Object-Oriented Style

Daniel P. Friedman

dfried@indiana.edu

Indiana University

Goals of the talk

- Explain conventional OOP

Goals of the talk

- Explain conventional OOP
- Super method call

Goals of the talk

- Explain conventional OOP
- Super method call
- Object method call

Goals of the talk

- Explain conventional OOP
- Super method call
- Object method call
- Using a style for OOP

Meta-goals of the talk

- Make explicit what's static

Meta-goals of the talk

- Make explicit what's static
- Use variables instead of symbols

Meta-goals of the talk

- Make explicit what's static
- Use variables instead of symbols
- Recursion only through it (self or this)

Meta-goals of the talk

- Make explicit what's static
- Use variables instead of symbols
- Recursion only through it (self or this)
- Make what's global potentially local

Meta-goals of the talk

- Make explicit what's static
- Use variables instead of symbols
- Recursion only through it (self or this)
- Make what's global potentially local
- defines could be lets

Meta-goals of the talk

- Make explicit what's static
- Use variables instead of symbols
- Recursion only through it (self or this)
- Make what's global potentially local
- defines could be lets
- define-syntaxes could be let-syntaxes

Structure of the talk

- What is a Style?

Structure of the talk

- What is a Style?
- Familiar Examples

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments
- Installation of Position Environments

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments
- Installation of Position Environments
- Interface Operators

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments
- Installation of Position Environments
- Interface Operators
- The Style

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments
- Installation of Position Environments
- Interface Operators
- The Style
- Familiar Example in the Style

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments
- Installation of Position Environments
- Interface Operators
- The Style
- Familiar Example in the Style
- Protocols in the Style

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments
- Installation of Position Environments
- Interface Operators
- The Style
- Familiar Example in the Style
- Protocols in the Style
- Three ways to Lift Methods

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments
- Installation of Position Environments
- Interface Operators
- The Style
- Familiar Example in the Style
- Protocols in the Style
- Three ways to Lift Methods
- Hygienic Macros (See paper)

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments
- Installation of Position Environments
- Interface Operators
- The Style
- Familiar Example in the Style
- Protocols in the Style
- Three ways to Lift Methods
- Hygienic Macros (See paper)
- Lexical Scope vs. Protected Scope

Structure of the talk

- What is a Style?
- Familiar Examples
- Position Environments
- Installation of Position Environments
- Interface Operators
- The Style
- Familiar Example in the Style
- Protocols in the Style
- Three ways to Lift Methods
- Hygienic Macros (See paper)
- Lexical Scope vs. Protected Scope
- Conclusions

What is a style?

- An encoding of an idiom

What is a style?

- An encoding of an idiom
- Encode everything that matters

What is a style?

- An encoding of an idiom
- Encode everything that matters
- Advantage of programming languages

What is a style?

- An encoding of an idiom
- Encode everything that matters
- Advantage of programming languages
- They hide idioms

What is a style?

- An encoding of an idiom
- Encode everything that matters
- Advantage of programming languages
- They hide idioms
- Disadvantage of programming languages

What is a style?

- An encoding of an idiom
- Encode everything that matters
- Advantage of programming languages
- They hide idioms
- Disadvantage of programming languages
- They hide idioms

What is a style?

- An encoding of an idiom
- Encode everything that matters
- Advantage of programming languages
- They hide idioms
- Disadvantage of programming languages
- They hide idioms
- A style makes explicit what’s implicit

Continuation-Passing is a style

- An encoding of call/cc

Continuation-Passing is a style

- An encoding of call/cc
- Encode every continuation

Continuation-Passing is a style

- An encoding of call/cc
- Encode every continuation
- Advantage of programming languages

Continuation-Passing is a style

- An encoding of call/cc
- Encode every continuation
- Advantage of programming languages
- We don’t see all the continuations

Continuation-Passing is a style

- An encoding of call/cc
- Encode every continuation
- Advantage of programming languages
- We don’t see all the continuations
- Disadvantage of programming languages

Continuation-Passing is a style

- An encoding of call/cc
- Encode every continuation
- Advantage of programming languages
- We don’t see all the continuations
- Disadvantage of programming languages
- Understanding call/cc is hard

Continuation-Passing is a style

- An encoding of call/cc
- Encode every continuation
- Advantage of programming languages
- We don’t see all the continuations
- Disadvantage of programming languages
- Understanding call/cc is hard
- But, not if you learn CPS first.

