ((person-part (person self name)))
    (make-handler
     'professor
     (make-methods
      'WHOAREYOU?
      (lambda () (list 'prof (ask person-part
                        'WHOAREYOU?)))
      'LECTURE
      (lambda (notes)
        (cons 'therefore
          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



4/11/05

8/28

### Arrogant-Prof class



```

(define ap1 (create-arrogant-prof 'perfect))

(ask ap1 'whoareyou?)
=> (prof perfect)

(ask ap1 'say '(the sky is blue))
=> (the sky is blue obviously)
  
```

4/11/05

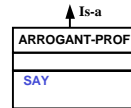
9/28

### Arrogant-Prof implementation

```

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

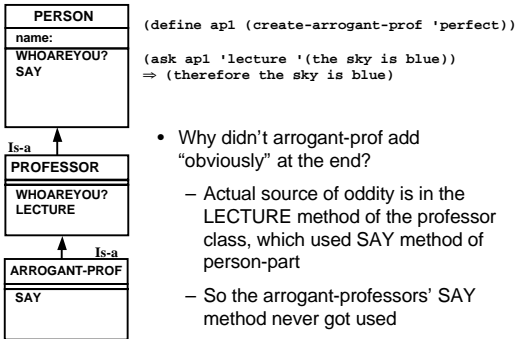
(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (make-handler
     'arrogant-prof
     (make-methods
      'SAY
      (lambda (stuff)
        (append (ask prof-part 'say stuff)
                 (list 'obviously))))
      prof-part)))
  
```



4/11/05

10/28

### Arrogant-Prof oddity



```

(define ap1 (create-arrogant-prof 'perfect))

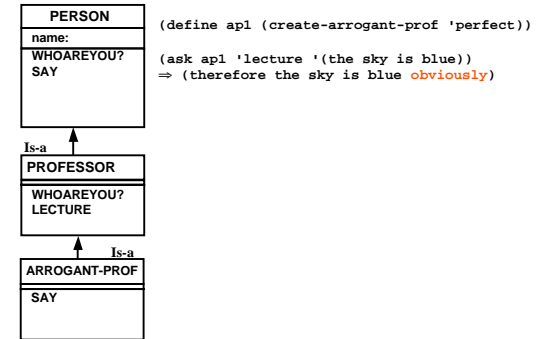
(ask ap1 'lecture '(the sky is blue))
=> (therefore the sky is blue)
  
```

- 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-professors' SAY method never got used

4/11/05

11/28

### Arrogant-Prof oddity corrected



```

(define ap1 (create-arrogant-prof 'perfect))

(ask ap1 'lecture '(the sky is blue))
=> (therefore the sky is blue obviously)
  
```

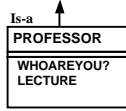
4/11/05

12/28

### 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)))
```



4/11/05

13/28

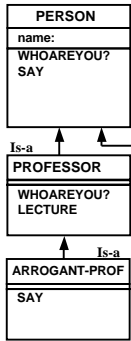
### When to ask self vs. ask a part?

- 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!

4/11/05

14/28

### 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)
```

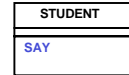
4/11/05

15/28

### Student implementation

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

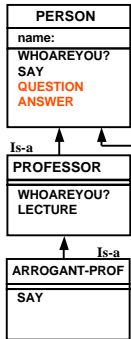
(define (student self name)
  (let ((person-part (person self name)))
    (make-handler
     'student
     (make-methods
      'SAY
      (lambda (stuff)
        (append '(excuse me but)
                 (ask person-part 'say stuff))))
      person-part)))
```



4/11/05

16/28

### Question and Answer



```
(define p1 (create-person 'joe))
(define s1 (create-student 'bert))

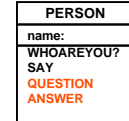
(ask s1 'question p1
 '(why is the sky blue))
=> (bert i do not know about why is the sky blue)
```

4/11/05

17/28

### 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

4/11/05

18/28

### Arrogant-Prof – specialized “answer”

```

(define s1 (create-student 'bert))
(define prof1 (create-professor 'fred))
(define apl (create-arrogant-prof 'perfect'))

(ask s1 'question apl
      '(why is the sky blue))
⇒ (this should be obvious to you obviously)

(ask prof1 'question apl
          '(why is the sky blue))
⇒ (but you wrote a paper about why
    is the sky blue obviously)

```

4/11/05 19/28

### 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
                   (append '(but you wrote a paper about)
                           query)))
             (else (ask prof-part 'answer whom query))))
     prof-part)))

```

4/11/05 20/28

### 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

4/11/05 21/28

### 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 ←

4/11/05 22/28

### 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”

4/11/05 23/28

### Singer implementation

```

(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)))

```

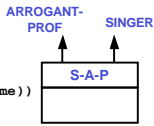
• The singer is a “base” class (its only superclass is root)

4/11/05 24/28

## 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))))
```



4/11/05

25/28

## 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

4/11/05

26/28

## 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))))))
```

4/11/05

27/28

## 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

4/11/05

28/28