

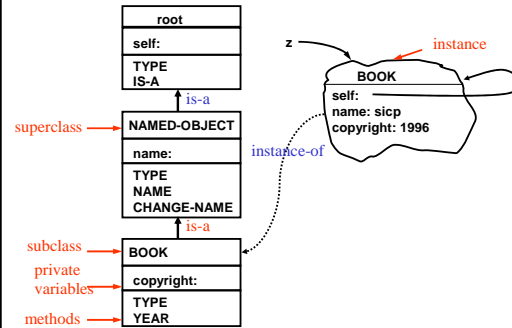
## Different Views of Object-Oriented System

- An **abstract view**
    - class and instance diagrams
    - terminology: messages, **methods**, **inheritance**, superclass, subclass, ...
  - Scheme OO system **user view**
    - conventions on **how to write** Scheme code to:
      - define classes
        - **inherit** from other classes
      - create instances
        - use instances (invoke **methods**)
- Scheme OO system **implementer view** (under the covers)
- How implement instances, classes, inheritance, types

4/6/2004

1

## Reminder: Example Class/Instance Diagram

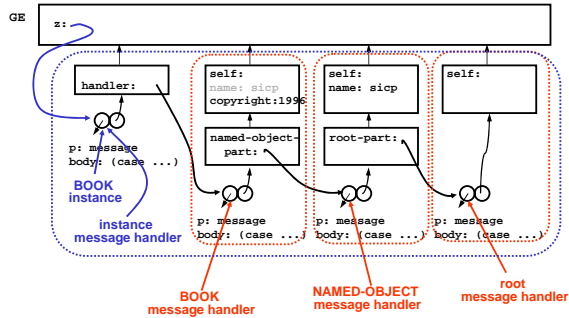


4/6/2004

2

## Implementer's View of this in Environ. Model

```
(define z (create-book 'sicip 1996))
```



4/6/2004

3

## Implementer's View: Instances

```
(define (make-instance)
  (let ((handler #f))
    (lambda (message)
      (case message
        ((SET-HANDLER!)
         (lambda (handler-proc)
           (set! handler handler-proc)))
        (else (get-method message handler))))))

(define (create-instance maker . args)
  (let* ((instance (make-instance))
        (handler (apply maker instance args)))
    (ask instance 'SET-HANDLER! handler)
    instance))
```

4/6/2004

4

## Implementer's View: get-method and ask

- 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)
```

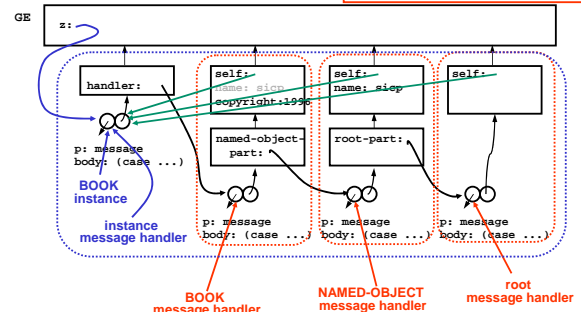
4/6/2004

5

## Implementer's View of this in Environ. Model

```
(define z (create-book 'sicip 1996))
```

Warning! Actual version in Project 4 is slightly different, but has the same general flavor!!



4/6/2004

6

## User's View: Why a "self" variable?

- Every class definition has access to a "self" variable
  - `self` is a pointer to the **entire** instance
- Why need this? How or when use `self` ?
  - When implementing a method, sometimes you "ask" **a part** of yourself to do something
    - E.g. inside a BOOK method, we might...  
(ask **named-object-part** 'CHANGE-NAME 'mit-sicp)
  - However, sometimes we want to ask the whole instance to do something
    - E.g. inside a subclass, we might  
(ask **self** 'YEAR)
  - This mostly matters when we have subclass methods that **shadow superclass methods**, and we want to invoke one of those shadowing methods from inside the superclass
- Next: An example OO design to illustrate our OO system

4/6/2004

7

## 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

4/6/2004

8

## Person class

PERSON
name:
TYPE
WHOWAREYOU?
SAY

```
(define p1 (create-person 'joe))
(ask p1 'whoareyou?)
⇒ joe

(ask p1 'say '(the sky is blue))
⇒ (the sky is blue)
```

4/6/2004

9

## Person class implementation

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

(define (person self name)
  (let ((root-part (make-root-object self)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'person root-part)))
        ((WHOWAREYOU?) (lambda () name))
        ((SAY) (lambda (stuff) stuff))
        (else (get-method message root-part)))))))
```