Mutual-Recursive Example

```
(define vr vector-ref)

(define eo-procs
  (vector
    (lambda (it n)
      (if (zero? n) #t
          ((vr it 1) it (- n 1))))
    (lambda (it n)
      (if (zero? n) #f
          ((vr it 0) it (- n 1))))))

> ((vr eo-procs 0) eo-procs 5)
#f
```

Familiar Example: Color Points

- One chain

Familiar Example: Color Points

- One chain
- <o>: Root

Familiar Example: Color Points

- One chain
- <o>: Root
- <p>: Points:
  `x`, `y`;
  `move`, `get-loc`, `diag`

Familiar Example: Color Points

- One chain
- <o>: Root
- <p>: Points:
  `x`, `y`;
  `move`, `get-loc`, `diag`
- <cp>: Color Points:
  `hue`;
  `get-hue`, `diag&set`

Familiar Example: Color Points

- One chain
- `<o>`: Root
- `<p>`: Points:
  `x`, `y`;
  `move`, `get-loc`, `diag`
- `<cp>`: Color Points:
  `hue`;
  `get-hue`, `diag&set`
- `<scp>`: Stationary Color Points:
  `y`;
  `move`, `show-y`

Familiar Example: Shadows

- Host Class = Host Shadow + Super Class

Familiar Example: Shadows

- Host Class = Host Shadow + Super Class
- One chain

Familiar Example: Shadows

- Host Class = Host Shadow + Super Class
- One chain
- «o»: Root Shadow

Familiar Example: Shadows

- Host Class = Host Shadow + Super Class
- One chain
- «o»: Root Shadow
- «p»: Point Shadow

Familiar Example: Shadows

- Host Class = Host Shadow + Super Class
- One chain
- «o»: Root Shadow
- «p»: Point Shadow
- «cp»: Color Point Shadow

Familiar Example: Shadows

- Host Class = Host Shadow + Super Class
- One chain
- «`o`»: Root Shadow
- «`p`»: Point Shadow
- «`cp`»: Color Point Shadow
- «`scp`»: Stationary Color Point Shadow

Points (no details)

```
(define-syntax <<p>>
  (extend-shadow <<o>> (x y)
    ([move    (method (dx dy) ---)]
     [get-loc (method () ---)]
     [diag    (method (a)
                (move it a a))])))

(define <p>
  (create-class <<p>> <o>))
```

Points

```
(define-syntax <<p>>
  (extend-shadow <<o>> (x y)
    ([move    (method (dx dy)
                (set! x (+ x dx))
                (set! y (+ y dy)))]
     [get-loc (method ()
                (list x y))]
     [diag    (method (a)
                (move it a a))])))
(define <p>
  (create-class <<p>> <o>))
```

Color Points

```
(define-syntax <<cp>>
  (extend-shadow <<p>> (hue)
    ([get-hue  (method () hue)]
     [diag&set (method (a)
                 (diag it a)
                 (set! hue a))])))

(define <cp>
  (create-class <<cp>> <p>))
```

Stationary Color Points

```
(define-syntax <<scp>>
  (extend-shadow <<cp>> (y)
    ([move   (method (x^ y^)
               (show-y it))]
     [diag   (method (a)
               (write hue)
               (diag sup a))]
     [show-y (method ()
               (display y))])))
(define <scp>
  (create-class <<scp>> <cp>))
```

Position Environments

A map from variables to positions
Represented by a list of pairs.

