25

Object-Oriented Lisp

This chapter discusses object-oriented programming in Lisp. Common Lisp includes a set of operators for writing object-oriented programs. Collectively they are called the Common Lisp Object System, or CLOS. Here we consider CLOS not just as a way of writing object-oriented programs, but as a Lisp program itself. Seeing CLOS in this light is the key to understanding the relation between Lisp and object-oriented programming.

25.1 Plus ça Change

Object-oriented programming means a change in the way programs are organized. This change is analogous to the one that has taken place in the distribution of processor power. In 1970, a multi-user computer system meant one or two big mainframes connected to a large number of dumb terminals. Now it is more likely to mean a large number of workstations connected to one another by a network. The processing power of the system is now distributed among individual users instead of centralized in one big computer.

Object-oriented programming breaks up traditional programs in much the same way: instead of having a single program which operates on an inert mass of data, the data itself is told how to behave, and the program is implicit in the interactions of these new data "objects."

For example, suppose we want to write a program to find the areas of two-dimensional shapes. One way to do this would be to write a single function which looked at the type of its argument and behaved accordingly:

```
(defun area (x)
  (cond ((rectangle-p x) (* (height x) (width x)))
        ((circle-p x) (* pi (expt (radius x) 2)))))
```

The object-oriented approach is to make each object able to calculate its own area. The `area` function is broken apart and each clause distributed to the appropriate class of object; the `area` method of the `rectangle` class might be

```
#'(lambda (x) (* (height x) (width x)))
```

and for the `circle` class,

```
#'(lambda (x) (* pi (expt (radius x) 2)))
```

In this model, we ask an object what its area is, and it responds according to the method provided for its class.

The arrival of CLOS might seem a sign that Lisp is changing to embrace the object-oriented paradigm. Actually, it would be more accurate to say that Lisp is staying the same to embrace the object-oriented paradigm. But the principles underlying Lisp don't have a name, and object-oriented programming does, so ○ there is a tendency now to describe Lisp as an object-oriented language. It would be closer to the truth to say that Lisp is an extensible language in which constructs for object-oriented programming can easily be written.

Since CLOS comes pre-written, it is not false advertising to describe Lisp as an object-oriented language. However, it would be limiting to see Lisp as merely that. Lisp is an object-oriented language, yes, but not because it has adopted the object-oriented model. Rather, that model turns out to be just one more permutation of the abstractions underlying Lisp. And to prove it we have CLOS, a program written in Lisp, which makes Lisp an object-oriented language.

The aim of this chapter is to bring out the connection between Lisp and object-oriented programming by studying CLOS as an example of an embedded language. This is also a good way to understand CLOS itself: in the end, nothing explains a language feature more effectively than a sketch of its implementation. In Section 7.6, macros were explained this way. The next section gives a similar sketch of how to build object-oriented abstractions on top of Lisp. This program provides a reference point from which to describe CLOS in Sections 25.3–25.6.

25.2 Objects in Plain Lisp

We can mold Lisp into many different kinds of languages. There is a particularly direct mapping between the concepts of object-oriented programming and the fundamental abstractions of Lisp. The size of CLOS tends to obscure this fact. So

before looking at what we can do with CLOS, let's see what we can do with plain Lisp.

Much of what we want from object-oriented programming, we have already in Lisp. We can get the rest with surprisingly little code. In this section, we will define an object system sufficient for many real applications in two pages of code. Object-oriented programming, at a minimum, implies

1. objects which have properties
2. and respond to messages,
3. and which inherit properties and methods from their parents.

In Lisp, there are already several ways to store collections of properties. One way would be to represent objects as hash-tables, and store their properties as entries within them. We then have access to individual properties through `gethash`:

```
(gethash 'color obj)
```

Since functions are data objects, we can store them as properties too. This means that we can also have methods; to invoke a given method of an object is to funcall the property of that name:

```
(funcall (gethash 'move obj) obj 10)
```

We can define a Smalltalk style message-passing syntax upon this idea:

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

so that to tell obj to move 10 we can say

```
(tell obj 'move 10)
```

In fact, the only ingredient plain Lisp lacks is inheritance, and we can provide a rudimentary version of that in six lines of code, by defining a recursive version of `gethash`:

```
(defun rget (obj prop)
  (multiple-value-bind (val win) (gethash prop obj)
    (if win
        (values val win)
        (let ((par (gethash 'parent obj)))
          (and par (rget par prop))))))
```

```
(defun rget (obj prop)
  (some2 #'(lambda (a) (gethash prop a))
         (get-ancestors obj)))

(defun get-ancestors (obj)
  (labels ((getall (x)
             (append (list x)
                     (mapcan #'getall
                             (gethash 'parents x)))))
    (stable-sort (delete-duplicates (getall obj))
                 #'(lambda (x y)
                     (member y (gethash 'parents x))))))

(defun some2 (fn lst)
  (if (atom lst)
      nil
      (multiple-value-bind (val win) (funcall fn (car lst))
        (if (or val win)
            (values val win)
            (some2 fn (cdr lst))))))
```

Figure 25.1: Multiple inheritance.

If we just use `rget` in place of `gethash`, we will get inherited properties and methods. We specify an object's parent thus:

```
(setf (gethash 'parent obj) obj2)
```

So far we have only single inheritance—an object can only have one parent. But we can have multiple inheritance by making the `parent` property a list, and defining `rget` as in Figure 25.1.

