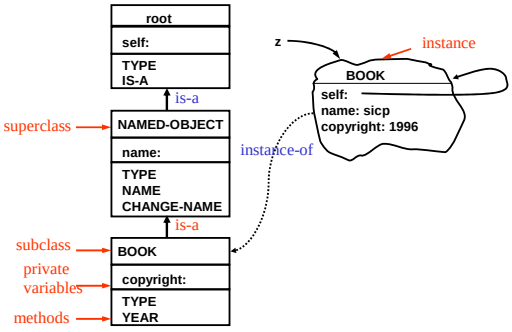
Different Views of Object-Oriented System

- 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)

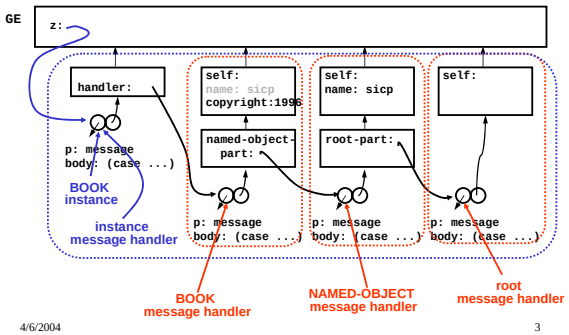
➔Scheme OO system **implementer view** (under the covers)
  - How implement instances, classes, inheritance, types

Reminder: Example Class/Instance Diagram



Implementer's View of this in Environ. Model

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



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))
```


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.

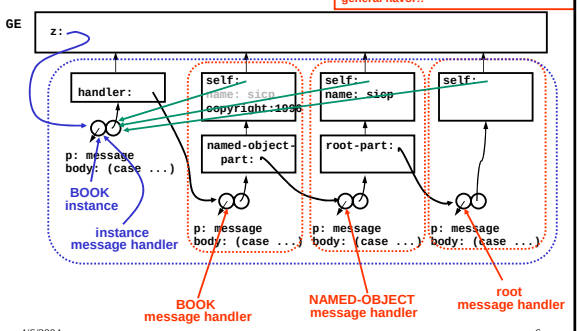
```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method args)
        (error "No method for message" message))))
```

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

Implementer's View of this in Environ. Model

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

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



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

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

Person class


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

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


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)))
       ((WHOAREYOU?)       (lambda () name))
       ((SAY)              (lambda (stuff) stuff))
       (else (get-method message root-part))))))
```


Professor class

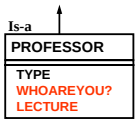

```
(define prof1 (create-professor 'fred))

(ask prof1 'say '(the sky is blue))
⇒ (the sky is blue)
```


Professor class – with own methods


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

(ask prof1 'whoareyou?)
⇒ (prof fred)

(ask prof1 'lecture '(the sky is blue))
⇒ (therefore the sky is blue)
```

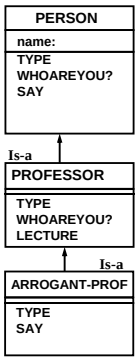
A professor's **lecture** method will use the person **say** method.

Professor class implementation



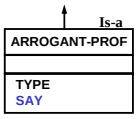

```
(define (create-professor name)
  (create-instance professor name))

(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))))))
```


Arrogant-Prof class



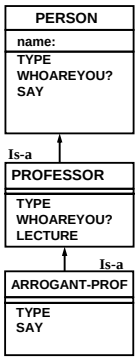

```
(define ap1 (create-arrogant-prof 'perfect))

(ask ap1 'whoareyou?)
⇒ (prof perfect)