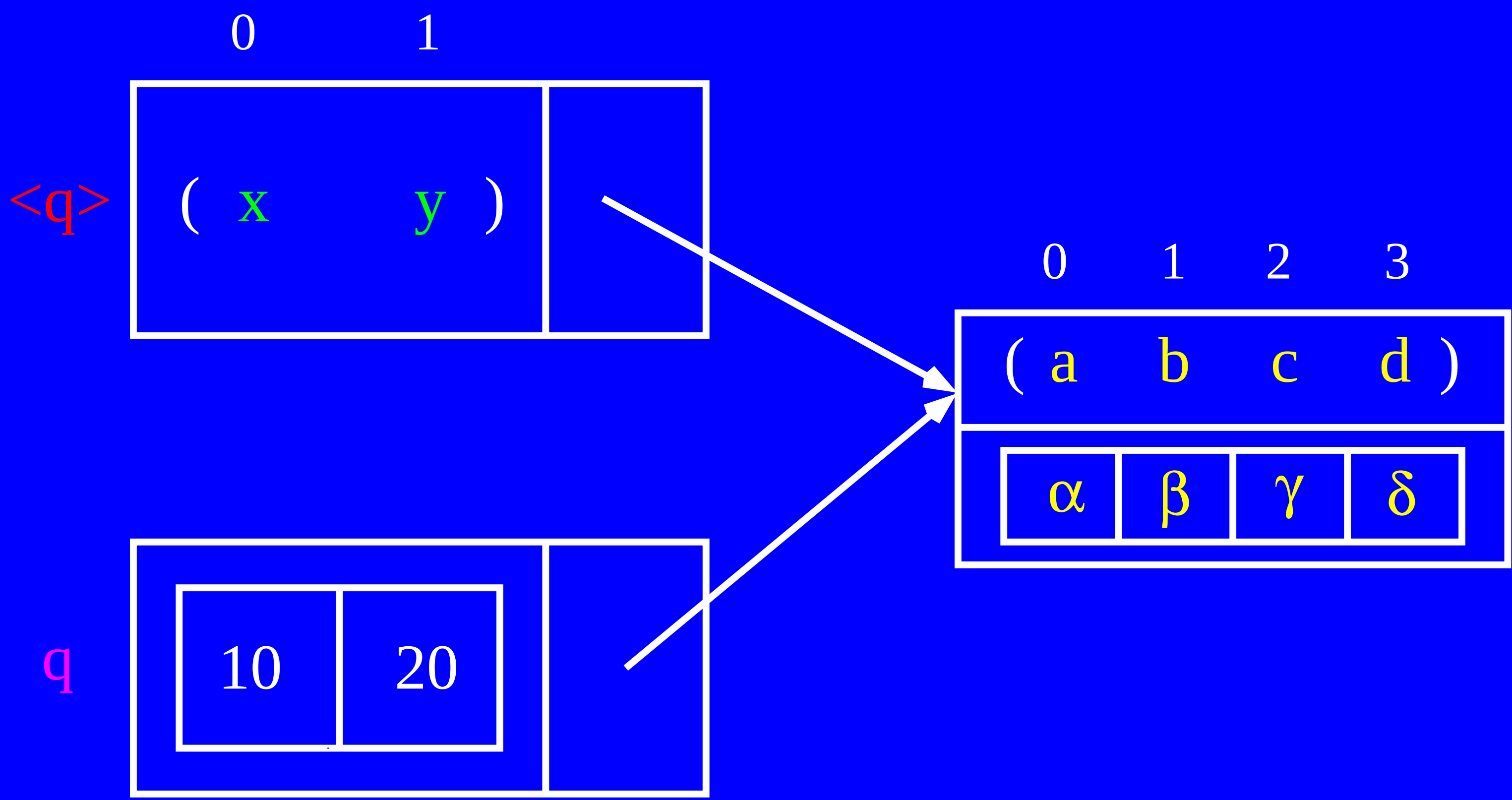
```
(define penv '([a 0][b 1][c 2]))
(define qenv '([a 0][d 1]))

(append-env penv qenv)

==>> ([a 0][b 1][c 2][a 3][d 4])
```

Installation

```
(list 5 3 1 2 6 4)

(let* ([a 0][b 1][c 2][a 3][d 4])
  (list 5 a b c 6 d))

(let ([b 1][c 2][a 3][d 4])
  (list 5 a b c 6 d))

==>> (5 3 1 2 6 4)
```

Data Structures

Classes, Objects, Fields, and Methods



Five Interface operators

Asumme α, β, γ, and δ are closures.

```
<q> = ((x y) (a b c d) #(α β γ δ))
q = ‘(#(10 20) . ,(cdr <q>))

(fx <q> ’(y)) = (x y y)
(mx <q> ’(e)) = (a b c d e)

(fp q 1) = 20
(fp! q 1 30) = unspecified
(fp q 1) = 30
(mp q 2) = γ
(mp <q> 2) = γ
```

Binding variables to values

- Because we know that there is a one-to-one correspondence between the variables in the field environment of a class and the positions in the field vector of its associated class, we can think of the fields in the vector as if they had a name.

Binding variables to values

- Because we know that there is a one-to-one correspondence between the variables in the field environment of a class and the positions in the field vector of its associated class, we can think of the fields in the vector as if they had a name.
- Because we know that there is a one-to-one correspondence between the variables in the method environment of a class and the positions in the method vector of the class, we can think of the methods in the vector as if they had a name.

