11 Objects

It was amazing to me, and it is still amazing, that people could not imagine what the psychological difference would be to have an interactive terminal. You can talk about it on a blackboard until you are blue in the face, and people would say, "Oh, yes, but why do you need that?"... We used to try to think of all these analogies, like describing it in terms of the difference between mailing a letter to your mother and getting on the telephone. To this day I can still remember people only realizing when they saw a real demo, say, "Hey, it talks back. Wow! You just type that and you got an answer."
Fernando Corbató (who worked on Whirlwind in the 1950s),
Charles Babbage Institute interview, 1989

So far, we have seen two main approaches for solving 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.
- Data-centric programming (introduced in Chapter 5, and extended to imperative programming with 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.

All computational problems involve both data and procedures. All procedures act on some form of data; without data they can have no meaningful inputs and outputs. Any data-focused design must involve some procedures to perform computations using that data.

In this chapter, we overcome a weakness of both previous approaches: the data and the procedures that manipulate it are separate. Packaging procedures and data together leads to a new problem-solving approach known as *object-oriented programming*.

object-oriented programming

Unlike many programming languages, Scheme does not provide special built-in support for objects[1]. We can, however, build an object system ourselves using simpler expressions (primarily the procedure-making lambda

[1]This refers to the standard Scheme language, not the many extended Scheme languages provided by DrScheme. The MzScheme language does provide additional constructs for supporting objects, but we do not cover them in this book.

expressions). By building an object system from simple components, we provide a clearer and deeper understanding of how object systems work.

The next section introduces techniques for programming with objects that combine state with procedures that manipulate that state. Section 11.2 describes a powerful technique for programming with objects by implementing new objects that add or modify the behaviors of previously implemented objects. Section 11.3 provides some historical background on the development of object-oriented programming.

11.1 Packaging Procedures and State

Recall our counter from Section 10.1:

```
(define (update-counter!) (set! counter (+ counter 1)) counter)
```

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. Hence, 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.

11.1.1 Encapsulation

It would be more useful to package the counter variable with the procedure that manipulates it. Then we could create as many counters as we want, each with its own counter variable to manipulate.

The statefull application rule (from Section 10.2.2) suggests a way to do this: evaluating an application creates a new environment, so a counter variable is defined in the application environment is only visible through body of the created procedure.

The *make-counter* procedure creates a counter object that packages the *count* variable with the procedure that increases its value:

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

Each application of *make-counter* produces a new object that is a procedure with its own associated *count* variable. Protecting state so it can only be manipulated in controlled ways is known as *encapsulation*.

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. Hence, the only way to manipulate the counter value is through the counter object.

An equivalent *make-counter* definition uses a let-expression to make the initialization of the *count* variable clearer:

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

Figure 11.1 depicts the environment after creating two counter objects and applying one of them.

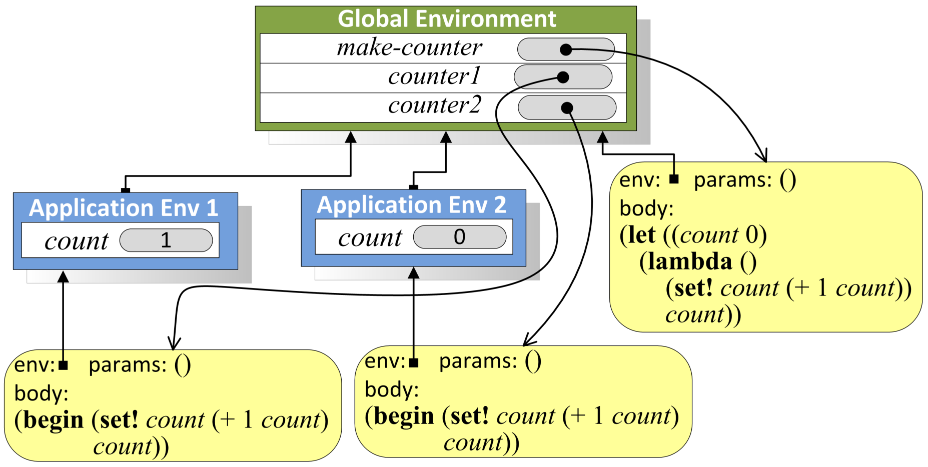


Figure 11.1. Environment produced by evaluating:
(**define** *counter1* (*make-counter*))
(**define** *counter2* (*make-counter*))
(*counter1*)

11.1.2 Messages

The object produced by *make-counter* is limited to only one behavior: every time it is applied the associated count variable is increased by one and the new value is output. To produce more useful objects, we need a way to combine state with multiple behaviors. For example, we might want a counter that return the current count, reset the count, and increment the count.