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


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))))))
```

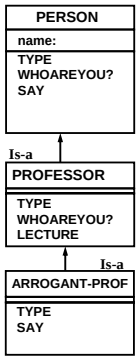

Arrogant-Prof oddity




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

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

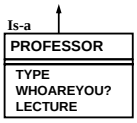
- Why didn't arrogant-prof add "obviously" at the end?
  - Actual source of oddity is in the professor class, which used SAY method of person-part
  - So the arrogant-professors' SAY method never got used

Arrogant-Prof oddity corrected




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

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

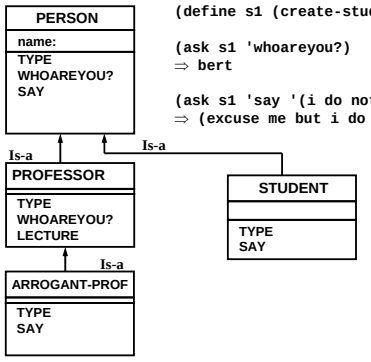

Professor class – revised implementation




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

(define (professor self name)
  (let ((person-part (person self name)))
    (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))))))
```

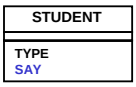
self

Student class



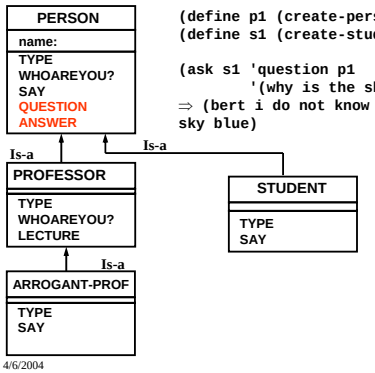

```
(define s1 (create-student 'bert))

(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)))
    (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))))))
```


Question and Answer

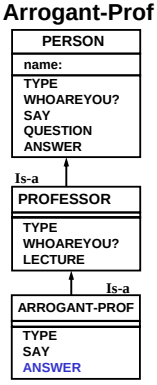



```
(define p1 (create-person 'joe))
(define s1 (create-student 'bert))

(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)))
    (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))))))
```


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)))
    (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))))))
```

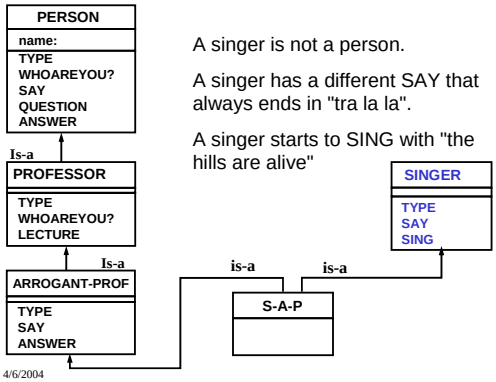
Lessons from our example class hierarchy

- 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"

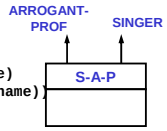
Singer implementation

```
(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))))))
```

SINGER
TYPE
SAY
SING

- 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)))
    (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))))))
```


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

- 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))
```


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

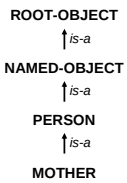
- Object-Oriented Programming Systems (OOPS)
  - **Abstract view:** class and instance diagrams
  - **User view:** how to define classes, create instances
  - **Implementation view:** how we layer notion of object classes, instances, and inheritance on top of standard Scheme

OOPS – One more example

- Goal: See an example that distinguishes between
  - "is-a" or inheritance relationships
  - "has-a" or local variable relationships

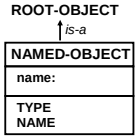
- Idea:
  - A **`person`** class with parent-child relationships

Some Classes for Family Relationships



- 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)

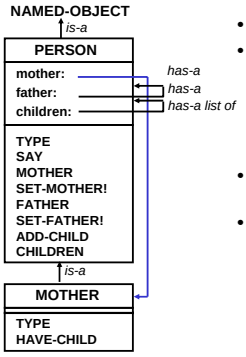
Named-object class definition




