

Menu

- Objects Review
- Object-Oriented Programming
- Inheritance

CS150 Spring 2006 BC: Lecture 21: Inheritance 2 Computer Science
at the UNIVERSITY of VIRGINIA

Objects

- When we package state and procedures together we have an object
- Programming with objects is object-oriented programming

CS150 Spring 2006 BC: Lecture 21: Inheritance 3 Computer Science
at the UNIVERSITY of VIRGINIA

Counter in Scheme

```
(define (make-ocounter)
  ((lambda (count)
    (lambda (message)
      (if (eq? message 'reset) (set! count 0)
          (if (eq? message 'next)
              (set! count (+ 1 count))
              (if (eq? message 'how-many)
                  count))))))
  0))
```

CS150 Spring 2006 BC: Lecture 21: Inheritance 4 Computer Science
at the UNIVERSITY of VIRGINIA

Counter in Scheme using let

```
(define (make-ocounter)
  (let ((count 0))
    (lambda (message)
      (if (eq? message 'reset) (set! count 0)
          (if (eq? message 'next)
              (set! count (+ 1 count))
              (if (eq? message 'how-many)
                  count))))))
```

CS150 Spring 2006 BC: Lecture 21: Inheritance 5 Computer Science
at the UNIVERSITY of VIRGINIA

Defining ask

(ask Object Method)

```
> (ask bcounter 'how-many)
0
> (ask bcounter 'next)
> (ask bcounter 'how-many)
1
```

```
(define (ask object message)
  (object message))
```

CS150 Spring 2006 BC: Lecture 21: Inheritance 6 Computer Science
at the UNIVERSITY of VIRGINIA

make-number

```
(define make-number
  (lambda (n)
    (lambda (message)
      (cond
        ((eq? message 'value)
         (lambda (self) n))
        ((eq? message 'add)
         (lambda (self other)
           (+ (ask self 'value)
              (ask other 'value))))))))))
```

Why don't we just use n?

ask with arguments

```
(define (ask object message)
  (object message))
```

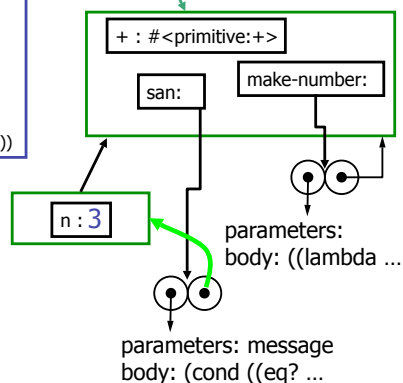
The . means take all the rest of the parameters and make them into a list.

```
(define (ask object message . args)
  (apply (object message) object args))
```

```
(define make-number
  (lambda (n)
    (lambda (message)
      (cond
        ((eq? message 'value)
         (lambda (self) n))
        ((eq? message 'add)
         (lambda (self other)
           (+ (ask self 'value)
              (ask other 'value))))))))))
```

```
> (define san
    (make-number 3))
> (ask san 'value)
3
> (ask san 'add
    (make-number 4))
7
```

global environment



make-fraction

```
(define make-fraction
  (lambda (numerator denominator)
    (lambda (message)
      (cond
        ((eq? message 'value)
         (lambda (self) (/ numerator denominator)))
        ((eq? message 'add)
         (lambda (self other)
           (+ (ask self 'value) (ask other 'value))))
        ((eq? message 'get-numerator)
         (lambda (self) numerator))
        ((eq? message 'get-denominator)
         (lambda (self) denominator))
        )))))
```

Same as in make-number

Note: our add method evaluates to a number, not a fraction object (which would be better).

Why is redefining add a bad thing?

- Cut-and-paste is easy but...
- There could be many such methods (subtract, multiply, print, etc.)
- Making the code bigger makes it harder to understand
- If we fix a bug in the number "add" method, we have to remember to fix the version in make-fraction also (and real, complex, float, etc.)

Inheritance

There are many kinds of numbers...