We do this by adding a *message* parameter to the procedure produced by *make-counter* that selects from different behavior procedures:

```
(define (make-counter)
  (let ((count 0))
    (lambda (message)
      (if (eq? message 'get-count) count
          (if (eq? message 'reset!) (set! count 0)
              (if (eq? message 'next!) (set! count (+ 1 count))
                  (error "Unrecognized message")))))))
```

Like the earlier *make-counter*, this procedure produces a procedure with an environment that contains a frame containing a place named *count*.

The produced procedure takes a *message* parameter, and its body is an if-expression that produces a different behavior depending on the input message. The input is a Symbol.

A Symbol is a sequence of characters preceded by a quote character such as 'next!. Two Symbols are equal (as determined by the *eq?* procedure) if their sequences of characters are identical.

The running time of the *eq?* procedure on symbol type inputs is constant; it does not increase with the length of the symbols since the symbols can be represented internally as small numbers and compared quickly using number equality. This makes symbols a more efficient way of selecting object behaviors than Strings, and a more memorable way to select behaviors than using Numbers.

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

```
> (define counter (make-counter))
> (counter 'next!)
> (counter 'get-count)
1
> (counter 'previous!)
Unrecognized message
```

Conditional expressions. If an object has many different behaviors, the nested if-expressions for selecting the behavior associated with the *message* input can get quite cumbersome. Scheme provides a compact conditional-expression for combining many if-expressions into one smaller expression.

The grammar for the conditional-expression is:

$$\begin{array}{lll} \mathit{Expression} & ::\Rightarrow & \mathit{CondExpression} \\ \mathit{CondExpression} & ::\Rightarrow & (\textbf{cond}\ \mathit{CondClauseList}) \\ \mathit{CondClauseList} & ::\Rightarrow & \mathit{CondClause}\ \mathit{CondClauseList} \\ \mathit{CondClauseList} & ::\Rightarrow & \epsilon \\ \mathit{CondClause} & ::\Rightarrow & (\mathit{Expression}_{predicate}\ \mathit{Expression}_{consequent}) \end{array}$$

The evaluation rule for a conditional-expression is:

Evaluation Rule 9: Conditional. A conditional-expression of the form, (**cond**) has no value. To evaluate a conditional-expression of the form,

$$\begin{array}{l} (\textbf{cond}\ (\mathit{Expression}_{p1}\ \mathit{Expression}_{c1}) \\ \qquad\quad (\mathit{Expression}_{p2}\ \mathit{Expression}_{c2}) \\ \qquad\quad \ldots \\ \qquad\quad (\mathit{Expression}_{pk}\ \mathit{Expression}_{ck})) \end{array}$$

in the current execution environment, E, first evaluate $\mathit{Expression}_{p1}$ in E. If it evaluates to true, the value of the conditional-expression is the value of $\mathit{Expression}_{c1}$ in E. Otherwise, the value of the conditional-expression is the value in E of the conditional-expression,

$$\begin{array}{l} (\textbf{cond}\ (\mathit{Expression}_{p2}\ \mathit{Expression}_{c2}) \\ \qquad\quad \ldots \\ \qquad\quad (\mathit{Expression}_{pk}\ \mathit{Expression}_{ck})) \end{array}$$

The evaluation rule is defined recursively. It continues through the predicate expressions until finding the first predicate that evaluates to *true*. Then, the value of the conditional-expression is the value of the consequent-expression associated with the first true predicate. If all the predicate expressions evaluate to *false*, the conditional-expression has no value.

