Fundamentals of Computer Science I (CS151.02 2007S)

# Objects in Scheme

**Summary:** We consider a mechanism for grouping information together, but limiting access to the internal representation. We refer to the values build in this way as *objects*, although the resultant values lack some of the key aspects of the objects used in traditional object-oriented programming.

**Contents:**



# Representing Compound Information

In our exploration of Scheme, we've seen a number of *data structures* that allow us to organize data. A list is a dynamic data structure with a variable number of components. A vector is a data structure with a fixed number of components, each of which you can quickly access by number.

As you've seen, we often build data structures for a particular kind of data. For example, we might choose to represent the information for a student with ten components:

- 0: a string for the student's last name;
- 1: a string for the student's first name;
- 2: a string for student's student id number;
- 3: a string for the student's major;
- 4: an integer for the students' graduation year;
- 5: a list of symbols (`'check`, `'minus`, `'check-plus`, etc.) for homework grades;
- 6: a list of real numbers for exam grades;
- 7: a list of strings (`"A"`, `"B-"`, etc.) for project grades;
- 8: a single real number for participation grade; and
- 9: a list of strings for extra credit events.

For example, we might store those components in a vector, with the last name at position 0, the first name at position 1, and so on and so forth. One strategy is to then tell other programmers about this representation, and let them rely on the structure (e.g., someone who wants to change the last name would do a `vector-set!` for position 0).

However, if we change the representation, other programmers must then change their code. Hence, it is often better to provide procedures to give other programmers access to our structure. If they use only our procedures, then we can change the representation freely (as long as we change our procedures correspondingly) and their code will continue to work.

Here are some sample procedures for a student data type.

```
(define make-student
  (lambda (last-name first-name id major graduation-year)
    (cond
      ((not (string? last-name))
       (error "make-student: parameter 0 (last-name) must be a string"))
      ((not (string? first-name))
       (error "make-student: parameter 1 (first-name) must be a string"))
      ((not (string? id))
       (error "make-student: parameter 3 (id) must be a string"))
      ((not (string? major))
       (error "make-student: parameter 4 (major) must be a string"))
      ((not (integer? graduation-year))
       (error "make-student: parameter 5 (graduation-year) must be an integer"))
      ((or (< graduation-year 2000) (> graduation-year 2010))
       (error "make-student: parameter 6 (graduation-year) must be in the range [2000 ... 2010]"))
      (else
       (vector last-name first-name id major graduation-year
               null null null 90 null)))))

(define get-student-last-name
  (lambda (student)
    (vector-ref student 0)))

...

(define get-student-ec
  (lambda (student)
    (vector-ref student 9)))

...

(define set-student-name!
  (lambda (student first-name last-name)
    (cond
      ((not (string? first-name))
       (error "set-student-name: parameter 1 (last-name) must be a string"))
      ((not (string? first-name))
       (error "set-student-name: parameter 2 (first-name) must be a string"))
      (else
       (vector-set! student 0 first-name)
       (vector-set! student 1 last-name)))))

...

(define add-student-ec!
  (lambda (student event)
    (if (not (string? event))
        (error "add-student-ec: parameter must be a string")
        (vector-set student 9
                    (append (vector-ref student 9)
                            (list event))))))

...
```

```
(define count-student-ec
  (lambda (student)
    (length (vector-ref student 9))))
```

You can find the complete set of code in `student.scm`.

