Object-Oriented Programming — Below the line view

This document documents the Object Oriented Programming system for CS 61A in terms of its implementation in Scheme. It assumes that you already know what the system does, i.e. that you've read "Object-Oriented Programming — Above the line view." Also, this handout will assume a knowledge of how to implement message passing and local state variables in Scheme, from chapters 2.3 and 3.1 of A&S. (Chapter 3.2 from A&S will also be helpful.)

Almost all of the work of the object system is handled by the special form `define-class`. When you type a list that begins with the symbol `define-class`, Scheme translates your class definition into Scheme code to implement that class. This translated version of your class definition is written entirely in terms of `define`, `let`, `lambda`, `set!`, and other Scheme functions that you already know about.

We will focus on the implementation of the three main technical ideas in OOP: message passing, local state, and inheritance.

Message Passing

The text introduces message-passing with this example from Section 2.3.3 (page 141):

```
(define (make-rectangular x y)
  (define (dispatch m)
    (cond ((eq? m 'real-part) x)
          ((eq? m 'imag-part) y)
          ((eq? m 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? m 'angle) (atan y x))
          (else
           (error "Unknown op -- MAKE-RECTANGULAR" m))))
  dispatch)
```

In this example, a complex number object is represented by a dispatch procedure. The procedure takes a *message* as its argument, and returns a number as its result. Later, in Section 3.1.1 (page 173), the text uses a refinement of this representation in which the dispatch procedure returns a *procedure* instead of a number. The reason they make this change is to allow for extra arguments to what we are calling the *method* that responds to a message. The user says

```
((acc 'withdraw) 100)
```

Evaluating this expression requires a two-step process: First, the dispatch procedure (named `acc`) is invoked with the message `withdraw` as its argument. The dispatch procedure returns the withdraw method procedure, and that second procedure is invoked with `100` as its argument to do the actual work. All of an object's activity comes from invoking its method procedures; the only job of the object itself is to return the right procedure when it gets sent a message.

Any OOP system that uses the message-passing model must have some below-the-line mechanism for associating methods with messages. In Scheme, with its first-class procedures, it is very natural to use a dispatch procedure as the association mechanism. In some other language the object might instead be represented as an array of message-method pairs.

If we are treating objects as an abstract data type, programs that use objects shouldn't have to know that we happen to be representing objects as procedures. The two-step notation for invoking a method violates this abstraction barrier. To fix this we invent the `ask` procedure:

```
(define (ask object message . args)
  (let ((method (object message)))     ; Step 1: invoke dispatch procedure
    (if (method? method)
        (apply method args)            ; Step 2: invoke the method
        (error "No method" message (cadr method)))))
```

`Ask` carries out essentially the same steps as the explicit notation used in the text. First it invokes the dispatch procedure (that is, the object itself) with the message as its argument. This should return a method (another procedure). The second step is to invoke that method procedure with whatever extra arguments have been provided to `ask`.

The body of `ask` looks more complicated than the earlier version, but most of that has to do with error-checking: What if the object doesn't recognize the message we send it? These details aren't very important. `Ask` does use two features of Scheme that we haven't discussed before:

The dot notation used in the formal parameter list of `ask` means that it accepts any number of arguments. The first two are associated with the formal parameters `object` and `message`; all the remaining arguments (zero or more of them) are put in a list and associated with the formal parameter `args`.

The procedure `apply` takes a procedure and a list of arguments and applies the procedure to the arguments. The reason we need it here is that we don't know in advance how many arguments the method will be given; if we said `(method args)` we would be giving the method *one* argument, namely, a list.

In our OOP system, you generally send messages to instances, but you can also send some messages to classes, namely the ones to examine class variables. When you send a message to a class, just as when you send one to an instance, you get back a method. That's why we can use `ask` with both instances and classes. (The OOP system itself also sends the class an `instantiate` message when you ask it to create a new instance.) Therefore, both the class and each instance is represented by a dispatch procedure. The overall structure of a class definition looks something like this:

```
(define (class-dispatch-procedure class-message)
  (cond ((eq? class-message 'some-var-name) (lambda () (get-the-value)))
        (...)
        ((eq? class-message 'instantiate)
         (lambda (instantiation-var ...)
           (define (instance-dispatch-procedure instance-message)
             (cond ((eq? instance-message 'foo) (lambda ...))
                   (...)
                   (else (error "No method in instance")) ))
           instance-dispatch-procedure))
        (else (error "No method in class")) ))
```

(Please note that this is *not* exactly what a class really looks like. In this simplified version we have left out many details. The only crucial point here is that there are two dispatch procedures, one inside the other.) In each dispatch procedure, there is a cond with a clause for each allowable message. The consequent expression of each clause is a lambda expression that defines the corresponding method. (In the text, the examples often use named method procedures, and the consequent expressions are names rather than lambdas. We found it more convenient this way, but it doesn't really matter.)

