

## Class 49: Object Basics

**Held:** Monday, April 30, 2007

**Summary:** Today we consider how to make *objects* that encapsulate values and provide capabilities for manipulating those values.

### Related Pages:

- EBoard.
- Reading: Modeling Objects in Scheme.

### Notes:

- EC for the Alumna Scholar talk tonight.
- Are there questions on the exam?
- We'll start class by talking about homework 16.
- For tomorrow, you may want to review the reading for today.

### Overview:

- Representing Compound Values.
- Introduction to Objects.
- Procedures as Objects.
- Adding State.

## Extending Records

- As we've seen in our experiments with representing compound values, there are strengths and weaknesses to simply choosing a representation and writing procedures to work with that implementation.
- Strengths: Access parts by procedure; Relatively easy to use.
- Weaknesses: Not fully encapsulated; hard to separate core operations from external operations (since they're called the same way); hard to limit access.
- In the late 1960's, some computer scientists decided to extend the idea of representing data into something they call an *object*
- Objects group data.
- Objects can also do things.
- You can't directly access the parts of an object.
- Rather, you ask the object to do things or tell you things.
- The requests you send to objects are called *messages*.
- Traditional objects also provide a number of other advantages. We'll focus on encapsulation.

## Objects in Scheme

- Scheme doesn't include objects as a built-in type. Hence, we have to implement them ourselves.
- The trick that we recommend is that you implement objects as procedures that take a *message* as a parameter.
- Traditionally, the messages begin with a colon.
- Here's a simple object that will respond when you greet it or leave it.

```
(define greeter
  (lambda (message)
    (cond
      ((eq? message ':enter) (display "Hello") (newline))
      ((eq? message ':leave) (display "Goodbye") (newline))
      (else (error "Unknown Message")))))
```

- Here's how we use it

```
> (greeter ':enter)
Hello
> (greeter ':leave)
Goodbye
> (greeter ':sleep)
Unknown Message
```

## Adding State

- But how do we have an object keep track of information about itself?
- We build a local symbol table that is only accessible to the procedure.
- We can build such a table by putting a `let` *outside* the `lambda` for the procedure.

```
(define fixed-value
  (let ((value 5))
    (lambda (message)
      (cond
        ((eq? message ':get) value)
        (else (error "fixed-value:" "unknown message"))))))
```

- Typically, we use vectors to encapsulate our state because we know how to mutate vectors.

```
(define incrementable-value
  (let ((value (vector 0)))
    (lambda (message)
      (cond
        ((eq? message ':get) (vector-ref value 0))
        ((eq? message ':add!)
         (vector-set! value 0
                      (+ 1 (vector-ref value 0))))
        (else (error "fixed-value:" "unknown message"))))))
```

- And an example of its use

```
> (incrementable-value ':get)
0
> (incrementable-value ':add!)
> (incrementable-value ':get)
1
```

## Lab

- If there is time (unlikely), you can begin the lab on object-oriented programming.
- 

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.