Another way to define the evaluation rule for conditionals is to show how a conditional-expression can be transformed into an if-expression:

Evaluation Rule 9: Conditional (using if)**.** The conditional-expression (**cond**) has no value. All other conditional-expressions are of the form (**cond** (*Expression*$_{p1}$ *Expression*$_{c1}$) *Rest*) where *Rest* is a list of conditional clauses. The value of such a conditional-expression is the value of the if-expression:

(**if** *Expression*$_{p1}$ *Expression*$_{c1}$ (**cond** *Rest*))

This is also a recursive evaluation rule since the transformed expression still includes a conditional-expression, but is shorter and simpler since it takes advantage of the already defined evaluation rule for an if-expression.

The conditional-expression can be used to define *make-counter* more clearly than the nested if-expressions:

```
(define (make-counter)
  (let ((count 0))
    (lambda (message)
      (cond ((eq? message 'get-count) count)
            ((eq? message 'reset!)   (set! count 0))
            ((eq? message 'next!)    (set! count (+ 1 count)))
            (true (error "Unrecognized message"))))))
```

For linguistic convenience, Scheme provides a special syntax **else** for use in conditional-expressions. When used as the predicate in the last conditional clause it means the same thing as *true*. So, we could write the last clause as

```
(else (error "Unrecognized message"))
```

with the same exact meaning as above.

Sending messages. A more natural way to interact 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 is a simple procedure that does this:

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

It simply applies the *object* input to the *message* input. Later, we will develop more complex versions of the *ask* procedure to provide a more powerful object model.

Using the *ask* procedure, the interactions with our counter object could be expressed as:

```
> (define counter (make-counter))
> (ask counter 'next!)
> (ask counter 'get-count)
1
> (ask counter 'previous!)
Unrecognized message
```

Message parameters. Sometimes it is useful to have behaviors that take additional parameters. For example, we may want to support a message *adjust!* that increases the counter value by an input value.

To support such behaviors, we generalize the behaviors so that the result of applying the message dispatching procedure is itself a procedure. The procedures for *reset!*, *next!*, and *get-count* take no parameters; the procedure for *adjust!* takes one parameter.

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

We also need to also change the *ask* procedure to pass in the extra arguments. So far, all the procedures we have defined take a fixed number of operands. To allow *ask* to work for procedures that take a variable number of arguments, we use a special definition construct:

$$\textit{Definition} ::\Rightarrow \ (\textbf{define}\ (\textit{Name Parameters} \ \textbf{.}\ \textit{Name}_{\textit{Rest}})\ \textit{Expression})$$

The name following the dot is bound to all the remaining operands combined into a list. This means the defined procedure can be applied to n or more operands where n is the number of names in *Parameters*. If there are only n operand expressions, the value bound to $\textit{Name}_{\textit{Rest}}$ is null. If there are $n + k$ operand expressions, the value bound to $\textit{Name}_{\textit{Rest}}$ is a list containing the values of the last k operand expressions.

To apply the procedure we use the built-in *apply* procedure which takes two inputs, a Procedure and a List. It applies the procedure to the values in the list, 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 with two or more parameters to invoke methods with any number of arguments:

```
> (define counter (make-adjustable-counter))
> (ask counter 'adjust! 5)
> (ask counter 'next!)
> (ask counter 'get-count)
6
```

11.1.3 Object Terminology

object An *object* is an entity that packages state and procedures.

instance variables The state variables that are part of an object are called *instance variables*. The instance variables are stored in places that are part of the application environment for the object. This means they are encapsulated with the object and can only be accessed through the object. An object produced by (*make-counter*) defines a single instance variable, *count*.

methods The procedures that are part of an object are called *methods*. Methods may provide information about the state of an object (we call these *observers*) or modify the state of an object (we call these *mutators*). An object produced by (*make-counter*) provides three methods: *reset!* (a mutator), *next!* (a mutator), and *get-count* (an observer).

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

class A *class* is a kind of object. Classes are similar to data types. They define a set of possible values and operations (methods in the object terminology) for manipulating those values.

By convention, we call the constructor for a class *make-<class>* where *<class>* is the name of the class. Hence, an instance of the *counter* class is the result produced when the *make-counter* procedure is applied.

