

Object-Oriented Programming

Records, Revisited

As you may recall, one of the key issues in the design of records is that the record designer have some control over the use of records. In particular, the designer might want to require that some fields be fixed and allow others to be mutable. The designer may also want to limit the legal values of some fields.

As we saw in the corresponding lab, it is relatively easy for someone other than the designer/implementer of the record to modify the record in “inappropriate” ways. For example, suppose that someone has written a student record type and someone else has written a related set of utilities. We might hope that the following would only behave correctly:

```
(load "student.ss")
(load "sams-student-stuff.ss") ; includes compute-gpa

(define report-gpa
  (lambda (student)
    (if (not (student? student))
        (error "Bozo, that's not a student")
        (display (compute-gpa student)))))
```

If we haven't looked at the code for `compute-gpa`, we have no guarantee as to whether or not the student record is still correct afterwards. (Other than crossing our fingers.) The `compute-gpa` procedure may have gone behind the scenes and changed a field.

It will also fairly clear to anyone who uses our records that we've chosen to implement records with vectors. They can determine this fact by printing out a record that they've created. Hence, at a more detailed level, there's nothing to stop someone from changing a fixed field of a record by using `vector-set!`.

We'd like to *encapsulate* our implementation so that we can hide how our records are implemented and restrict how they're used.

Objects: Records that Protect Their Fields

One of the basic ideas of the programming paradigm called *object-oriented programming* is to encapsulate the data so as to intercept low-level interventions and treat them as errors. An *object* is a data structure that permits access to and modification of its elements only through a fixed set of procedures -- the object's *methods*. One cannot “peek inside” an object; one is limited to the procedures provided.

To request the execution of one of these methods, one *sends* the object a *message* that names the desired method, providing any additional parameters that the object will need as part of the message. Attempting to send an object a message that does not name one of its methods simply causes an error. The custom is to precede the message names with colons.

Objects in Scheme

The Scheme standard does not include objects. However, you can implement an object as a procedure that takes messages as parameters and inspects them before acting on them. Since Scheme typically does not allow one to look inside procedures, procedures provide an appropriate form of encapsulation.

How do we store data for use within the procedure? We can use vectors to build the storage locations that are protected by the procedure.

Here's a simple example -- an object named `sample-box` that contains only one field, `contents`, and responds to only one message, `' :show-contents`.

```
;;; Value
;;; sample-box
;;; Type:
;;; object
;;; Purpose:
;;; To provide a sample "box"; something whose value you
;;; can look at but not change.
;;; Valid Messages:
;;; :show-contents
;;; Get the contents of the box.
(define sample-box
  (let ((contents (vector 42)))
    (lambda (message)
      (if (eq? message ' :show-contents)
          (vector-ref contents 0)
          (error "sample-box: unrecognized message")))))
```

That is,

- Build a new symbol table with the `let` that contains one name-to-value mapping (that is, it maps `contents` to a one-element vector that contains 42).
- Build and return a procedure that takes a message as a parameter. Since the `lambda` falls within the `let`, it has access to that new symbol table and *nothing else has direct access*.

We can test our sample object by trying to set the contents to 0.

```
> (sample-box ' :show-contents)
42
> (sample-box ' :set-contents-to-zero!)
sample-box: unrecognized message
> (sample-box ' :set-contents 0)
sample-box: unrecognized message
> (set! (sample-box ' :show-contents) 0)
set!: not an identifier at: (sample-box (quote :show-contents)) ...
> (set! contents 0)
set!: cannot set undefined identifier: contents
> (sample-box ' :show-contents)
42
```

All the attempts to modify the contents field of `sample-box` fail. Sending it the message `' :set-contents-to-zero!` doesn't work, because the procedure is not set up to receive such a message. And you can't reach the actual `contents` variable from outside the `sample-box` procedure because that identifier is bound to the storage location that contains 42 *only inside the body of the let-expression*.

Changing Object Values

One could revise the procedure so that it would accept the message `' :set-contents-to-zero!`:

```
;;; Value
;;; zeroable-box
;;; Type:
;;; object
;;; Purpose:
;;; To provide a sample "box"; something whose value you
;;; can look at and change to 0
;;; Valid Messages:
;;; :show-contents
;;; Get the contents of the box.
;;; :set-to-zero!
;;; Set the contents of the box to 0.
(define zeroable-box
  (let ((contents (vector 57)))
    (lambda (message)
      (cond ((eq? message ' :show-contents)
             (vector-ref contents 0))
            ((eq? message ' :set-contents-to-zero!)
             (vector-set! contents 0 0))
            (else (error "zeroable-box: unrecognized message"))))))

> (zeroable-box ' :show-contents)
57
> (zeroable-box ' :set-contents-to-zero!)
> (zeroable-box ' :show-contents)
0
```

Of course, there is no way for anyone to set the contents of this particular object to anything *except* zero. Now that the box has been zeroed its contents will remain zero forever.

