

# FP vs. OOP: case studies in misunderstandings

Panicz Maciej Godek

[godek.maciek@gmail.com](mailto:godek.maciek@gmail.com)  
@PaniczGodek

**Racketfest, 23.03.2019**

# Definition of a paradigm

**paradigm** – a distinct set of concepts or thought patterns, including theories, research methods, postulates, and standards for what constitutes a legitimate contribution to a field. [Wikipedia]



# Definition of functional programming

functional programming – a style of programming where a programmer is allowed only to define pure functions, by composing them from other pure functions



AS WE HAVE SEEN,  
THE SET! OPERATOR  
ENABLES US TO MODEL  
OBJECTS THAT HAVE  
LOCAL STATE.

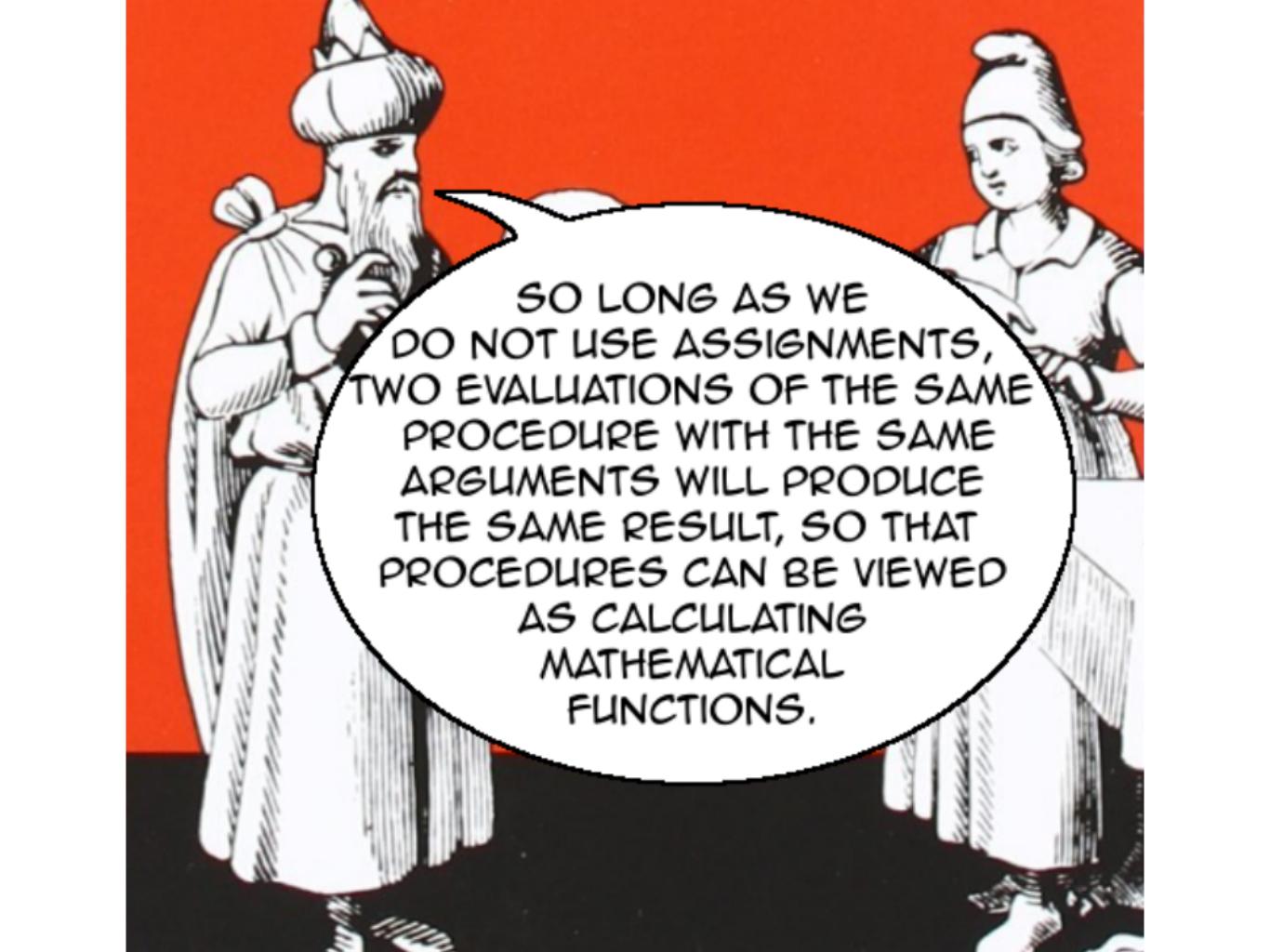


HOWEVER,  
THIS ADVANTAGE  
COMES AT  
A PRICE.

Equal  
Apple



OUR PROGRAMMING  
LANGUAGE CAN  
NO LONGER BE  
INTERPRETED IN TERMS  
OF THE SUBSTITUTION  
MODEL OF PROCEDURE  
APPLICATION [...]



SO LONG AS WE  
DO NOT USE ASSIGNMENTS,  
TWO EVALUATIONS OF THE SAME  
PROCEDURE WITH THE SAME  
ARGUMENTS WILL PRODUCE  
THE SAME RESULT, SO THAT  
PROCEDURES CAN BE VIEWED  
AS CALCULATING  
MATHEMATICAL  
FUNCTIONS.

PROGRAMMING WITHOUT  
ANY USE OF ASSIGNMENTS,  
AS WE DID THROUGHOUT  
THE FIRST TWO CHAPTERS OF THIS BOOK,  
IS ACCORDINGLY KNOWN AS  
**FUNCTIONAL PROGRAMMING**



# Referential opacity

```
(define (compile expression result)
  (match expression
    ('(, <*> , a , b)
     (let ((a+ (name a))
           (b+ (name b)))
       '(', @ (compile a a+)
            , @ (compile b b+)
            , result = , a+ , <*> , b+))))
```

```
(_
  ' ())) )
```

# Referential opacity

```
(define (compile expression result)
  (match expression
    ('(, <*> , a , b)
     (let ((a+ (name a))
           (b+ (name b)))
       '(', @ (compile a a+)
            , @ (compile b b+)
            , result = , a+ , <*> , b+))))
```

```
(_
  ' ())) )
```

# Referential opacity

```
(define (compile expression result)
  (match expression
    ('(, <*> , a , b)
     (let ((a+ (name a))
          (b+ (name b)))
       ' (, @ (compile a a+)
              , @ (compile b b+)
              , result = , a+ , <*> , b+)))))
  (_
   ' ())) )
```

# Referential opacity

```
(define (compile expression result)
  (match expression
    ('(, <*> , a , b)
     (let ((a+ (name a))
           (b+ (name b)))
       `',(,@(compile a a+)
             ,@(compile b b+)
             ,result = ,a+ ,<*> ,b+)))))
  (_
   ' ())) )
```

# Referential opacity

```
(define (compile expression result)
  (match expression
    ('(, <*> , a , b)
     (let ((a+ (name a))
           (b+ (name b)))
       '(', @ (compile a a+)
            , @ (compile b b+)
            , result = , a+ , <*> , b+))))  
(_  
' ()))
```

# Referential opacity

```
(define name
  (let ((counter 0))
    (lambda (expression)
      (cond ((symbol? expression)
              expression)
            (else
              (set! counter (+ counter 1))
              (symbol-append 't
                (number->symbol counter)))))))
```

# Referential opacity

```
(define name
  (let ((counter 0))
    (lambda (expression)
      (cond ((symbol? expression)
              expression)
            (else
              (set! counter (+ counter 1))
              (symbol-append 't
                (number->symbol counter)))))))
```

# Referential opacity

```
(define name
  (let ((counter 0))
    (lambda (expression)
      (cond ((symbol? expression)
              expression)
            (else
              (set! counter (+ counter 1))
              (symbol-append 't
                (number->symbol counter)))))))
```

# Referential opacity

```
(define name
  (let ((counter 0))
    (lambda (expression)
      (cond ((symbol? expression)
              expression)
            (else
              (set! counter (+ counter 1))
              (symbol-append 't
                (number->symbol counter)))))))
```

# Referential opacity

```
(compile '(* (+ x y) (/ z w)) 'result)
```

# Referential opacity

```
(match '(* (+ x y) (/ z w))  
  ('(, <*> , a , b)  
   (let ((a+ (name a))  
         (b+ (name b)))  
     '(', @compile a a+)  
     , @compile b b+)  
     (result = , a+ , <*> , b+))))  
  (_  
   ' ( ) ))
```

# Referential opacity

```
(match '(* (+ x y) (/ z w))  
  ('(, <*> , a , b)  
   (let ((a+ (name a))  
         (b+ (name b)))  
     ' (, @ (compile a a+)  
            , @ (compile b b+)  
            (result = , a+ , <*> , b+))))  
  (_  
   ' ()))
```

# Referential opacity

```
(let ((a+ (name ' (+ x y) ))  
      (b+ (name ' (/ z w) )) )  
  ` (, @ (compile ' (+ x y) a+)  
    , @ (compile ' (/ z w) b+)  
    (result = , a+ * , b+) ) )
```

# Referential opacity

```
(let ((a+ (name ' (+ x y)))  
      (b+ (name ' (/ z w)) ))  
  ` (, @ (compile ' (+ x y) a+)  
    , @ (compile ' (/ z w) b+)  
    (result = , a+ * , b+)) )
```

# Referential opacity

```
(let ((a+ (name ' (+ x y)))  
      (b+ (name ' (/ z w))))  
  ` (, @ (compile ' (+ x y) a+)  
            , @ (compile ' (/ z w) b+)  
            (result = , a+ * , b+)) )
```

# Referential opacity

```
(let ((a+ 't1)
      (b+ 't2))
  `(~@compile '(+ x y) a+)
   ,~@compile '(/ z w) b+)
  (result = ,a+ * ,b+)) )
```

# Referential opacity

```
'(, @compile '(+ x y) 't1)
 , @compile '(/ z w) 't2)
(result = t1 * t2))
```

# Referential opacity

```
'(, @compile '(+ x y) 't1)
 , @compile '(/ z w) 't2)
(result = t1 * t2))
```

# Referential opacity

```
' ((t1 = x + y)
  (t2 = z / w)
  (result = t1 * t2))
```

# Non-determinism

```
(define (shuffle l)
  (let ((n (length l)))
    (if (is n < 2)
        l
        (let ((left right
              (split-at l
                        (+ (random (- n 1))
                           1))))
          (if (= (random 2) 1)
              ` (,@(shuffle right)
                  ,@(shuffle left))
              ` (,@(shuffle left)
                  ,@(shuffle right)))))))
```

# Non-determinism

```
(define (shuffle l)
  (let ((n (length l)))
    (if (is n < 2)
        l
        (let ((left right
              (split-at l
                         (+ (random (- n 1))
                            1))))
          (if (= (random 2) 1)
              ` (,@(shuffle right)
                  ,@(shuffle left))
              ` (,@(shuffle left)
                  ,@(shuffle right)))))))
```

# Non-determinism

```
(define (shuffle l)
  (let ((n (length l)))
    (if (is n < 2)
        l
        (let ((left right
              (split-at l
                (+ (random (- n 1))
                    1))))
          (if (= (random 2) 1)
              ` (,@(shuffle right)
                  ,@(shuffle left))
              ` (,@(shuffle left)
                  ,@(shuffle right)))))))
```

# Non-determinism

```
(define (shuffle l)
  (let ((n (length l)))
    (if (is n < 2)
        l
        (let ((left right
              (split-at l
                         (+ (random (- n 1))
                            1))))
          (if (= (random 2) 1)
              `(@(shuffle right)
                 ,@ (shuffle left))
              `(@(shuffle left)
                 ,@ (shuffle right)))))))
```

# Non-determinism

```
(define (shuffle l)
  (let ((n (length l)))
    (if (is n < 2)
        l
        (let ((left right
              (split-at l
                         (+ (random (- n 1))
                            1))))
          (if (= (random 2) 1)
              ` (,@(shuffle right)
                  ,@(shuffle left))
              ` (,@(shuffle left)
                  ,@(shuffle right)))))))
```

# Non-determinism

```
(shuffle ' (a b c) )
```

# Non-determinism

```
(let ((n (length '(a b c))))
  (if (is n < 2)
      '(a b c)
      (let ((left right
            (split-at '(a b c)
                      (+ (random (- n 1))
                         1))))
        (if (= (random 2) 1)
            `',(,@(shuffle right)
                  ,@(shuffle left))
            `',(,@(shuffle left)
                  ,@(shuffle right)))))))
```