Exercise 11.1. Modify the *make-counter* definition to add a *previous!* method that decrements the counter value by one.

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

Exercise 11.2. [⋆] Define a *variable-counter* object that provides these methods:

- *set-increment!*: Number → Void — sets the increment amount for this counter.
- *next!*: Void → Void — modifies the value of the counter by adding the increment amount to it.
- *get-count*: Void → Number — outputs the current value of the counter.

The *make-variable-counter* takes one Number as input and produces a *variable-counter* object with initial counter value of 0 and an initial increment value that is the value of the input. Here are some sample interactions using a *variable-counter* object:

```
> (define vcounter (make-variable-counter 1))
> (ask vcounter 'next!)
> (ask vcounter 'set-increment! 2)
> (ask vcounter 'next!)
> (ask vcounter 'get-count)
3
```

11.2 Inheritance

Objects are particularly well-suited to programs that involve modeling real or imaginary worlds 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 most simulated worlds) are complex. Suppose we are implementing a game that simulates a typical university. It might include 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 also 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). A *student* has all the behaviors of a normal *person*, as well as some behaviors particular to a *student* such as choosing a major and graduating.

Figure 11.2 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*.

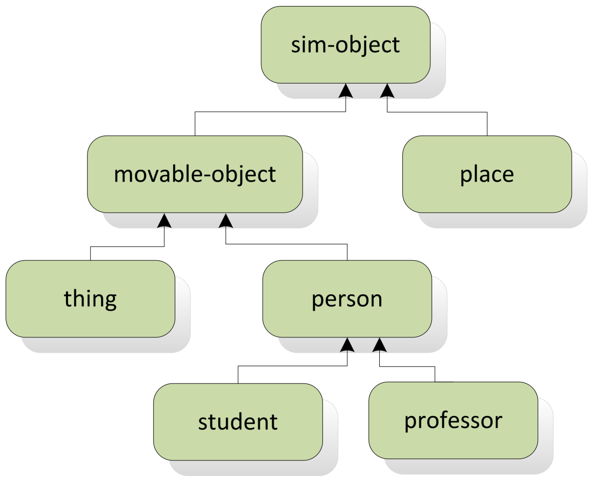


Figure 11.2. Inheritance Hierarchy.

To implement a *student* class, we want to reuse 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 say *student* is a subclass of *person*. The reused class is known as the *superclass*, so *person* is the superclass of *student*. A class can have many subclasses but only one superclass.[2]

subclass

superclass

Our goal is to be able to reuse superclass methods in subclasses. When a

[2]Some object systems (such as the one provided by the C++ programming language) allow a class to have more than one superclass. This can be confusing, though. If a class has two superclasses and both define methods with the same name, it may be ambiguous which of the methods is used when it is invoked on an object of the subclass. In our object system, a class may have only one superclass.

method is invoked in a subclass, if the subclass does not provide a definition of the method, then the definition of the method in the superclass is used. This can continue up the superclass chain. For instance, *student* is a subclass of *person*, which is a subclass of *movable-object*, which is a subclass of *sim-object*.

Hence, if the *sim-object* class defines a *get-name* method, when the *get-name* method is invoked on a *student* object, the implementation of *get-name* in the *sim-object* class will be used (as long as neither *person* nor *movable-object* defines its own *get-name* method).

When one class implementation uses the methods from another class we say the subclass *inherits* from the superclass. Inheritance is a powerful way to obtain many different objects with a small amount of code.

inherits

11.2.1 Implementing Subclasses

To implement inheritance we need to change class definitions so that if a requested method is not defined by the subclass, the method defined by its superclass will be used.

The *make-sub-object* procedure does this. It takes two inputs, a superclass object and the object dispatching procedure of the subclass, and produces an instance of the subclass. This is a procedure that takes a message as input and outputs the method corresponding to that message. If the method is defined by the subclass, it will be the subclass method. If the method is not defined by the subclass, it will be the superclass method.

```
(define (make-sub-object super subproc)
  (lambda (message)
    (let ((method (subproc message)))
      (if method
          method
          (super message)))))
```

When an object produced by (*make-sub-object obj proc*) is applied to a message, it will first apply the subclass dispatch procedure to the message to find an appropriate method if one is defined. If no method is defined by the subclass implementation, it evaluates to (*super message*), the method associated with the *message* in the superclass.

