Chapter 10

Objects

> *By the word operation, we mean any process which alters the mutual relation of two or more things, be this relation of what kind it may. This is the most general definition, and would include all subjects in the universe. Again, it might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine... Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.*
>
> Ada Byron, Countess of Lovelace (around 1843)

So far, we have seen two different approaches for using computing to solve problems:

- Functional programming (introduced in Chapter 4) — to solve a complex problem, break it into a group of simpler procedures and find a way to compose those procedures to solve the problem.
- Imperative (data-centric) programming (introduced in Chapter 5, and extended with state mutation in the previous chapter) — to solve a complex problem, think about how to represent the data the problem involves, and develop procedures to manipulate that data.

In fact, all computational problems involve both data and procedures. All proce-

dures act on some form of data; without data they can have no meaningful inputs and outputs. Any data-based design must involve some procedures to manipulate that data; otherwise, we can only represent static values, and cannot perform any computation. In this chapter, we will overcome a weakness of previous approaches, namely, that the data and the procedures that manipulate it are separate. Packaging procedures and data together leads to a new approach to problem solving, known as *object-oriented* programming.

Unlike many modern languages, Scheme[1] provides no built-in support for objects. Instead, we create an object system ourselves using simpler expressions (primarily the procedure-making lambda expressions). By building an object system from simple components, we provide a clearer and deeper understanding of how other object systems work.

10.1 Packaging Procedures and State

Recall our counter from Section 9.1.2. The `update-counter!` procedure increments the value of the `counter` variable, which is stored in the global environment, and evaluates to the resulting counter value. Every time an application of `update-counter!` is evaluated, we expect to obtain a value one larger than the previous application. This only works, however, if there are no other evaluations that modify the `counter` variable. Using this implementation, we can only have one counter: there is only one `counter` place in the global environment. If we want to have a second counter, we would need to define a new variable (such as `counter2`, and implement a new procedure, `update-counter2!`, that is identical to `update-counter!`, but manipulates `counter2` instead. For each new counter, we would need a new variable and a new procedure. This is possible, but clearly unsatisfactory.

Instead, what would be more useful is if we could package the counter variable with the procedure that manipulates it. This would create a counter object, and we could create as many as we want, each with its own counter variable to manipulate. The new application rule (from Section 9.1.4) provides a way to do this. Evaluating an application creates a new environment, so if the counter variable is

[1] This means the standard Scheme language, not the extended Scheme languages provided by DrScheme. The MzScheme language does provide additional constructs for supporting objects.

defined in the new environment it will be visible only to the procedure body. The `make-counter` procedure below creates a counter object.

```
(define (make-counter)
  ((lambda (count)
     (lambda ()
       (set! count (+ 1 count))
       count))
   0))
```

This is equivalent to the definition below, which uses a let expression to make the initialization of `count` clearer:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ 1 count))
      count)))
```

Figure 10.1 depicts the environment after evaluating:

```
(define counter1 (make-counter))
(define counter2 (make-counter))
(counter1)
```

The procedure for manipulating the counter state is now packaged with the state it manipulates (the `count` variable in the let expression). Each application of `make-counter` creates a new frame containing its own `count` place. This is known as *encapsulation*. The `count` place is encapsulated with the counter object. Whereas the previous counter used the global environment to store the counter in a way that could be manipulated by other expressions, this version encapsulates the counter variable with the counter object in a way that the only way to manipulate the counter value is to use the counter object. The evaluation of `(counter1)` increments the value in the `count` place associated with the `counter1` procedure's environment.

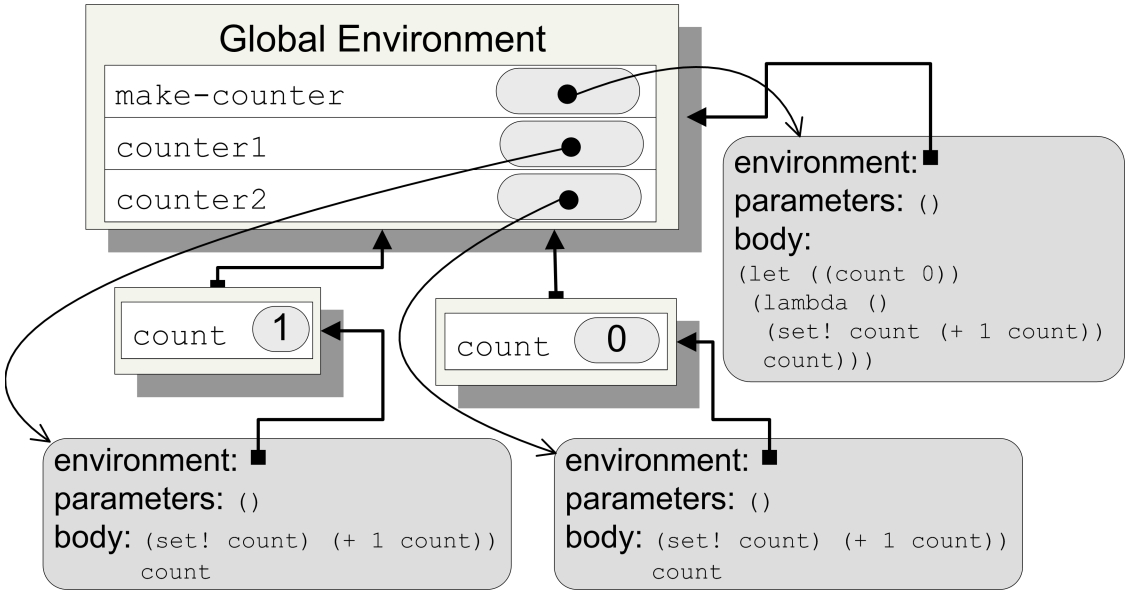

Figure 10.1: Packaging procedures and state.

10.1.1 Messages

The counter object is limited to only one behavior — the object is a procedure that takes no parameters, and every time it is applied the associated count variable is increased by one and the new value is output. To produce more interesting objects, we need a way to combine state with more than one procedure. For example, we might want a counter that can perform several behaviors including resetting the count, incrementing the count, and providing the current count.

We can do this by modifying our `make-counter` procedure to produce a procedure that takes one parameter. The input parameter to the resulting procedure is a message used to select one of the behaviors.

```
(define (make-counter)
  (let ((count 0))
    (lambda (message)
      (cond ((eq? message 'reset!)
             (set! count 0))
```

```
        ((eq? message 'next!)
         (set! count (+ 1 count)))
        ((eq? message 'current)
         count)
        (else (error "Unrecognized message"))))))
```

As with the earlier `make-counter`, the new `make-counter` procedure produces a procedure with an environment that contains a frame containing a place named `count` initialized to 0. The procedure takes a `message` parameter, and its body is a conditional expression that produces a different behavior depending on the input message. The input is a symbol, a sequence of characters preceded by a quote, `'`. Two symbols are equal (as determined by the `eq?` procedure) if the sequence of characters after the quote is identical. Symbols are a more convenient and efficient way of selecting the object behavior than strings would be, and easier to remember and understand than using numbers.

Here are some sample interactions using the object produced by `make-counter`:

```
> (define counter (make-counter))
> (counter 'current)
0
> (counter 'next!)
> (counter 'next!)
> (counter 'current)
2
> (counter 'reset!)
> (counter 'current)
0
> (counter 'previous!)
♯ Unrecognized message
```

A more natural way of interacting with objects is to define a generic procedure that takes an object and a message as its parameters, and send the message to the object. The `ask` procedure defined below is a simple procedure for doing this; later in this chapter, we will develop more complex versions of the `ask` procedure that allow us to define a more powerful object model.

```
(define (ask object message)
```

```
      (object message))
```

Our `ask` procedure simply applies the `object` input to the `message` input. Using the `ask` procedure, the interactions above could be expressed as:

```
> (define counter (make-counter))
> (ask counter 'current)
0
> (ask counter 'next!)
> (ask counter 'next!)
> (ask counter 'current)
2
> (ask counter 'reset!)
> (ask counter 'current)
0
> (ask counter 'previous!)
♯ Unrecognized message
```

It is useful to sometime have behaviors that take additional parameters. For example, we may want to support a message `adjust!` that can add an input value to the counter. To suppose such behaviors, we generalize the behaviors so that the result of matching behavior is a procedure. The procedures for `reset!`, `next!`, and `current` take no parameters; the procedure for `adjust!` takes one parameter.