# Non-determinism

```
(let ((n 3))
  (if (is n < 2)
    ' (a b c)
    (let ((left right
           (split-at ' (a b c)
                     (+ (random (- n 1))
                        1))))
      (if (= (random 2) 1)
        ` (,@(shuffle right)
            ,@(shuffle left))
        ` (,@(shuffle left)
            ,@(shuffle right))))))
```

# Non-determinism

```
(if (is 3 < 2)
  ' (a b c)
  (let ((left right
         (split-at ' (a b c)
                   (+ (random (- 3 1))
                      1)))))

  (if (= (random 2) 1)
      ` (,@(shuffle right)
          ,@(shuffle left))
      ` (,@(shuffle left)
          ,@(shuffle right))))
```

# Non-determinism

```
(let ((left right
      (split-at '(a b c)
      (+ (random 2)
         1)))))

(if (= (random 2) 1)
  ` (,@(shuffle right)
    ,@(shuffle left))
  ` (,@(shuffle left)
    ,@(shuffle right))))
```

# Non-determinism

```
(let ((left right
      (split-at '(a b c)
                (+ (random 2)
                   1)))))

(if (= (random 2) 1)
    ` (,@(shuffle right)
       ,@(shuffle left))
    ` (,@(shuffle left)
       ,@(shuffle right))))
```

# Non-determinism

```
(let* ((x (random 2))
      (left right
            (split-at ' (a b c)
                      (+ x
                          1)))))

(if (= x 1)
    ` (, @ (shuffle right)
           , @ (shuffle left))
    ` (, @ (shuffle left)
           , @ (shuffle right))))
```

# Non-determinism

```
(let ((left right
      (split-at '(a b c)
                (+ (random 2)
                   1)))))

(if (= (random 2) 1)
    ` (,@(shuffle right)
       ,@(shuffle left))
    ` (,@(shuffle left)
       ,@(shuffle right))))
```

# Non-determinism

```
(let ((left right (split-at '(a b c)
                               (+ (random 2) 1))))
  (if (= (random 2) 1)
      `',(,@(shuffle right)
        ,@(shuffle left))
      `',(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                               (+ (random 2) 1))))
  (if (= (random 2) 1)
      `',(,@(shuffle right)
        ,@(shuffle left))
      `',(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                               (+ (random 2) 1))))
  (if (= (random 2) 1)
      `',(,@(shuffle right)
        ,@(shuffle left))
      `',(,@(shuffle left)
        ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                               (+ (random 2) 1))))
  (if (= (random 2) 1)
      `',(,@(shuffle right)
        ,@(shuffle left))
      `',(,@(shuffle left)
        ,@(shuffle right))))
```

# Non-determinism

```
(let ((left right (split-at '(a b c)
                               (+ 0 1))))
     (if (= 0 1)
         `',(,@(shuffle right)
           ,@(shuffle left))
         `',(,@(shuffle left)
           ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                               (+ 1 1))))
     (if (= 0 1)
         `',(,@(shuffle right)
           ,@(shuffle left))
         `',(,@(shuffle left)
           ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                               (+ 0 1))))
     (if (= 1 1)
         `',(,@(shuffle right)
           ,@(shuffle left))
         `',(,@(shuffle left)
           ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c)
                               (+ 1 1))))
     (if (= 1 1)
         `',(,@(shuffle right)
           ,@(shuffle left))
         `',(,@(shuffle left)
           ,@(shuffle right))))
```

# Non-determinism

```
(let ((left right (split-at '(a b c)
                               1)))
  (if (= 0 1)
      `',(,@(shuffle right)
        ,@(shuffle left))
      `',(,@(shuffle left)
        ,@(shuffle right)))))

(let ((left right (split-at '(a b c)
                               2)))
  (if (= 0 1)
      `',(,@(shuffle right)
        ,@(shuffle left))
      `',(,@(shuffle left)
        ,@(shuffle right)))))

(let ((left right (split-at '(a b c)
                               1)))
  (if (= 1 1)
      `',(,@(shuffle right)
        ,@(shuffle left))
      `',(,@(shuffle left)
        ,@(shuffle right)))))

(let ((left right (split-at '(a b c)
                               2)))
  (if (= 1 1)
      `',(,@(shuffle right)
        ,@(shuffle left))
      `',(,@(shuffle left)
        ,@(shuffle right)))))
```

## Non-determinism

# Non-determinism

```
' (a b c)  
' (a c b)
```

```
' (b c a)  
' (c b a)
```

```
' (a b c)  
' (b a c)
```

```
' (c a b)  
' (c b a)
```

# Non-determinism

' (a b c)  
' (a c b)

' (b c a)  
' (c b a)

' (a b c)  
' (b a c)

' (c a b)  
' (c b a)

# Purity

```
main :: IO ()  
main = do {  
    name <- getLine;  
    putStrLn "Hello "++name;  
}
```

# FP vs OOP

2 + 2

(+) :: Num a => a -> a -> a

2.+ (2)

# FP vs OOP

$$2 + 2$$

(+) :: Num a => a -> a -> a

2.+ (2)

# FP vs OOP

$$2 + 2$$

(+) :: Num a => a -> a -> a

2.+ (2)





I COINED THE TERM  
"OBJECT ORIENTED  
PROGRAMMING"



Smalltalk is the recursion  
on the notion of the computer  
itself

# The notion of the computer (live demo)

1.rkt

<https://github.com/panicz/sracket>

# The notion of the computer

```
(define (draggable-rectangle left top width
                             height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) ...)
        (`(mouse-down ,x ,y) ...)
        (`(mouse-up ,x ,y) ...)
        (`(mouse-move ,x ,y ,dx ,dy) ...)
        (`(embraces? ,x ,y) ...)
        (`(as-image) ...)
        (`(mouse-out) ...)
        (_ #false))))
```

# The notion of the computer

```
(define (draggable-rectangle left top width height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(mouse-out) ...)
        (_ #false))))
```

# The notion of the computer

```
(define (draggable-rectangle left top width
                             height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) `,(left ,top))
        (`(mouse-down ,x ,y) ...)
        (`(mouse-up ,x ,y) ...)
        (`(mouse-move ,x ,y ,dx ,dy) ...)
        (`(embraces? ,x ,y) ...)
        (`(as-image) ...)
        (`(mouse-out) ...)
        (_ #false))))
```

# The notion of the computer

```
(define (draggable-rectangle left top width
                             height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) ...)
        (`(mouse-down ,x ,y)
         (set! dragged? #true))
        (`(mouse-up ,x ,y) ...)
        (`(mouse-move ,x ,y ,dx ,dy) ...)
        (`(embraces? ,x ,y) ...)
        (`(as-image) ...)
        (`(mouse-out) ...)
        (_ #false))))
```

# The notion of the computer

```
(define (draggable-rectangle left top width
                             height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) ...)
        (`(mouse-down ,x ,y) ...)
        (`(mouse-up ,x ,y)
         (set! dragged? #false))
        (`(mouse-move ,x ,y ,dx ,dy) ...)
        (`(embraces? ,x ,y) ...)
        (`(as-image) ...)
        (`(mouse-out) ...)
        (_ #false))))
```

# The notion of the computer

```
(define (draggable-rectangle left top width
                             height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) ...)
        (`(mouse-down ,x ,y) ...)
        (`(mouse-up ,x ,y) ...)
        (`(mouse-move ,x ,y ,dx ,dy)
           (when dragged?
             (set! left (+ left dx))
             (set! top (+ top dy))))
        (`(embraces? ,x ,y) ...)
        (`(as-image) ...)
        (`(mouse-out) ...)))
```

# The notion of the computer

```
(define (draggable-rectangle left top width
                             height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) ...)
        (`(mouse-down ,x ,y) ...)
        (`(mouse-up ,x ,y) ...)
        (`(mouse-move ,x ,y ,dx ,dy) ...)
        (`(embraces? ,x ,y)
           (and (is-left <= x <= (+ left width))
                (is-top <= y <= (+ top height)))))
        (`(as-image) ...)
        (`(mouse-out) ...)
        (_ #false))))
```

# The notion of the computer

```
(define (draggable-rectangle left top width
                             height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) ...)
        (`(mouse-down ,x ,y) ...)
        (`(mouse-up ,x ,y) ...)
        (`(mouse-move ,x ,y ,dx ,dy) ...)
        (`(embraces? ,x ,y) ...)
        (`(as-image) image)
        (`(mouse-out) ...)
        (_ #false))))
```

# The notion of the computer

```
(define (draggable-rectangle left top width
                             height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) ...)
        (`(mouse-down ,x ,y) ...)
        (`(mouse-up ,x ,y) ...)
        (`(mouse-move ,x ,y ,dx ,dy) ...)
        (`(embraces? ,x ,y) ...)
        (`(as-image) ...)
        (`(mouse-out)
           (set! dragged? #false))
        (_ #false))))
```

# The notion of the computer

```
(define (draggable-rectangle left top width
                             height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) ...)
        (`(mouse-down ,x ,y) ...)
        (`(mouse-up ,x ,y) ...)
        (`(mouse-move ,x ,y ,dx ,dy) ...)
        (`(embraces? ,x ,y) ...)
        (`(as-image) ...)
        (`(mouse-out) ...)
        (_ #false))))))
```

# The notion of the computer

```
(define stage
  (let* ((',(width ,height) (screen-size))
         (elements ...)
         (image (rectangle width height 0)))
    (hovered-element #false))
  (lambda message
    (match message
      ('(as-image) ...)
      ('(mouse-down ,x ,y) ...)
      ('(mouse-up ,x ,y) ...)
      ('(mouse-move ,x ,y ,dx ,dy) ...)
    ))))
```

# The notion of the computer

```
(define stage
  (let* ((`(,width ,height) (screen-size))
         (elements (map (lambda (_)
                           (draggable-rectangle
                             (random (- width 50))
                             (random (- height 50))
                             50 50 (random color)))
                           (range 10)))
         (image (rectangle width height 0)))
    (hovered-element #false)))
  (lambda message
    (match message
      (`(as-image) ...)
      (`(mouse-down ,x ,y) ...)
      (`(mouse-up ,x ,y) ...)
      (`(mouse-move ,x ,y ,dx ,dy) ...)
    ))))
```

# The notion of the computer

```
(define stage
  (let* ((',(width ,height) (screen-size))
         (elements ...)
         (image (rectangle width height 0)))
    (hovered-element #false))
  (lambda message
    (match message
      ('(as-image)
       (fill-image! image 0)
       (fold-right (lambda (e i)
                     (let ((',(x ,y) (e 'position)))
                       (draw-image! (e 'as-image) x y i)
                       i))
                  image elements)
       image)
      ('(mouse-down ,x ,y) ...)
      ('(mouse-up ,x ,y) ...)
      ('(mouse-move ,x ,y ,dx ,dy) ...)
    ))))
```

# The notion of the computer

```
(define stage
  (let* ((`(,width ,height) (screen-size))
         (elements ...)
         (image (rectangle width height 0)))
    (hovered-element #false))
  (lambda message
    (match message
      (`(as-image) ...)
      (`(mouse-down ,x ,y)
       (when hovered-element
         (hovered-element 'mouse-down x y)))
      (`(mouse-up ,x ,y) ...)
      (`(mouse-move ,x ,y ,dx ,dy) ...)
    ))))
```

# The notion of the computer

```
(define stage
  (let* ((',(width ,height) (screen-size))
         (elements ...)
         (image (rectangle width height 0)))
    (hovered-element #false))
  (lambda message
    (match message
      ('(as-image) ...)
      ('(mouse-down ,x ,y) ...)
      ('(mouse-up ,x ,y)
       (when hovered-element
         (hovered-element 'mouse-up x y)))
      ('(mouse-move ,x ,y ,dx ,dy) ...)
      ))))
```

# The notion of the computer

```
(define stage
  (let* ((',(width ,height) (screen-size))
         (elements ...)
         (image (rectangle width height 0)))
    (hovered-element #false))
  (lambda message
    (match message
      ('(as-image) ...)
      ('(mouse-down ,x ,y) ...)
      ('(mouse-up ,x ,y) ...)
      ('(mouse-move ,x ,y ,dx ,dy)
        (let ((hovered (find (lambda (_)
                               (_ 'embraces? x y))
                               elements)))
          (unless (eq? hovered hovered-element)
            (when hovered-element (hovered-element 'mouse-out))
            (when hovered (hovered 'mouse-over))
            (set! hovered-element hovered))
          (when hovered (hovered 'mouse-move x y dx dy))))))
  ))))
```