With single inheritance, when we wanted to retrieve some property of an object, we just searched recursively up its ancestors. If the object itself had no information about the property we wanted, we looked at its parent, and so on. With multiple inheritance we want to perform the same kind of search, but our job is complicated by the fact that an object's ancestors can form a graph instead of a simple list. We can't just search this graph depth-first. With multiple parents we can have the hierarchy shown in Figure 25.2: `a` is descended from `b` and `c`, which are both descended from `d`. A depth-first (or rather, height-first) traversal would go `a`, `b`, `d`, `c`, `d`. If the desired property were present in both `d` and `c`, we would

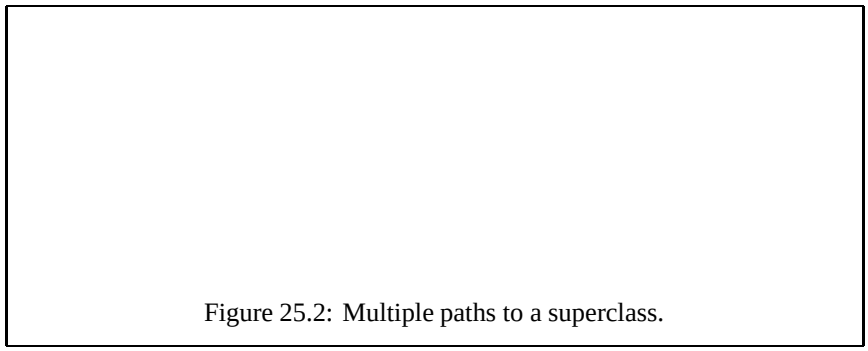

Figure 25.2: Multiple paths to a superclass.

get the value stored in `d`, not the one stored in `c`. This would violate the principle that subclasses override the default values provided by their parents.

If we want to implement the usual idea of inheritance, we should never examine an object before one of its descendants. In this case, the proper search order would be `a`, `b`, `c`, `d`. How can we ensure that the search always tries descendants first? The simplest way is to assemble a list of all the ancestors of the original object, sort the list so that no object appears before one of its descendants, and then look at each element in turn.

This strategy is used by `get-ancestors`, which returns a properly ordered list
○ of an object and its ancestors. To sort the list, `get-ancestors` calls `stable-sort` instead of `sort`, to avoid the possibility of reordering parallel ancestors. Once the list is sorted, `rget` merely searches for the first object with the desired property. (The utility `some2` is a version of `some` for use with functions like `gethash` that indicate success or failure in the second return value.)

The list of an object's ancestors goes from most specific to least specific: if `orange` is a child of `citrus`, which is a child of `fruit`, then the list will go `(orange citrus fruit)`.

When an object has multiple parents, their precedence goes left-to-right. That is, if we say

```
(setf (gethash 'parents x) (list y z))
```

then `y` will be considered before `z` when we look for an inherited property. For example, we can say that a patriotic scoundrel is a scoundrel first and a patriot second:

```
> (setq scoundrel (make-hash-table)
        patriot (make-hash-table)
        patriotic-scoundrel (make-hash-table))
#<Hash-Table C4219E>
```

```
(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (setf (gethash 'parents obj) parents)
    (ancestors obj)
    obj))

(defun ancestors (obj)
  (or (gethash 'ancestors obj)
      (setf (gethash 'ancestors obj) (get-ancestors obj))))

(defun rget (obj prop)
  (some2 #'(lambda (a) (gethash prop a))
         (ancestors obj)))
```

Figure 25.3: A function to create objects.

```
> (setf (gethash 'serves scoundrel) 'self
        (gethash 'serves patriot)   'country
        (gethash 'parents patriotic-scoundrel)
                 (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget patriotic-scoundrel 'serves)
SELF
T
```

Let's make some improvements to this skeletal system. We could begin with a function to create objects. This function should build a list of an object's ancestors at the time the object is created. The current code builds these lists when queries are made, but there is no reason not to do it earlier. Figure 25.3 defines a function called `obj` which creates a new object, storing within it a list of its ancestors. To take advantage of stored ancestors, we also redefine `rget`.

Another place for improvement is the syntax of message calls. The `tell` itself is unnecessary clutter, and because it makes verbs come second, it means that our programs can no longer be read like normal Lisp prefix expressions:

```
(tell (tell obj 'find-owner) 'find-owner)
```

We can get rid of the `tell` syntax by defining each property name as a function, as in Figure 25.4. The optional argument `meth?`, if true, signals that this property should be treated as a method. Otherwise it will be treated as a slot, and the value retrieved by `rget` will simply be returned. Once we have defined the name of either kind of property,

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget obj ',name)))
     (defsetf ,name (obj) (val)
       `(setf (gethash ',',name ,obj) ,val))))

(defun run-methods (obj name args)
  (let ((meth (rget obj name)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))
```

Figure 25.4: Functional syntax.

```
(defprop find-owner t)
```

we can refer to it with a function call, and our code will read like Lisp again:

```
(find-owner (find-owner obj))
```

Our previous example now becomes somewhat more readable:

```
> (progn
    (setq scoundrel (obj))
    (setq patriot (obj))
    (setq patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self)
    (setf (serves patriot) 'country)
    (serves patriotic-scoundrel))
SELF
T
```

In the current implementation, an object can have at most one method of a given name. An object either has its own method, or inherits one. It would be convenient to have more flexibility on this point, so that we could combine local and inherited methods. For example, we might want the `move` method of some object to be the `move` method of its parent, but with some extra code run before or afterwards.

To allow for such possibilities, we will modify our program to include before-, after-, and around-methods. Before-methods allow us to say "But first, do this." They are called, most specific first, as a prelude to the rest of the method call. After-methods allow us to say "P.S. Do this too." They are called, most specific last, as an epilogue to the method call. Between them, we run what used to be the whole method, and is now called the *primary method*. The value of this call is the one returned, even if after-methods are called later.

Before- and after-methods allow us to wrap new behavior around the call to the primary method. Around-methods provide a more drastic way of doing the same thing. If an around-method exists, it will be called *instead* of the primary method. Then, at its own discretion, the around-method may itself invoke the primary method (via `call-next`, which will be provided in Figure 25.7).

To allow auxiliary methods, we modify `run-methods` and `rget` as in Figures 25.5 and 25.6. In the previous version, when we ran some method of an object, we ran just one function: the most specific primary method. We ran the first method we encountered when searching the list of ancestors. With auxiliary methods, the calling sequence now goes as follows:

1. The most specific around-method, if there is one.
2. Otherwise, in order:
   (a) All before-methods, from most specific to least specific.
   (b) The most specific primary method (what we used to call).
   (c) All after-methods, from least specific to most specific.

Notice also that instead of being a single function, a method becomes a four-part structure. To define a (primary) method, instead of saying:

```
(setf (gethash 'move obj) #'(lambda ...))
```

we say:

```
(setf (meth-primary (gethash 'move obj)) #'(lambda ...))
```

For this and other reasons, our next step should be to define a macro for defining methods.

Figure 25.7 shows the definition of such a macro. The bulk of this code is taken up with implementing two functions that methods can use to refer to other methods. Around- and primary methods can use `call-next` to invoke the *next method*, which is the code that would have run if the current method didn't exist. For example, if the currently running method is the only around-method, the next

```
(defstruct meth  around before primary after)

(defmacro meth- (field obj)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (and (meth-p ,gobj)
            (,(symb 'meth- field) ,gobj)))))

(defun run-methods (obj name args)
  (let ((pri (rget obj name :primary)))
    (if pri
        (let ((ar (rget obj name :around)))
          (if ar
              (apply ar obj args)
              (run-core-methods obj name args pri)))
        (error "No primary ~A method for ~A." name obj))))

(defun run-core-methods (obj name args &optional pri)
  (multiple-value-prog1
    (progn (run-befores obj name args)
           (apply (or pri (rget obj name :primary))
                  obj args))
    (run-afters obj name args)))

(defun rget (obj prop &optional meth (skip 0))
  (some2 #'(lambda (a)
             (multiple-value-bind (val win) (gethash prop a)
               (if win
                   (case meth (:around  (meth- around val))
                              (:primary (meth- primary val))
                              (t (values val win))))))
         (nthcdr skip (ancestors obj))))
```

Figure 25.5: Auxiliary methods.

method would be the usual sandwich of before-, most specific primary, and after-methods. Within the most specific primary method, the next method would be the second most specific primary method. Since the behavior of `call-next` depends on where it is called, it is never defined globally with a `defun`, but is defined locally within each method defined by `defmeth`.

```
(defun run-befores (obj prop args)
  (dolist (a (ancestors obj))
    (let ((bm (meth- before (gethash prop a))))
      (if bm (apply bm obj args)))))

(defun run-afters (obj prop args)
  (labels ((rec (lst)
             (when lst
               (rec (cdr lst))
               (let ((am (meth- after
                                (gethash prop (car lst)))))
                 (if am (apply am (car lst) args))))))
    (rec (ancestors obj))))
```

Figure 25.6: Auxiliary methods (continued).

An around- or primary method can use `next-p` to check whether there is a next method. If the current method is the primary method of an object with no parents, for example, there would be no next method. Since `call-next` yields an error when there is no next method, `next-p` should usually be called to test the waters first. Like `call-next`, `next-p` is defined locally within individual methods.

The new macro `defmeth` is used as follows. If we just want to define the `area` method of the `rectangle` object, we say

```
(setq rectangle (obj))
(defprop height)
(defprop width)
(defmeth (area) rectangle (r)
  (* (height r) (width r)))
```

Now the area of an instance is calculated according to the method of the class:

```
> (let ((myrec (obj rectangle)))
    (setf (height myrec) 2
          (width myrec)  3)
    (area myrec))
6
```

```
(defmacro defmeth ((name &optional (type :primary))
                   obj parms &body body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (unless (meth-p (gethash ',name ,gobj))
         (setf (gethash ',name ,gobj) (make-meth)))
       (setf (,(symb 'meth- type) (gethash ',name ,gobj))
             ,(build-meth name type gobj parms body)))))

(defun build-meth (name type gobj parms body)
  (let ((gargs (gensym)))
    `#'(lambda (&rest ,gargs)
         (labels
           ((call-next ()
              ,(if (or (eq type :primary)
                       (eq type :around))
                   `(cnm ,gobj ',name (cdr ,gargs) ,type)
                   '(error "Illegal call-next.")))
            (next-p ()
              ,(case type
                 (:around
                  `(or (rget ,gobj ',name :around 1)
                       (rget ,gobj ',name :primary)))
                 (:primary
                  `(rget ,gobj ',name :primary 1))
                 (t nil))))
           (apply #'(lambda ,parms ,@body) ,gargs)))))

(defun cnm (obj name args type)
  (case type
    (:around  (let ((ar (rget obj name :around 1)))
                (if ar
                    (apply ar obj args)
                    (run-core-methods obj name args))))
    (:primary (let ((pri (rget obj name :primary 1)))
                (if pri
                    (apply pri obj args)
                    (error "No next method."))))))
```

Figure 25.7: Defining methods.

```
(defmacro undefmeth ((name &optional (type :primary)) obj)
  `(setf (,(symb 'meth- type) (gethash ',name ,obj))
         nil))
```

Figure 25.8: Removing methods.

In a more complicated example, suppose we have defined a `backup` method for the `filesystem` object:

```
(setq filesystem (obj))
(defmeth (backup :before) filesystem (fs)
  (format t "Remember to mount the tape.~%"))
(defmeth (backup) filesystem (fs)
  (format t "Oops, deleted all your files.~%")
  'done)
(defmeth (backup :after) filesystem (fs)
  (format t "Well, that was easy.~%"))
```

The normal sequence of calls will be as follows:

```
> (backup (obj filesystem))
Remember to mount the tape.
Oops, deleted all your files.
Well, that was easy.
DONE
```

Later we want to know how long backups take, so we define the following around-method:

```
(defmeth (backup :around) filesystem (fs)
  (time (call-next)))
```

Now whenever `backup` is called on a child of `filesystem` (unless more specific around-methods intervene) our around-method will be called. It calls the code that would ordinarily run in a call to `backup`, but within a call to `time`. The value returned by `time` will be returned as the value of the call to `backup`:

```
> (backup (obj filesystem))
Remember to mount the tape.
Oops, deleted all your files.
Well, that was easy.
Elapsed Time = .01 seconds
DONE
```

Once we are finished timing the backups, we will want to remove the around-method. That can be done by calling `undefmeth` (Figure 25.8), which takes the same first two arguments as `defmeth`:

```
(undefmeth (backup :around) filesystem)
```

Another thing we might want to alter is an object's list of parents. But after any such change, we should also update the list of ancestors of the object and all its children. So far, we have no way of getting from an object to its children, so we must also add a `children` property.

Figure 25.9 contains code for operating on objects' parents and children. Instead of getting at parents and children via `gethash`, we use the operators `parents` and `children`. The latter is a macro, and therefore transparent to `setf`. The former is a function whose inversion is defined by `defsetf` to be `set-parents`, which does everything needed to maintain consistency in the new doubly-linked world.

To update the ancestors of all the objects in a subtree, `set-parents` calls `maphier`, which is like a `mapc` for inheritance hierarchies. As `mapc` calls a function on every element of a list, `maphier` calls a function on an object and all its descendants. Unless they form a proper tree, the function could get called more than once on some objects. Here this is harmless, because `get-ancestors` does the same thing when called multiple times.

Now we can alter the inheritance hierarchy just by using `setf` on an object's `parents`:

```
> (progn (pop (parents patriotic-scoundrel))
         (serves patriotic-scoundrel))
COUNTRY
T
```

When the hierarchy is modified, affected lists of children and ancestors will be updated automatically. (The `children` are not meant to be manipulated directly, but they could be if we defined a `set-children` analogous to `set-parents`.) The last function in Figure 25.9 is `obj` redefined to use the new code.

As a final improvement to our system, we will make it possible to specify new ways of combining methods. Currently, the only primary method that gets called is the most specific (though it can call others via `call-next`). Instead we might like to be able to combine the results of the primary methods of each of an object's ancestors. For example, suppose that `my-orange` is a child of `orange`, which is a child of `citrus`. If the `props` method returns `(round acidic)` for `citrus`, `(orange sweet)` for `orange`, and `(dented)` for `my-orange`, it would be convenient to be able to make `(props my-orange)` return the *union* of all these values: `(dented orange sweet round acidic)`.

```
(defmacro children (obj)
  `(gethash 'children ,obj))

(defun parents (obj)
  (gethash 'parents obj))

(defun set-parents (obj pars)
  (dolist (p (parents obj))
    (setf (children p)
          (delete obj (children p))))
  (setf (gethash 'parents obj) pars)
  (dolist (p pars)
    (pushnew obj (children p)))
  (maphier #'(lambda (obj)
               (setf (gethash 'ancestors obj)
                     (get-ancestors obj)))
           obj)
  pars)

(defsetf parents set-parents)

(defun maphier (fn obj)
  (funcall fn obj)
  (dolist (c (children obj))
    (maphier fn c)))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (setf (parents obj) parents)
    obj))
```

Figure 25.9: Maintaining parent and child links.

We could have this if we allowed methods to apply some function to the values of all the primary methods, instead of just returning the value of the most specific. Figure 25.10 contains a macro which allows us to define the way methods are combined, and a new version of `run-core-methods` which can perform method combination.

We define the form of combination for a method via `defcomb`, which takes a method name and a second argument describing the desired combination. Or-

```
(defmacro defcomb (name op)
  `(progn
     (defprop ,name t)
     (setf (get ',name 'mcombine)
           ,(case op
              (:standard nil)
              (:progn '#'(lambda (&rest args)
                           (car (last args))))
              (t op)))))

(defun run-core-methods (obj name args &optional pri)
  (let ((comb (get name 'mcombine)))
    (if comb
        (if (symbolp comb)
            (funcall (case comb (:and #'comb-and)
                                (:or  #'comb-or))
                     obj name args (ancestors obj))
            (comb-normal comb obj name args))
        (multiple-value-prog1
          (progn (run-befores obj name args)
                 (apply (or pri (rget obj name :primary))
                        obj args))
          (run-afters obj name args)))))

(defun comb-normal (comb obj name args)
  (apply comb
         (mapcan #'(lambda (a)
                     (let* ((pm (meth- primary
                                       (gethash name a)))
                            (val (if pm
                                     (apply pm obj args))))
                       (if val (list val))))
                 (ancestors obj))))
```

Figure 25.10: Method combination.

dinarily this second argument should be a function. However, it can also be one of `:progn`, `:and`, `:or`, or `:standard`. With the former three, primary methods will be combined as though according to the corresponding operator, while `:standard` indicates that we want the traditional way of running methods.

```
(defun comb-and (obj name args ancs &optional (last t))
  (if (null ancs)
      last
      (let ((pm (meth- primary (gethash name (car ancs)))))
        (if pm
            (let ((new (apply pm obj args)))
              (and new
                   (comb-and obj name args (cdr ancs) new)))
            (comb-and obj name args (cdr ancs) last)))))

(defun comb-or (obj name args ancs)
  (and ancs
       (let ((pm (meth- primary (gethash name (car ancs)))))
         (or (and pm (apply pm obj args))
             (comb-or obj name args (cdr ancs))))))
```

Figure 25.11: Method combination (continued).

The central function in Figure 25.10 is the new `run-core-methods`. If the method being called has no `mcombine` property, then the method call proceeds as before. Otherwise the `mcombine` of the method is either a function (like `+`) or a keyword (like `:or`). In the former case, the function is just applied to a list of the values returned by all the primary methods.[1] In the latter, we use the function associated with the keyword to iterate over the primary methods.

The operators `and` and `or` have to be treated specially, as in Figure 25.11. They get special treatment not just because they are special forms, but because they short-circuit evaluation:

```
> (or 1 (princ "wahoo"))
1
```

Here nothing is printed because the `or` returns as soon as it sees a non-nil argument. Similarly, a primary method subject to `or` combination should never get called if a more specific method returns true. To provide such short-circuiting for `and` and `or`, we use the distinct functions `comb-and` and `comb-or`.

To implement our previous example, we would write:

```
(setq citrus (obj))
(setq orange (obj citrus))
```

[1] A more sophisticated version of this code could use `reduce` to avoid consing here.