- Whole Numbers (0, 1, 2, ...)
- Integers (-23, 73, 0, ...)
- Fractions (1/2, 7/8, ...)
- Floating Point (2.3, 0.0004, 3.14159)

- But they can't all do the same things
 - We can get the denominator of a fraction, but not of an integer

make-fraction

```
(define (make-fraction numer denom)
  (let ((super (make-number #f)))
    (lambda (message)
      (cond
        ((eq? message 'value)
         (lambda (self) (/ numer denom)))
        ((eq? message 'get-denominator)
         (lambda (self) denom))
        ((eq? message 'get-numerator)
         (lambda (self) numer))
        (else
         (super message))))))
```

Using Fractions

```
> (define half (make-fraction 1 2))
> (ask half 'value)
1/2
> (ask half 'get-denominator)
2
> (ask half 'add (make-number 1))
3/2
> (ask half 'add half)
1
```

```
> (trace ask)
> (trace eq?)
> (ask half 'add half)
|(ask #<procedure> add #<procedure>)
| (eq? add value) | (ask #<procedure> value)
| #f | (eq? value value)
| (eq? add get-denominator) | #t
| #f | 1/2
| (eq? add get-numerator) | (ask #<procedure> value)
| #f | (eq? value value)
| (eq? add value) | #t
| #f | 1/2
| (eq? add add) | 1
| #t | 1
```

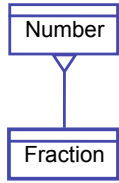
make-number make-fraction

```
> (trace ask)
> (trace eq?)
> (ask half 'add half)
|(ask #<procedure> add #<procedure>)
| (eq? add value) | (ask #<procedure> value)
| #f | (eq? value value)
| (eq? add get-denominator) | #t
| #f | 1/2
| (eq? add get-numerator) | (ask #<procedure> value)
| #f | (eq? value value)
| (eq? add value) | #t
| #f | 1/2
| (eq? add add) | 1
| #t | 1
```

Inheritance

Inheritance is using the definition of one class to make another class

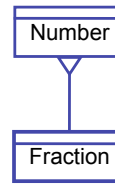
make-fraction uses **make-number** to inherit the behaviors of number



- **English**
A Fraction is a kind of Number.
- **C++**
Fraction is a derived class whose base class is Number
- **Java**
Fraction extends Number.
- **Eiffel**
Fraction inherits from Number.
- **Beta**
Fraction is a subpattern of Number.
- **Smalltalk (72) (and Squeak 05)**
Don't have inheritance!

Note: people sometimes draw this different ways

CS 150:



- Fraction inherits from Number.
- Fraction is a subclass of Number.
- The superclass of Fraction is Number.

Subtyping

- Subtyping is very important in statically typed languages (like C, C++, C#, Java, Pascal) where you have to explicitly declare a type for all variables:

```
method Number add (Number n) { ... }
```

Because of subtyping, either a `Number` or a `Fraction` (subtype of `Number`) could be passed as the argument

- We won't cover subtyping (although we will talk more about types later)

Who's Hungry?

```

(define (make-sammich name calories)
  (let ((tasty #f))
    (lambda (message)
      (case message
        ((class) (lambda (self) 'sandwich))
        ((name) (lambda (self) name))
        ((calories) (lambda (self) calories))
        ((eat) (lambda (self)
                  (display (if (ask self 'tasty) "Yummy" "Yuck"))))
        ((add-mayonnaise) (lambda (self) (ask self 'set-tasty #t)))
        ((set-tasty) (lambda (self t) (set! tasty t)))
        ((tasty) (lambda (self) tasty)))
      )))
  )
)

```

Hello, awesome

Who's Hungry?

```

> (define club (make-sammich "club" 1000))
> (ask club 'name)
"club"
> (ask club 'calories)
1000
> (ask club 'eat)
Yuck
> (ask club 'add-mayonnaise)
> (ask club 'eat)
Yummy

```

Who's Hungry?

```

> (define happy (make-sammich "grilled cheese" 500))
> (ask happy 'name)
"grilled cheese"
> (ask happy 'calories)
500
> (ask happy 'eat)
Yuck
> (ask happy 'add-mayonnaise)
> (ask happy 'eat)
Yummy

```

This is clearly not right.

Who's Hungry?

- Need to set up some state at "prep" time:

```
(define (make-grilled-cheese)
  (let ((super (make-sammich "grilled cheese" 500)))
    (begin
      (ask super 'set-tasty #t) It's already tasty
      (lambda (message)
        (case message
          ((add-mayonnaise)
           (lambda (self)
            (ask self 'set-tasty #f))) Who does that?
          (else (super message)))))) Otherwise, it's a sammich
```

Who's Hungry?

```
> (define happy (make-grilled-cheese))
> (ask happy 'name)
"grilled cheese"
> (ask happy 'calories)
500
> (ask happy 'eat)
Yummy
> (ask happy 'add-mayonnaise)
> (ask happy 'eat)
Yuck
```

This is much better.

Charge

- PS6
 - Programming an adventure game
 - Using objects and inheritance
 - Due two weeks from today
- Eat a sammich