# Recursion on the notion of the computer (live demo)

2.rkt

<https://github.com/panicz/sracket>

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(mouse-over) ...)
        ('(mouse-out) ....))))
```

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(mouse-over) ...)
        ('(mouse-out) ....))))
```

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) `',(left ,top))
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(mouse-over) ...)
        ('(mouse-out) ....))))
```

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y)
         (if hovered-element
             (hovered-element 'mouse-down (- x left) (- y top))
             (when draggable?
               (set! dragged? #true))))
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(mouse-over) ...)
        ('(mouse-out) ....))))
```

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y)
         (when hovered-element
           (hovered-element 'mouse-up (- x left) (- y top)))
           (set! dragged? #false)))
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(mouse-over) ...)
        ('(mouse-out) ....))))
```

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy)
         (cond (dragged?
                  (set! left (+ left dx))
                  (set! top (+ top dy)))
                (else
                  (let ((hovered (find (_ 'embraces? (- x left) (- y top))
                                         elements)))
                    (unless (eq? hovered hovered-element)
                      (when hovered-element (hovered-element 'mouse-out))
                      (when hovered (hovered 'mouse-over))
                      (set! hovered-element hovered))
                    (when hovered
                      (hovered 'mouse-move (- x left) (- y top) dx dy))))))
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(mouse-over) ...)
        ('(mouse-out) ...))))
```

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y)
         (and (is left <= x <= (+ left width))
              (is top <= y <= (+ top height))))
        ('(as-image) ...)
        ('(mouse-over) ...)
        ('(mouse-out) ...)))))
```

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image)
         (fill-image! image 0)
         (fold-right (lambda (e i)
                      (let ((`(,x ,y) (e 'position)))
                        (draw-image! (e 'as-image) x y i)
                        i))
                     image elements))
        ('(mouse-over) ...)
        ('(mouse-out) ...))))
```

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(mouse-over) #false)
        ('(mouse-out) ....))))
```

# Recursion on the notion of the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color
            #:draggable? [draggable? #true] . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(mouse-over) ...)
        ('(mouse-out) (set! dragged? #false))))))
```

# Recursion on the notion of the computer

```
(define stage
  (let ((`(,w ,h) (screen-size)))
    (box #:left 0 #:top 0 #:width w #:height h
         #:background-color #x000000 #:draggable? #false
         (box #:left 10 #:top 10 #:width 200 #:height 200
              #:background-color #x77cc00
              (box #:left 10 #:top 10 #:width 50 #:height 50
                   #:background-color #x0077cc
                   (box #:left 5 #:top 5 #:width 20 #:height 20
                        #:background-color #xff0000))
              (box #:left 140 #:top 140 #:width 50 #:height 50
                   #:background-color #xcc0077
                   #:draggable? #false))
        (box #:left (- w 210) #:top (- h 210)
             #:width 200 #:height 200
             #:background-color #x7700cc
             (box #:left 140 #:top 140 #:width 50 #:height 50
                  #:background-color #xcc7700))))
```

# Beyond the computer (live demo)

3.rkt

<https://github.com/panicz/sracket>

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ...)
        ('(install-element! ,element) ...)
        (_ #false)))
      self)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ...)
        ('(install-element! ,element) ...)
        (_ #false)))
      self)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) `',(left ,top))
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ... )
        ('(install-element! ,element) ...)
        (_ #false)))
      self)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y)
         (let ((acquired (self 'acquire-hovered-element!)))
           (when acquired
             (set! dragged-element acquired))))
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ...))
        ('(install-element! ,element) ...)
        (_ #false)))
      self)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y)
         (when dragged-element
           (self 'install-element! dragged-element)
           (set! dragged-element #false))
         (self 'mouse-move x y 0 0)
         (when hovered-element
           (hovered-element 'mouse-up (- x left) (- y top))))
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ...))
        ('(install-element! ,element) ...)
        (_ #false)))
    self)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy)
         (when dragged-element
           (dragged-element 'move-by! dx dy)))
        (let ((hovered (find (_ 'embraces? (- x left) (- y top))
                             elements)))
            (unless (eq? hovered hovered-element)
              (when hovered-element
                (hovered-element 'mouse-out))
              (when hovered
                (hovered 'mouse-over))
              (set! hovered-element hovered)))
            (when hovered
              (hovered 'mouse-move (- x left) (- y top) dx dy))))
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ...)
        ('(install-element! ,element) ...)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y)
         (and (is-left? x left)
              (is-top? y top)))
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ...)
        ('(install-element! ,element) ...)
        (_ #false)))
      self)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image)
         (fill-image! image color)
         (fold-right (lambda (element image)
                       (let ((',(x ,y) (element 'position)))
                         (draw-image! (element 'as-image) x y image)
                         image))
                     image
                     `(@(if dragged-element
                           `(,dragged-element)
                           '())
                      ,@elements)))
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ...)
        ('(install-element! ,element) ...)
        (_ #false)))
      self)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy)
         (set! left (+ left dx))
         (set! top (+ top dy)))
        ('(acquire-hovered-element!) ...)
        ('(install-element! ,element) ...)
        (_ #false)))
      self)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
            #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!)
         (and hovered-element
              (or (and-let* ((acquired (hovered-element
                                         'acquire-hovered-element!))
                           ('(,x ,y) (hovered-element 'position)))
                  (acquired 'move-by! x y)
                  acquired)
                  (let ((acquired hovered-element))
                    (set! hovered-element #false)
                    (set! elements (filter (isnt _ eq? acquired)
                                           elements)))
                  acquired)))
        ('(install-element! ,element) ...)
        (_ #false))))
```

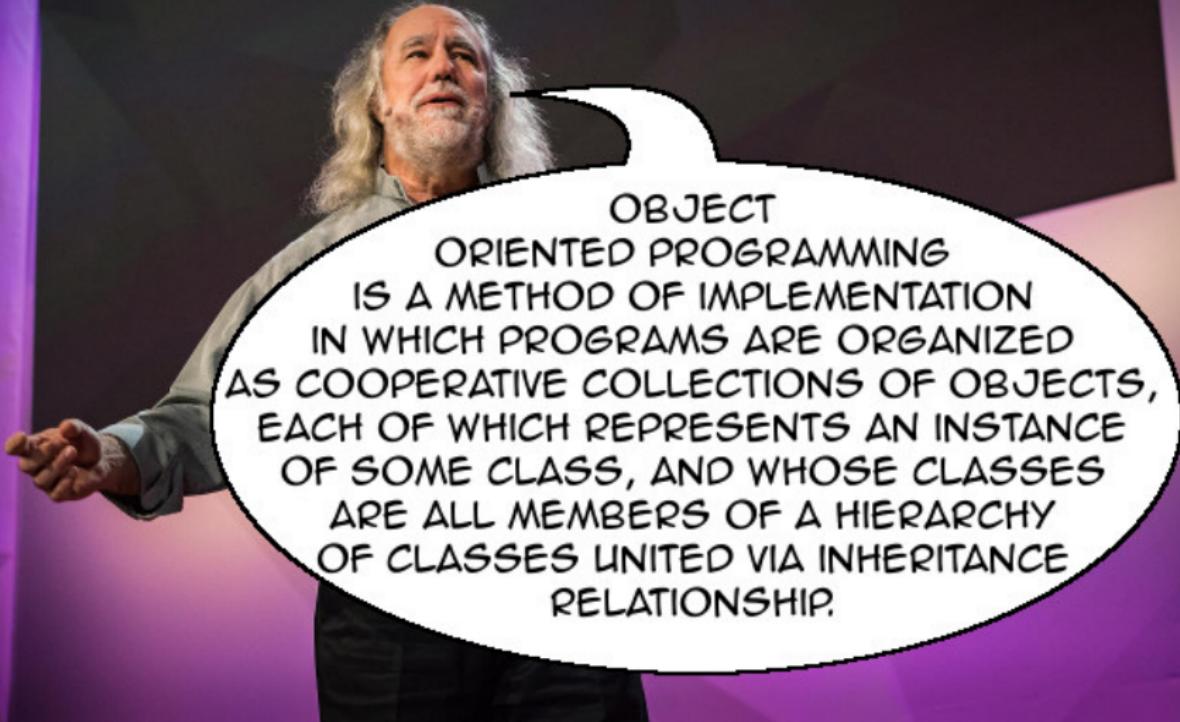
# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ...)
        ('(install-element! ,element)
         (if hovered-element
             (let ((',(x ,y) (hovered-element 'position)))
               (element 'move-by! (- x) (- y))
               (hovered-element 'install-element! element))
             (set! elements '(,element . ,elements))))
         (_ #false)))
      self)))
```

# Beyond the computer

```
(define (box #:left left #:top top #:width width #:height height
           #:background-color color . elements)
  (let ((dragged-element #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) ...)
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(embraces? ,x ,y) ...)
        ('(as-image) ...)
        ('(move-by! ,dx ,dy) ...)
        ('(acquire-hovered-element!) ...)
        ('(install-element! ,element) ...)
        (_ #false)))
      self)))
```





OBJECT  
ORIENTED PROGRAMMING  
IS A METHOD OF IMPLEMENTATION  
IN WHICH PROGRAMS ARE ORGANIZED  
AS COOPERATIVE COLLECTIONS OF OBJECTS,  
EACH OF WHICH REPRESENTS AN INSTANCE  
OF SOME CLASS, AND WHOSE CLASSES  
ARE ALL MEMBERS OF A HIERARCHY  
OF CLASSES UNITED VIA INHERITANCE  
RELATIONSHIP.

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

## “Pillars”:

- abstraction
- encapsulation
- inheritance
- polymorphism

## Principles (Booch et al.):

- abstraction
- encapsulation
- modularity
- hierarchy

# The four pillars of OOP

“Pillars”:

- **abstraction**
- **encapsulation**
- **inheritance**
- **polymorphism**

Principles (Booch et al.):

- **abstraction**
- **encapsulation**
- **modularity**
- **hierarchy**

We're talking about

# Abstraction

We're talking about

# Abstraction



John A De Goes  
@jdegoes

Following

An abstraction is a set of types, operations on these types, and laws that give meaning to the operations across all contexts.

In functional programming, we call abstractions type classes (or sometimes modules), and contrast them with design patterns.

12:24 PM - 17 Sep 2018

24 Retweets 78 Likes



5

24

78



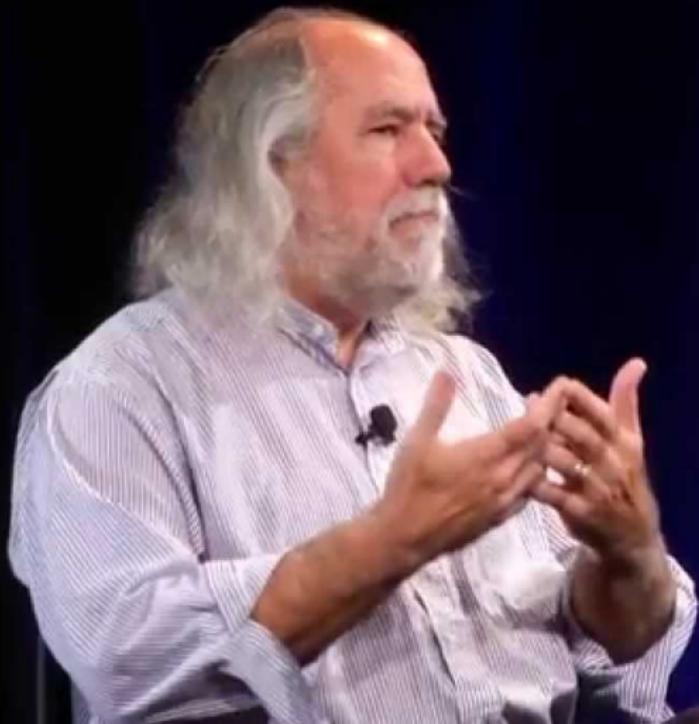
[HIRE ERIC](#)