The Style Template

```
(list
  (fx <super> '(field-var ...))
  (mx <super> '(method-var ...))
  (vector
    (lambda (it arg ...) ---)
    (lambda (it arg ...) ---)
    (mp <super> 2)
    (lambda (it arg ...) ---)
    (mp <super> 4)
    (mp <super> 5)
    (lambda (it arg ...) ---)
    (lambda (it arg ...) ---)
    (lambda (it arg ...) ---)))
```

The Style (Part 1)

- The arguments to fx are the super class and the fresh field variables. These variables cannot contain duplicates, and their order matters.

The Style (Part 1)

- The arguments to fx are the super class and the fresh field variables. These variables cannot contain duplicates, and their order matters.
- The arguments to mx are the super class and the fresh method variables. These variables cannot contain duplicates, their order matters, and they are different from those in the super class.

The Style (Part 2)

- Each method of the host class is determined in one of three ways

The Style (Part 2)

- Each method of the host class is determined in one of three ways
- 1. The result of evaluating an expression (e.g., an mp expression), yielding a *contributed* (or *inherited*) method, or

The Style (Part 2)

- Each method of the host class is determined in one of three ways
- 1. The result of evaluating an expression (e.g., an `mp` expression), yielding a *contributed* (or *inherited*) method, or
- 2. The result of evaluating an expression (e.g., a `lambda` expression), yielding a *replaced* (or *overridden*) method, or

The Style (Part 2)

- Each method of the host class is determined in one of three ways
- 1. The result of evaluating an expression (e.g., an mp expression), yielding a *contributed* (or *inherited*) method, or
- 2. The result of evaluating an expression (e.g., a `lambda` expression), yielding a *replaced* (or *overridden*) method, or
- 3. The result of evaluating an expression (e.g., a `lambda` expression), yielding a *fresh* method.

The Style (Part 3)

- A method is contributed if its associated variable is in the domain of the super method environment, and it is not replaced.

The Style (Part 3)

- A method is contributed if its associated variable is in the domain of the super method environment, and it is not replaced.
- The contributed methods fill in the vector with `(mp <super> position)`.

The Style (Part 3)

- A method is contributed if its associated variable is in the domain of the super method environment, and it is not replaced.
- The contributed methods fill in the vector with `(mp <super> position)`.
- As the method vector is filled in, each method must fit into the right position. The replaced and contributed methods must be in the same position as in their super class. The fresh methods follow these, and they must be in the order they appear in the call to mx.

The Style (Part 4)

- Some methods are not expressed as a procedure built from a `lambda` expression, but those that are, have it, which may be bound to an object or a class, as their first argument.

The Style (Part 4)

- Some methods are not expressed as a procedure built from a `lambda` expression, but those that are, have it, which may be bound to an object or a class, as their first argument.
- When it is bound to an object, we can reference or update its fields through a constant position in its field vector.

The Style (Part 4)

- Some methods are not expressed as a procedure built from a `lambda` expression, but those that are, have it, which may be bound to an object or a class, as their first argument.
- When it is bound to an object, we can reference or update its fields through a constant position in its field vector.
- Every object uses the same position in its field vector for each field defined by the class of its object.

The Style (Part 5)

- To invoke a method of an object or a class, first obtain the method through a constant position of a method vector and invoke it on some object or class and perhaps some additional arguments.

The Style (Part 5)

- To invoke a method of an object or a class, first obtain the method through a constant position of a method vector and invoke it on some object or class and perhaps some additional arguments.
- If the method is from an object, then its first argument is the object.

The Style (Part 5)

- To invoke a method of an object or a class, first obtain the method through a constant position of a method vector and invoke it on some object or class and perhaps some additional arguments.
- If the method is from an object, then its first argument is the object.
- If the method is from a class, then its first argument is either the class or it.

The Style (Part 5)

- To invoke a method of an object or a class, first obtain the method through a constant position of a method vector and invoke it on some object or class and perhaps some additional arguments.
- If the method is from an object, then its first argument is the object.
- If the method is from a class, then its first argument is either the class or it.
- There are no restrictions on the method bodies.

The Style (Part 6)

- If a class is passed to an interface operator, it should be the host's super class.

The Style (Part 6)

- If a class is passed to an interface operator, it should be the host's super class.
- There are no more constraints on how the method vector is built.

