CS61A Lecture 15

2011-07-14

Colleen Lewis

REVIEW: define the `animal` class

```
STk> (define animal1 (instantiate animal 'fred))
animal1
STk> (ask animal1 'age)
0
STk> (ask animal1 'eat)
yum
STk> (ask animal1 'name)
fred
```

Do you want: A) live coding B) Chalk C) PowerPoint

Inheritance

Set another **class** as a **parent** and then use all of their **methods**!

dogs inherit from animals

(& can call parent methods)

```
(define-class (dog)
  (parent (animal 'doggy-name)))
STk> (define dog1 (instantiate dog))
dog1
STk> (ask dog1 'eat)
yum
```

Can call methods of **parent** on instances of the **child class**

I don't have an `'eat` method. Let me ask my **parent**

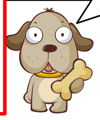
dogs inherit from animals

(& can call parent's automatically generated methods)

```
(define-class (dog)
  (parent (animal 'doggy-name)))
STk> (define dog1 (instantiate dog))
dog1
STk> (ask dog1 'age)
0
```

Can call automatically generated methods in the **parent class** with instances of the **child class**

I don't have an `'age` method. Let me ask my **parent**



Children <u>can not</u> access **parent**'s `instance-vars` directly

```
(define-class (dog)
  (parent (animal 'doggy-name))
  (method (say-name)
    (se 'woof name)))
```

BAD BAD BAD!

This doesn't work!

`self`

A way to ask yourself to call **methods**

```
define-class (dog)
  (parent (animal 'doggy-name))
  (method (say-name)
      (se 'woof (ask self 'name))))
```

You can ask `self` things

Excessively tricky case

```
(define-class (tricky)
  (instance-vars (x 3))
  (method (weird x)
     (* x (ask self 'x))))

(define trick (instantiate tricky))
STk> (ask trick 'weird 4)
```

A) 9 B) 16 C) 12 D) Other

You can do recursion with methods

```
(define-class (math-wiz)
  (method (factorial n)
      (if (< n 2)
          1
          (* n
             (ask self
                  'factorial
                  (- n 1))))))
```

Overriding methods

The `surfer1` class overrides the parent's `say` method

```
(define-class (person)
  (method (say sent)
      (se sent '!)))

(define-class (surfer1)
  (parent (person))
  (method (say sent)
      (se sent 'dude)))
```

Creating a person object

```
(define-class (person)
  (method (say sent)
      (se sent '!)))
```

```
STk> (define p1 (instantiate person))
p1
STk> (ask p1 'say '(happy birthday))
(happy birthday !)
```

Creating a surfer1 object

```
(define-class (surfer1)
  (parent (person))
  (method (say sent)
      (se sent 'dude)))
```

I want it to work more like the **parent**

```
STk> (define s1 (instantiate surfer1))
s1
STk> (ask s1 'say '(happy birthday))
(happy birthday dude)
```

usual

Explicitly call the **parent's method**

Call the usual method (the one you had overridden)

```
(define-class (person)
  (method (say sent)
      (se sent '!)))

(define-class (surfer2)
  (parent (person))
  (method (say sent)
      (usual 'say (se sent 'dude))))
```

Call the usual method (the one you had overridden)

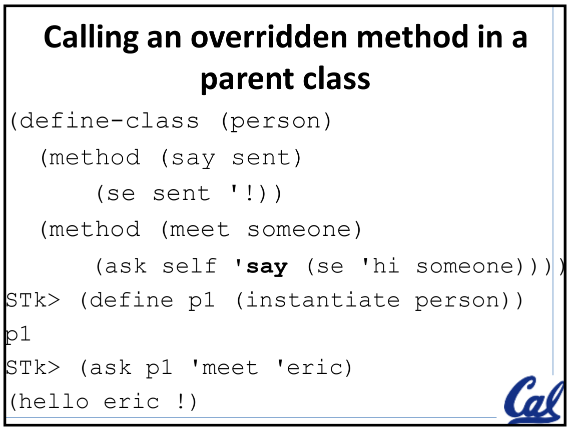
```
(define-class (surfer2)
  (parent (person))
  (method (say sent)
      (usual 'say (se sent 'dude))))
```

```
STk> (define s2 (instantiate surfer2))
s2
STk> (ask s2 'say '(happy birthday))
(happy birthday dude !)
```

Would this have worked?