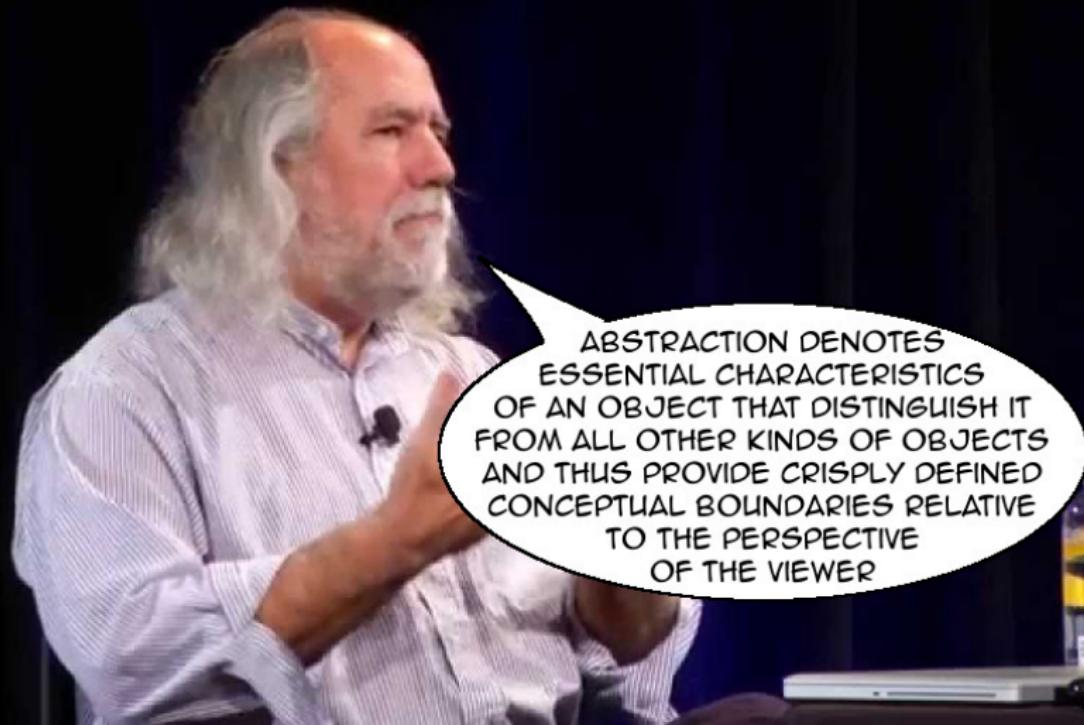
## WHAT IS AN ABSTRACTION?

ERIC NORMAND · UPDATED AUGUST 15, 2018

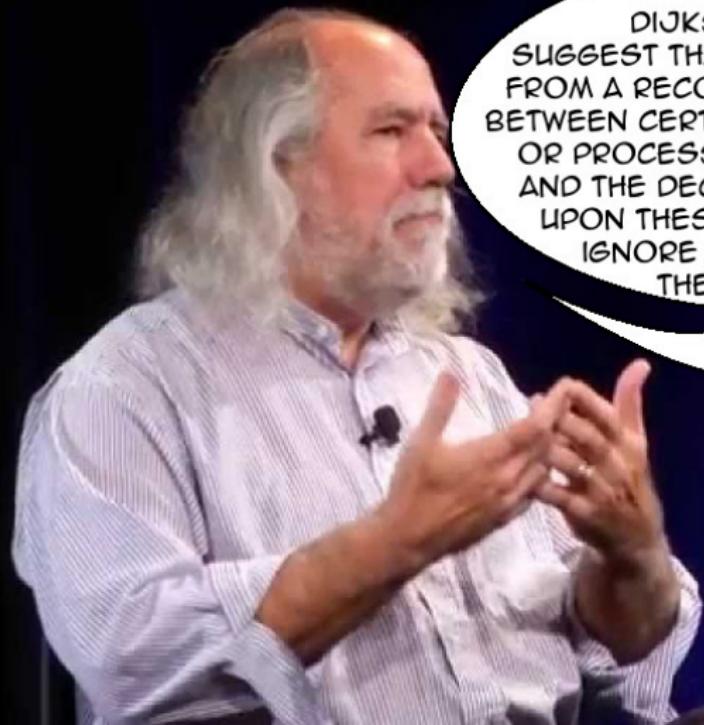
*Summary: We explore some of the background behind the meaning of the word abstraction and why we do it.*

For a term we use so much in our field, there are very few definitions of *abstraction*. And when I gave my talk called [Building Composable Abstractions](#), that was a persistent question: what did I mean by *abstraction*? I'm going to be talking about abstractions a lot, both on this





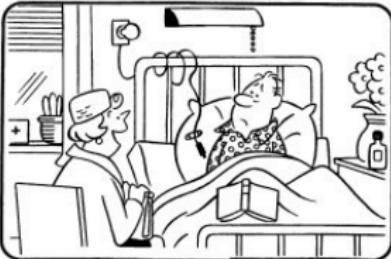
ABSTRACTION DENOTES  
ESSENTIAL CHARACTERISTICS  
OF AN OBJECT THAT DISTINGUISH IT  
FROM ALL OTHER KINDS OF OBJECTS  
AND THUS PROVIDE CRISPLY DEFINED  
CONCEPTUAL BOUNDARIES RELATIVE  
TO THE PERSPECTIVE  
OF THE VIEWER

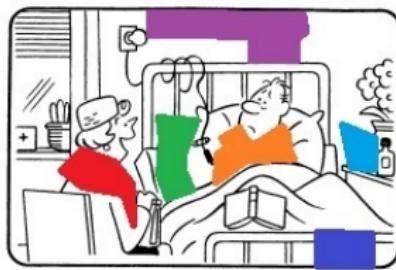


DAHL,  
DIJKSTRA AND HOARE  
SUGGEST THAT "ABSTRACTION ARISES  
FROM A RECOGNITION OF SIMILARITIES  
BETWEEN CERTAIN OBJECTS, SITUATIONS  
OR PROCESSES IN THE REAL WORLD,  
AND THE DECISION TO CONCENTRATE  
UPON THESE SIMILARITIES AND TO  
IGNORE FOR THE TIME BEING  
THE DIFFERENCES"

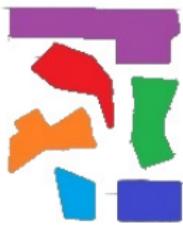


Find at least 6 differences between the panels – if you can!





these holes  
are



*in*



# Definition of abstraction I

**abstraction** the process of intentionally omitting (leaving out) certain detail in some mental representation

## Definition of abstraction II

abstraction a mental representation which contains *holes* that can be filled with some content (concretized)

We're talking about

# Inheritance

We're talking about

# Inheritance





INHERITANCE [...] MEANS  
THAT YOU CAN SPECIFY THAT  
A CLASS IS A SPECIAL KIND OF  
ANOTHER CLASS [...].



INHERITANCE [...] MEANS  
THAT YOU CAN SPECIFY THAT  
A CLASS IS A SPECIAL KIND OF  
ANOTHER CLASS [...].

THE UNDERLYING  
IDEA IS [...] THAT OF CLASSIFYING  
THINGS BY INCREASING DEGREES  
OF SPECIALIZATION



INHERITANCE [...] MEANS THAT YOU CAN SPECIFY THAT A CLASS IS A SPECIAL KIND OF ANOTHER CLASS [...].

THE UNDERLYING IDEA IS [...] THAT OF CLASSIFYING THINGS BY INCREASING DEGREES OF SPECIALIZATION

THE BOTANIST CARL VON LINNEE, BETTER KNOWN AS LINNEUS, POPULARIZED THIS WAY OF THINKING ABOUT PLANTS (AS WELL AS ANIMALS) IN THE EIGHTEENTH CENTURY. [...]

A middle-aged woman with short brown hair and glasses is speaking directly to the camera. She is wearing a red jacket over a dark shirt. Behind her is a large bookshelf filled with books and some papers on a desk. Four speech bubbles are overlaid on the image, pointing towards her mouth as if she is speaking.

INHERITANCE [...] MEANS  
THAT YOU CAN SPECIFY THAT  
A CLASS IS A SPECIAL KIND OF  
ANOTHER CLASS [...].

THE UNDERLYING  
IDEA IS [...] THAT OF CLASSIFYING  
THINGS BY INCREASING DEGREES  
OF SPECIALIZATION

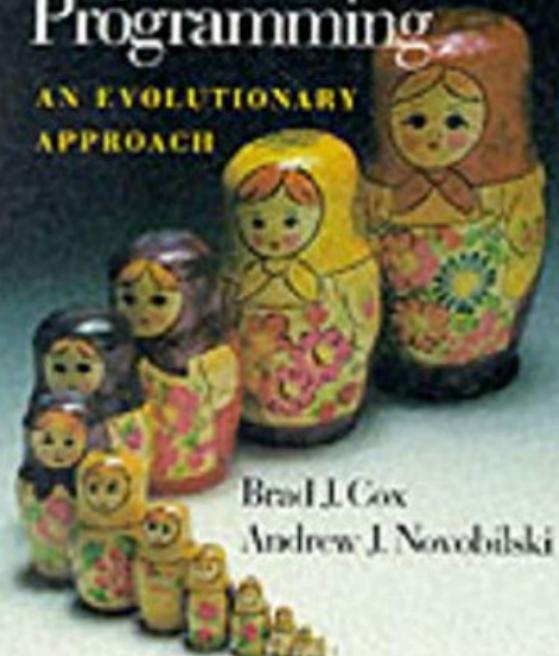
THE BOTANIST CARL VON LINNEE,  
BETTER KNOWN AS LINNEUS,  
POPULARIZED THIS WAY OF THINKING  
ABOUT PLANTS (AS WELL AS ANIMALS)  
IN THE EIGHTEENTH CENTURY. [...]

THE IDEA EXTENDED  
THE VARY WAY IN WHICH  
PEOPLE COULD THINK  
ABOUT THE WORLD

SECOND EDITION

# Object-Oriented Programming

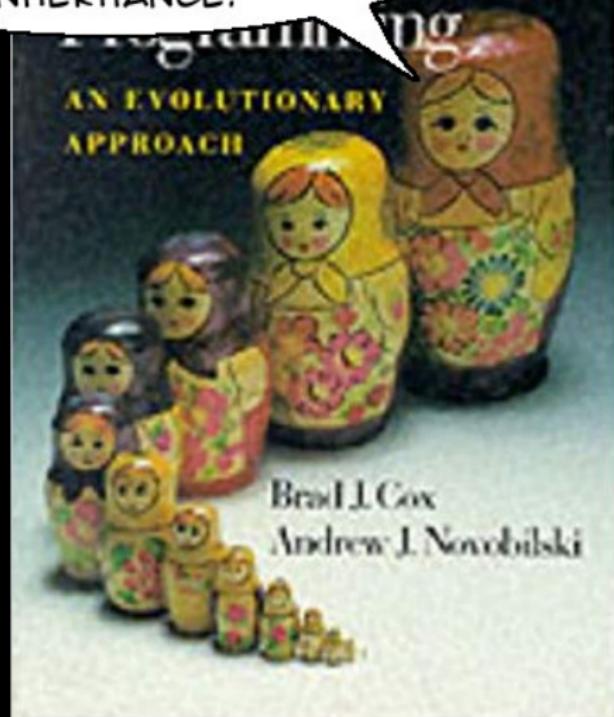
AN EVOLUTIONARY  
APPROACH



Brad J. Cox

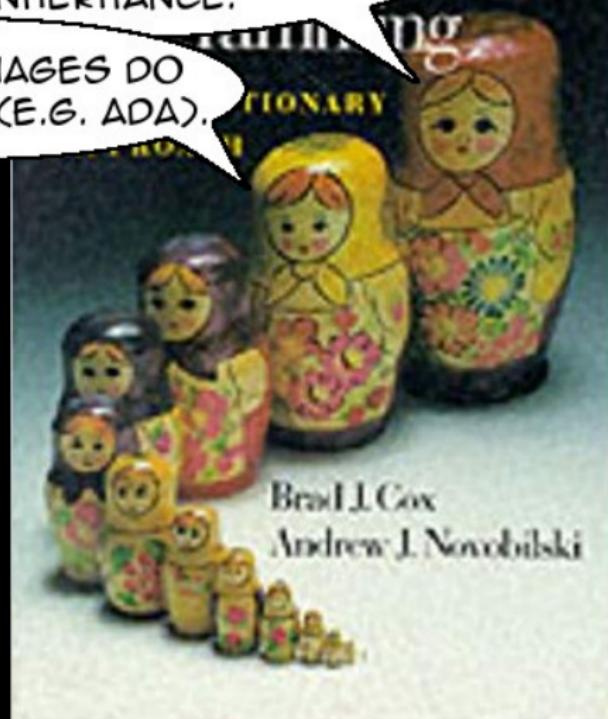
Andrew J. Novobilski

IT WOULD BE POSSIBLE TO STOP  
WITH THE ENCAPSULATION CONCEPT  
AND BUILD A LANGUAGE THAT PROVIDES  
CLASSES AND INSTANCES, BUT NOT  
INHERITANCE.

