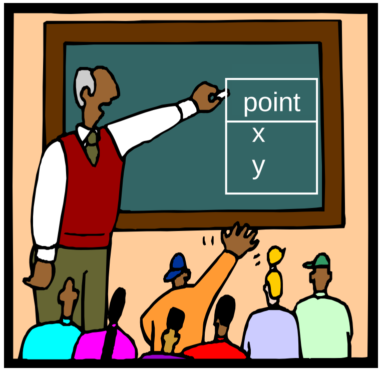
6.001 recitation 16   4/13/07

object-oriented programming

- procedures with state
- classes, instances
- Scheme object system



Dr. Kimberle Koile

programming styles: procedural vs object-oriented

- **Procedural programming:**

  Organize system around **procedures** that operate on data.

```
(do-something <data> <arg> ...)
(do-another-thing <data>)
```

- **Object-based programming:**

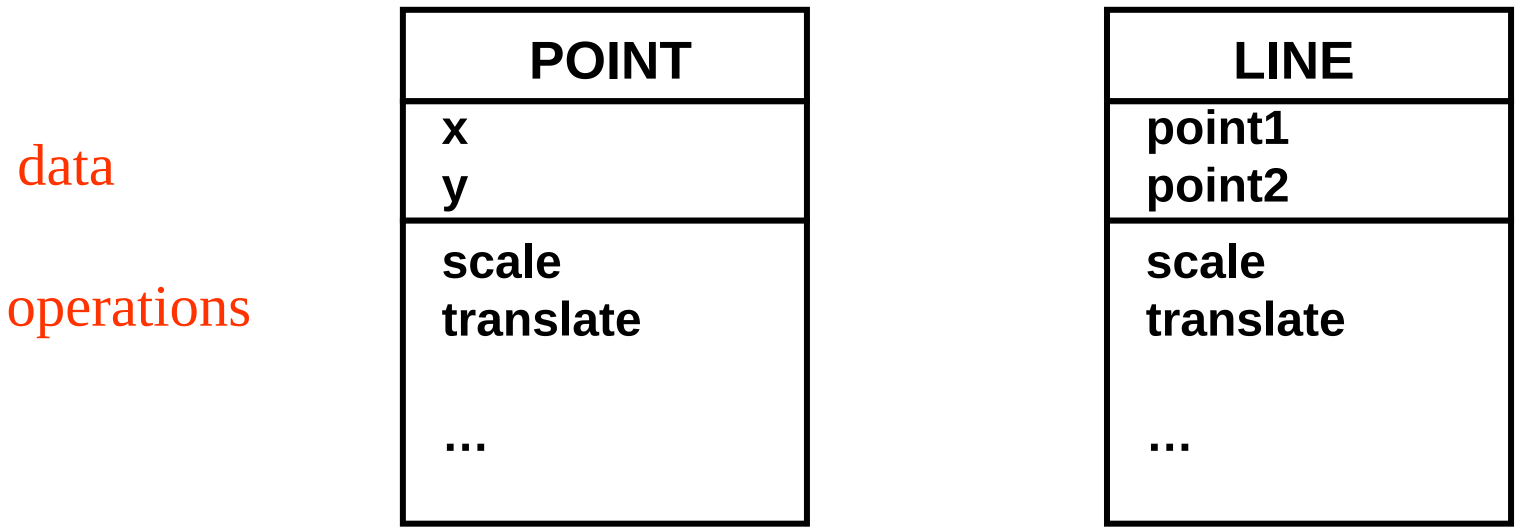
  Organize system around **objects** that receive messages.

  (<object> 'do-something <arg>)

  (<object> 'do-another-thing)

  An object encapsulates data and operations.

Scheme: data objects as procedures with state



data objects and message passing

```
(define (point x y)
   (lambda (msg)
      (cond ((eq? msg 'x)  x)
            ((eq? msg 'y)  y)
            ((eq? msg 'point?)  #t))))
```

How do you define a point for (3 4)?