Points in the Style

```
(define <p>
  (list
    (fx <o> '(x y))
    (mx <o> '(move get-loc diag))
    (vector
      (lambda (it dx dy)
        (fp! it 0 (+ (fp it 0) dx))
        (fp! it 1 (+ (fp it 1) dy)))
      (lambda (it)
        (list (fp it 0) (fp it 1)))
      (lambda (it a)
        ((mp it 0) it a a)))))
```

Color Points in the Style

```
(define <cp>
  (list
    (fx <p> '(hue))
    (mx <p> '(get-hue diag&set))
    (vector
      (mp <p> 0)
      (mp <p> 1)
      (mp <p> 2)
      (lambda (it) (fp it 2))
      (lambda (it a)
        ((mp it 1) it a)
        (fp! it 2 a)))))
```

Stat. Color Points in the Style

```
(define <scp>
  (list
    (fx <cp> '(y))
    (mx <cp> '(show-y))
    (vector
      (lambda (it x^ y^)
        ((mp it 5) it))
      (mp <cp> 1)
      (lambda (it a)
        (write (fp it 2))
        ((mp <cp> 2) it a))
      (mp <cp> 3)
      (mp <cp> 4)
      (lambda (it)
        (display (fp it 3))))))
```

Three Protocols

- sup is a lexical variable

Three Protocols

- sup is a lexical variable
- Installing method environments

Three Protocols

- sup is a lexical variable
- Installing method environments
- Installing field environments

Stationary Color Points: super

```
(define <scp>
  (let ([sup <cp>])
    (list
      (fx sup '(y))
      (mx sup '(show-y))
      (vector
        (lambda (it x^ y^)
          ((mp sup 5) it))
        (mp sup 1)
        (lambda (it a)
          (write (fp it 2))
          ((mp sup 2) it a))
        (mp sup 3)
        (mp sup 4)
        (lambda (it)
          (display (fp it 3)))))))
```

Installing **method** envs: (Part 1)

```
(define <scp>
  (let ([move 0]
        [get-loc 1]
        [diag 2]
        [get-hue 3]
        [diag&set 4]
        [show-y 5]
    (let ([sup <cp>])
      --------------------------))))
```

Installing method envs: (Part 2)

```
(list
  (fx sup '(y))
  (mx sup '(show-y))
  (vector
    (lambda (it x^ y^)
      ((mp it show-y) it))
    (mp sup get-loc)
    (lambda (it a)
      (write (fp it 2))
      ((mp sup diag) it a))
    (mp sup get-hue)
    (mp sup diag&set)
    (lambda (it)
      (display (fp it 3))))
```

Installing field envs: (Part 1)

```
(define <scp>
  (let* ([x 0][y 1][hue 2][y 3])
    (let ([move 0]
          [get-loc 1]
          [diag 2]
          [get-hue 3]
          [diag&set 4]
          [show-y 5])
      (let ([sup <cp>])
        ---------------------))))
```

Installing field envs: (Part 2)

```
(list
  (fx sup '(y))
  (mx sup '(show-y))
  (vector
    (lambda (it x^ y^)
      ((mp it show-y) it))
    (mp sup get-loc)
    (lambda (it a)
      (write (fp it hue))
      ((mp sup diag) it a))
    (mp sup get-hue)
    (mp sup diag&set)
    (lambda (it)
      (display (fp it y)))))
```

Fully-colorized (Part 1)

```
(define <scp>
  (let* ([x 0][y 1][hue 2][y 3])
    (let ([move 0]
          [get-loc 1]
          [diag 2]
          [get-hue 3]
          [diag&set 4]
          [show-y 5])
      (let ([sup <cp>])
        ---------------------)))))
```

Everything above this line remains unchanged.

Fully-colorized (Part 2)

```
(list
  (fx sup '(y))
  (mx sup '(show-y))
  (vector
    (lambda (it x^ y^)
      ((mp it show-y) it))
    (mp sup get-loc)
    (lambda (it a)
      (write (fp it hue))
      ((mp sup diag) it a))
    (mp sup get-hue)
    (mp sup diag&set)
    (lambda (it)
      (display (fp it y)))))
```

Three Ways to Lift Methods

- Naive Lifting

Three Ways to Lift Methods

- Naive Lifting
- Triply-Nested let

Three Ways to Lift Methods

- Naive Lifting
- Triply-Nested let
- Quadruply-Nested let

Naive Lifting: (Part 2)

```
(let ([move   (lambda ---)]
      [diag   (lambda ---)]
      [show-y (lambda ---)])
   (list
     (fx sup '(y))
     (mx sup '(show-y))
     (vector
       move
       (mp sup get-loc)
       diag
       (mp sup get-hue)
       (mp sup diag&set)
       show-y)))
```