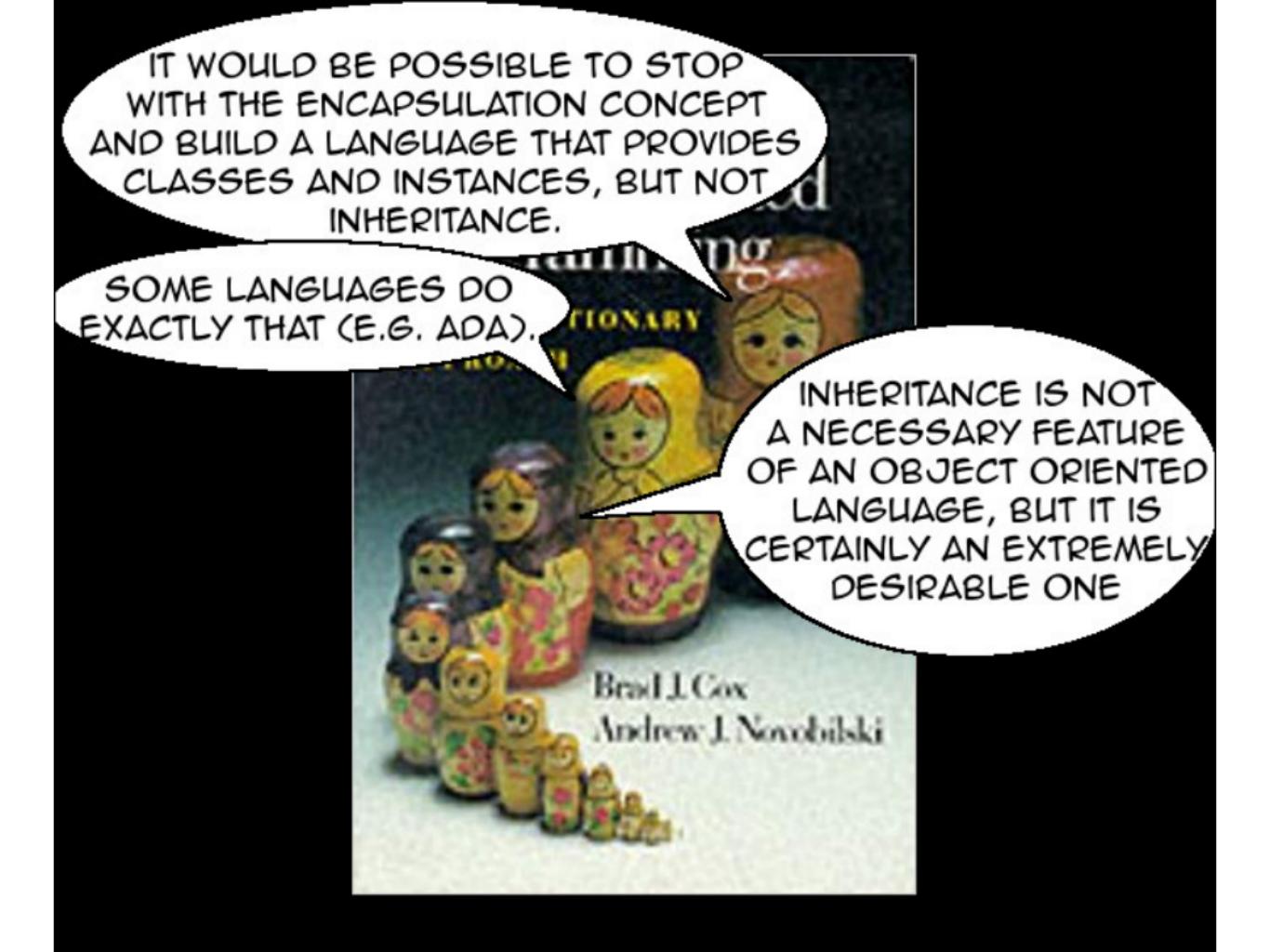


IT WOULD BE POSSIBLE TO STOP  
WITH THE ENCAPSULATION CONCEPT  
AND BUILD A LANGUAGE THAT PROVIDES  
CLASSES AND INSTANCES, BUT NOT  
INHERITANCE.

SOME LANGUAGES DO  
EXACTLY THAT (E.G. ADA).

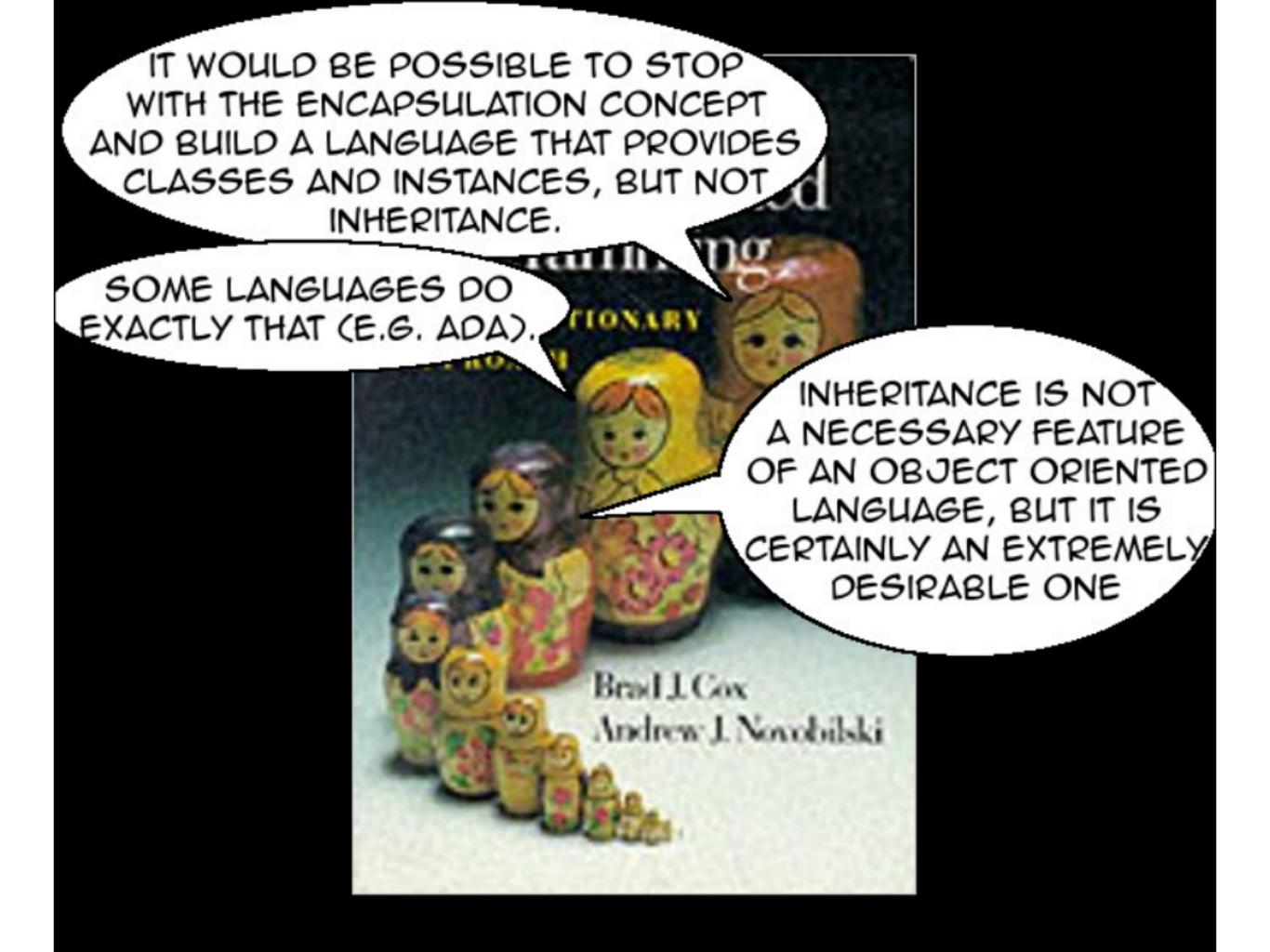


Brad J. Cox  
Andrew J. Novobilski



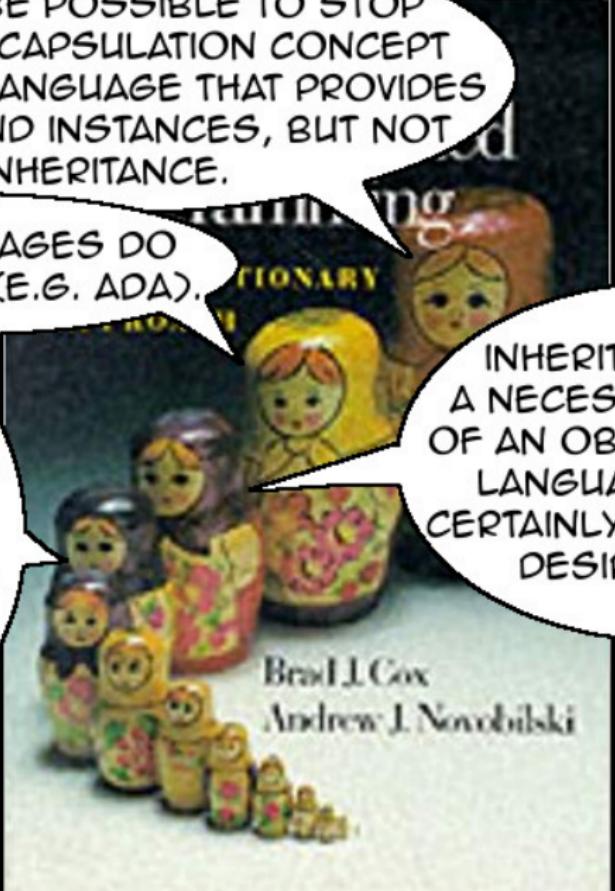
IT WOULD BE POSSIBLE TO STOP  
WITH THE ENCAPSULATION CONCEPT  
AND BUILD A LANGUAGE THAT PROVIDES  
CLASSES AND INSTANCES, BUT NOT  
INHERITANCE.

SOME LANGUAGES DO  
EXACTLY THAT (E.G. ADA).



INHERITANCE IS NOT  
A NECESSARY FEATURE  
OF AN OBJECT ORIENTED  
LANGUAGE, BUT IT IS  
CERTAINLY AN EXTREMELY  
DESIRABLE ONE

Brad L. Cox  
Andrew J. Novobilski



IT WOULD BE POSSIBLE TO STOP WITH THE ENCAPSULATION CONCEPT AND BUILD A LANGUAGE THAT PROVIDES CLASSES AND INSTANCES, BUT NOT INHERITANCE.

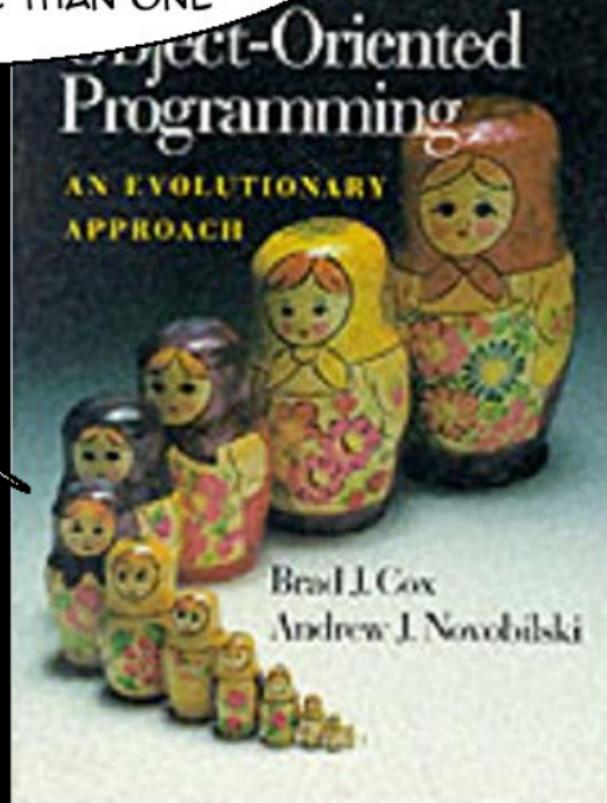
SOME LANGUAGES DO EXACTLY THAT (E.G. ADA).