References to self. It is useful to add an extra parameter to all methods so the object on which the method was invoked is visible. Otherwise, the object will lose its special behaviors as it is moves up the superclasses. We call this the *self* object (in some languages it is called the *this* object instead).

To support this, we modify the *ask* procedure 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. So, the *counter* constructor is defined as:

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

Subclassing counter. Since subclass objects cannot see the instance variables of their superclass objects directly, if we want to provide a versatile counter class we need to also provide a *set-count!* method for setting the value of the counter to an arbitrary value. For reasons that will become clear later, we should use *set-count!* everywhere the value of the *count* variable is changed instead of setting it directly:

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

Previously, we defined *make-adjustable-counter* by repeating all the code from *make-counter* and adding an *adjust!* method. With inheritance, we can define *make-adjustable-counter* as a subclass of *make-counter* without repeating any code:

```
(define (make-adjustable-counter)
  (make-sub-object
   (make-counter)
   (lambda (message)
     (cond
      ((eq? message 'adjust!)
       (lambda (self val)
         (ask self 'set-count! (+ (ask self 'get-count) val))))
      (else false)))))
```

We use *make-sub-object* to create an object that inherits the behaviors from one class, and extends those behaviors by defining new methods in the subclass implementation. The new *adjust!* method takes one Number parameter (in addition to the *self* object that is passed to every method) and adds that number to the current counter value. It cannot use (**set!** *count* (+ *count val*)) directly, though, since the *count* variable is defined in the application environment of its superclass object and is not visible within *adjustable-counter*. Hence, it accesses the counter using the *set-count!* and *get-count* methods provided by the superclass.

Suppose we create an *adjustable-counter* object:

```
(define acounter (make-adjustable-counter))
```

Consider what happens when (*ask acounter* 'adjust! 3) is evaluated. The *acounter* object is the result of the application of *make-sub-object*, so it is the procedure

```
(lambda (message)
  (let ((method (subproc message)))
    (if method method (super message))))
```

where *super* is the *counter* object resulting from evaluating (*make-counter*) and *subproc* is the procedure created by the lambda-expression in *make-adjustable-counter*. The body of *ask* evaluates (*object message*) to find the method associated with the input message, in this case 'adjust!. The *acounter* object takes the *message* input and evaluates the let-expression:

```
(let ((method (subproc message))) ...)
```

The result of applying *subproc* to *message* is the *adjust!* procedure defined by *make-adjustable-counter*:

```
(lambda (self val)
  (ask self 'set-count! (+ (ask self 'get-count) val)))
```

Since this is not false, the predicate of the if-expression is non-false and the value of the consequent expression, *method*, is the result of the procedure application. The *ask* procedure uses *apply* to apply this procedure to the *object* and *args* parameters. The *object* is the *acounter* object, and the *args* is the list of the extra parameters, in this case (3).

Thus, the *adjust!* method procedure is applied to the *acounter* object and 3. The body of the *adjust!* method uses *ask* to invoke the *set-count!* method on the *self* object. As with the first invocation, the body of *ask* evaluates (*object message*) to find the method. In this case, the subclass implementation provides no *set-count!* method so the result of (*subproc message*) in the application of the subclass object is false. Hence, the alternate expression is evaluated: (*super message*). This evaluates to the method associated with the *set-count!* message in the superclass. The *ask* body will apply this method to the *self* object, setting the value of the counter to the new value.

We can define new classes by defining subclasses of previously defined classes. For example, *reversible-counter* inherits from *adjustable-counter*:

```
(define (make-reversible-counter)
  (make-subobject
   (make-adjustable-counter)
   (lambda (message)
     (cond
       ((eq? message 'previous!) (lambda (self) (ask self 'adjust! −1)))
       (else false)))))
```

The *reversible-counter* object defines the *previous!* method which provides a new behavior. If the message to a *adjustable-counter* object is not *previous!*, the method from its superclass, *adjustable-counter* is used. 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.