PERSON
name:
TYPE
WHOWAREYOU?
SAY

4/6/2004

10

## Professor class

PERSON
name:
TYPE
WHOWAREYOU?
SAY

Is-a

PROFESSOR
TYPE

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

4/6/2004

12

## Professor class – with own methods

PERSON
name:
TYPE
WHOWAREYOU?
SAY

Is-a

PROFESSOR
TYPE
WHOWAREYOU?
LECTURE

```
(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.

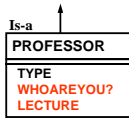
4/6/2004

13

## Professor class implementation

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

(define (professor self name)
  (let ((person-part (person self name)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'professor person-part)))
        ((WHOAREYOU?)
         (lambda () (list 'prof
                           (ask person-part 'WHOAREYOU?))))
        ((LECTURE)
         (lambda (notes)
           (cons 'therefore (ask person-part 'say notes))))
        (else (get-method message person-part)))))))
```



4/6/2004

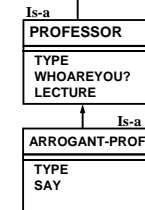
14

## Arrogant-Prof class

```
(define apl (create-arrogant-prof 'perfect))

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

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



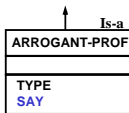
4/6/2004

15

## Arrogant-Prof implementation

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

(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'arrogant-prof prof-part)))
        ((SAY) (lambda (stuff)
                  (append (ask prof-part 'say stuff)
                          (list 'obviously)))))
        (else (get-method message prof-part)))))))
```



4/6/2004

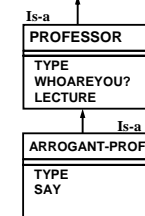
16

## Arrogant-Prof oddity

```
(define apl (create-arrogant-prof 'perfect))

(ask apl '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 professor class, which used SAY method of person-part
  - So the arrogant-professors' SAY method never got used



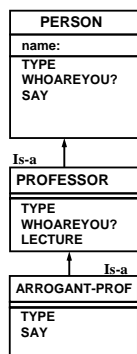
4/6/2004

17

## Arrogant-Prof oddity corrected

```
(define apl (create-arrogant-prof 'perfect))

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



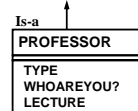
4/6/2004

18

## Professor class – revised implementation

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

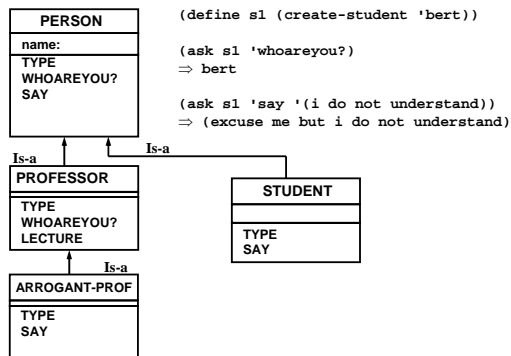
(define (professor self name)
  (let ((person-part (person self name)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'professor person-part)))
        ((WHOAREYOU?) (lambda () (list 'prof name)))
        ((LECTURE) (lambda (notes)
                      (cons 'therefore
                            (ask person-part 'say notes))))
        (else (get-method message person-part)))))))
```



4/6/2004

19

## Student class



4/6/2004

20

## Student implementation

STUDENT
TYPE
SAY

```

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

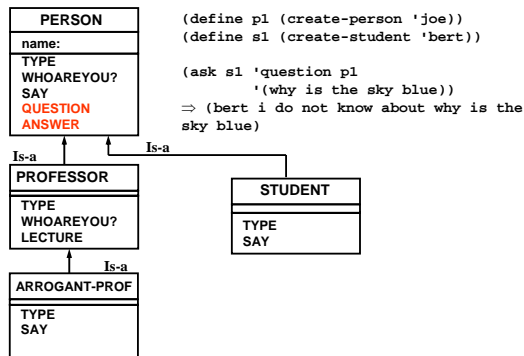
(define (student self name)
  (let ((person-part (person self name)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'student person-part)))
        ((SAY) (lambda (stuff)
                  (append '(excuse me but)
                          (ask person-part 'say stuff))))
        (else (get-method message person-part)))))))

```

4/6/2004

21

## Question and Answer



4/6/2004

22

## Person class – added methods

PERSON
name:
TYPE
WHOAREYOU?
SAY
QUESTION
ANSWER