Local State

You learned in section 3.1 that the way to give a procedure a local state variable is to define that procedure inside another procedure that establishes the variable. That outer procedure might be the implicit procedure in the let special form, as in this example from page 171:

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
         balance)
          "Insufficient funds"))))
```

In the OOP system, there are three kinds of local state variables: class variables, instance variables, and instantiation variables. Although instantiation variables are just a special kind of instance variable above the line, they are implemented differently. Here is another simplified view of a class definition, this time leaving out all the message passing stuff and focusing on the variables:

```
(define class-dispatch-procedure
  (LET ((CLASS-VAR1 VAL1)
        (CLASS-VAR2 VAL2) ...)
    (lambda (class-message)
      (cond ((eq? class-message 'class-var1) (lambda () class-var1))
            ...
            ((eq? class-message 'instantiate)
             (lambda (INSTANTIATION-VARIABLE1 ...)
               (LET ((INSTANCE-VAR1 VAL1)
                     (INSTANCE-VAR2 VAL2) ...)
                 (define (instance-dispatch-procedure instance-message)
                   ...)
                 instance-dispatch-procedure)))))))
```

The scope of a class variable includes the class dispatch procedure, the instance dispatch procedure, and all of the methods within those. The scope of an instance variable does not include the class dispatch procedure in its methods. Each invocation of the class instantiate method gives rise to a new set of instance variables, just as each new bank account in the book has its own local state variables.

Why are class variables and instance variables implemented using `let`, but not instantiation variables? The reason is that class and instance variables are given their (initial) values by the class definition itself. That's what `let` does: It establishes the connection between a name and a value. Instantiation variables, however, don't get values until each particular instance of the class is created, so we implement these variables as the formal parameters of a `lambda` that will be invoked to create an instance.

Inheritance and Delegation

Inheritance is the mechanism through which objects of a child class can use methods from a parent class. Ideally, all such methods would just be part of the repertoire of the child class; the parent's procedure definitions would be "copied into" the Scheme implementation of the child class.

The actual implementation in our OOP system, although it has the same purpose, uses a somewhat different technique called *delegation.* Each object's dispatch procedure contains entries only for the methods of its own class, not its parent classes. But each object has, in an instance variable, an object of its parent class. To make it easier to talk about all these objects and classes, let's take an example that we looked at before:

```
(define-class (checking-account init-balance)
  (parent (account init-balance))
  (method (write-check amount)
    (ask self 'withdraw (+ amount 0.10)) ))
```

Let's create an instance of that class:

```
(define Gerry-account (instantiate checking-account 20000))
```

Then the object named `Gerry-account` will have an instance variable named `my-account` whose value is an instance of the `account` class. (The variables `my-whatever` are created automatically by `define-class`.)

What good is this parent instance? If the dispatch procedure for `Gerry-account` doesn't recognize some message, then it reaches the `else` clause of the `cond`. In an object without a parent, that clause will generate an error message. But if the object does have a parent, the `else` clause passes the message on to the parent's dispatch procedure:

```
(define (make-checking-account-instance init-balance)
  (LET ((MY-ACCOUNT (INSTANTIATE ACCOUNT INIT-BALANCE)))
    (lambda (message)
      (cond ((eq? message 'write-check) (lambda (amount) ...))
            ((eq? message 'init-balance) (lambda () init-balance))
            (ELSE (MY-ACCOUNT MESSAGE)) ))))
```

(Naturally, this is a vastly simplified picture. We've left out the class dispatch procedure, among other details. There isn't really a procedure named `make-checking-account-instance` in the implementation; this procedure is really the `instantiate` method for the class, as we explained earlier.)

When we send `Gerry-account` a `write-check` message, it's handled in the straightforward way we've been talking about before this section. But when we send `Gerry-account` a `deposit` message, we reach the `else` clause of the `cond` and the message is delegated to the parent `account` object. That object (that is, its dispatch procedure) returns a method, and `Gerry-account` returns the method too.

The crucial thing to understand is why the `else` clause does *not* say