ALSO, THE SIMPLE LINEAR SCHEME TO BE DESCRIBED THERE IS NOT THE ONLY WAY TO PROVIDE INHERITANCE

INHERITANCE IS NOT A NECESSARY FEATURE OF AN OBJECT ORIENTED LANGUAGE, BUT IT IS CERTAINLY AN EXTREMELY DESIRABLE ONE

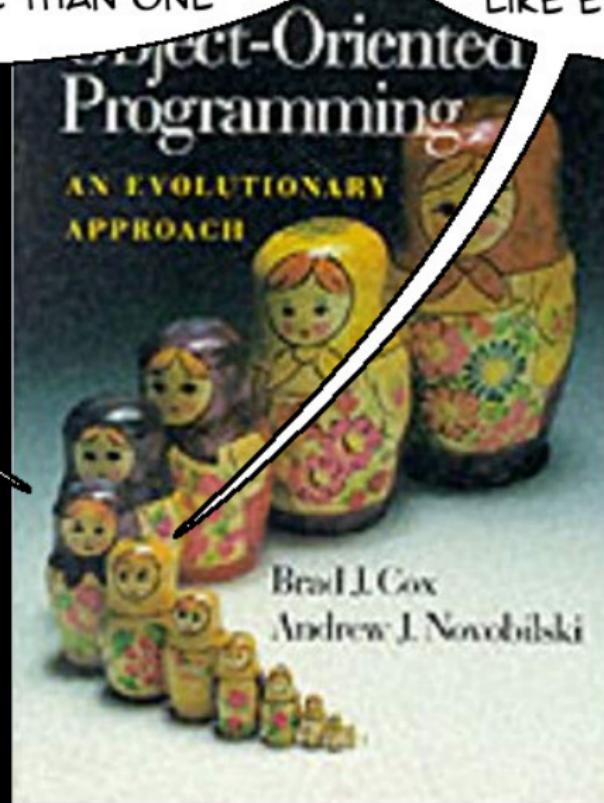
Brad L. Cox  
Andrew J. Novobilski

FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS



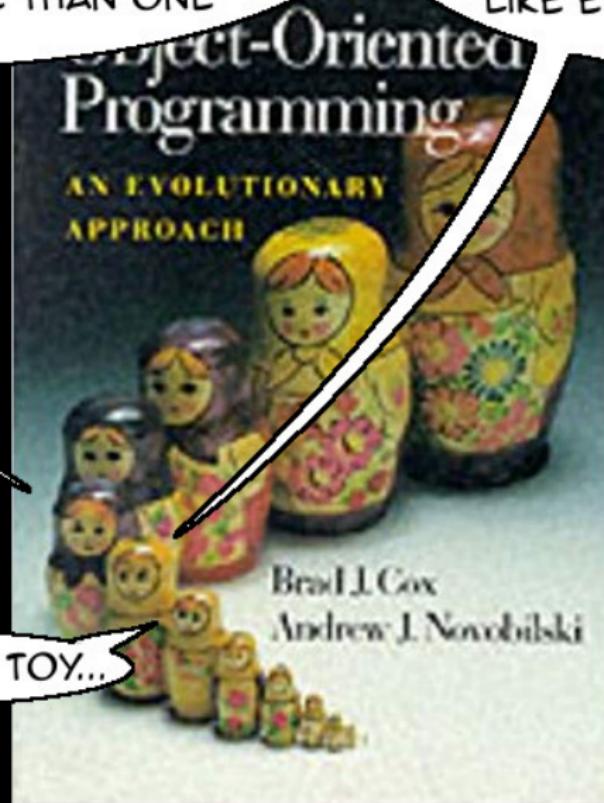
FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS

THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS



FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS

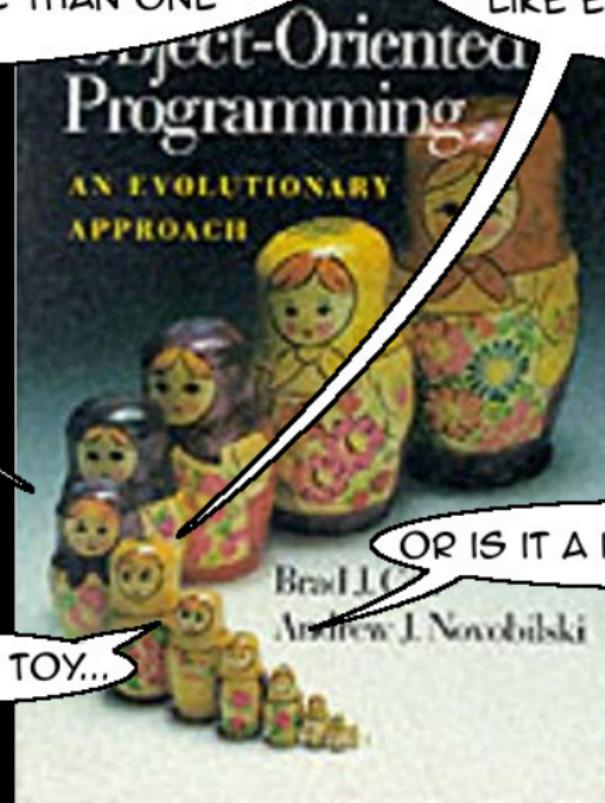
THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS



IS IT A KIND OF TOY...

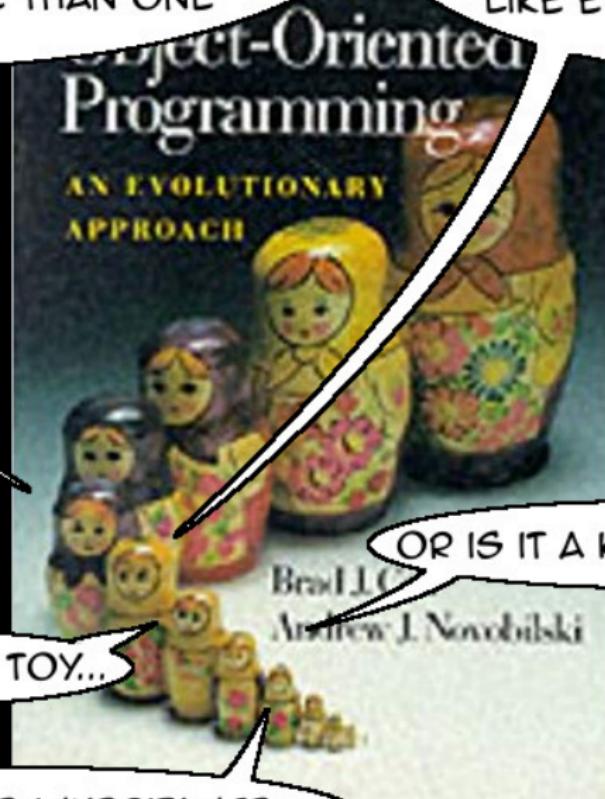
FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS

THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS



FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS

THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS



IS IT A KIND OF TOY...

WITH MULTIPLE INHERITANCE...

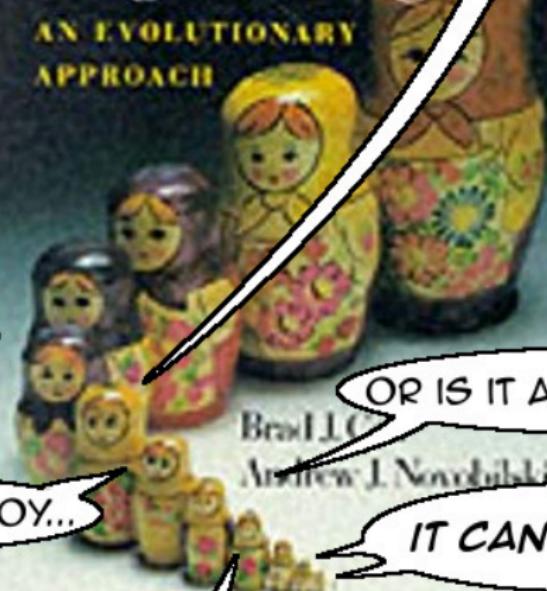
OR IS IT A KIND OF TRUCK?

FOR EXAMPLE, SOME LANGUAGES PROVIDE MULTIPLE INHERITANCE, WHICH MEANS THAT CLASSES CAN HAVE MORE THAN ONE SUPERCLASS

THESE LANGUAGES PROVIDE MORE GENERALITY, SOLVING, FOR EXAMPLE, PROBLEMS LIKE EXPRESSING WHAT A TOY TRUCK IS

# Object-Oriented Programming

AN EVOLUTIONARY APPROACH



Brad J.C.  
Andrew J. Novakowski

IS IT A KIND OF TOY...

OR IS IT A KIND OF TRUCK?

WITH MULTIPLE INHERITANCE...

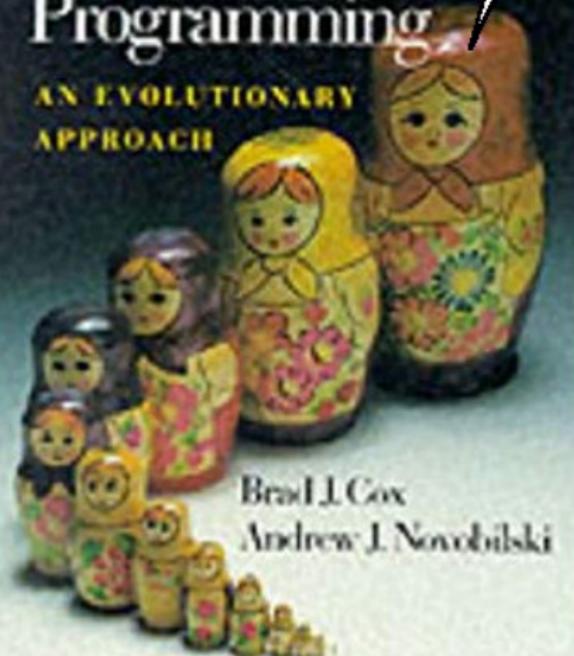
IT CAN BE BOTH!