```
(define (make-counter)
  (let ((count 0))
    (lambda (message)
      (cond ((eq? message 'reset!)
             (lambda () (set! count 0)))
            ((eq? message 'next!)
             (lambda () (set! count (+ 1 count))))
            ((eq? message 'current)
             (lambda () count))
```

```
        ((eq? message 'adjust!)
         (lambda (val) (set! count (+ count val))))
        (else (error "Unrecognized message"))))))
```

To make this work, we need to also change the `ask` procedure to pass in the extra arguments. So far, all the procedures we have seen take a fixed number of operands. To allow `ask` to work for procedures that take a variable number of arguments, we use a new define construct. If we precede last parameter in a parameter list with a `.`, it means that all the rest of the operands will be combined into a list, and the value of that list will be bound to the last parameter name. To apply the procedure we use the `apply` primitive procedure which takes two inputs, a procedure and an operand list. It applies the procedure to the operands, extracting them from the list as each operand in order.

```
(define (ask object message . args)
  (apply (object message) args))
```

We can use the new `ask` procedure to invoke methods with any number of arguments:

```
> (define counter (make-counter))
> (ask counter 'current)
0
> (ask counter 'adjust!  5)
> (ask counter 'current)
5
> (ask counter 'next!)
> (ask counter 'current)
6
```

10.1.2 Object Terminology

An *object* is an entity that packages state and procedures. We call the state that is part of an object its *instance variables*, and the procedures that are part of an object its *methods*. Methods may give information about the state of an object (we call these *observers* or modify the state of an object (we call these *mutators*. Our

counter object provides four methods: `reset!` (a mutator), `next!` (a mutator), `adjust!` (a mutator), and `current` (an observer).

We also need procedures for creating new objects, such as the `make-counter` procedure above. We call these procedures *constructors*. Once an object is created, it is only manipulated by using the object's methods. We *invoke* a method of an object by sending the object a message. This is analogous to applying a procedure.

A *class* is a kind of object. By convention, we call the constructor for a *class*, `make-`*`class`*. Hence, an object of the `counter` class is the result produced when the `make-counter` procedure is applied.

Exercise 10.1. Modify the `make-counter` definition to add a `previous!` method that decrements the counter value by one. ♢

Exercise 10.2. In Section 8.1.4 we developed a tree representation using cons pairs. This approach is simple and was adequate for our purposes (implementing a sorting procedure), but provides no encapsulation. For this exercise, you are to develop an object for representing trees.

a. Define a `make-tree` constructor that creates a `tree` object. The constructor should take three parameters, the `left`, `element`, and `right`, where the `element` is a value, and `left` and `right` are tree objects. Your tree class should have methods for getting the element of a tree node, getting the left child, and getting the right child.

b. Rewrite the `insert-sort-tree` procedure from Section 8.1.4 to use your new tree object instead of the earlier tree representation that used cons pairs. You will have to think carefully about how to represent the null tree.

c. Discuss the differences between the two implementations. What are the advantages and disadvantages of using the tree object?

d. Make your tree object mutable by defining the methods for setting the left subtree, right subtree, and elements (i.e., `set-left!`, `set-right!`, and `set-element!`. Can these be used to improve your `insert-sort-tree` implementation?

♢

10.2 Inheritance