```
(setq my-orange (obj orange))

(defmeth (props) citrus (c) '(round acidic))
(defmeth (props) orange (o) '(orange sweet))
(defmeth (props) my-orange (m) '(dented))

(defcomb props #'(lambda (&rest args) (reduce #'union args)))
```

after which `props` would return the union of all the primary method values: [2]

```
> (props my-orange)
(DENTED ORANGE SWEET ROUND ACIDIC)
```

Incidentally, this example suggests a choice that you only have when doing object-oriented programming in Lisp: whether to store information in slots or methods.

Afterward, if we wanted the `props` method to return to the default behavior, we just set the method combination back to standard:

```
> (defcomb props :standard)
NIL
> (props my-orange)
(DENTED)
```

Note that before- and after-methods only run in standard method combination. However, around-methods work the same as before.

The program presented in this section is intended as a model, not as a real foundation for object-oriented programming. It was written for brevity rather than efficiency. However, it is at least a working model, and so could be used for experiments and prototypes. If you do want to use the program for such purposes, one minor change would make it much more efficient: don't calculate or store ancestor lists for objects with only one parent.

25.3 Classes and Instances

The program in the previous section was written to resemble CLOS as closely as such a small program could. By understanding it we are already a fair way towards understanding CLOS. In the next few sections we will examine CLOS itself.

In our sketch, we made no syntactic distinction between classes and instances, or between slots and methods. In CLOS, we use the `defclass` macro to define a class, and we declare the slots in a list at the same time:

[2] Since the combination function for `props` calls `union`, the list elements will not necessarily be in this order.

```
(defclass circle ()
  (radius center))
```

This expression says that the `circle` class has no superclasses, and two slots, `radius` and `center`. We can make an instance of the circle class by saying:

```
(make-instance 'circle)
```

Unfortunately, we have defined no way of referring to the slots of a `circle`, so any instance we make is going to be rather inert. To get at a slot we define an accessor function for it:

```
(defclass circle ()
  ((radius :accessor circle-radius)
   (center :accessor circle-center)))
```

Now if we make an instance of a circle, we can set its `radius` and `center` slots by using `setf` with the corresponding accessor functions:

```
> (setf (circle-radius (make-instance 'circle)) 2)
2
```

We can do this kind of initialization right in the call to `make-instance` if we define the slots to allow it:

```
(defclass circle ()
  ((radius :accessor circle-radius :initarg :radius)
   (center :accessor circle-center :initarg :center)))
```

The `:initarg` keyword in a slot definition says that the following argument should become a keyword parameter in `make-instance`. The value of the keyword parameter will become the initial value of the slot:

```
> (circle-radius (make-instance 'circle
                   :radius 2
                   :center '(0 . 0)))
2
```

By declaring an `:initform`, we can also define slots which initialize themselves. The `visible` slot of the `shape` class

```
(defclass shape ()
  ((color   :accessor shape-color   :initarg :color)
   (visible :accessor shape-visible :initarg :visible
            :initform t)))
```

will be set to `t` by default:

```
> (shape-visible (make-instance 'shape))
T
```

If a slot has both an initarg and an initform, the initarg takes precedence when it is specified:

```
> (shape-visible (make-instance 'shape :visible nil))
NIL
```

Slots are inherited by instances and subclasses. If a class has more than one superclass, it inherits the union of their slots. So if we define the class `screen-circle` to be a subclass of both `circle` and `shape`,

```
(defclass screen-circle (circle shape)
  nil)
```

then instances of `screen-circle` will have four slots, two inherited from each grandparent. Note that a class does not have to create any new slots of its own; this class exists just to provide something instantiable that inherits from both `circle` and `shape`.

The accessors and initargs work for instances of `screen-circle` just as they would for instances of `circle` or `shape`:

```
> (shape-color (make-instance 'screen-circle
                              :color 'red :radius 3))
RED
```

We can cause every `screen-circle` to have some default initial `color` by specifying an initform for this slot in the `defclass`:

```
(defclass screen-circle (circle shape)
  ((color :initform 'purple)))
```

Now instances of `screen-circle` will be purple by default,

```
> (shape-color (make-instance 'screen-circle))
PURPLE
```

though it is still possible to initialize the slot otherwise by giving an explicit `:color` initarg.

In our sketch of object-oriented programming, instances inherited values directly from the slots in their parent classes. In CLOS, instances do not have slots in the same way that classes do. We define an inherited default for instances by defining an initform in the parent class. In a way, this is more flexible, because as well as being a constant, an initform can be an expression that returns a different value each time it is evaluated:

```
(defclass random-dot ()
  ((x :accessor dot-x :initform (random 100))
   (y :accessor dot-y :initform (random 100))))
```

Each time we make an instance of a `random-dot` its x- and y-position will be a random integer between `0` and `99`:

```
> (mapcar #'(lambda (name)
              (let ((rd (make-instance 'random-dot)))
                (list name (dot-x rd) (dot-y rd))))
          '(first second third))
((FIRST 25 8) (SECOND 26 15) (THIRD 75 59))
```

In our sketch, we also made no distinction between slots whose values were to vary from instance to instance, and those which were to be constant across the whole class. In CLOS we can specify that some slots are to be *shared*—that is, their value is the same for every instance. We do this by declaring the slot to have `:allocation :class`. (The alternative is for a slot to have `:allocation :instance`, but since this is the default there is no need to say so explicitly.) For example, if all owls are nocturnal, then we can make the `nocturnal` slot of the `owl` class a shared slot, and give it the initial value `t`:

```
(defclass owl ()
  ((nocturnal :accessor owl-nocturnal
              :initform t
              :allocation :class)))
```

Now every instance of the `owl` class will inherit this slot:

```
> (owl-nocturnal (make-instance 'owl))
T
```

If we change the "local" value of this slot in an instance, we are actually altering the value stored in the class:

```
> (setf (owl-nocturnal (make-instance 'owl)) 'maybe)
MAYBE
> (owl-nocturnal (make-instance 'owl))
MAYBE
```

This could cause some confusion, so we might like to make such a slot read-only. When we define an accessor function for a slot, we create a way of both reading and writing the slot's value. If we want the value to be readable but not writable, we can do it by giving the slot just a reader function, instead of a full-fledged accessor function:

```
(defclass owl ()
  ((nocturnal :reader owl-nocturnal
              :initform t
              :allocation :class)))
```

Now attempts to alter the `nocturnal` slot of an instance will generate an error:

```
> (setf (owl-nocturnal (make-instance 'owl)) nil)
>>Error: The function (SETF OWL-NOCTURNAL) is undefined.
```

25.4 Methods

Our sketch emphasized the similarity between slots and methods in a language which provides lexical closures. In our program, a primary method was stored and inherited in the same way as a slot value. The only difference between a slot and a method was that defining a name as a slot by

```
(defprop area)
```

made `area` a function which would simply retrieve and return a value, while defining it as a method by

```
(defprop area t)
```

made `area` a function which would, after retrieving a value, funcall it on its arguments.

In CLOS the functional units are still called methods, and it is possible to define them so that they each seem to be a property of some class. Here we define an `area` method for the `circle` class:

```
(defmethod area ((c circle))
  (* pi (expt (circle-radius c) 2)))
```

The parameter list for this method says that it is a function of one argument which applies to instances of the `circle` class.

We invoke this method like a function, just as in our sketch:

```
> (area (make-instance 'circle :radius 1))
3.14...
```

We can also define methods that take additional arguments:

```
(defmethod move ((c circle) dx dy)
  (incf (car (circle-center c)) dx)
  (incf (cdr (circle-center c)) dy)
  (circle-center c))
```

If we call this method on an instance of `circle`, its center will be shifted by ⟨dx,dy⟩:

```
> (move (make-instance 'circle :center '(1 . 1)) 2 3)
(3 . 4)
```

The value returned by the method reflects the circle's new position.

As in our sketch, if there is a method for the class of an instance, and for superclasses of that class, the most specific one runs. So if `unit-circle` is a subclass of `circle`, with the following `area` method

```
(defmethod area ((c unit-circle)) pi)
```

then this method, rather than the more general one, will run when we call `area` on an instance of `unit-circle`.

When a class has multiple superclasses, their precedence runs left to right. By defining the class `patriotic-scoundrel` as follows

```
(defclass scoundrel nil nil)
(defclass patriot nil nil)
(defclass patriotic-scoundrel (scoundrel patriot) nil)
```

we specify that patriotic scoundrels are scoundrels first and patriots second. When there is an applicable method for both superclasses,

```
(defmethod self-or-country? ((s scoundrel))
  'self)

(defmethod self-or-country? ((p patriot))
  'country)
```

the method of the `scoundrel` class will run:

```
> (self-or-country? (make-instance 'patriotic-scoundrel))
SELF
```

The examples so far maintain the illusion that CLOS methods are methods *of* some object. In fact, they are something more general. In the parameter list of the `move` method, the element `(c circle)` is called a *specialized* parameter; it says that this method applies when the first argument to `move` is an instance of the `circle` class. In a CLOS method, *more than one parameter can be specialized.* The following method has two specialized and one optional unspecialized parameter:

```
(defmethod combine ((ic ice-cream) (top topping)
                    &optional (where :here))
  (append (list (name ic) 'ice-cream)
          (list 'with (name top) 'topping)
          (list 'in 'a
                (case where
                  (:here  'glass)
                  (:to-go 'styrofoam))
                'dish)))
```

It is invoked when the first two arguments to `combine` are instances of `ice-cream` and `topping`, respectively. If we define some minimal classes to instantiate

```
(defclass stuff () ((name :accessor name :initarg :name)))
(defclass ice-cream (stuff) nil)
(defclass topping (stuff) nil)
```

then we can define and run this method:

```
> (combine (make-instance 'ice-cream :name 'fig)
           (make-instance 'topping :name 'olive)
           :here)
(FIG ICE-CREAM WITH OLIVE TOPPING IN A GLASS DISH)
```

When methods specialize more than one of their parameters, it is difficult to continue to regard them as properties of classes. Does our `combine` method belong to the `ice-cream` class or the `topping` class? In CLOS, the model of objects responding to messages simply evaporates. This model seems natural so long as we invoke methods by saying something like:

```
(tell obj 'move 2 3)
```

Then we are clearly invoking the `move` method of `obj`. But once we drop this syntax in favor of a functional equivalent:

```
(move obj 2 3)
```

then we have to define `move` so that it *dispatches* on its first argument—that is, looks at the type of the first argument and calls the appropriate method.

Once we have taken this step, the question arises: why only allow dispatching on the first argument? CLOS answers: why indeed? In CLOS, methods can specialize any number of their parameters—and not just on user-defined classes, but on Common Lisp types,[3] and even on individual objects. Here is a `combine` method that applies to strings:

[3] Or more precisely, on the type-like classes that CLOS defines in parallel with the Common Lisp type hierarchy.

```
(defmethod combine ((s1 string) (s2 string) &optional int?)
  (let ((str (concatenate 'string s1 s2)))
    (if int? (intern str) str)))
```

Which means not only that methods are no longer properties of classes, but that we can use methods without defining classes at all.

```
> (combine "I am not a " "cook.")
"I am not a cook."
```

Here the second parameter is specialized on the symbol `palindrome`:

```
(defmethod combine ((s1 sequence) (x (eql 'palindrome))
                    &optional (length :odd))
  (concatenate (type-of s1)
               s1
               (subseq (reverse s1)
                       (case length (:odd 1) (:even 0)))))
```

This particular method makes palindromes of any kind of sequence elements: [4]

```
> (combine '(able was i ere) 'palindrome)
(ABLE WAS I ERE I WAS ABLE)
```

At this point we no longer have object-oriented programming, but something more general. CLOS is designed with the understanding that beneath methods there is this concept of dispatch, which can be done on more than one argument, and can be based on more than an argument's class. When methods are built upon this more general notion, they become independent of individual classes. Instead of adhering conceptually to classes, methods now adhere to other methods with the same name. In CLOS such a clump of methods is called a *generic function.* All our `combine` methods implicitly define the generic function `combine`.

We can define generic functions explicitly with the `defgeneric` macro. It is not necessary to call `defgeneric` to define a generic function, but it can be a convenient place to put documentation, or some sort of safety-net for errors. Here we do both:

```
(defgeneric combine (x y &optional z)
  (:method (x y &optional z)
     "I can't combine these arguments.")
  (:documentation "Combines things."))
```

[4] In one (otherwise excellent) Common Lisp implementation, `concatenate` will not accept `cons` as its first argument, so this call will not work.

Since the method given here for `combine` doesn't specialize any of its arguments, it will be the one called in the event no other method is applicable.