INHERITANCE IS A TOOL FOR  
ORGANIZING, BUILDING  
AND USING REUSABLE  
CLASSES

SECOND EDITION

# Object-Oriented Programming

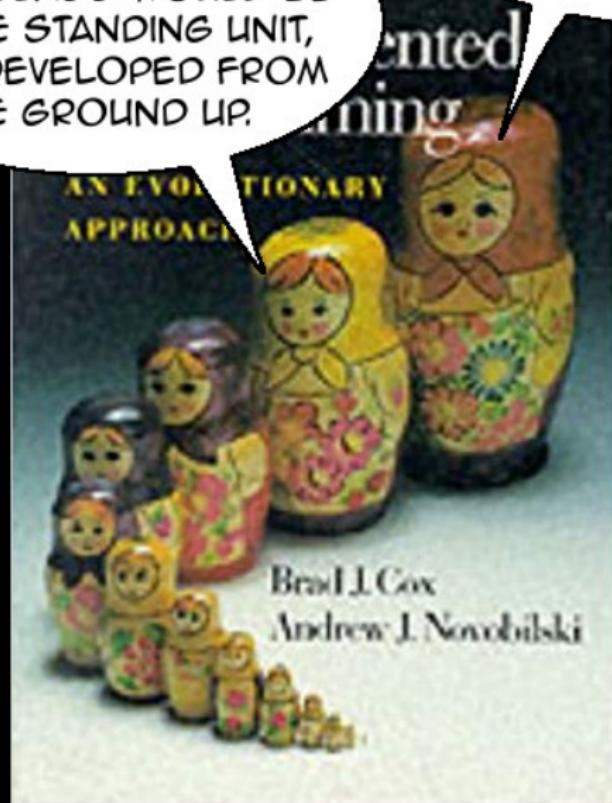
AN EVOLUTIONARY  
APPROACH



Brad J. Cox  
Andrew J. Novobilski

INHERITANCE IS A TOOL FOR  
ORGANIZING, BUILDING  
AND USING REUSABLE  
CLASSES

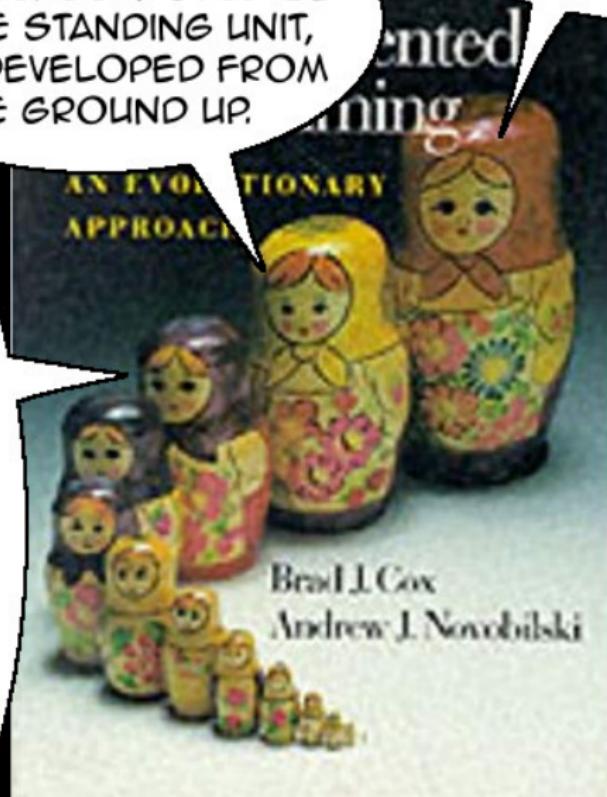
WITHOUT INHERITANCE,  
EVERY CLASS WOULD BE  
A FREE STANDING UNIT,  
EACH DEVELOPED FROM  
THE GROUND UP.



INHERITANCE IS A TOOL FOR  
ORGANIZING, BUILDING  
AND USING REUSABLE  
CLASSES

WITHOUT INHERITANCE,  
EVERY CLASS WOULD BE  
A FREE STANDING UNIT,  
EACH DEVELOPED FROM  
THE GROUND UP.

DIFFERENT  
CLASSES  
WOULD  
BEAR NO  
RELATIONSHIP  
WITH ONE  
ANOTHER,  
SINCE THE  
DEVELOPER  
OF EACH  
PROVIDES  
METHODS IN  
WHATEVER  
MANNER  
HE CHOOSES

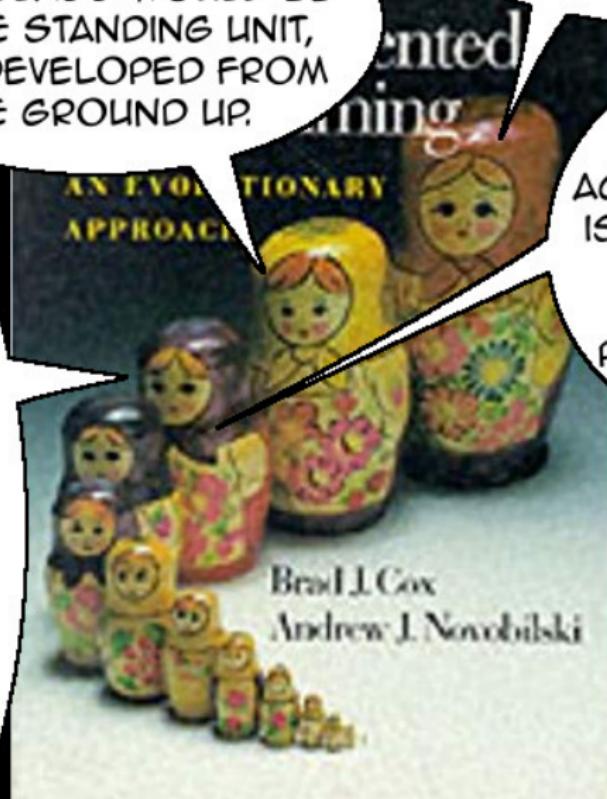


INHERITANCE IS A TOOL FOR  
ORGANIZING, BUILDING  
AND USING REUSABLE  
CLASSES

WITHOUT INHERITANCE,  
EVERY CLASS WOULD BE  
A FREE STANDING UNIT,  
EACH DEVELOPED FROM  
THE GROUND UP.

DIFFERENT  
CLASSES  
WOULD  
BEAR NO  
RELATIONSHIP  
WITH ONE  
ANOTHER,  
SINCE THE  
DEVELOPER  
OF EACH  
PROVIDES  
METHODS IN  
WHATEVER  
MANNER  
HE CHOOSES

ANY  
CONSISTENCY  
ACROSS CLASSES  
IS THE RESULT OF  
DISCIPLINE  
ON PART OF  
PROGRAMMERS.



INHERITANCE IS A TOOL FOR ORGANIZING, BUILDING AND USING REUSABLE CLASSES

WITHOUT INHERITANCE, EVERY CLASS WOULD BE A FREE STANDING UNIT, EACH DEVELOPED FROM THE GROUND UP.

DIFFERENT CLASSES WOULD BEAR NO RELATIONSHIP WITH ONE ANOTHER, SINCE THE DEVELOPER OF EACH PROVIDES METHODS IN WHATEVER MANNER HE CHOOSES.

ANY CONSISTENCY ACROSS CLASSES IS THE RESULT OF DISCIPLINE ON PART OF PROGRAMMERS.



INHERITANCE MAKES IT POSSIBLE TO DEFINE NEW SOFTWARE IN THE SAME WAY WE INTRODUCE A NEW CONCEPT TO A NEWCOMER...

INHERITANCE IS A TOOL FOR ORGANIZING, BUILDING AND USING REUSABLE CLASSES

WITHOUT INHERITANCE, EVERY CLASS WOULD BE A FREE STANDING UNIT, EACH DEVELOPED FROM THE GROUND UP.

DIFFERENT CLASSES WOULD BEAR NO RELATIONSHIP WITH ONE ANOTHER, SINCE THE DEVELOPER OF EACH PROVIDES METHODS IN WHATEVER MANNER HE CHOOSES.

ANY CONSISTENCY ACROSS CLASSES IS THE RESULT OF DISCIPLINE ON PART OF PROGRAMMERS.



INHERITANCE MAKES IT POSSIBLE TO DEFINE NEW SOFTWARE IN THE SAME WAY WE INTRODUCE A NEW CONCEPT TO A NEWCOMER...

BY COMPARING IT WITH SOMETHING THAT IS ALREADY FAMILIAR

class A

class A



instance a

class

A

construction



instance

a

class

A

B

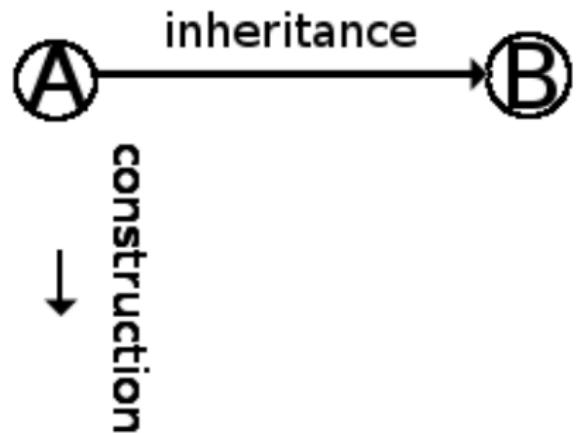
instance

a

construction



class



instance

a

class

instance

A

a

↑

B

construction



class

A

mix-in



B

instance

a

construction



class

A

mix-in



B

construction

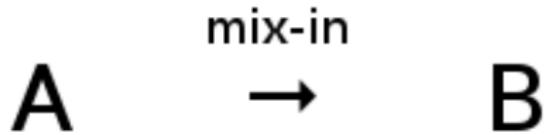


instance

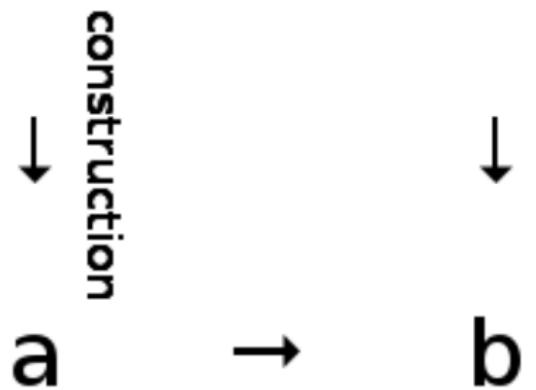
a

b

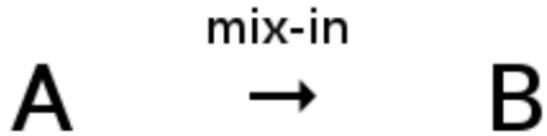
class



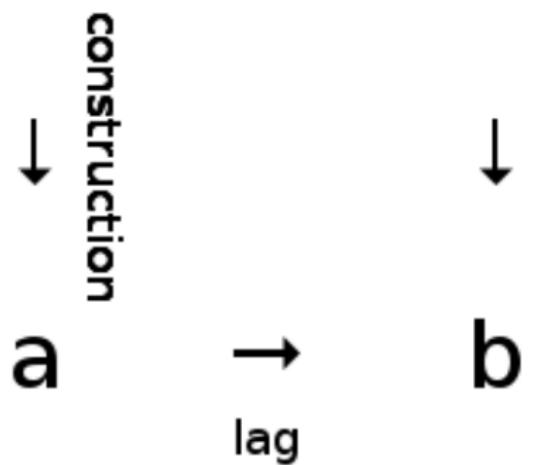
instance



class



instance



# Lagging (live demo)

4.rkt

<https://github.com/panicz/sracket>

# Lagging

```
(define (bit #:left left #:top top #:width width #:height height
           #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) '(,left ,top))
        ('(size) '(,width ,height))
        ('(extents) '(,left ,top ,(+ left width) ,(+ top height)))
        ('(resize-by! ,dx ,dy)
          (set! width (max 0 (+ width dx)))
          (set! height (max 0 (+ height dy)))
          (set! image (rectangle width height color)))
        ('(move-by! ,dx ,dy)
          (set! left (+ left dx))
          (set! top (+ top dy)))
        ('(embraces? ,x ,y)
          (and (is left <= x <= (+ left width))
               (is top <= y <= (+ top height))))
        ('(as-image)
          (fill-image! image color)
          image)
        (_ #false)))
    self)))
```

# Lagging

```
(define (bit #:left left #:top top #:width width #:height height
           #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) '(,left ,top))
        ('(size) '(,width ,height))
        ('(extents) '(,left ,top ,(+ left width) ,(+ top height)))
        ('(resize-by! ,dx ,dy)
          (set! width (max 0 (+ width dx)))
          (set! height (max 0 (+ height dy)))
          (set! image (rectangle width height color)))
        ('(move-by! ,dx ,dy)
          (set! left (+ left dx))
          (set! top (+ top dy)))
        ('(embraces? ,x ,y)
          (and (is left <= x <= (+ left width))
               (is top <= y <= (+ top height))))
        ('(as-image)
          (fill-image! image color)
          image)
        (_ #false)))
    self)))
```

# Lagging

```
(define (bit #:left left #:top top #:width width #:height height
           #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) '(,left ,top))
        ('(size) '(,width ,height))
        ('(extents) '(,left ,top ,(+ left width) ,(+ top height)))
        ('(resize-by! ,dx ,dy)
          (set! width (max 0 (+ width dx)))
          (set! height (max 0 (+ height dy)))
          (set! image (rectangle width height color)))
        ('(move-by! ,dx ,dy)
          (set! left (+ left dx))
          (set! top (+ top dy)))
        ('(embraces? ,x ,y)
          (and (is left <= x <= (+ left width))
               (is top <= y <= (+ top height))))
        ('(as-image)
          (fill-image! image color)
          image)
        (_ #false)))
    self)))
```