Here's a sample session:

```
> (define sam (make-student "Rebelsky" "Samuel" "00000032"
                            "Computer Science" 2010))
> (display-student sam)
Record for Rebelsky, Samuel [ID: 00000032]
Major: Computer Science; Class: 2010
No Homework.
No Exams.
No Projects.
Participation: 90
No Extra Credit.
> (add-student-homework! sam 'minus)
> (add-student-homework! sam 'check)
> (add-student-homework! sam 'minus)
> (add-student-exam! sam 90)
> (add-student-exam! sam 80) ; slump!
> (add-student-exam! sam 100) ; recovered!
> (add-student-exam! sam 0) ; skipped the final
> (add-student-ec! sam 'trogdor)
add-student-ec: parameter must be a string
> (add-student-ec! sam "trogdor")
> (add-student-ec! sam "anti-train concert")
> (count-student-ec sam)
2
> (add-student-project sam "B")
reference to undefined identifier: add-student-project
> (add-student-project! sam "B")
> (add-student-project! sam "C")
> (display-student sam)
Record for Rebelsky, Samuel [ID: 00000032]
Major: Computer Science; Class: 2010
Homework: minus, check, minus
Exams: 90, 80, 100, 0
Projects: B, C
Participation: 90
Extra Credit: trogdor, anti-train concert
```

If we did not want to permit other programmers to change particular parts (e.g., we don't generally want student-id numbers to change) or wanted to limit the kinds of possible access (e.g., you can add forms of extra credit, but not remove them), we would not provide procedures that gave that extra access.

```
> (set-student-id sam "bozo")
reference to undefined identifier: set-student-id
```

# Problems With This Technique

This technique (of providing both representation and procedures that use that representation) has many advantages, as suggested above. For many, the most compelling advantage is that *client code* (procedures that use our student representation) need not change when the representation changes.

Another possible advantage is that we prevent naive programmers from doing inappropriate things to our structure (e.g., storing a symbol in the list of events). Unfortunately, we haven't really prevented such inappropriate behavior, since clients can still determine the representation we use and then modify things directly.

```
> (define sam (make-student "Rebelsky" "Samuel"
                            "00000032" "Computer Science"
                            2010))
> (count-student-ec sam)
0
> (add-student-ec sam 'trogdor)
add-student-ec: parameter must be a string
> sam
#10("Rebelsky" "Samuel" "00000032" "Computer Science" 2010 null null null 90 null)
> (vector-set! sam 9 'trogdor)
> (get-student-ec sam)
trogdor
> (count-student-ec sam)
length: expects argument of type <proper list>; given trogdor
```

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

# Objects: Representations that Protect Their Contents

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 these attempts to modify the contents field of `sample-box` fail, as will all attempts. 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.*

In fact, we can't even see that vector (as we could with students)

```
> sample-box
#<procedure:sample-box>
```

# 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"))))))
```

Here's a simple interaction with the box.

```
> (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. If we want the box to change, we might add an `':increment!` message.

```
;;; Value
;;;   another-box
;;; Type:
;;;   object
;;; Purpose:
;;;   To provide a sample "box"; something whose value you
;;;   can look at, set to 0, and increment
;;; Valid Messages:
;;;   :show-contents
;;;     Get the contents of the box.
;;;   :set-to-zero!
;;;     Set the contents of the box to 0.
```

```
;;;   :increment!
;;;     Add 1 to the contents of the box.
(define another-box
  (let ((contents (vector 0)))
    (lambda (message)
      (cond ((eq? message ':show-contents)
             (vector-ref contents 0))
            ((eq? message ':set-contents-to-zero!)
             (vector-set! contents 0 0))
            ((eq? message ':increment!)
             (vector-set! contents 0 (+ (vector-ref contents 0) 1)))
            (else (error "zeroable-box: unrecognized message"))))))
```

Our interactions with this box are similar.

```
> (another-box ':show-contents)
0
> (another-box ':increment!)
> (another-box ':increment!)
> (another-box ':increment!)
> (another-box ':show-contents)
3
> (another-box ':set-contents-to-zero!)
> (another-box ':show-contents)
0
> (another-box ':increment!)
> (another-box ':show-contents)
1
```

What if we want to include a value with a message, such as when we want to change the boxed value to a particular new value? We'll see later in this reading.