(define my-pt (point 3 4))

How do you define a procedure called x that returns the x value of a point?

```
(define (x pt)
   (pt 'x))
```

data objects and message passing

(1)

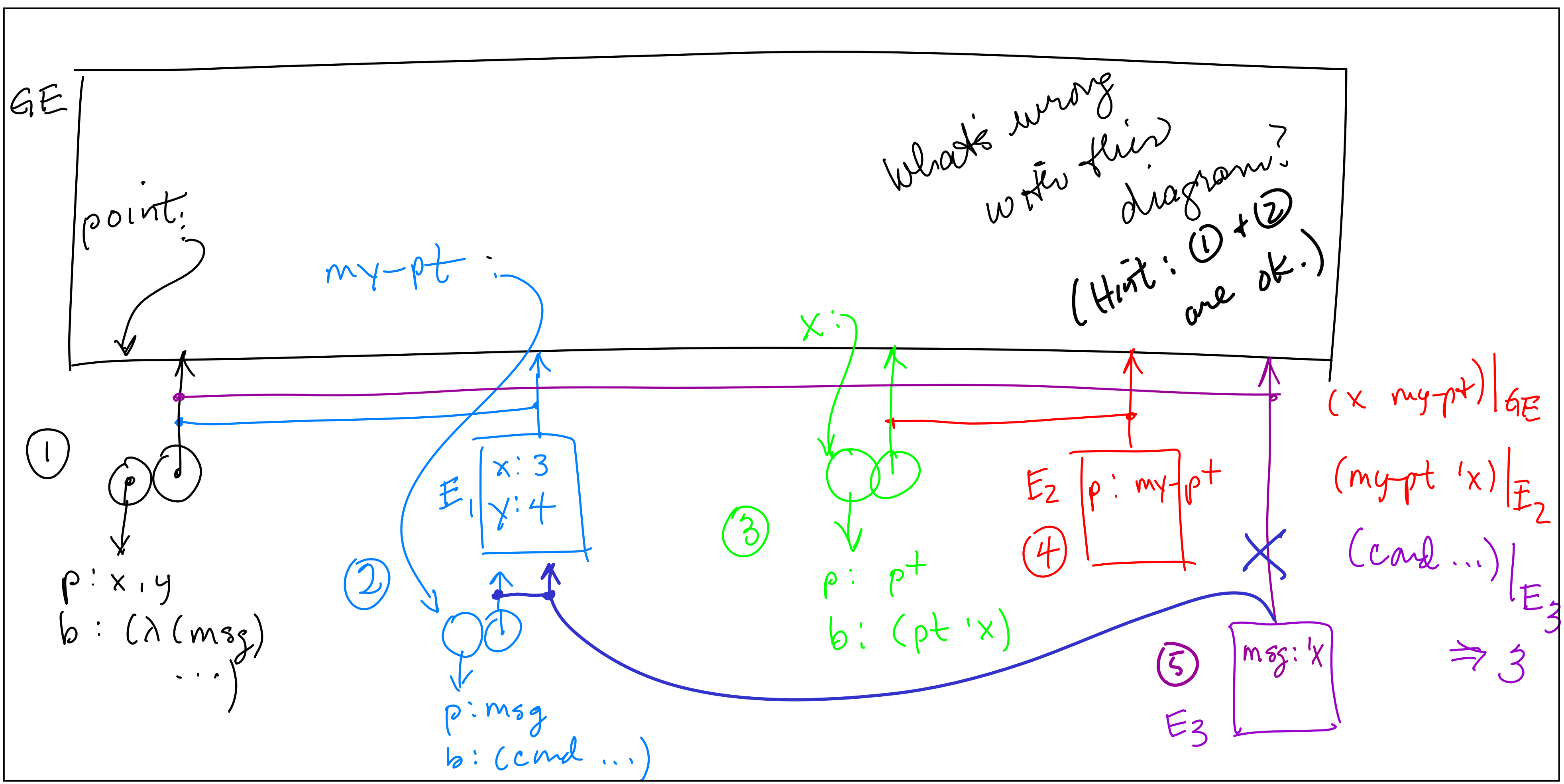
```
(define (point x y)
  (lambda (msg)
    (cond ((eq? msg 'x)  x)
          ((eq? msg 'y)  y)
          ((eq? msg 'point?)  #t))))
```

(5)

(2) (define my-pt (point 3 4))

(3) (define (x pt) (pt 'x))

(4) (x my-pt)



data objects and message passing

Add a method `set-x!` for setting x's value. (Hint: One way requires a change to a part of the procedure not boxed.)