One of the appeals of objects is they can be used to represent objects in a real or imagined world that a program is modeling. Objects are particularly well-suited to programs that such as graphical user interfaces (modeling windows, files, and folders on a desktop), simulations (modeling physical objects in the real world and their interactions), and games (modeling creatures and things in an imagined world).

Objects in the real world, or a simulated world, are complex. Suppose we are implementing a game that simulates a typical university. It will involve many different kinds of objects including places (which are stationary and may contain other objects), things, and people. There are many different kinds of people, such as students and professors. All objects in our game have a name and a location; some objects have methods for talking and moving. We could define classes independently for all of the object types, but this would involve a lot of duplicate effort. It would also make it hard to add a new behavior to all of the objects in the game without modifying many different procedures.

The solution is to define more specialized kinds of objects using the definitions of other objects. For example, a `student` is a kind of `person`, which is a kind of `movable-object`, which is a kind of `sim-object` (simulation object). To implement the `student` class, we would like to use methods from the `person` class without needing to duplicate them in the `student` implementation. We call the more specialized class (in this case the `student` class) the *subclass*, and the more general class (in this case the `person` class) the *superclass*. That is to say, `student` is a subclass of `person`, and `person` is a superclass of `student`. When one class implementation uses the methods from another class we say the subclass *inherits* from the superclass.

Figure 10.2 illustrates some possible inheritance relationships for a university simulator. The arrows point from subclasses to their superclass. Note that a class may be both a subclass to another class, and a superclass to a different class. For example, `person` is a subclass of `movable-object`, but a superclass of `student` and `professor`. The inheritance relationships can continue up the tree. The superclass of `movable-object` is `sim-object`.

Our goal is to be able to reuse superclass methods in subclasses. When a method is invoked in a subclass, if the subclass does not provide a definition of the method,

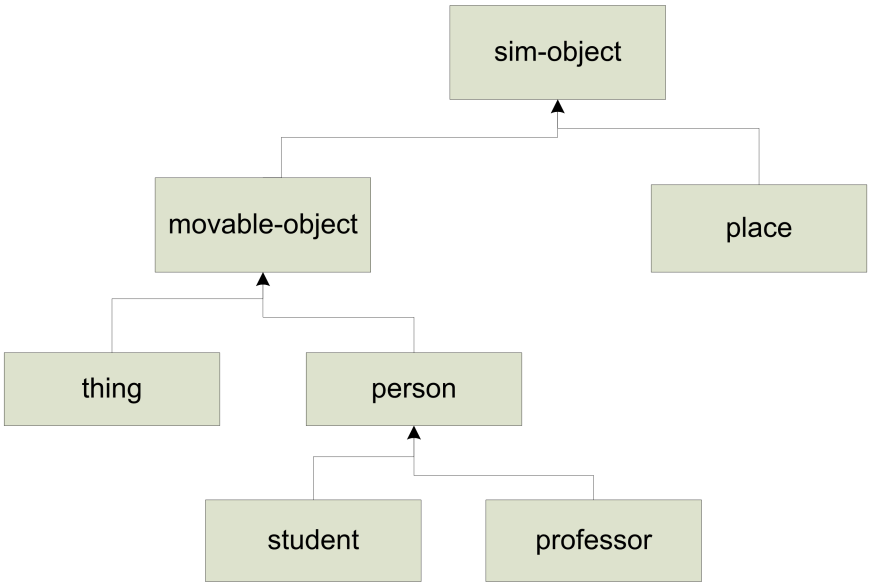

Figure 10.2: Inheritance Hierarchy.

then the definition of the method in the superclass is used. This can continue up the inheritance chain. For instance, `student` inherits from `person`, which inherits from `movable-object`, which inherits from `sim-object`. This means if `sim-object` provides a `get-name` method, when the `get-name` method is invoked of a `student` object, the implementation of `get-name` in the `sim-object` class will be used (as long as none of the other classes replace the method). If a subclass replaces a method defined by its superclass, then the subclass method *overrides* the superclass method. When the method is invoked on a subclass object, the new method will be used.

To implement inheritance we need to change class definitions so that if a requested method is not defined by the subclass, the superclass method will be used. The `make-subobject` procedure does this. It takes two inputs: the superclass object and the implementation of the subclass. It produces a new subclass object. When the resulting object is applied to a message, it will first apply the subclass implementation to the message to find an appropriate method if one is defined. If no method is defined by the subclass implementation, it produces the result of applying the superclass object to the message. Sometimes it will be useful to obtain the superclass object. The `make-subobject` defines the `super` method for

doing this. Every subclass object created using `make-subobject` will have a method `super` defined that produces the superclass object.