# 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`.

We might start by building a single switch.

```
(define switch
  (let ((state (vector #f)))
    (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"))))))
```

However, when we want more than one, we need a procedure that builds switches. Hence, we need to write a procedure that returns something like the previous object. We call something that returns objects a *constructor*. 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 ’:type)
               ’switch)
              ((eq? message ’:->string)
               (string-append "#<switch>("
                              (if (vector-ref state 0) "on" "off")
                              ")"))
              ((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")))))))
```

The ordering of lambdas and lets is important. 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 `state` 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.

You'll note that we've added support for two other messages, `’:type` and `’:->string` message. It is good practice to regularly include those two methods.

```
> (define overhead-lights (make-switch))
> (define board-lights (make-switch))
> (overhead-lights ’:show-position)
off
> (board-lights ’:show-position)
off
> (board-lights ’:toggle!)
> (board-lights ’:show-position)
on
> (overhead-lights ’:show-position)
```

```
off
> (overhead-lights ':toggle!)
> (overhead-lights ':->string)
"#<switch>(on)"
> (overhead-lights ':type)
switch
```

# Methods with 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 vector that contains the one value
    ; accessed by the object.
    (let ((contents (vector 0)))
      ; Respond to messages with additional parameters
      (lambda (message . parameters)
        (cond
          ; [type]
          ;   Get the type
          ((eq? message ':type)
           'growing-box)
          ; [:->string]
          ;   Convert to a sting (typically for output)
          ((eq? message ':->string)
           (string-append "#<growing-box>("
                          (number->string (vector-ref contents 0))
                          ")"))
          ; [: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)
growing-box:replace-with: the argument must exceed the current contents
> (growable ':show-contents)
5
> (growable ':replace-with! 'foo)
growing-box:replace-with: the argument must be an integer
> (growable ':replace-with!)
growing-box:replace-with: an argument is required
> (growable ':show-contents)
5
> (growable ':replace-with! 7)
> (growable ':->string)
"#<growing-box>(7)"
```

# Objects with Multiple Fields

All the objects that we've seen so far have stored only one value. However, since we use a vector to keep track of the value, we can certainly store more than one value in the vector. For example, suppose we want something that keeps track of the number of times we get the answers "yes" and "no". We'll use position 0 to keep track of the yes answers and position 1 to keep track of the no answers.

```
(define make-yesno
  (lambda ()
    (let ((counts (vector 0 0)))
      (lambda (message)
        (cond
          ; [:type]
          ((eq? message ':type)
           'yesno)
          ; [:->string]
          ;  Convert to a string
          ((eq? message ':->string)
           (string-append "<yesno>("
                          "yes:" (number->string (vector-ref counts 0))
                          ", no:" (number->string (vector-ref counts 1))
                          ")"))
          ; [:yes!]
          ;  Increment the number of yes responses.
          ((eq? message ':yes!)
           (vector-set! counts 0 (+ 1 (vector-ref counts 0))))
          ; [:no!]
          ;  Increment the number of no responses.
          ((eq? message ':no!)
           (vector-set! counts 1 (+ 1 (vector-ref counts 1))))
          ; [:report!]
          ;  Print a report of responses.
          ((eq? message 'report!)
           (display "Yes appeared ")
           (display (vector-ref counts 0))
           (display " times.")
           (newline)
           (display "No appeared ")
           (display (vector-ref counts 1))
           (display " times.")
           (newline))
          (else
           (error "#<yesno>: unrecognized message")))))))
```

For example,

```
> (define yn (make-yesno))
> (yn ':yes!)
> (yn ':report!)
Yes appeared 1 times.
No appeared 0 times.
> (yn ':no!)
> (yn ':no!)
> (yn ':no!)
> (yn ':report!)
```

```
Yes appeared 1 times.
No appeared 3 times.
> (yn ':no!)
> (yn ':no!)
```

# Student Objects

At the beginning of this reading, we considered one mechanism for representing students: a vector with associated procedures. Now that you've seen how to build objects, we can consider how to represent students as objects. However, that is a subject for another day.

---