```
> (combine #'expt "chocolate")
"I can't combine these arguments."
```

Before, this call would have generated an error.

Generic functions impose one restriction that we don't have when methods are properties of objects: when all methods of the same name get joined into one generic function, their parameter lists must agree. That's why all our `combine` methods had an additional optional parameter. After defining the first `combine` method to take up to three arguments, it would have caused an error if we attempted to define another which only took two.

CLOS requires that the parameter lists of all methods with the same name be *congruent*. Two parameter lists are congruent if they have the same number of required parameters, the same number of optional parameters, and compatible use of `&rest` and `&key`. The actual keyword parameters accepted by different methods need not be the same, but `defgeneric` can insist that all its methods accept a certain minimal set. The following pairs of parameter lists are all congruent:

```
(x)             (a)
(x &optional y) (a &optional b)
(x y &rest z)   (a b &rest c)
(x y &rest z)   (a b &key c d)
```

and the following pairs are not:

```
(x)             (a b)
(x &optional y) (a &optional b c)
(x &optional y) (a &rest b)
(x &key x y)    (a)
```

Redefining methods is just like redefining functions. Since only required parameters can be specialized, each method is uniquely identified by its generic function and the types of its required parameters. If we define another method with the same specializations, it overwrites the original one. So by saying:

```
(defmethod combine ((x string) (y string)
                    &optional ignore)
  (concatenate 'string x " + " y))
```

we redefine what `combine` does when its first two arguments are strings.

Unfortunately, if instead of redefining a method we want to remove it, there is no built-in converse of `defmethod`. Fortunately, this is Lisp, so we can write

```
(defmacro undefmethod (name &rest args)
  (if (consp (car args))
      (udm name nil (car args))
      (udm name (list (car args)) (cadr args))))

(defun udm (name qual specs)
  (let ((classes (mapcar #'(lambda (s)
                             `(find-class ',s))
                         specs)))
    `(remove-method (symbol-function ',name)
                    (find-method (symbol-function ',name)
                                 ',qual
                                 (list ,@classes)))))
```

Figure 25.12: Macro for removing methods.

one. The details of how to remove a method by hand are summarized in the implementation of `undefmethod` in Figure 25.12. We use this macro by giving arguments similar to those we would give to `defmethod`, except that instead of giving a whole parameter list as the second or third argument, we give just the class-names of the required parameters. So to remove the `combine` method for two strings, we say:

```
(undefmethod combine (string string))
```

Unspecialized arguments are implicitly of class `t`, so if we had defined a method with required but unspecialized parameters:

```
(defmethod combine ((fn function) x &optional y)
  (funcall fn x y))
```

we could get rid of it by saying

```
(undefmethod combine (function t))
```

If we want to remove a whole generic function, we can do it the same way we would remove the definition of any function, by calling `fmakunbound`:

```
(fmakunbound 'combine)
```

25.5 Auxiliary Methods and Combination

Auxiliary methods worked in our sketch basically as they do in CLOS. So far we have seen only primary methods, but we can also have before-, after- and around-methods. Such auxiliary methods are defined by putting a qualifying keyword after the method name in the call to `defmethod`. If we define a primary `speak` method for the `speaker` class as follows:

```
(defclass speaker nil nil)

(defmethod speak ((s speaker) string)
  (format t "~A" string))
```

Then calling `speak` with an instance of `speaker` just prints the second argument:

```
> (speak (make-instance 'speaker)
         "life is not what it used to be")
life is not what it used to be
NIL
```

By defining a subclass `intellectual` which wraps before- and after-methods around the primary `speak` method,

```
(defclass intellectual (speaker) nil)

(defmethod speak :before ((i intellectual) string)
  (princ "Perhaps "))

(defmethod speak :after ((i intellectual) string)
  (princ " in some sense"))
```

we can create a subclass of speakers which always have the last (and the first) word:

```
> (speak (make-instance 'intellectual)
         "life is not what it used to be")
Perhaps life is not what it used to be in some sense
NIL
```

In standard method combination, the methods are called as described in our sketch: all the before-methods, most specific first, then the most specific primary method, then all the after-methods, most specific last. So if we define before- or after-methods for the `speaker` superclass,

```
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

they will get called in the middle of the sandwich:

```
> (speak (make-instance 'intellectual)
         "life is not what it used to be")
Perhaps I think life is not what it used to be in some sense
NIL
```

Regardless of what before- or after-methods get called, the value returned by the generic function is the value of the most specific primary method—in this case, the `nil` returned by `format`.

This changes if there are around-methods. If one of the classes in an object's family tree has an around-method—or more precisely, if there is an around-method specialized for the arguments passed to the generic function—the around-method will get called first, and the rest of the methods will only run if the around-method decides to let them. As in our sketch, an around- or primary method can invoke the next method by calling a function: the function we defined as `call-next` is in CLOS called `call-next-method`. There is also a `next-method-p`, analogous to our `next-p`. With around-methods we can define another subclass of `speaker` which is more circumspect:

```
(defclass courtier (speaker) nil)

(defmethod speak :around ((c courtier) string)
  (format t "Does the King believe that ~A? " string)
  (if (eq (read) 'yes)
      (if (next-method-p) (call-next-method))
      (format t "Indeed, it is a preposterous idea.~%"))
  'bow)
```

When the first argument to `speak` is an instance of the `courtier` class, the courtier's tongue is now guarded by the around-method:

```
> (speak (make-instance 'courtier) "kings will last")
Does the King believe that kings will last? yes
I think kings will last
BOW
> (speak (make-instance 'courtier) "the world is round")
Does the King believe that the world is round? no
Indeed, it is a preposterous idea.
BOW
```

Note that, unlike before- and after-methods, the value returned by the around-method is returned as the value of the generic function.

Generally, methods are run as in this outline, which is reprinted from Section 25.2:

1. The most specific around-method, if there is one.
2. Otherwise, in order:
   (a) All before-methods, from most specific to least specific.
   (b) The most specific primary method.
   (c) All after-methods, from least specific to most specific.