```
(define (make-subobject super imp)
  (lambda (message)
    (if (eq? message 'super)
        (lambda (self) super)
        (let ((method (imp message)))
          (if method
              method
              (super message))))))
```

It is useful to add an extra parameter to all methods so the object on which the method was invoked is visible. We call this the `self` object (in some languages it is called the *this* object instead). To support this, the `ask` procedure is modified to pass in the object parameter to the method:

```
(define (ask object message . args)
  (apply (object message) object args))
```

All methods now take the `self` object as their first parameter, and may take additional parameters. For instance, we define the `counter` constructor procedure:

```
(define (make-counter)
  (let ((count 0))
    (lambda (message)
      (cond ((eq? message 'reset!)
             (lambda (self) (set! count 0)))
            ((eq? message 'next!)
             (lambda (self) (set! count (+ 1 count))))
            ((eq? message 'current)
             (lambda (self) count))
            ((eq? message 'adjust!)
             (lambda (self val) (set! count (+ count val))))
            (else (error "Unrecognized message"))))))
```

We can use `make-subobject` to create an object that inherits the behaviors from one class, and extends those behaviors by defining new methods in the subclass implementation. For example, `make-pcounter` below defines the `pcounter` class which extends the `counter` class with a `previous!` method.

```
(define (make-pcounter)
  (make-subobject
   (make-counter)
   (lambda (message)
     (if (eq? message 'previous!)
         (lambda (self)
           (ask self 'adjust! -1))
         #f))))
```

The `pcounter` object defines a `previous!` method which provides a new behavior. If the message is not `'previous!`, however, it evaluates to `#f`. The `make-subobject` procedure will apply the superclass object to the message when the subclass implementation evaluates to false (that is, when the subclass does not define a method for the input message). Hence, for all other messages, the superclass method is used. Note that within the `previous!` method we use `ask` to invoke the `adjust!` method on the `self` object. Since the subclass implementation does not provide an `adjust!` method, this will result in the superclass method being applied.

Suppose we define a subclass of `pcounter` that is not allowed to have negative counter values. If the counter would reach a negative number, instead of setting the counter to the new value, it produces an error message and maintains the counter at zero. We can do this by *overriding* the `adjust!` method. This replaces the superclass implementation of the method with a new implementation.

```
(define (make-poscounter)
  (make-subobject
   (make-pcounter)
```

```
(lambda (message)
  (if (eq? message 'adjust!)
      (lambda (self val)
        (if (< (+ (ask self 'current) val) 0)
            (error "Negative count")
            (ask (ask self 'super) 'adjust! val)))
      #f))))
```

Now, consider what happens when we evaluate:

```
> (define poscount (make-poscounter))
> (ask poscount 'next!)
> (ask poscount 'previous!)
> (ask poscount 'previous!)
♯ Negative count
> (ask poscount 'current)
0
```

For the first `ask`, the `next!` method is invoked on a `poscounter` object. Since the `poscounter` implementation does not define a `next!` method, the message is sent to the superclass, `pcounter`. The `pcounter` implementation also does not define a `next!` method, so the message is passed up to its superclass, `counter`. This class defines a `next!` method, so that method is used.

For the next `ask`, the `previous!` method is invoked. As before, the `poscounter` implementation does not define a `previous!` method, so the message is sent to the superclass. Here, `pcounter` does define a `previous!` method. Its implementation involves an invocation of the `adjust!` method:

```
(ask self 'adjust!  -1)
```

Note that this invocation is done on the `self` object, which is an instance of the `poscounter` class. Hence, the `adjust!` method is found from the `poscounter` class implementation. This is the method that overrides the `adjust!` method defined by the `counter` class. Hence, the second invocation of `previous!` produces the "Negative count" error and does not adjust the count to -1.

The property this object system has where the method invoked depends on the object is known as *dynamic dispatch*. The method that will be used for an invocation

depends on the `self` object. In this case, for example, it means that when we examine the implementation of the `previous!` method in the `pcounter` class it is not possible to determine what procedure will be applied for the method invocation, `(ask self 'adjust! -1)`. It depends on the actual `self` object: if it is a `poscounter` object, the `adjust!` method defined by `poscounter` is used; if it is a `pcounter` object, the `adjust!` method defined by `counter` (the superclass of `pcounter`) is used.

Dynamic dispatch provides for a great deal of expressiveness. It enables us to use the same code to produce many different behaviors by overriding methods in subclasses. This is very useful, but also very dangerous — it makes it impossible to reason about what a given procedure does, without knowing about all possible subclasses. For example, we cannot make any claims about what the `previous!` method in `pcounter` actually does, without knowing what the `adjust!` method does in all subclasses of `pcounter`.

Exercise 10.3. Define a new subclass of `poscounter` where the increment for each `next!` method application is a parameter to the constructor procedure. For example, `(make-var-counter 0.1)` would produce a counter object whose counter has value 0.1 after one invocation of the `next!` method. ♢

Exercise 10.4. Define a `countdown` class that simulates a rocket launch countdown (that is, it starts at some intial value, and counts down to zero, at which point the rocket is launched). Can you implement `countdown` as a subclass of `counter`? ♢

10.3 Summary

I invented the term "Object-Oriented" and I can tell you I did not have C++ in mind.
Alan Kay

An object is an entity that packages state and procedures that manipulate that state together. By packaging state and procedures together, we can encapsulate state in ways that enable more elegant and robust programs. Inheritance allows an implementation of one class to reuse or override methods in another class, known as its superclass. Programming using objects and inheritance enables a style of problem solving known as object-oriented programming in which we solve problems by modeling a problem instance using objects.