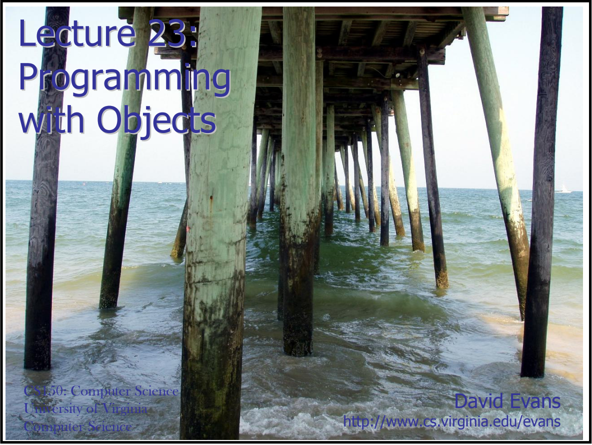


Lecture 23: Programming with Objects

CS150: Computer Science
University of Virginia
Computer Science

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

Reminder

- Start thinking of ideas of PS9 and discussing them on the forum
  http://www.cs.virginia.edu/forums/viewforum.php?f=28
  - You can also vote in the "should we have a quiz Monday" poll
    http://www.cs.virginia.edu/forums/viewtopic.php?t=1651

http://www.sportsline.com/collegebasketball/scoreboard

Problem-Solving Strategies

- PS1-PS4: Functional Programming
  - Focused on **procedures**
  - Break a problem into procedures that can be combined to solve it
- PS5: Imperative Programming
  - Focused on **data**
  - Design data for representing a problem and procedures for updating that data

Problem-Solving Strategies

- PS6: "Object-Oriented Programming"
  - Focused on **objects**: package procedures and state
  - Model a problem by dividing it into objects
  - Lots of problems in real (and imaginary) worlds can be thought of this way

Counter Object

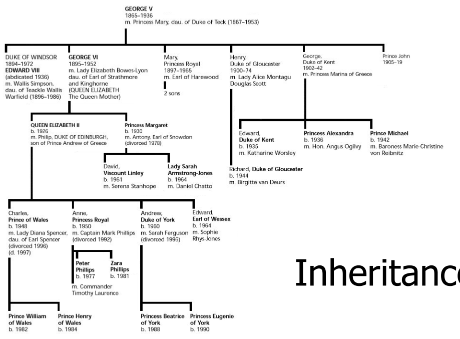
```
(define (make-counter)
  (let ((count 0))          Instance variable
    (lambda (message)
      (cond ((eq? message 'reset!)     Methods
             (set! count 0))
            ((eq? message 'next!)
             (set! count (+ 1 count)))
            ((eq? message 'current) count)
            (else
              (error "Unrecognized message"))))))
```

Defining **ask**

(ask *Object Method*)

```
> (define bcounter (make-counter))
> (ask bcounter 'current)
0
> (ask bcounter 'next)
> (ask bcounter 'current)
1
```

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


GEORGE V 1865–1936 m. Princess Mary, dau. of Duke of Teck (1867–1953)
DUKE OF WINDSOR 1894–1972 EDWARD VIII (abdicated 1936) m. Wallis Simpson, dau. of Teackle Wallis Warfield (1896–1986)
GEORGE VI 1895–1952 m. Lady Elizabeth Bowes-Lyon dau. of Earl of Strathmore and Kinghorne (QUEEN ELIZABETH The Queen Mother)
Mary, Princess Royal 1897–1965 m. Earl of Harewood
2 sons
Henry, Duke of Gloucester 1900–74 m. Lady Alice Montagu Douglas Scott
George, Duke of Kent 1902–42 m. Princess Marina of Greece
Prince John 1905–19
QUEEN ELIZABETH II b. 1926 m. Philip, DUKE OF EDINBURGH, son of Prince Andrew of Greece
Princess Margaret b. 1930 m. Antony, Earl of Snowdon (divorced 1978)
David, Viscount Linley b. 1961 m. Serena Stanhope
Lady Sarah Armstrong-Jones b. 1964 m. Daniel Chatto
Edward, Duke of Kent b. 1935 m. Katharine Worsley
Princess Alexandra b. 1936 m. Hon. Angus Ogilvy
Prince Michael b. 1942 m. Baroness Marie-Christine von Reibnitz
Richard, Duke of Gloucester b. 1944 m. Birgitte van Deurs
Charles, Prince of Wales b. 1948 m. Lady Diana Spencer, dau. of Earl Spencer (divorced 1996) (d. 1997)
Anne, Princess Royal b. 1950 m. Captain Mark Phillips (divorced 1992)
Andrew, Duke of York b. 1960 m. Sarah Ferguson (divorced 1996)
Edward, Earl of Wessex b. 1964 m. Sophie Rhys-Jones
Peter Phillips b. 1977
Zara Phillips b. 1981
m. Commander Timothy Laurence
Prince William of Wales b. 1982
Prince Henry of Wales b. 1984
Princess Beatrice of York b. 1988
Princess Eugenie of York b. 1990

