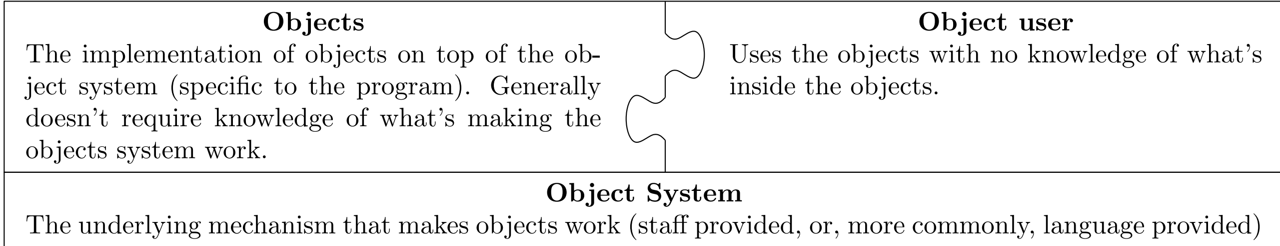
6.001 — Tutorial 8

Structure and Interpretation of Computer Programs — April 23, 2007

Object-Oriented Programming

OOPs



Creating and Using Objects

With our full-blown object system, here's what the code for a class looks like:

```scheme
;; Define the class constructor
(define (create-TYPE arg1 arg2 ...)
  (create-instance TYPE arg1 arg2 ...))

;; Define the class
(define (TYPE self arg1 arg2 ...)
  ;; Capture internal state, including instances of superclasses to
  ;; delegate to. If there are no better superclasses, don't forget
  ;; to inherit from root-object
  (let ((super1-part (super1 self args...))
        (super2-part (super2 self args...))
        ;; Other superclasses and local state
        )
    ;; This function has to return the handler for class of type, so
    ;; create that handler
    (make-handler
      ;; The handler needs to know the name of this object's type
      'TYPE
      ;; make-methods bundles together the methods for the handler
      (make-methods
        'METHOD1 (lambda (args...) ...)
        'METHOD2 (lambda (args...) ...))
      ;; Finally, the handler needs to know where to go for messages
      ;; that weren't defined here
      super1-part
      super2-part)))

;; Instantiate the class
(define inst (create-TYPE arg1 arg2 ...))

;; Ask the instance to do something (ie, pass it a message)
(ask inst 'METHOD1 arg1 arg2 ...)

;; Or you can get the method directly as a procedure (useful in
;; conjunction with higher order procedures, since this allows you
;; to pass message handlers/methods around like regular procedures)
(define method-of-inst (get-method inst 'METHOD1))
(method-of-inst arg1 arg2 ...)
```

- To call method X in another object O, use `(ask O 'X ...)`
- To call overridden method X from X in a subclass, use `(ask whatever-part 'X ...)`. This lets you override a method in a superclass without losing access to its functionality.
- To call another method Y in this class, use `(ask self 'Y ...)`. This asks the *whole* object, instead of just part of it.