11.2.2 Overriding Methods

In addition to adding new methods, subclasses can replace the definitions of methods defined in the superclass. When 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.

overrides

For example, we can define a subclass of *reversible-counter* 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 *set-count!* method. This replaces the superclass implementation of the method with a new implementation.

```
(define (make-positive-counter)
  (make-subobject
   (make-reversible-counter)
   (lambda (message)
     (cond
      ((eq? message 'set-count!)
       (lambda (self val) (if (< val 0) (error "Negative count")
                                        ...)))
      (else false)))))
```

What should go where the ... is? When the value to set the count to is not negative, what should happen is the count is set as it would be by the superclass *set-count!* method. In the *positive-counter* code though, there is no way to access the *count* variable since it is in the superclass procedure's application environment. There is also no way to invoke the superclass' *set-count!* method since it has been overridden by *positive-counter*.

The solution is to provide a way for the subclass object to obtain its superclass object. We can do this by adding a *get-super* method to the object produced by *make-sub-object*:

```
(define (make-sub-object super subproc)
  (lambda (message)
    (if (eq? message 'get-super)
        (lambda (self) super)
        (let ((method (subproc message)))
          (if method method (super message))))))
```

Thus, when an object produced by *make-sub-object* is passed the *get-super* message it returns a method that produces the *super* object. The rest of the procedure is the same as before, so for every other message it behaves like the earlier *make-sub-object* procedure.

With the *get-super* method we can define the *set-count!* method for *positive-counter*, replacing the ... with:

```
(ask (ask self 'get-super) 'set-count! val))
```

Figure 11.3 shows the subclasses that inherit from *counter* and the methods they define or override.

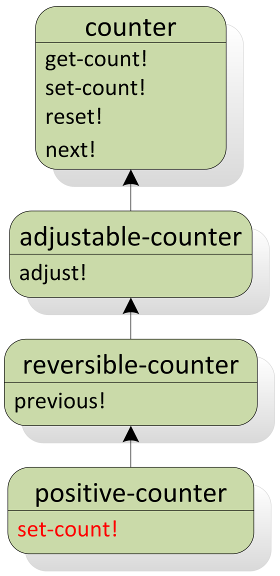


Figure 11.3. Counter class hierarchy.

Consider these sample interactions with a *positive-counter* object:

```
> (define poscount (make-positive-counter))
> (ask poscount 'next!)
> (ask poscount 'previous!)
> (ask poscount 'previous!)
Negative count
> (ask poscount 'get-count)
0
```

For the first *ask* application, the *next!* method is invoked on a *positive-counter* object. Since the *positive-counter* class does not define a *next!* method, the message is sent to the superclass, *reversible-counter*. The *reversible-counter* implementation also does not define a *next!* method, so the message is passed up to its superclass, *adjustable-counter*. This class also does not define a *next!* method, so the message is passed up to its superclass, *counter*. The *counter* class defines a *next!* method, so that method is used.

For the next *ask*, the *previous!* method is invoked. As before, the *positive-counter* class does not define a *previous!* method, so the message is sent to the superclass. Here, *reversible-counter* does define a *previous!* method. Its implementation involves an invocation of the *adjust!* method: (*ask self* 'adjust! −1). This invocation is done on the *self* object, which is an instance of the *positive-counter* class. Hence, the *adjust!* method is found from the *positive-counter* class implementation. This is the method that overrides the *adjust!* method defined by the *adjustable-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 *reversible-counter* 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 *positive-counter* object, the *adjust!* method defined by *positive-counter* is used; if it is a *reversible-counter* object, the *adjust!* method defined by *adjustable-counter* class (the superclass of *reversible-counter*) is used.

dynamic dispatch

This is a lot of work to implement a simple counter, but the value of encapsulation and inheritance increases as programs get more complex. Programming with objects allows a programmer to manage complexity by hiding implementation details inside the objects from how those objects are 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 *reversible-counter* actually does, without knowing what the *adjust!* method does in all subclasses of *reversible-counter*.

Exercise 11.3. Define a *countdown* class that simulates a rocket launch countdown: it starts at some initial value, and counts down to zero, at which point the rocket is launched. Can you implement *countdown* as a subclass of *counter*?