```
(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))
```

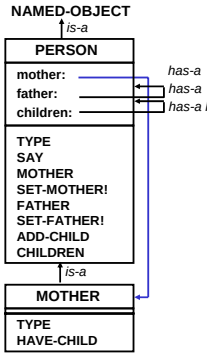
- Very simple state and behavior: a local name, which the user can access through NAME method.

Some Family Relationships – Class Diagram



- person inherits from named-object
- local state: a person now…
  - *has-a* mother (of type mother)
  - *has-a* father (of type person)
  - *has-a list of* children (of type person)
- additional person methods to manage state
- a mother inherits from person
  - adds the have-child method

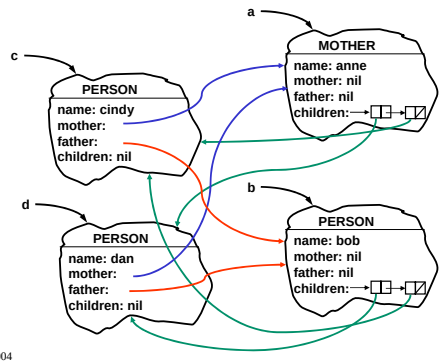
Some Family Relationships – Behaviors



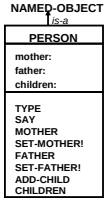

```
(define a (create-mother 'anne))
(define b (create-person 'bob))
(ask a 'name)  ;Value: anne
(ask b 'name)  ;Value: bob
(ask a 'type)
;Value: (mother person named-object root)

(ask b 'type)
;Value: (person named-object root)

(define c (ask a 'have-child b 'cindy))
(define d (ask a 'have-child b 'dan))
(names-of (ask a 'children))
;Value: (dan cindy)

(names-of (ask b 'children))
;Value: (dan cindy)

(ask d 'name)                ;Value: dan
(ask (ask d 'mother) 'name)  ;Value: anne
```

mother father

Some Family Relationships – Instance Diagram

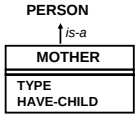


Person Class Definition




```
(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))))))
```

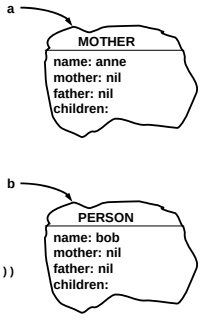

Mother Class Definition




```
(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))))))
```

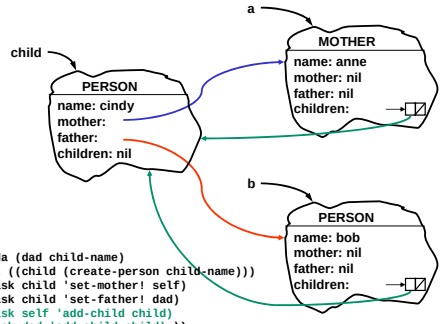

Some Family Relationships – Instance Diagram

```
(define a (create-mother 'anne))
(define b (create-person 'bob))
```



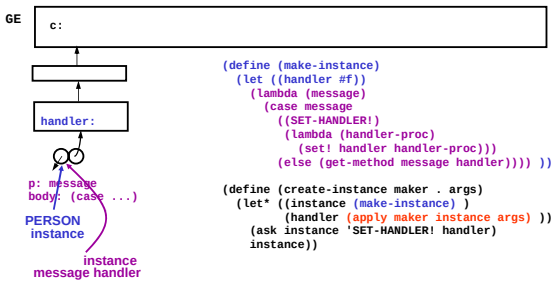

```
(lambda (dad child-name)
  (let ((child (create-person child-name)))
    (ask child 'set-mother! self)
    (ask child 'set-father! dad)
    (ask self 'add-child child)
    (ask dad 'add-child child)))
```


Some Family Relationships – Instance Diagram



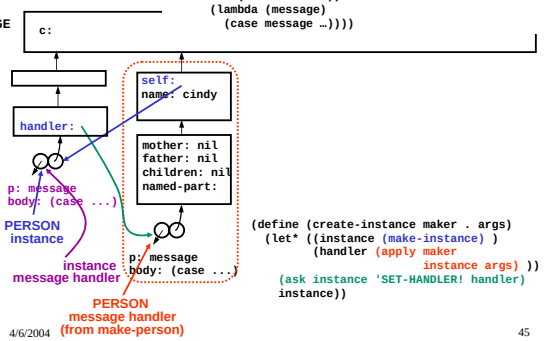