```
(else (ask my-parent message))
```

The `Gerry-account` dispatch procedure takes a message as its argument, and returns *a method* as its result. `Ask`, you'll recall, carries out a two-step process in which it first gets the method and then invokes that method. Within the dispatch procedure we only want to get the method, not invoke it. (Somewhere there is an invocation of `ask` waiting for `Gerry-account`'s dispatch procedure to return a method, which `ask` will then invoke.)

There is one drawback to the delegation technique. As we mentioned in the above-the-line handout, when we ask `Gerry-account` to `deposit` some money, the `deposit` method only has access to the local state variables of the `account` class, not those of the `checking-account` class. Similarly, the `write-check` method doesn't have access to the `account` local state variables like `balance`. You can see why this limitation occurs: Each method is a procedure defined within the scope of one or the other class procedure, and Scheme's lexical scoping rules restrict each method to the variables whose scope contains it. The technical distinction between *inheritance* and *delegation* is that an inheritance-based OOP system does not have this restriction.

We can get around the limitation by using messages that ask the other class (the child asks the parent, or *vice versa*) to return (or modify) one of its variables. The `(ask self 'withdraw ...)` in the `write-check` method is an example.

Bells and Whistles

The simplified Scheme implementation shown above hides several complications in the actual OOP system. What we have explained so far is really the most important part of the implementation, and you shouldn't let the details that follow confuse you about the core ideas. We're giving pretty brief explanations of these things, leaving out the gory details.

One complication is multiple inheritance. Instead of delegating an unknown message to just one parent, we have to try more than one. The real `else` clauses invoke a procedure called `get-method` that accepts any number of objects (i.e., dispatch procedures) as arguments, in addition to the message. `Get-method` tries to find a method in each object in turn; only if all of the parents fail to provide a method does it give an error message. (There will be a `my-`whatever variable for each of the parent classes.)

Another complication that affects the `else` clause is the possible use of a `default-method` in the class definition. If this optional feature is used, the body of the `default-method` clause becomes part of the object's `else` clause.

When an instance is created, the `instantiate` procedure sends it an `initialize` message. Every dispatch procedure automatically has a corresponding method. If the `initialize` clause is used in define-class, then the method includes that code. But even if there is no initialize clause, the OOP system has some initialization tasks of its own to perform.

In particular, the initialization must provide a value for the self variable. Every initialize method takes the desired value for self as an argument. If there are no parents or children involved, self is just another name for the object's own dispatch procedure. But if an instance is the my-whatever of some child instance, then self should mean that child. The solution is that the child's initialize method invokes the parent's initialize method with the child's own self as the argument. (Where does the child get its self argument? It is provided by the instantiate procedure.)

Finally, usual involves some complications. Each object has a send-usual-to-parent method that essentially duplicates the job of the ask procedure, except that it only looks for methods in the parents, as the else clause does. Invoking usual causes this method to be invoked.

A useful feature

To aid in your understanding of the below-the-line functioning of this system, we have provided a way to look at the translated Scheme code directly, i.e., to look at the below-the-line version of a class definition. To look at the definition of the class foo, for example, you type

```
(show-class 'foo)
```

If you do this, you will see the complete translation of a define-class, including all the details we've been glossing over. But you should now understand the central issues well enough to be able to make sense of it.

We end this document with one huge example showing every feature of the object system. Here are the above-the-line class definitions:

```
(define-class (person) (method (smell-flowers) 'Mmm!))
(define-class (fruit-lover fruit) (method (favorite-food) fruit))

(define-class (banana-holder name)
  (class-vars (list-of-banana-holders '()))
  (instance-vars (bananas 0))
  (method (get-more-bananas amount)
    (set! bananas (+ bananas amount)))
  (default-method 'sorry)
  (parent (person) (fruit-lover 'banana))
  (initialize
   (set! list-of-banana-holders (cons self list-of-banana-holders))) )
```

On the next page we show the translation of the banana-holder class definition into ordinary Scheme. Of course this is hideously long, since we have artificially defined the class to use every possible feature at once. The translations aren't meant to be read by people, ordinarily. The comments in the translated version were added just for this handout; you won't see comments if you use show-class yourself.

```
(define banana-holder
 (let ((list-of-banana-holders '()))                    ;; class vars set up
  (lambda (class-message)                               ;; class dispatch proc
    (cond
     ((eq? class-message 'list-of-banana-holders)
      (lambda () list-of-banana-holders))
     ((eq? class-message 'instantiate)
      (lambda (name)                                    ;; Instantiation vars
        (let ((self '())                                ;; Instance vars
              (my-person (instantiate-parent person))
              (my-fruit-lover (instantiate-parent fruit-lover 'banana))
              (bananas 0))
          (define (dispatch message)                    ;; Object dispatch proc
            (cond
             ((eq? message 'initialize)                 ;; Initialize method:
              (lambda (value-for-self)                  ;;  set up self variable
                (set! self value-for-self)
                (ask my-person 'initialize self)
                (ask my-fruit-lover 'initialize self)
                (set! list-of-banana-holders            ;;  user's init code
                      (cons self list-of-banana-holders))))
             ((eq? message 'send-usual-to-parent)       ;; How USUAL works
              (lambda (message . args)
                (let ((method (get-method
                               'banana-holder
                               message
                               my-person
                               my-fruit-lover)))
                  (if (method? method)
                      (apply method args)
                      (error "No USUAL method" message 'banana-holder)))))
             ((eq? message 'name) (lambda () name))
             ((eq? message 'bananas) (lambda () bananas))
             ((eq? message 'list-of-banana-holders)
              (lambda () list-of-banana-holders))
             ((eq? message 'get-more-bananas)
              (lambda (amount) (set! bananas (+ bananas amount))))
             (else                                      ;; Else clause:
              (let ((method (get-method
                             'banana-holder
                             message
                             my-person
                             my-fruit-lover)))
                (if (method? method)                    ;; Try delegating...
                    method
                    (lambda args 'sorry))))))           ;; default-method

          dispatch)))                                   ;; Class' instantiate
                                                        ;; proc returns object
     (else (error "Bad message to class" class-message))))))
```