The Object System

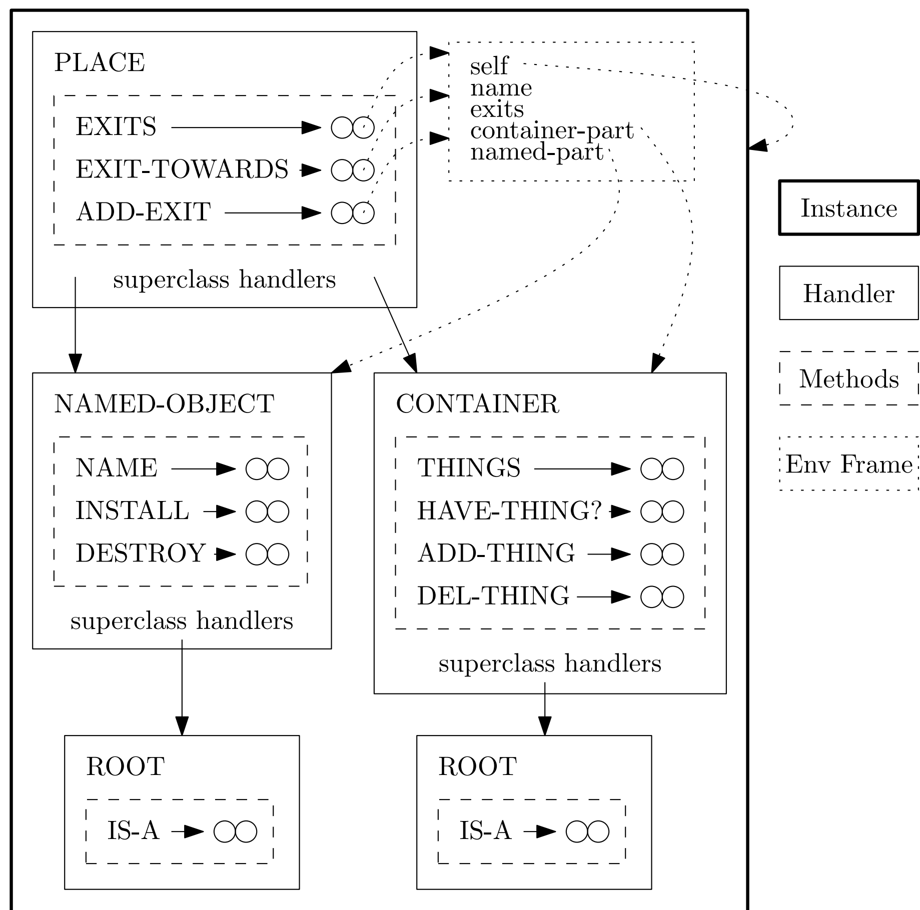
The provided object system interface uses three abstractions.

Methods Created by `make-methods`. Simply a mapping between method names and handler procedures.

Handlers Created by `make-handler`. Consists of the type name (a symbol), methods (as created by `make-methods`), and a list of *handlers* (not instances!) for the super classes.

Instances Created by `create-instance`. Consists of a self pointer and a handler (which may in turn contain handlers for superclasses).

Here's an example of what a PLACE instance looks like and how it is composed of handlers and methods:



ask self Versus ask part

What do the last three expressions evaluate to?

```
(define (create-foo) (create-instance foo))
(define (foo self)
  (let ((root-part (root self)))
    (make-handler 'foo
                  (make-methods 'A (lambda () 1)
                                'B (lambda () 2))
                  root-part)))
(define (create-bar) (create-instance bar))
(define (bar self)
  (let ((foo-part (foo self)))
    (make-handler 'bar
                  (make-methods 'A (lambda () 3)
                                'B (lambda () (+ (ask foo-part 'B)
                                                 (ask self 'A))))
                  foo-part)))
(define (create-baz) (create-instance baz))
(define (baz self)
  (let ((bar-part (bar self)))
    (make-handler 'baz
                  (make-methods 'A (lambda () 4)
                                'B (lambda () (ask bar-part 'B)))
                  bar-part)))
(ask (create-foo) 'B)     =>
(ask (create-bar) 'B)     =>
(ask (create-baz) 'B)     =>
```

Brainteaser: Classless Object Systems

Consider an object system in which everything's an object (ie, there's no huge distinction between classes and instances). For example, if we ask an object in our object system for its type, it returns a list of symbols, which are only names. Thus, classes are not, themselves, first-class (we can pass around the name, but we can't use that to, say, instantiate the class or ask it what methods it has). We could imaging making classes objects, so if we ask an object what type it is, it just returns an object representing the class it was instantiated from. If we want to instantiate a class, we simply ask that class to create an instance of itself. In such a system, classes become *metaobjects*, and the conventions for using them (such as what method one calls to instantiate an instance) form the *metaobject protocol*. Since every object has a type and now our classes are just objects, our classes also have types. These are known as *metaclasses*. This, of course, is just another object, which itself has must have a class!

Your first challenge is to wrap your head around metaclasses. Your second challenge is to figure out something you can do with them.