```
(lambda (dad child-name)
  (let ((child (create-person child-name)))
    (ask child 'set-mother! self)
    (ask child 'set-father! dad)
    (ask self 'add-child child)
    (ask dad 'add-child child) ))
```

Result of
`(create-person 'cindy) =>`
`(create-instance make-person 'cindy)`



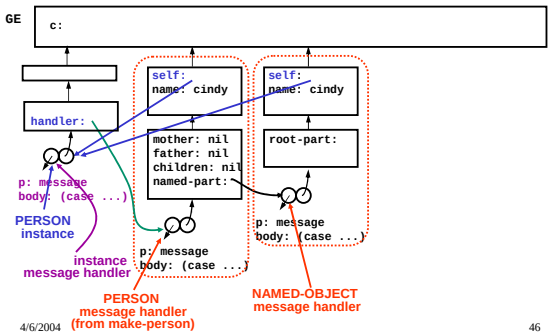
Result of
`(create-person 'c`
`(create-instance`

```
(define (make-person self name)
  (let ((named-part (make-named-object self name))
        (mother nil)
        (father nil)
        (children nil))
    (lambda (message)
      (case message …))))
```



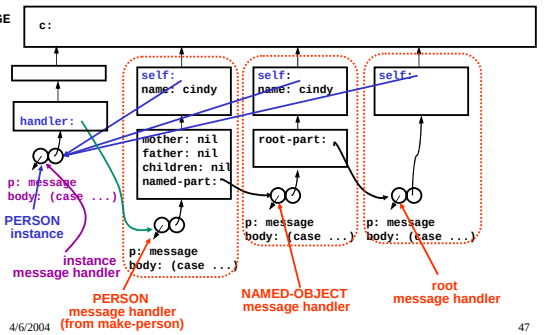
Result of
`(create-person 'c`
`(create-instance`

```
(define (make-named-object self name)
  (let ((root-part (make-root-object self)))
    (lambda (message)
      (case message …))))
```

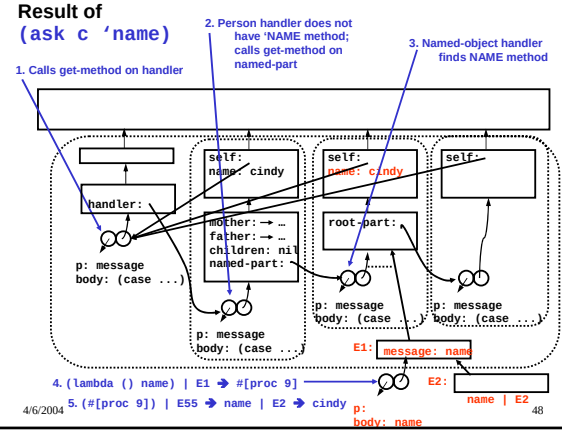


Result of
`(create-person 'c`
`(create-instance`

```
(define (make-named-object self name)
  (let ((root-part (make-root-object self) ))
    (lambda (message)
      (case message …))))
```



Result of
`(ask c 'name)`



Summary

- Classes in our system
  - May have local state and local methods. Local state can:
    - include primitive data (e.g. a **`name`** symbol)
    - indicate relationships with other objects (e.g. pointers to other instances in the system)
  - May inherit state and methods
    - By way of internal handlers generated thru "make-<superclass>" parts
- Instances in our system
  - Have a starting "instance" (self) object in env. model
  - Instance contains a series of message/state handlers for each class in inheritance chain
  - You need to gain experience with this!