Triply-Nested let: (Part 2)

```
(let ([h-move   (lambda ---)]
      [h-diag   (lambda ---)]
      [h-show-y (lambda ---)])
  (let ([move       (mp sup move)]
        [get-loc  (mp sup get-loc)]
        [diag       (mp sup diag)]
        [get-hue  (mp sup get-hue)]
        [diag&set (mp sup diag&set)])
```

Nested let: (Part 3)

Everything below this line remains unchanged.

```
-------------------------------------
    (let ([move   h-move]
          [diag   h-diag]
          [show-y h-show-y])
      (list
        (fx sup '(y))
        (mx sup '(show-y))
        (vector
          move
          diag
          get-loc
          get-hue
          diag&set
          show-y)))
```

Quadruply-Nested let: (Part 2)

```
(let ([s-move       (mp sup move)]
      [s-get-loc    (mp sup get-loc)]
      [s-diag       (mp sup diag)]
      [s-get-hue    (mp sup get-hue)]
      [s-diag&set   (mp sup diag&set)])
  (let ([h-move     (lambda ---)]
        [h-diag     (lambda ---)]
        [h-show-y   (lambda ---)])
    (let ([move      s-move]
          [get-loc   s-get-loc]
          [diag      s-diag]
          [get-hue   s-get-hue]
          [diag&set  s-diag&set])
      -----------------------------
```

Zoom in on Methods: (Part 3)

```
(let ([h-move
        (lambda (it x^ y^)
          ((mp it show-y) it))]
      [h-diag
        (lambda (it a)
          (write (fp it hue))
          (s-diag it a))]
      [h-show-y
        (lambda (it)
          (display (fp it y)))]))
```

Zoom out on diag: (Part 3)

```
(let* ([x 0][y 1][hue 2][y 3])
  (let ([diag 2] ...)
    (let ([sup <cp>] ...)
      (let ([s-diag ---] ...)
        (let ([h-diag ---] ...)
          (let ([diag s-diag] ...)
            (let ([diag h-diag] ...)
              (list
                (fx ...)
                (mx ...)
                (vector
                  ... diag ...))))))))))
```

Two Scope Issues

- Lexical Scope

Two Scope Issues

- Lexical Scope
- Protected Scope

Two Scope Issues

- Lexical Scope
- Protected Scope
- Which one should shadow the other?

Two Scope Issues

- Lexical Scope
- Protected Scope
- Which one should shadow the other?
- Where should one shadow the other?

Two Scope Issues

- Lexical Scope
- Protected Scope
- Which one should shadow the other?
- Where should one shadow the other?
- Assume that we have extended <scp> with the method show-y.

Lexical Scope (Part 1)

```
[show-y
  (let ([hue* "outside "]
        [diag* (lambda (x y)
                 (display
                   "moving "))])
    (method ()
      (display hue)
      (diag* 5 5)
      (let ([hue "inside "]
            [diag (lambda (n self)
                    (diag self n))])
        (display hue)
        (diag 5 it))))]
```

Lexical Scope (Part 2)

```
[show-y
  (let ([hue "outside "]
        [diag (lambda (x y)
                (display
                  "moving "))])
    (method ()
      (display hue)
      (diag 5 5)
      (let ([hue "inside "]
            [diag (lambda (n self)
                    (diag self n))])
        (display hue)
        (diag 5 it)))))]
```

Lexical Scope (Part 3)

```
(define <e>-maker
  (lambda (x)
    (let-syntax
      ([<<e>>
         (extend-shadow <<scp>> ()
           ([e
              (begin
                (write x)
                (let ([y 1])
                  (method (q . a)
                    (+ x y q
                     (car a))))))]))])
      (lambda (s)
        (create-class <<e>> s)))))
```

Conclusions

- Meta-goal: Everything as static as possible

Conclusions

- Meta-goal: Everything as static as possible
- Goal: Clarified super and object method calls

Conclusions

- Meta-goal: Everything as static as possible
- Goal: Clarified super and object method calls
- Continuation-Passing Style vs.
  Object-Oriented Style

Conclusions

- Meta-goal: Everything as static as possible
- Goal: Clarified super and object method calls
- Continuation-Passing Style vs.
  Object-Oriented Style
- Both get their power by harnessing properties
  with an extra argument