Making Several Objects of the Same Type

In the preceding examples, we have created only one object of each type, but it is not difficult to write a higher-order constructor procedure that can be called repeatedly, to build and return any number of objects of a given type. Suppose, for example, that we want to build several *switches*, each of which is an object with one field (a Boolean value) and responding to only two messages: `' :show-position`, which returns `' on` if the field contains `#t` and `' off` if it contains `#f`, and `' :toggle!`, which changes the field from `#t` to `#f` or from `#f` to `#t`. Here's a constructor for switches:

```

;;; Procedure:
;;;   make-switch
;;; Parameters:
;;;   None
;;; Purpose:
;;;   Creates a new switch in the off position.
;;; Produces:
;;;   newswitch, a switch
;;; Preconditions:
;;;   None
;;; Postconditions:
;;;   newswitch is an object which responds to two messages:
;;;     :show-position
;;;       Shows the current position ('on or 'off)
;;;     :toggle!
;;;       Switches the current position
(define make-switch
  (lambda ()
    (let ((state (vector #f))) ; All switches are off when manufactured.
      (lambda (message)
        (cond ((eq? message ':show-position)
              (if (vector-ref state 0) 'on 'off))
              ((eq? message ':toggle!)
               (vector-set! state 0 (not (vector-ref state 0))))
              (else (error "switch: unrecognized message"))))))))

```

Now let's manufacture a couple of switches and show that they can be toggled independently:

Because the `make-switch` procedure enters the `let`-expression to create a new binding each time it is invoked, each switch that is returned by `make-switch` gets a separate static variable to put its state in. This static variable retains its contents unchanged even between calls to the object and independently of calls to any other object of the same type.

Methods with Additional Parameters

In all of the preceding examples, the messages received by the object have not included any additional parameters. Suppose that we want to define an object similar to `sample-box` except that one can replace the value in the `contents` field with any integer that is larger than the one that it currently contains, by giving it the message `':replace-with` and including the new, larger value. We can accommodate such messages by making the object a procedure of variable arity, requiring at least one argument (the name of the method to be applied) but allowing for more:

```

;;; Procedure:
;;;   make-growing-box
;;; Parameters:
;;;   None
;;; Purpose:
;;;   Creates a new box whose values you can change to larger values.
;;; Produces:
;;;   newbox, a box whose contents can change to larger values.
;;; Preconditions:
;;;   None
;;; Postconditions:

```

```

;;; newbox is an object which responds to two messages:
;;;   :show-contents
;;;     Get the contents of the box.
;;;   :replace-with! val
;;;     Set the contents of the box to val, provided val
;;;     is larger than the current contents of the box.
(define make-growing-box
  (lambda ()
    ; Build a new symbol table that contains the one value
    ; accessed by the object.
    (let ((contents (vector 0)))
      ; Respond to messages with additional parameters
      (lambda (message . parameters)
        (cond
          ; [:show-contents]
          ; Show the current contents of the box
          ((eq? message ':show-contents)
           (vector-ref contents 0))
          ; [:replace-with! val]
          ; Replace the contents of the box with val
          ((eq? message ':replace-with!)
           (cond
            ; We need at least one parameter
            ((null? parameters)
             (error "growing-box:replace-with!: requires an argument"))
            ; But no more than one
            ((not (null? (cdr parameters)))
             (error "growing-box:replace-with!: only one argument allowed"))
            (else
             (let ((new-contents (car parameters)))
               (cond
                ; That parameter needs to be an integer
                ((not (integer? new-contents))
                 (error "growing-box:replace-with: "
                        "the argument must be an integer"))
                ; Precondition: The new value must be larger
                ((<= new-contents (vector-ref contents 0))
                 (error "growing-box:replace-with: "
                        "the argument must exceed the current contents"))
                (else (vector-set! contents 0 new-contents)))))))
          ; [OTHER MESSAGE]
          ; No other messages are allowed
          (else (error
                 (string-append
                  "growing-box: unrecognized message "
                  (symbol->string message))))))))))

```

```

> (define growable (make-growing-box))
> box
<procedure>
> (growable ':show-contents)
0
> (growable ':replace-with! 5)
> (growable ':show-contents)
5
> (growable ':replace-with! 3)
growable:replace-with: the argument must exceed the current contents

```

```
> (growable ':show-contents)
5
> (growable ':replace-with! 'foo)
growable:replace-with: the argument must be an integer
> (growable ':replace-with!)
growable:replace-with: an argument is required
> (growable ':show-contents)
5
> (growable ':replace-with! 7)
> (growable ':show-contents)
7
```