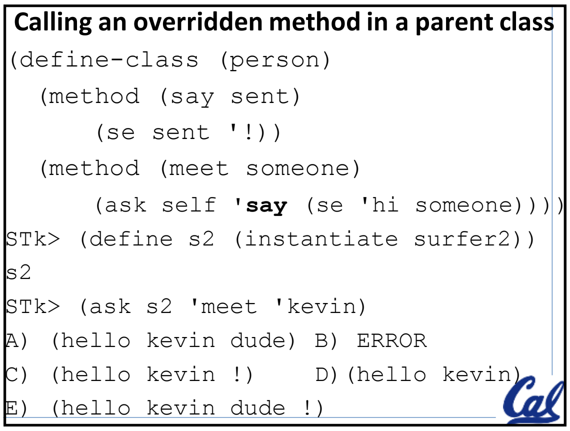
```
(define-class (person)
  (method (say sent)
      (se sent '!)))

(define-class (surfer2)
  (parent (person))
  (method (say sent)
      (ask self 'say (se sent 'dude))))
```

A) Yes B) No C) Sometimes



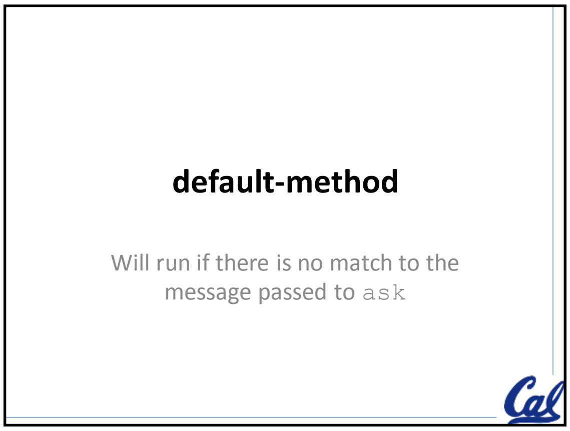
Calling an overridden method in a parent class

```
(define-class (person)
  (method (say sent)
      (se sent '!))
  (method (meet someone)
      (ask self 'say (se 'hi someone))))
STk> (define p1 (instantiate person))
p1
STk> (ask p1 'meet 'eric)
(hello eric !)
```



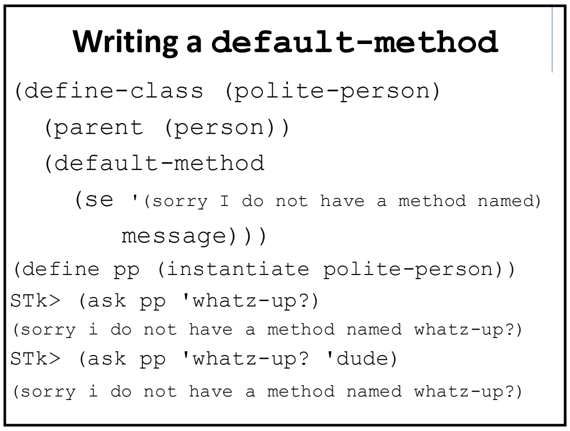
Calling an overridden method in a parent class

```
(define-class (person)
  (method (say sent)
      (se sent '!))
  (method (meet someone)
      (ask self 'say (se 'hi someone))))
STk> (define s2 (instantiate surfer2))
s2
STk> (ask s2 'meet 'kevin)
A) (hello kevin dude) B) ERROR
C) (hello kevin !)     D)(hello kevin)
E) (hello kevin dude !)
```



