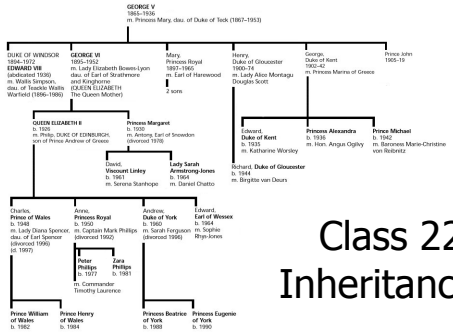


Class 22: Inheritance

CS150: Computer Science
University of Virginia
Computer Science

David Evans
http://www.cs.virginia.edu/evans

Menu

- Objects Review
- Object-Oriented Programming
- Inheritance

Objects

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

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))
```


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))))))
```


Defining **ask**

(ask *Object Method*)

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

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

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? (Well see why later today.)

ask with arguments

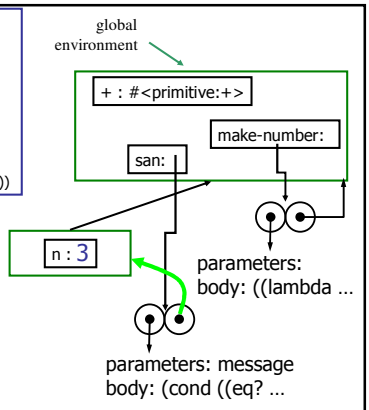
```
(define (ask object message)
  (object message))
```

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
```


Object-Oriented Programming

Simula

- Considered the first "object-oriented" programming language
- Language designed for *simula*tion by Kristen Nygaard and Ole-Johan Dahl (Norway, 1962)
- Had special syntax for defining classes that packages state and procedures together

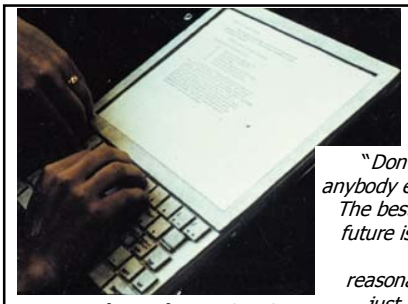
Counter in Simula

```
class counter;
  integer count;
  begin
    procedure reset(); count := 0; end;
    procedure next();
        count := count + 1; end;
    integer procedure how-many();
        how-many := count; end;
  end
```

XEROX Palo Alto Research Center (PARC)

1970s:

- Bitmapped display
- Graphical User Interface
  - Steve Jobs paid $1M to visit and PARC, and returned to make Apple Lisa/Mac
- Ethernet
- First personal computer (Alto)
- PostScript Printers
- **Object-Oriented Programming**



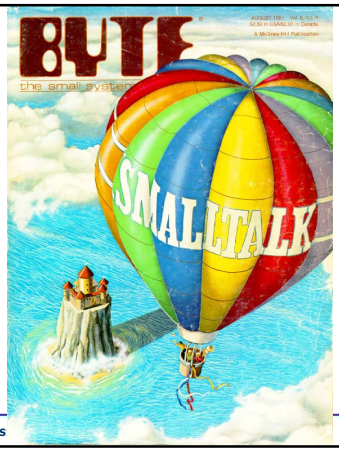
Dynabook, 1972

(Just a model)

"Don't worry about what anybody else is going to do… The best way to predict the future is to invent it. Really smart people with reasonable funding can do just about anything that doesn't violate too many of Newton's Laws!"
— Alan Kay, 1971

Dynabook 1972

- Tablet computer
- Intended as tool for learning
- Kay wanted children to be able to program it also
- Hallway argument, Kay claims you could define "the most powerful language in the world in a page of code"
- Proof: Smalltalk
  - Scheme is as powerful, but takes two pages



BYTE Magazine, August 1981

Smalltalk

- Everything is an *object*
- Objects communicate by sending and receiving *messages*
- Objects have their own state (which may contain other objects)
- How do you do 3 + 4?

  send the object **3** the message **"+ 4"**

Counter in Smalltalk

```
class name counter
   instance variable names count
   new count <- 0
   next count <- count + 1
   how-many ^ count
```

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
  (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 lots of number methods (subtract, multiply, print, etc.)
- Making the code bigger makes it harder to understand
- If we fix a problem in the number add method, we have to remember to fix the copy in make-fraction also (and real, complex, float, etc.)

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)
| #f
| (eq? add get-denominator)
| #f
| (eq? add get-numerator)
| #f
| (eq? add value)
| #f
| (eq? add add)
| #t
| (ask #<procedure> value)
| |(eq? value value)
| |#t
| 1/2
| (ask #<procedure> value)
| |(eq? value value)
| |#t
| 1/2
|1
1
```

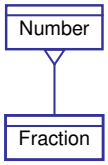

make-number
make-fraction

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


Inheritance

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

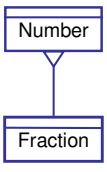
make-fraction uses make-number to *inherit* the behaviors of number



Note: people sometimes draw this different ways

- **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!

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 was the first object-oriented programmer?

By the word operation, we mean any process which alters the mutual relation of two or more things, be this relation of what kind it may. This is the most general definition, and would include all subjects in the universe. Again, it might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine... Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent. Ada, Countess of Lovelace, around 1830

Charge

- PS5: Due Monday
- PS6: Out Monday
  - Programming an adventure game using objects and inheritance