```

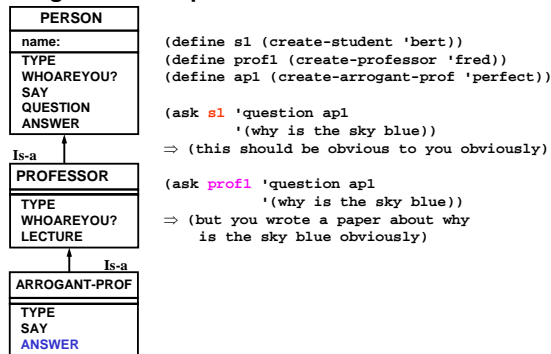
(define (person self name)
  (let ((root-part (root-object self)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'person root-part)))
        ((WHOAREYOU?) (lambda () name))
        ((SAY) (lambda (stuff) stuff))
        ((QUESTION)
         (lambda (of-whom 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)))))
        (else (get-method message root-part))))))

```

4/6/2004

23

## Arrogant-Prof – specialized “answer”



4/6/2004

24

## Arrogant-Prof: revised implementation

```

(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'arrogant-prof prof-part)))
        ((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))))
        (else (get-method message prof-part))))))

```

4/6/2004

25

## 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

4/6/2004

26

## Steps toward our Scheme OOPS:

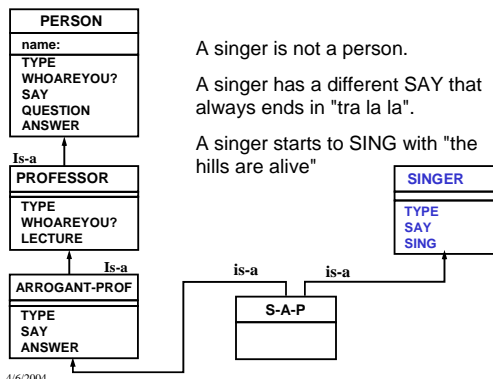
- Basic Objects
  - messages and methods convention
  - `self` variable to refer to oneself
- Inheritance
  - internal parts to inherit superclass behaviors
  - in local methods, can “ask” internal parts to do something
  - use get-method on superclass parts to find method if needed

- Multiple Inheritance ←

4/6/2004

27

## A Singer, and a Singing-Arrogant-Prof



4/6/2004

28

## Singer implementation

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

(define (singer self)
  (let ((root-part (root-object self)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'singer root-part)))
        ((SAY)
         (lambda (stuff) (append stuff '(tra la la))))
        ((SING)
         (lambda () (ask self 'say '(the hills are alive))))
        (else (get-method message root-part)))))))
```

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

4/6/2004

29

## 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)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'singing-arrogant-prof
                                   singer-part
                                   arr-prof-part)))
        (else (get-method message singer-part
                           arr-prof-part)))))))
```

4/6/2004

30

## 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

4/6/2004

31

## Implementation View: Multiple Inheritance

- How implement the more general get-method?
  - Just look through the supplied objects from left to right until the first matching method is found.

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

becomes

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

4/6/2004

32

## 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

4/6/2004

33

## OOPS – One more example

- Goal: See an example that distinguishes between
  - “is-a” or inheritance relationships
  - “has-a” or local variable relationships
- Idea:
  - A **person** class with parent-child relationships

4/6/2004

34

## Some Classes for Family Relationships

- ROOT-OBJECT  
 ↑ is-a  
 NAMED-OBJECT  
 ↑ is-a  
 PERSON  
 ↑ is-a  
 MOTHER
- Look at these classes (named-object, person, mother) from perspectives of
    - class diagrams
    - desired behaviors
    - instance diagrams
    - our class/method definitions
    - underlying representation (environment model)

4/6/2004

35

## Named-object class definition

```
(define (create-named-object name)
  (create-instance named-object name))

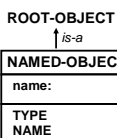
(define (named-object self name)
  (let ((root-part (root-object self)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'named-object root-part)))
        ((NAME) (lambda () name))
        (else (get-method message root-part))))))

