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))))))

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 \textit{n}? (Well see why later today.)

ask with arguments
(define (ask object message . args)
  (apply (object message) object args))

The \texttt{.} means take all the rest of the parameters and make them into a list.

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

The \texttt{.} means take all the rest of the parameters and make them into a list.

The \texttt{.} means take all the rest of the parameters and make them into a list.

Object-Oriented Programming

Simula
- Considered the first “object-oriented” programming language
- Language designed for \textit{simulation} by Kristen Nygaard and Ole-Johan Dahl (Norway, 1962)
- Had special syntax for defining classes that packages state and procedures together

Counter in Simula
\begin{verbatim}
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
\end{verbatim}
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

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
```
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

```scheme
(make-fraction)
(lambda (numerator denominator)
  (lambda (message)
    (cond
      ((eq? message 'value)
        (lambda (self) (/ numerator denominator)))
      ((eq? message 'get-denominator)
        (lambda (self) denominator))
      ((eq? message 'get-numerator)
        (lambda (self) numerator))
      (else
        (super message))))))
```

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.)
Inheritance

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

`make-fraction` uses `make-number` to inherit the behaviors of number.

**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