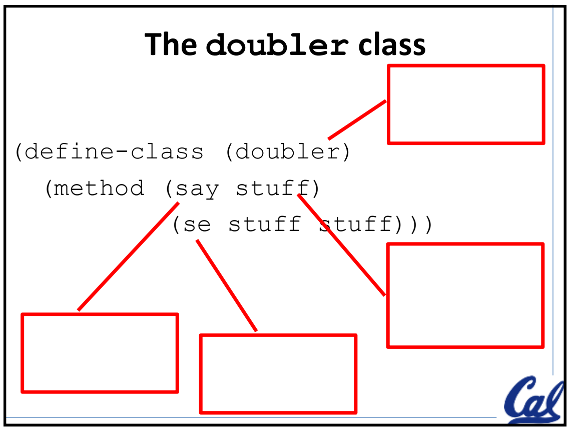
default-method

Will run if there is no match to the message passed to `ask`



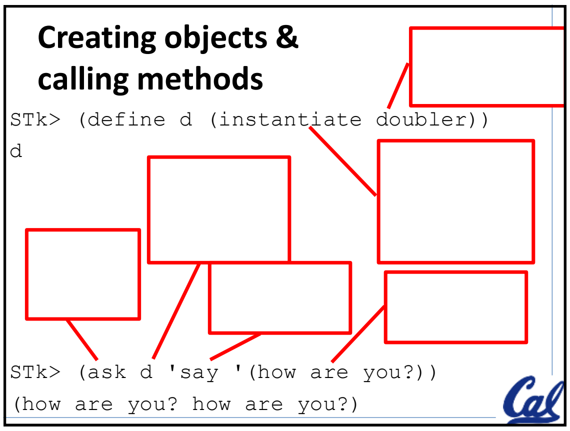
Writing a `default-method`

```
(define-class (polite-person)
  (parent (person))
  (default-method
     (se '(sorry I do not have a method named)
         message)))
(define pp (instantiate polite-person))
STk> (ask pp 'whatz-up?)
(sorry i do not have a method named whatz-up?)
STk> (ask pp 'whatz-up? 'dude)
(sorry i do not have a method named whatz-up?)
```



The `doubler` class

```
(define-class (doubler)
  (method (say stuff)
     (se stuff stuff)))
```



Creating objects & calling methods

```
STk> (define d (instantiate doubler))
d
STk> (ask d 'say '(how are you?))
(how are you? how are you?)
```

instance-vars

```
(define-class (counter)

  (instance-vars (count 0)   )

  (method (welcome)
     (se 'my 'count 'is count)))
```

Initializing class-vars

```
(define-class (beach-bum name)
  (class-vars (surfer-names '()))

  (initialize
    (set! surfer-names (se name surfer-names)))
  (method (say stuff)
       (se stuff 'dude)))
```

Rewriting a let as a lambda

Let review

```
(define (sum-sq a b)
  (let ((a2 (* a a))
        (b2 (* b b)))
     (+ a2 b2)))
STk> (sum-sq 2 3)
```

What does this return?

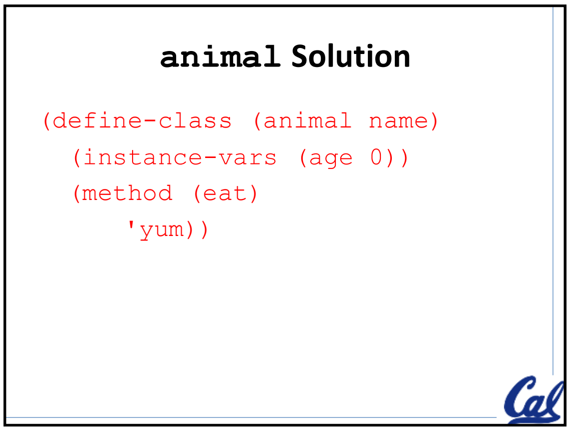
A) 9 B) 10 C) 11 D) 12 E)13

Let review

```
(define (sum-sq a b)
  (let (  (a2     (* a a))
          (b2     (* b b))   )
    (+ a2 b2)  ))

(define (sum-sq a b)
    ((lambda (a2 b2) (+ a2 b2))
      (* a a) (* b b))
```

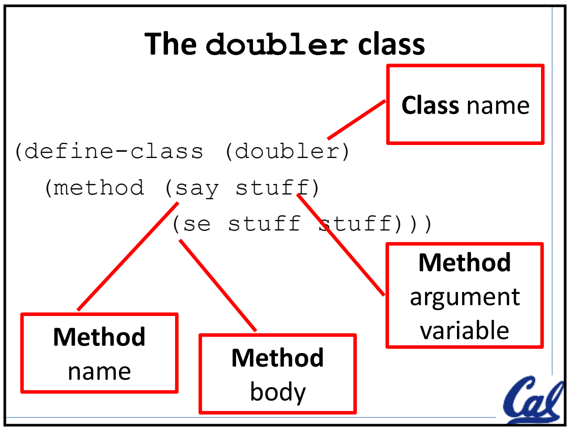
Rewrite the let with lambda

```
(define (funct x)
  (let ((a 3) (b 4) (c 6))
    (+ a b c x)))
```



animal Solution

```
(define-class (animal name)
  (instance-vars (age 0))
  (method (eat)
     'yum))
```