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))))))
```

Making Subobjects

```
(define (make-fraction numer denom)
   (make-subobject
     (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 #f)))))
```

Implementing make-subobject

```
(define (make-subobject super imp)
  (lambda (message)
    (if (eq? message 'super)
      (lambda (self) super)
      (let ((method (imp message)))
        (if method
          method
          (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                           | (ask #<procedure> value)
| (eq? add get-denominator)    | |(eq? value value)
| #f                           | |#t
| (eq? add get-numerator)      | 1/2
| #f                           | (ask #<procedure> value)
| (eq? add value)              | |(eq? value value)
| #f                           | |#t
| (eq? add add)                | 1/2
| #t                           |1
                               1
```

make-number
make-fraction

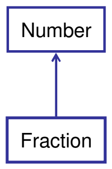
```
> (trace ask)
> (trace eq?)
> (ask half 'add half)
|(ask #<procedure> add #<procedure>)
| (eq? add value)
| #f                           | (ask #<procedure> value)
| (eq? add get-denominator)    | |(eq? value value)
| #f                           | |#t
| (eq? add get-numerator)      | 1/2
| #f  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

Speaking about Inheritance



Fraction *inherits* from Number.

Fraction is a *subclass* of Number.

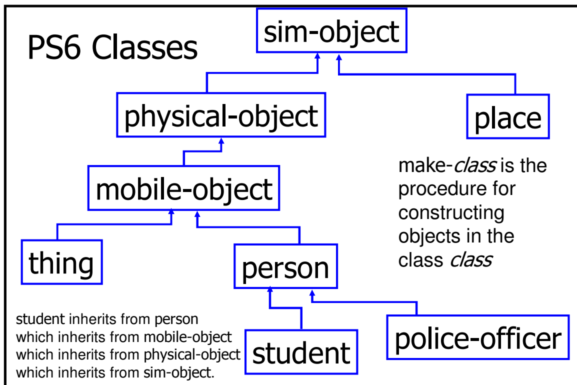
The *superclass* of Fraction is Number.

PS6

Make an adventure game programming with objects

Many objects in our game have similar properties and behaviors, so we use inheritance.

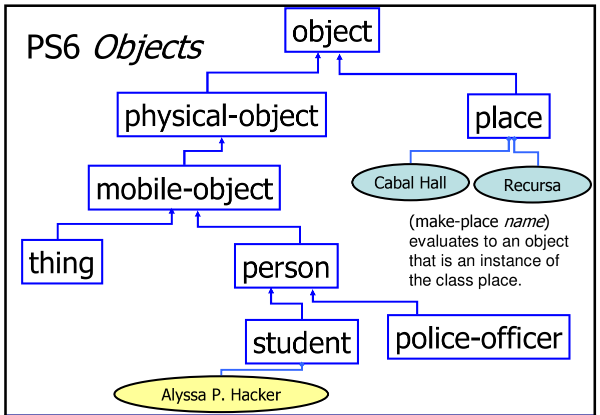
PS6 Classes



make-*class* is the procedure for constructing objects in the class *class*

student inherits from person which inherits from mobile-object which inherits from physical-object which inherits from sim-object.

PS6 *Objects*



(make-place *name*) evaluates to an object that is an instance of the class place.

Are there class hierarchies like this in the "real world" or just in fictional worlds like Charlottansville?

Charge

- Monday:
  - Quiz on GEB reading (depending on poll)
  - History of Object-Oriented Programming
- PS6 due Friday
- Start thinking about PS9 project ideas
  - Use the forum to find teammates and propose ideas