new arg needed for first answer below

```
(define (point x y)
   (lambda (msg)          arg
      (cond ((eq? msg 'x)  x)
            ((eq? msg 'y)  y)
```

```
((eq? msg 'set-x!)  (set!  x  arg))
or
((eq? msg 'set-x!)
  (lambda (arg) (set!  x  arg))
```

```
            ((eq? msg 'point?)  #t))))
```

new syntax: dot for variable number of args

```
(define (mul . args)
   (if  (null?  args)
         1
         (* (car args) (apply mul (cdr args)))))

(mull 2  3  4)  ;  => 24
(mul (list 2 3  4))   ; => error
(mul)  ;  => 1
```

```
(define (cons x y)
  (define (change-car new-car) (set! x new-car))
  (define (change-cdr new-cdr) (set! y new-cdr))
  (lambda (msg . args)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          ((eq? msg 'SET-CAR!)
           (change-car (first args)))
          ((eq? msg 'SET-CDR!)
           (change-cdr (first args)))
          (else (error "pair cannot" msg)))))
```

new syntax: dot for variable number of args

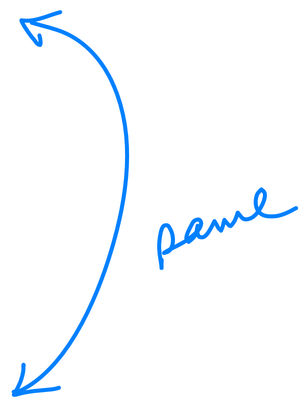
```
(add 1 2)
(add 1 2 3 4)
```

```
(define (add x y . rest)  ...)

   (add 1 2)          =>  x bound to 1
                          y bound to 2
                          rest bound to '()
   (add 1)            => error; requires 2 or more args
   (add 1 2 3)        => rest bound to (3)
   (add 1 2 3 4 5)    => rest bound to (3 4 5)
```

new syntax: case

```
(define (test1 x)
   (cond ((eq? x 'foo) 'got-foo)
          (eq? x 'foo2) 'got-foo)
          (eq? x 'bar)  'got-bar)
          (eq? x 'baz)  'got-baz)
          (else  (error  'uh-oh)))
```




```
(define (test2 x)
   (case x
     ((foo foo2)  'got-foo)
     ((bar)  'got-bar)
     ((baz)  'got-baz)
     (else (error 'uh-oh)))
```

```
(test2  'foo)   ; =>  'got-foo
(test2  'foo2)  ; =>  'got-foo
```

problems (from final exam spring '06)

```
(define (inst1)
  (lambda (msg)
       (let ((n  0))
           (case  msg
                 ((next)  (let ((val  n))  (set!  n  (+ n 1))  val))
                 ((reset)  (set!  n  0)  n)))))
```

each call resets n to 0

n is changed, but not returned

```
(define (inst2)
  (let ((n  0))
    (lambda (msg)
           (case  msg
                 ((next)  (let ((val  n))  (set!  n  (+ n 1))  val))
                 ((reset)  (set!  n  0)  n)))))
```

What is the sequence of values returned for the following expressions? List the values for all the expressions in order, not just the value of the last expression.

1. (define z (ints1))
   (z 'next)
   (z 'next)
   (z 'next)

nothing 0 0 0

2. (define z (ints2))
   (z 'next)
   (z 'next)
   (z 'next)