(define (names-of objects)
  ; Given a list of objects, returns a list of their names.
  (map (lambda (x) (ask x 'NAME)) objects))
```

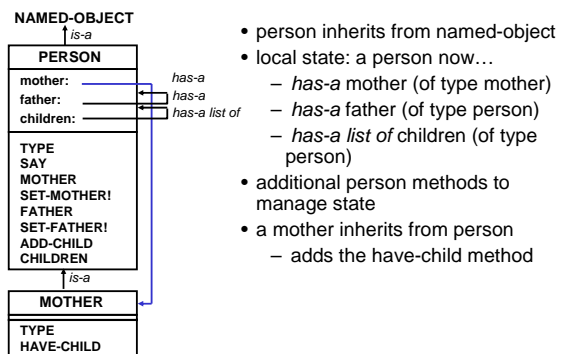
- Very simple state and behavior: a local name, which the user can access through NAME method.

4/6/2004

36



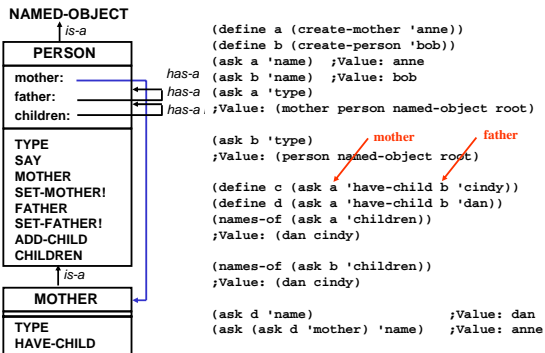
## Some Family Relationships – Class Diagram



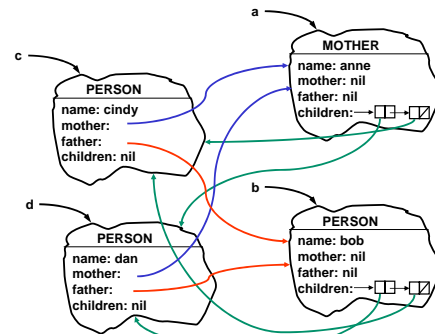
4/6/2004

37

## Some Family Relationships – Behaviors



## Some Family Relationships – Instance Diagram



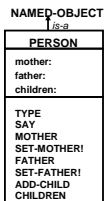
## Person Class Definition

```

(define (create-person name)
  (create-instance person name))

(define (person self name)
  (let ((named-part (named-object self name)))
    (mother nil)
    (father nil)
    (children nil))
  (lambda (message)
    (case message
      ((TYPE) (lambda () (type-extend 'person named-part)))
      ((SAY) (lambda (stuff) (display stuff)))
      ((MOTHER) (lambda () mother))
      ((FATHER) (lambda () father))
      ((CHILDREN) (lambda () children))
      ((SET-MOTHER!) (lambda (mom) (set! mother mom)))
      ((SET-FATHER!) (lambda (dad) (set! father dad)))
      ((ADD-CHILD) (lambda (child)
                      (set! children (cons child children))
                      child))
      (else (get-method message named-part))))))
  
```

4/6/2004 40



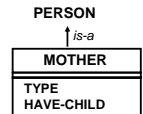
## Mother Class Definition

```

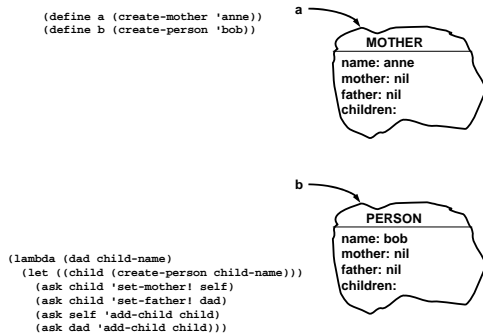
(define (create-mother name)
  (create-instance mother name))

(define (mother self name)
  (let ((person-part (person self name)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'mother person-part)))
        ((HAVE-CHILD)
         (lambda (dad child-name)
           (let ((child (create-person child-name)))
             (ask child 'set-mother! self)
             (ask child 'set-father! dad)
             (ask self 'add-child child)
             (ask dad 'add-child child))))
        (else (get-method message person-part))))))
  
```

4/6/2004 41



## Some Family Relationships – Instance Diagram



## Some Family Relationships – Instance Diagram