# Lagging

```
(define (bit #:left left #:top top #:width width #:height height
           #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) '(,left ,top))
        ('(size) '(,width ,height))
        ('(extents) '(,left ,top ,(+ left width) ,(+ top height)))
        ('(resize-by! ,dx ,dy)
          (set! width (max 0 (+ width dx)))
          (set! height (max 0 (+ height dy)))
          (set! image (rectangle width height color)))
        ('(move-by! ,dx ,dy)
          (set! left (+ left dx))
          (set! top (+ top dy)))
        ('(embraces? ,x ,y)
          (and (is left <= x <= (+ left width))
               (is top <= y <= (+ top height))))
        ('(as-image)
          (fill-image! image color)
          image)
        (_ #false)))
    self)))
```

# Lagging

```
(define (bit #:left left #:top top #:width width #:height height
           #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) '(,left ,top))
        ('(size) '(,width ,height))
        ('(extents) '(,left ,top ,(+ left width) ,(+ top height)))
        ('(resize-by! ,dx ,dy)
          (set! width (max 0 (+ width dx)))
          (set! height (max 0 (+ height dy)))
          (set! image (rectangle width height color)))
        ('(move-by! ,dx ,dy)
          (set! left (+ left dx))
          (set! top (+ top dy)))
        ('(embraces? ,x ,y)
          (and (is left <= x <= (+ left width))
               (is top <= y <= (+ top height))))
        ('(as-image)
          (fill-image! image color)
          image)
        (_ #false)))
    self))
```

# Lagging

```
(define (bit #:left left #:top top #:width width #:height height
           #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) '(,left ,top))
        ('(size) '(,width ,height))
        ('(extents) '(,left ,top ,(+ left width) ,(+ top height)))
        ('(resize-by! ,dx ,dy)
          (set! width (max 0 (+ width dx)))
          (set! height (max 0 (+ height dy)))
          (set! image (rectangle width height color)))
        ('(move-by! ,dx ,dy)
          (set! left (+ left dx))
          (set! top (+ top dy)))
        ('(embraces? ,x ,y)
          (and (is left <= x <= (+ left width))
               (is top <= y <= (+ top height))))
        ('(as-image)
          (fill-image! image color)
          image)
        (_ #false)))
    self)))
```

# Lagging

```
(define (bit #:left left #:top top #:width width #:height height
            #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) '(,left ,top))
        ('(size) '(,width ,height))
        ('(extents) '(,left ,top ,(+ left width) ,(+ top height)))
        ('(resize-by! ,dx ,dy)
          (set! width (max 0 (+ width dx)))
          (set! height (max 0 (+ height dy)))
          (set! image (rectangle width height color)))
        ('(move-by! ,dx ,dy)
          (set! left (+ left dx))
          (set! top (+ top dy)))
        ('(embraces? ,x ,y)
          (and (is left <= x <= (+ left width))
               (is top <= y <= (+ top height))))
        ('(as-image)
          (fill-image! image color)
          image)
        (_ #false)))
    self)))
```

# Lagging

```
(define (bit #:left left #:top top #:width width #:height height
           #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) '(,left ,top))
        ('(size) '(,width ,height))
        ('(extents) '(,left ,top ,(+ left width) ,(+ top height)))
        ('(resize-by! ,dx ,dy)
          (set! width (max 0 (+ width dx)))
          (set! height (max 0 (+ height dy)))
          (set! image (rectangle width height color)))
        ('(move-by! ,dx ,dy)
          (set! left (+ left dx))
          (set! top (+ top dy)))
        ('(embraces? ,x ,y)
          (and (is left <= x <= (+ left width))
               (is top <= y <= (+ top height))))
        ('(as-image)
          (fill-image! image color)
          image)
        (_ #false)))
    self)))
```

# Lagging

```
(define (bit #:left left #:top top #:width width #:height height
           #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        ('(position) '(,left ,top))
        ('(size) '(,width ,height))
        ('(extents) '(,left ,top ,(+ left width) ,(+ top height)))
        ('(resize-by! ,dx ,dy)
          (set! width (max 0 (+ width dx)))
          (set! height (max 0 (+ height dy)))
          (set! image (rectangle width height color)))
        ('(move-by! ,dx ,dy)
          (set! left (+ left dx))
          (set! top (+ top dy)))
        ('(embraces? ,x ,y)
          (and (is left <= x <= (+ left width))
               (is top <= y <= (+ top height))))
        ('(as-image)
          (fill-image! image color)
          image)
        (_ #false)))
    self)))
```

# Lagging

```
(define (containing origin elements)
  (let ((hovered-element #false))
    (lambda message
      (let ((`(,left ,top) (origin 'position)))
        (match message
          (`(mouse-down ,x ,y) ...)
          (`(mouse-up ,x ,y) ...)
          (`(mouse-move ,x ,y ,dx ,dy) ...)
          (`(add! ,element) ...)
          (`(remove! ,element) ...)
          (`(as-image) ...)
          (_ (apply origin message)))
        )))))
```

# Lagging

```
(define (containing origin elements)
  (let ((hovered-element #false))
    (lambda message
      (let ((`(,left ,top) (origin 'position)))
        (match message
          (`(mouse-down ,x ,y) ...)
          (`(mouse-up ,x ,y) ...)
          (`(mouse-move ,x ,y ,dx ,dy) ...)
          (`(add! ,element) ...)
          (`(remove! ,element) ...)
          (`(as-image) ...)
          (_ (apply origin message)))
        )))))
```

# Lagging

```
(define (transferable origin)
  (let ((hovered-element #false)
        (dragged-element #false))
    (define (self . message)
      (match message
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(acquire-element!) ...)
        ('(install-element! ,element) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(as-image) ...)
        (_ (apply origin message)))))

  self)))
```

# Lagging

```
(define (transferable origin)
  (let ((hovered-element #false)
        (dragged-element #false))
    (define (self . message)
      (match message
        ('(mouse-down ,x ,y) ...)
        ('(mouse-up ,x ,y) ...)
        ('(acquire-element!) ...)
        ('(install-element! ,element) ...)
        ('(mouse-move ,x ,y ,dx ,dy) ...)
        ('(as-image) ...)
        (_ (apply origin message)))))
  self))
```

# Lagging

```
(define (box #:left left #:top top
            #:width width #:height height
            #:background-color color
            . elements)

(transferable
 (containing
 (bit #:left left #:top top
       #:width width #:height height
       #:background-color color)
 elements)))
```

# Lagging

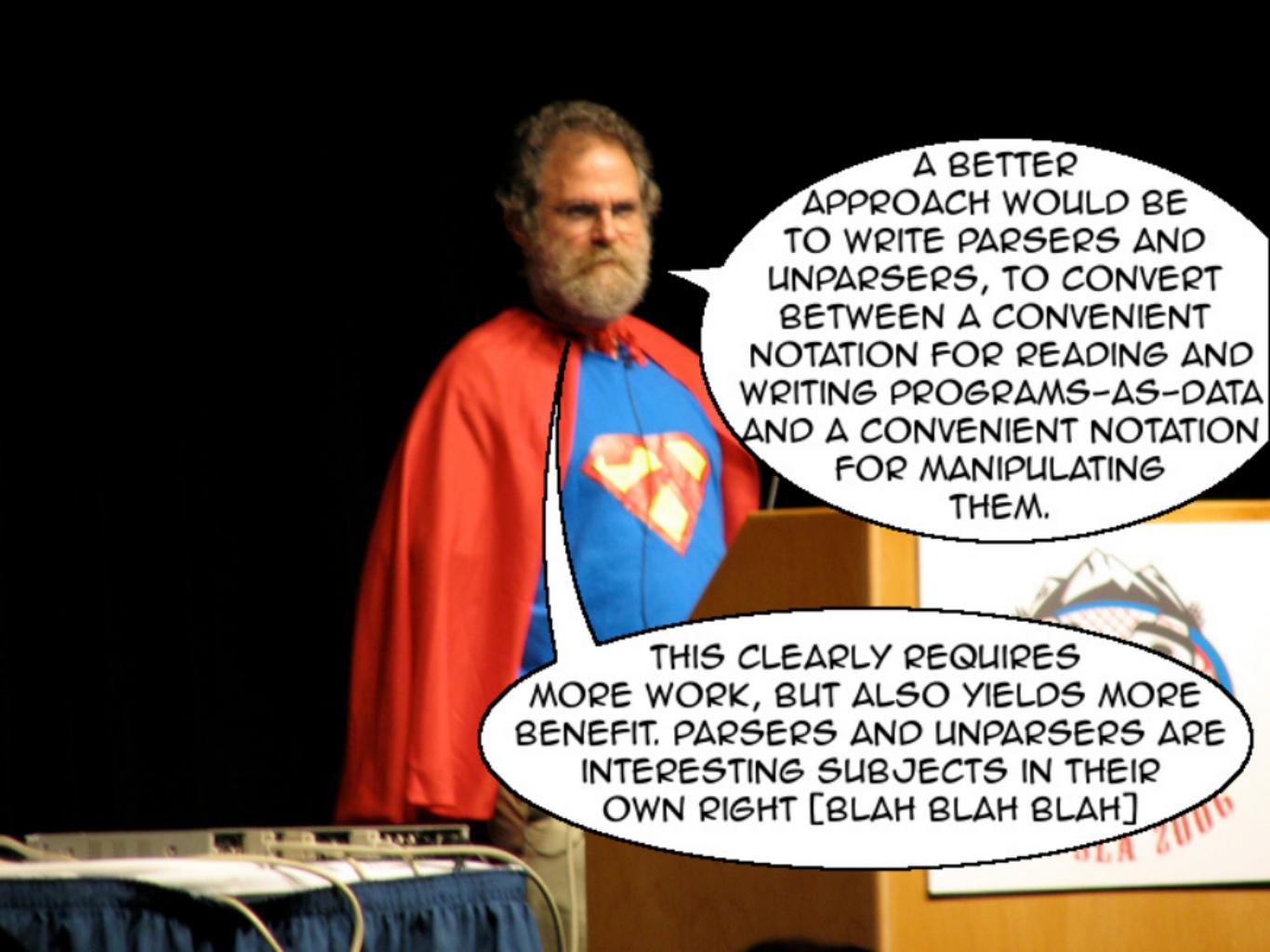
```
(define (box #:left left #:top top
            #:width width #:height height
            #:background-color color
            . elements)

(resizable
  (transferable
    (containing
      (bit #:left left #:top top
           #:width width #:height height
           #:background-color color)
      elements))))
```



A BETTER APPROACH WOULD BE TO WRITE PARSERS AND UNPARSERS, TO CONVERT BETWEEN A CONVENIENT NOTATION FOR READING AND WRITING PROGRAMS-AS-DATA AND A CONVENIENT NOTATION FOR MANIPULATING THEM.





A BETTER APPROACH WOULD BE TO WRITE PARSERS AND UNPARSERS, TO CONVERT BETWEEN A CONVENIENT NOTATION FOR READING AND WRITING PROGRAMS-AS-DATA AND A CONVENIENT NOTATION FOR MANIPULATING THEM.

THIS CLEARLY REQUIRES MORE WORK, BUT ALSO YIELDS MORE BENEFIT. PARSERS AND UNPARSERS ARE INTERESTING SUBJECTS IN THEIR OWN RIGHT [BLAH BLAH BLAH]

< Conversation with Jon Harrop

... X



SICP is a seriously out of date book [...]



SICP is a seriously out of date book [...]

I'm not sure whether parsing would actually bring any value to the topics that were covered by SICP. How would it be helpful to their logic circuit simulator, or to the picture language? How would it help with symbolic differentiation? How would they justify introduction of syntax? "Hey, listen, we have this straightforward way of representing expressions, and although we could get used to it and go on with our presentation of ideas, it looks a bit alien to us, so we will devote the next chapter to parsing, so that we could be writing it so that it would resemble things that we are more accustomed to"?





SICP is a seriously out of date book [...]

I'm not sure whether parsing would actually bring any value to the topics that were covered by SICP. How would it be helpful to their logic circuit simulator, or to the picture language? How would it help with symbolic differentiation? How would they justify introduction of syntax? "Hey, listen, we have this straightforward way of representing expressions, and although we could get used to it and go on with our presentation of ideas, it looks a bit alien to us, so we will devote the next chapter to parsing, so that we could be writing it so that it would resemble things that we are more accustomed to"?



What a bizarre question. Syntax is defacto standard. Syntax is everywhere, in every major language. Rather than asking for justification for the inclusion of syntax you should be trying to justify its exclusion.



# LIMB (live demo)

5.rkt

<https://github.com/panicz/sracket>