nothing 0 1 2

classes and instances

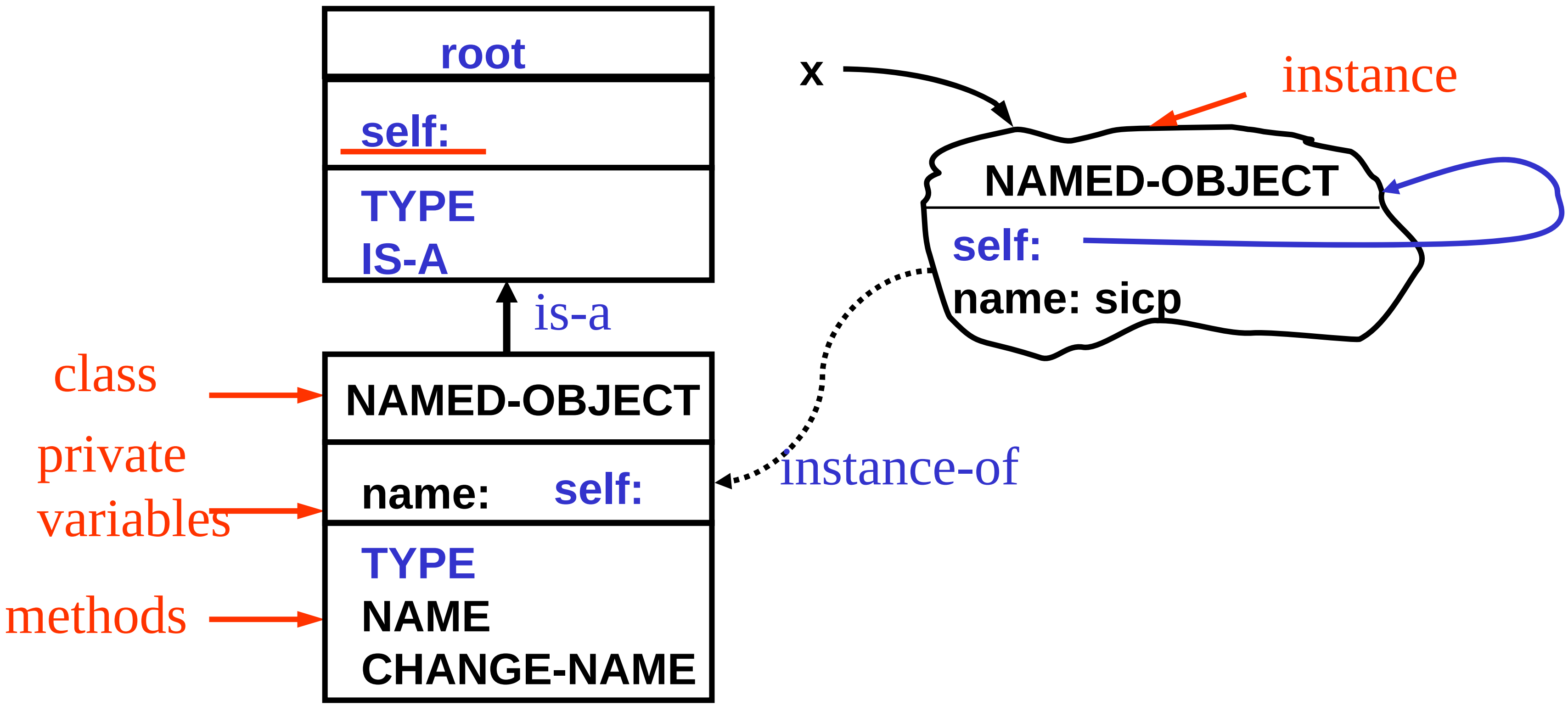
Class:

- specifies the common behavior of entities
- in Scheme, a "maker" procedure

Instance:

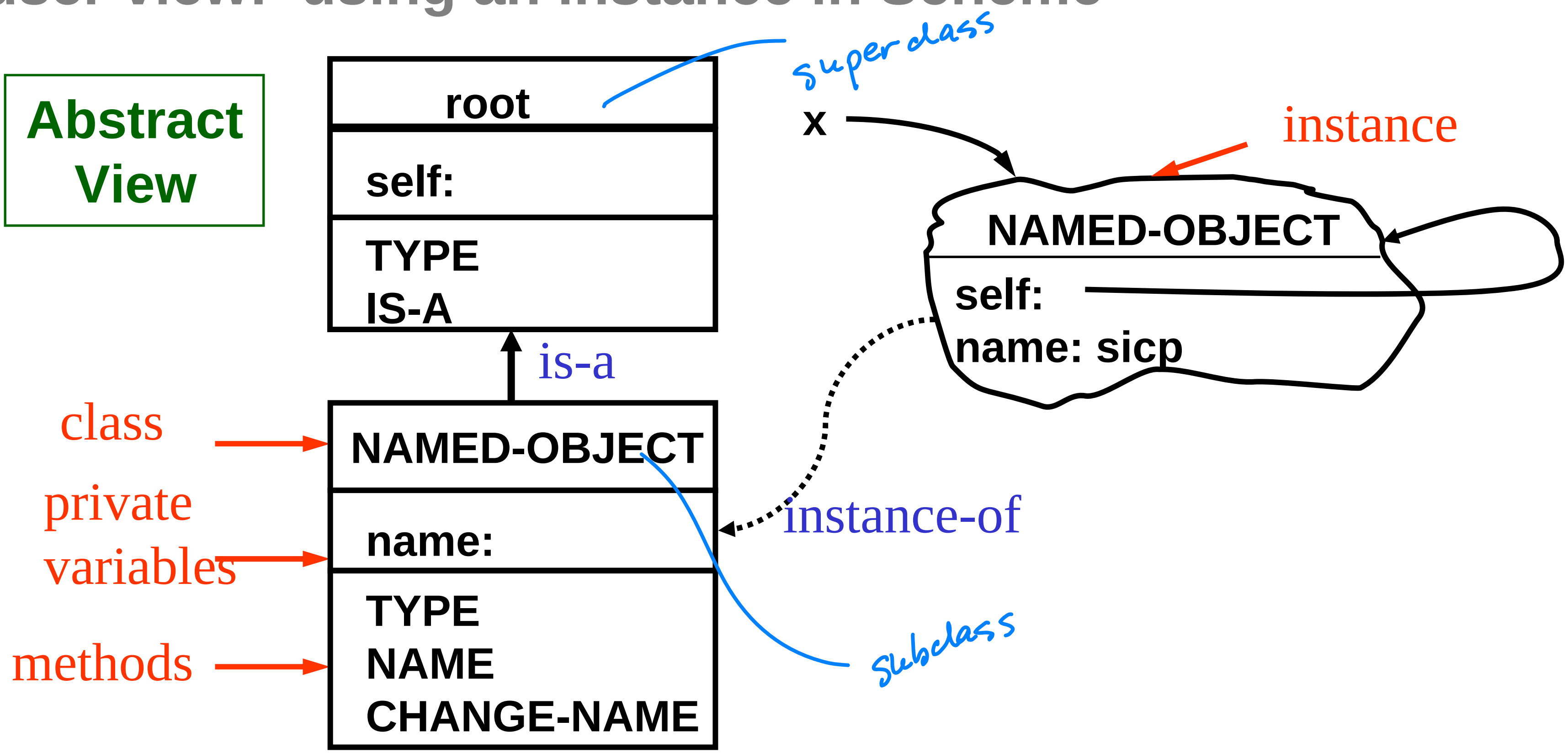
- A particular object or entity of a given class
- in Scheme, an instance is a message-handling procedure made by the maker procedure

object system in Scheme



- Named-object **inherits from our** root **class**
  - Gains a "self" variable
  - Gains an IS-A method
  - Specializes a TYPE method (i.e., overwrites)
  - Also gets a METHODS method