The doubler class

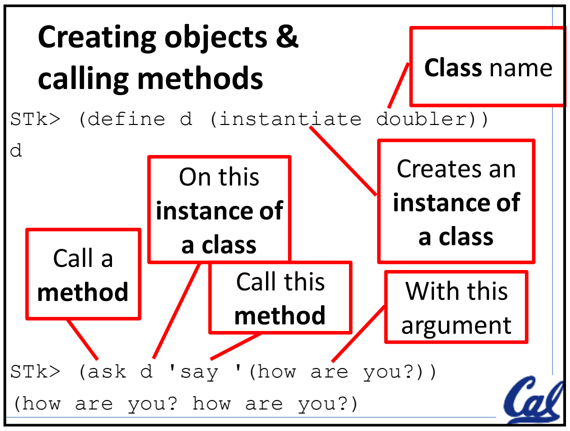
```
(define-class (doubler)
  (method (say stuff)
          (se stuff stuff)))
```

Class name

Method argument variable

Method name

Method body



Creating objects & calling methods

```
STk> (define d (instantiate doubler))
d
STk> (ask d 'say '(how are you?))
(how are you? how are you?)
```

Class name

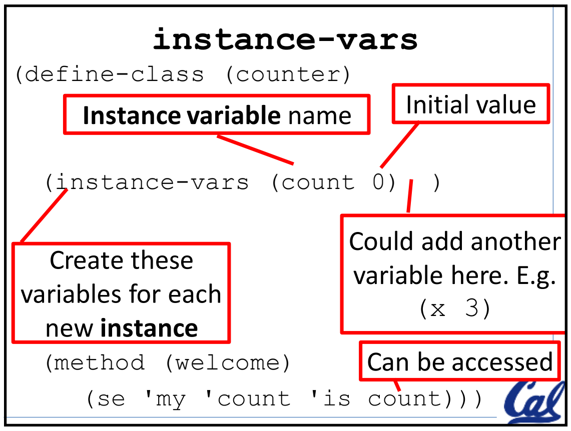
Creates an **instance of a class**

On this **instance of a class**

Call a **method**

Call this **method**

With this argument



instance-vars

```
(define-class (counter)
  (instance-vars (count 0)  )
  (method (welcome)
     (se 'my 'count 'is count)))
```

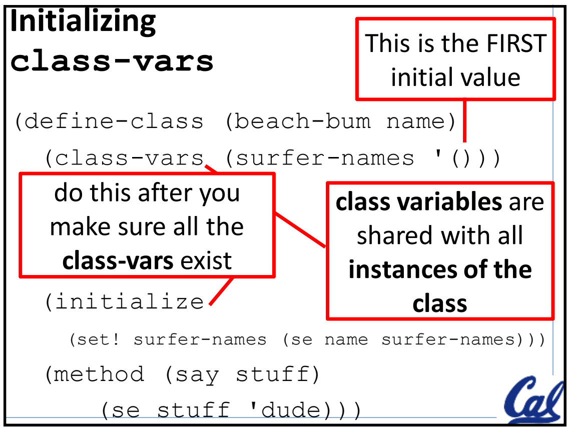
Instance variable name

Initial value

Create these variables for each new **instance**

Could add another variable here. E.g. (x 3)

Can be accessed



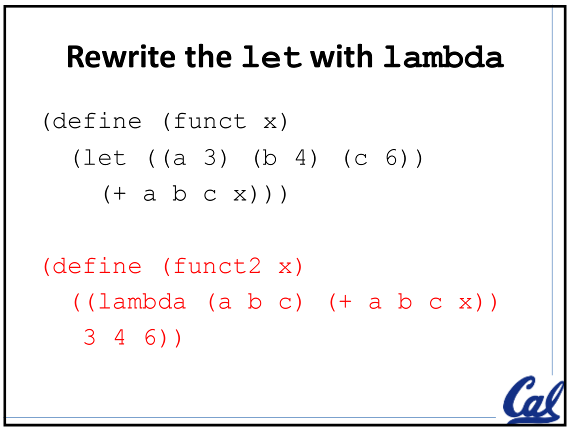
Initializing class-vars

```
(define-class (beach-bum name)
  (class-vars (surfer-names '()))
  (initialize
    (set! surfer-names (se name surfer-names)))
  (method (say stuff)
     (se stuff 'dude)))
```

This is the FIRST initial value

do this after you make sure all the **class-vars** exist

class variables are shared with all **instances of the class**



Rewrite the let with lambda

```
(define (funct x)
  (let ((a 3) (b 4) (c 6))
    (+ a b c x)))

(define (funct2 x)
  ((lambda (a b c) (+ a b c x))
   3 4 6))
```