CS4613 Lecture 6: Objects

David Bremner

September 20, 2025

Desugared objects

p. 87

Basic Object

```
(define o
  (lambda (m)
    (case m
      [(add1) (lambda (x) (+ x 1))]
      [(sub1) (lambda (x) (- x 1))])))
```

obj1

use

```
(test ((o 'sub1) 6) 5)
```

- why return a function (hint, think about method args)

Small simplifications

p. 87

```
(define (msg obj selector . args)
  (apply (obj selector) args))
```


```
(define-syntax snd
  (syntax-rules ()
    [(_ obj selector args ...)
     ((obj (quote selector)) args ...)]))
```

obj2

```
(test (msg o 'sub1 6) 5)
(test (snd o sub1 6) 5)
```

Constructors

Turn object declaration into function

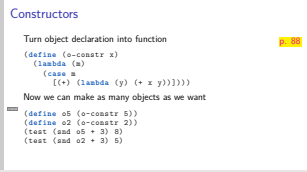
p. 88

```
(define (o-constr x)
  (lambda (m)
    (case m
      [(+) (lambda (y) (+ x y))])))
```

Now we can make as many objects as we want

const

```
(define o5 (o-constr 5))
(define o2 (o-constr 2))
(test (snd o5 + 3) 8)
(test (snd o2 + 3) 5)
```



1. This *looks* like infix syntax, but it's really just using + as a symbol

Constructors as Classes

p. 89

```
 class Thing {
     constructor(x) {
         this.x=x;
     }
     add(y) { return this.x + y; }
     sub(y) { return this.x - y; }
}

(define (class x)
  (lambda (m)
    (case m
      [(+) (lambda (y) (+ x y))]
      [(-) (lambda (y) (- x y))])))
```

Constructors as Classes

p. 89

Where is x stored?

```
(define (mk-O x)
  (lambda (m)
    (case m
      [(+) (lambda (y) (+ x y))]
      [(-) (lambda (y) (- x y))])))
```

const

```
(define O2 (mk-O 2))
(define O5 (mk-O 5))
(test (snd O2 - 3) -1)
(test (snd O5 + 7) 12)
```

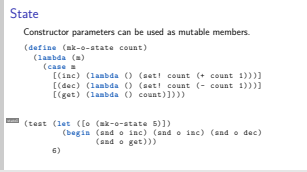
State

Constructor parameters can be used as mutable members.

```
(define (mk-o-state count)
  (lambda (m)
    (case m
      [(inc) (lambda () (set! count (+ count 1)))]
      [(dec) (lambda () (set! count (- count 1)))]
      [(get) (lambda () count)])))
```

state1

```
(test (let ([o (mk-o-state 5)])
        (begin (snd o inc) (snd o inc) (snd o dec)
               (snd o get)))
      6)
```


State

Constructor parameters can be used as mutable members.

```
(define (mk-o-state count)
  (lambda (m)
    (case m
      [(inc) (lambda () (set! count (+ count 1)))]
      [(dec) (lambda () (set! count (- count 1)))]
      [(get) (lambda () count)])))


(test (let ([o (mk-o-state 5)])
        (begin (snd o inc) (snd o inc) (snd o dec)
               (snd o get)))
      6)
```

1. Mutating the arguments to a function is also possible in e.g. C, even if considered in somewhat poor taste.
2. Note that everything is pass by value here, so we are mutating a local copy
3. There's a lot to think about here; the fact that the returned closures remember their defining environment after that function returns is crucial

More state

Objects are independent

p. 90

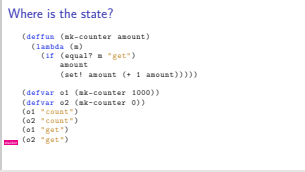
state2

```
(test (let ([o1 (mk-o-state 3)]
            [o2 (mk-o-state 3)])
        (begin (snd o1 inc) (snd o1 inc)
               (list (snd o1 get) (snd o2 get))))
      '(5 3))
```

Where is the state?

```
(deffun (mk-counter amount)
  (lambda (m)
    (if (equal? m "get")
        amount
        (set! amount (+ 1 amount)))))

(defvar o1 (mk-counter 1000))
(defvar o2 (mk-counter 0))
(o1 "count")
(o2 "count")
(o1 "get")
(o2 "get")
```