This way of combining methods is called standard method combination. As in our sketch, it is possible to define methods which are combined in other ways: for example, for a generic function to return the sum of all the applicable primary methods.

In our program, we specified how to combine methods by calling `defcomb`. By default, methods were combined as in the outline above, but by saying, for example,

```
(defcomb price #'+)
```

we could cause the function `price` to return the sum of all the applicable primary methods.

In CLOS this is called *operator* method combination. As in our program, such method combination can be understood as if it resulted in the evaluation of a Lisp expression whose first element was some operator, and whose arguments were calls to the applicable primary methods, in order of specificity. If we defined the `price` generic function to combine values with `+`, and there were no applicable around-methods, it would behave as though it were defined:

```
(defun price (&rest args)
  (+ (apply ⟨most specific primary method⟩ args)
     ⋮
     (apply ⟨least specific primary method⟩ args)))
```

If there are applicable around-methods, they take precedence, just as in standard method combination. Under operator method combination, an around-method can still call the next method via `call-next-method`. However, primary methods can no longer use `call-next-method`. (This is a difference from our sketch, where we left `call-next` available to such methods.)

In CLOS, we can specify the type of method combination to be used by a generic function by giving the optional `:method-combination` argument to `defgeneric`:

```
(defgeneric price (x)
  (:method-combination +))
```

Now the `price` method will use + method combination. If we define some classes with prices,

```
(defclass jacket nil nil)
(defclass trousers nil nil)
(defclass suit (jacket trousers) nil)

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)
```

then when we ask for the price of an instance of `suit`, we get the sum of the applicable `price` methods:

```
> (price (make-instance 'suit))
550
```

The following symbols can be used as the second argument to `defmethod` or in the `:method-combination` option to `defgeneric`:

```
+   and   append   list   max   min   nconc   or   progn
```

By calling `define-method-combination` you can define other kinds of method combination; see CLTL2, p. 830.

Once you specify the method combination a generic function should use, all methods for that function must use the same kind. Now it would cause an error if we tried to use another operator (or `:before` or `:after`) as the second argument in a `defmethod` for `price`. If we do want to change the method combination of `price` we must remove the whole generic function by calling `fmakunbound`.

25.6 CLOS and Lisp

CLOS makes a good example of an embedded language. This kind of program usually brings two rewards:

1. Embedded languages can be conceptually well-integrated with their environment, so that within the embedded language we can continue to think of programs in much the same terms.

2. Embedded languages can be powerful, because they take advantage of all the things that the base language already knows how to do.

CLOS wins on both counts. It is very well-integrated with Lisp, and it makes good use of the abstractions that Lisp has already. Indeed, we can often see Lisp through CLOS, the way we can see the shapes of objects through a sheet draped over them.

It is no accident that we usually speak to CLOS through a layer of macros. Macros do transformation, and CLOS is essentially a program which takes programs built out of object-oriented abstractions, and translates them into programs built out of Lisp abstractions.

As the first two sections suggested, the abstractions of object-oriented programming map so neatly onto those of Lisp that one could almost call the former a special case of the latter. The objects of object-oriented programming can easily be implemented as Lisp objects, and their methods as lexical closures. By taking advantage of such isomorphisms, we were able to provide a rudimentary form of object-oriented programming in just a few lines of code, and a sketch of CLOS in a few pages.

CLOS is a great deal larger and more powerful than our sketch, but not so large as to disguise its roots as an embedded language. Take `defmethod` as an example. Though CLTL2 does not mention it explicitly, CLOS methods have all the power of lexical closures. If we define several methods within the scope of some variable,

```
(let ((transactions 0))
  (defmethod withdraw ((a account) amt)
    (incf transactions)
    (decf (balance a) amt))
  (defmethod deposit ((a account) amt)
    (incf transactions)
    (incf (balance a) amt))
  (defun transactions ()
    transactions))
```

then at runtime they will share access to the variable, just like closures. Methods can do this because, underneath the syntax, they *are* closures. In the expansion of a `defmethod`, its body appears intact in the body of a sharp-quoted lambda-expression.

Section 7.6 suggested that it was easier to conceive of how macros work than what they mean. Likewise, the secret to understanding CLOS is to understand how it maps onto the fundamental abstractions of Lisp.

25.7 When to Object

The object-oriented style provides several distinct benefits. Different programs need these benefits to varying degrees. At one end of the continuum there are programs—simulations, for example—which are most naturally expressed in the abstractions of object-oriented programming. At the other end are programs written in the object-oriented style mainly to make them extensible.

Extensibility is indeed one of the great benefits of the object-oriented style. Instead of being a single monolithic blob of code, a program is written in small pieces, each labelled with its purpose. So later when someone else wants to modify the program, it will be easy to find the part that needs to be changed. If we want to change the way that objects of type `ob` are displayed on the screen, we change the `display` method of the `ob` class. If we want to make a new class of objects like `obs` but different in a few respects, we can create a subclass of `ob`; in the subclass, we change the properties we want, and all the rest will be inherited by default from the `ob` class. And if we just want to make a single `ob` which behaves differently from the rest, we can create a new child of `ob` and modify the child's properties directly. If the program was written carefully to begin with, we can make all these types of modifications without even looking at the rest of the code. From this point of view, an object-oriented program is a program organized like a table: we can change it quickly and safely by looking up the appropriate entry.

Extensibility demands the least from the object-oriented style. In fact, it demands so little that an extensible program might not need to be object-oriented at all. If the preceding chapters have shown anything, they have shown that Lisp programs do not have to be monolithic blobs of code. Lisp offers a whole range of options for extensibility. For example, you could quite literally have a program organized like a table: a program which consisted of a set of closures stored in an array.

If it's extensibility you need, you don't have to choose between an "object-oriented" and a "traditional" program. You can give a Lisp program exactly the degree of extensibility it needs, often without resorting to object-oriented techniques. A slot in a class is a global variable. And just as it is inelegant to use a global variable where you could use a parameter, it could be inelegant to build a world of classes and instances when you could do the same thing with less effort in plain Lisp. With the addition of CLOS, Common Lisp has become the most powerful object-oriented language in widespread use. Ironically, it is also the language in which object-oriented programming is least necessary.