user view: using an instance in Scheme



User View

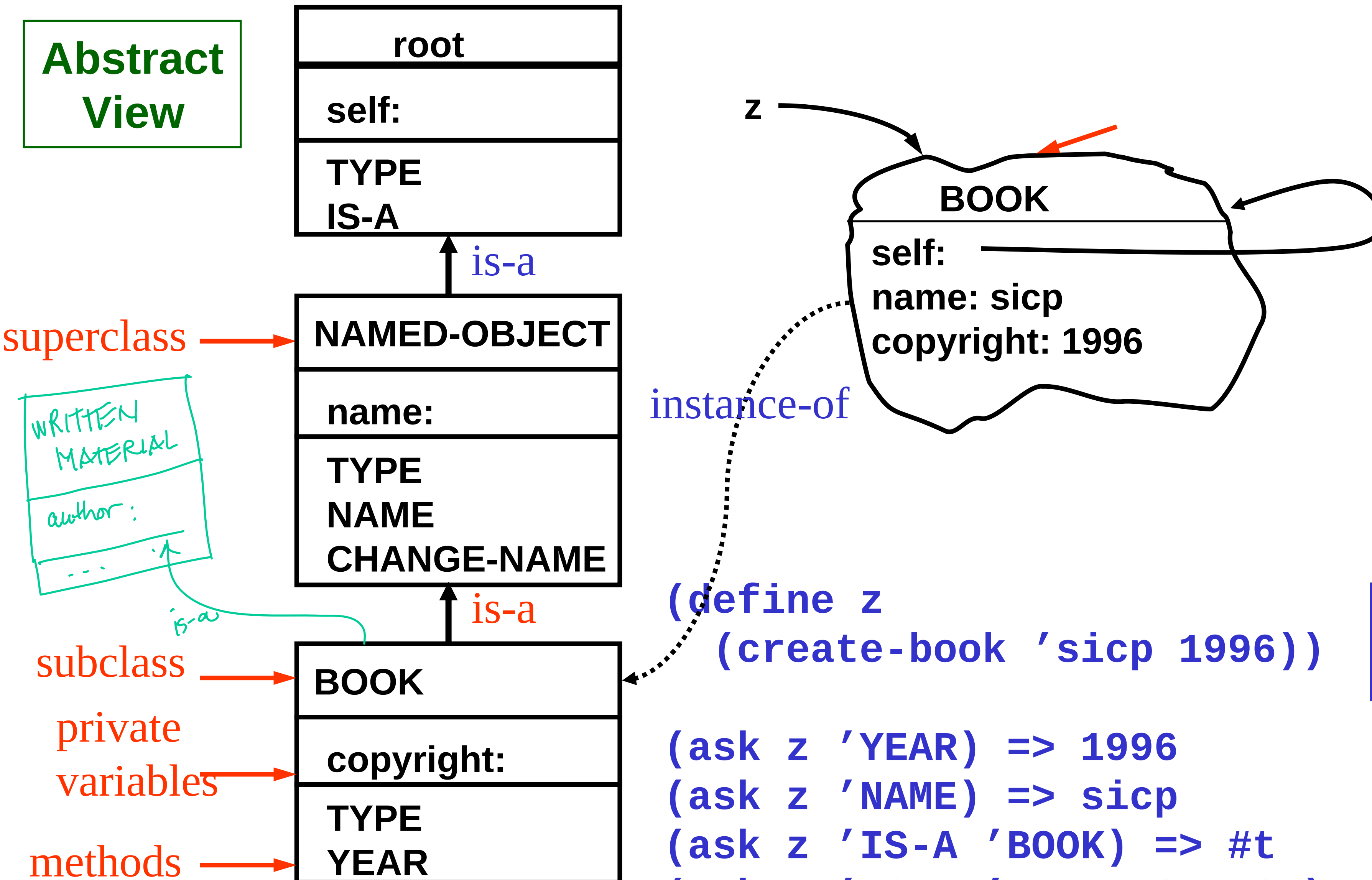
```
(define x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGE-NAME 'sicp-2nd-ed)
(ask x 'NAME) => sicp-2nd-ed

(ask x 'TYPE) => (named-object root)
(ask x 'IS-A 'NAMED-OBJECT) => #t
(ask x 'IS-A 'CLOCK) => #f
```

what does 'ask do?
get-method
+ apply

object system view in Scheme – with inheritance

Abstract View




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

(ask z 'YEAR) => 1996
(ask z 'NAME) => sicp
(ask z 'IS-A 'BOOK) => #t
(ask z 'IS-A 'NAMED-OBJECT) => #t
```

User's View

user's view example: BOOK class with inheritance

```
(define (book self name copyright)
  (let ((named-object-part (named-object self name)))
    (make-handler 'book
      (make-methods
        'YEAR (lambda () copyright))
      named-object)))
```

```
; create-book: symbol, number -> book
(define (create-book name copyright)
  (create-instance book name copyright))
```

user's view example: BOOK class with inheritance

new class

always first arg

local state for class

```
(define (book self name copyright)
  (let ((named-object-part (named-object self name)))
    (make-handler 'book
      (make-methods
        'YEAR (lambda () copyright))
      named-object)))
```

superclass

make superclass

new message

new method

use inherited methods

instance creator for new class

```
; create-book: symbol, number -> book
(define (create-book name copyright)
  (create-instance book name copyright))
```

MAKE-HANDLER does a lot of work

```
(define (make-handler typename methods . super-parts)
  (cond ;check for possible programmer errors
    ((not (symbol? typename))
     (error "bad typename" typename))
    ...
    (else
     (named-lambda (handler message)
      (case message
        ((TYPE)
         (lambda () (type-extend typename super-parts)))
        ((METHODS)
         (lambda ()
           (append (method-names methods)
                   (append-map
                    (lambda (x) (ask x 'METHODS)) super-parts))))
        (else (let ((entry (method-lookup message methods)))
               (if entry
                   (cadr entry)
                   (find-method-from-handler-list
                    message super-parts)))))))))
```