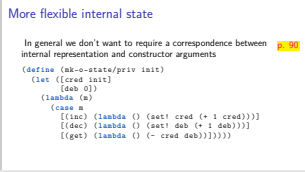
1. This simplifies the object pattern to directly run the code in question rather than returning a function that does the running
2. The answer to the title question is that the state is in environments, and those environments (or the variables in them, are actually mutated)

More flexible internal state

In general we don't want to require a correspondence between internal representation and constructor arguments

p. 90

```
(define (mk-o-state/priv init)
  (let ([cred init]
        [deb 0])
    (lambda (m)
      (case m
        [(inc) (lambda () (set! cred (+ 1 cred)))]
        [(dec) (lambda () (set! deb (+ 1 deb)))]
        [(get) (lambda () (- cred deb))]))))
```



1. This abstraction is part of the whole Object-Oriented concept: interface vs. implementation
2. This example is different from the one of same name in the book, to emphasize abstraction. Notice they have the same interface.

Interface is preserved

state3

```
(test (let ([o (mk-o-state/priv 5)])
        (begin (snd o inc) (snd o inc) (snd o dec)
               (snd o get)))
      6)
```

The “Class” Pattern revisited

p. 91

```
(define (class-w/-private constructor-params)
  (let ([private-vars ...] ...)
    ... the object pattern ...))
```

Which is equivalent to

```
(define class-w/-private
  (lambda (constructor-params)
    (let ([private-vars ...] ...)
      ... the object pattern ...)))
```

- set up for “let-over-lambda”

Static or “class members”

p. 91

```
(define mk-o-static
  (let ([counter 0]) ;; outside constructor
    (lambda (amnt)
      (begin
        (set! counter (+ 1 counter))
        (lambda (m)
          (case m
            [(inc) (lambda (n) (set! amnt (+ amnt n)))]
            [(dec) (lambda (n) (set! amnt (- amnt n)))]
            [(get) (lambda () amnt)]
            [(count) (lambda () counter)])))))))
```

Testing class with static members

Static members are common

p. 92

state4
```
(test (let ([o (mk-o-static 1000)])
        (snd o count))
      1)
(test (let ([o (mk-o-static 0)])
        (snd o count))
      2)
```

Private members are not

state4
```
(test (let ([o1 (mk-o-static 3)]
            [o2 (mk-o-static 3)])
        (begin (snd o1 inc 2) (snd o1 inc 2)
               (list (snd o1 get) (snd o2 get))))
      '(7 3))
```

Revised class pattern

p. 92

```
(define class-w/-private&static
  (let ([static-vars ...] ...)
    (lambda (constructor-params)
      (let ([private-vars ...] ...)
        ... the object pattern ...))))
```

Static on Stacker

p. 92

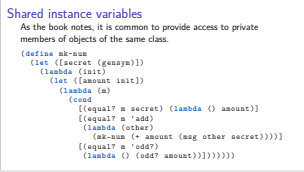
```
(defvar mk-o-static
  (let ([counter 0])
    (lambda (amount)
      (begin
        (set! counter (+ 1 counter))
        (lambda (m)
          (if (equal? m "get") (lambda () amount)
              (if (equal? m "count") counter
                  (error "no such member")))))))))
(defvar o1 (mk-o-static 1000))
(defvar o2 (mk-o-static 0))
(o1 "count") (o2 "count")
```

stacker

Shared instance variables

As the book notes, it is common to provide access to private members of objects of the same class.

```
(define mk-num
  (let ([secret (gensym)])
    (lambda (init)
      (let ([amount init])
        (lambda (m)
          (cond
            [(equal? m secret) (lambda () amount)]
            [(equal? m 'add)
             (lambda (other)
               (mk-num (+ amount (msg other secret))))]
            [(equal? m 'odd?)
             (lambda () (odd? amount))]))))))
```



1. This combines the just introduced static class pattern, with the new function `gensym`. The latter just makes a symbol that guarantees not to repeat, and is practically impossible to guess.
2. Some object oriented languages provide a more compile time solution for enforcing private access; we'll see some related ideas when we talk about types

Testing private sharing

state5

```
(define o2 (mk-num 2))
(define o3 (mk-num 3))
(define o4 (msg o2 'add o2))
(define o5 (msg o2 'add o3))
(test (msg o2 'odd?) #f)
(test (msg o3 'odd?) #t)
(test (msg o4 'odd?) #f)
(test (msg o5 'odd?) #t)
```