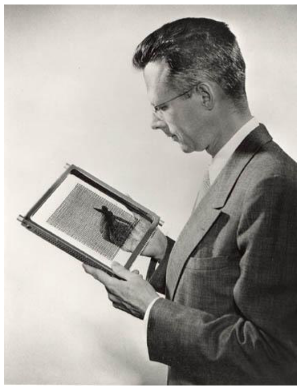
Exercise 11.4. Define the *variable-counter* object from Exercise 11.2 as a subclass of *counter*.

Exercise 11.5. Define a new subclass of *parameterizable-counter* where the increment for each *next!* method application is a parameter to the constructor procedure. For example, (*make-parameterizable-counter* 0.1) would produce a counter object whose counter has value 0.1 after one invocation of the *next!* method.

11.3 Object-Oriented Programming

Object-oriented programming is a style of programming where programs are broken down into objects that can be combined to solve a problem or model a simulated world. The notion of designing programs around object manipulations goes back at least to Ada (see the quote at the end if Chapter 6), but started in earnest in the early 1960s.

During World War II, the US Navy began to consider the possibility of building a airplane simulator for training pilots and aiding aircraft designers. At the time, pilots trained in mechanical simulators that were custom designed for particular airplanes.


Jay Forrester with magnetic-core memory (4096 bits)

The Navy wanted a simulator that could be used for multiple airplanes and could accurately model the aerodynamics of different airplanes. Project Whirlwind was started at MIT to build the simulator. The initial plans called for an analog computer which would need to be manually reconfigured to change the aerodynamics model to a different airplane. Jay Forrester learned about emerging projects to build digital computers, in particular the ENIAC project at the University of Pennsylvania which became operational in 1946, and realized that building a digital programmable computer would enable a much more flexible and powerful simulator, as well as a machine that could be used for many other purposes.

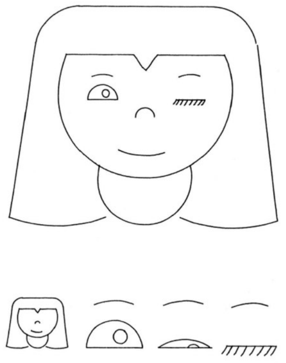
Before Whirlwind, all digital computers operated as batch processors: a programmer would create a program (typically described using a stack of punch cards) and submit this to the computer. A computer operator would set up the computer to run the program, after which it would run and (if everything works as hoped) produce a result. A flight simulator, though, requires direct interaction between a human user and the computer.

The first Whirlwind computer was designed in 1947 and operational by 1950. It was the first interactive programmable digital computer. Producing a machine that could perform the complex computations needed for a flight simulator fast enough to be used interactively required much faster and more reliable memory that was possible with available technologies based on storing electrostatic charges in vacuum tubes. Jay Forrester invented a much faster memory based known as magnetic-core memory. Magnetic-core memory stores a bit using magnetic polarity.

The interactiveness of the Whirlwind computer opened up many new possibilities for computing. Shortly after the first Whirlwind computer, Ken Olson led an effort to build a version of the computer using transistors. The successor to this machine became the TX-2, and Ken Olsen went on to found Digital Equipment Corporation (DEC) which pioneered the widespread use of moderately priced computers in science and industry. DEC was very successful in the 1970s and 1980s, but lacked the vision to see computers becoming inexpensive enough for individuals to own them, and suffered a long decline before eventually being bought by Compaq.

Ivan Sutherland, who was then a graduate student at MIT, had an opportunity to use the TX-2 machine. He developed a program called *Sketchpad* that was the first program to have an interactive graphical interface. Sketchpad allowed a used to draw and manipulate objects on the screen using a light pen. It was designed around objects and operations on those objects:

> In the process of making the Sketchpad system operate, a few very general functions were developed which make no reference at all to the specific types of entities on which they operate. These general functions give the Sketchpad system the ability to operate on a wide range of problems. The motivation for making the functions as general as possible came from the desire to get as much result as possible from the programming effort involved. For example, the general function for expanding instances makes it possible for Sketchpad to handle any fixed geometry subpicture. The rewards that come from implementing general functions are so great that the author has become reluctant to write any programs for specific jobs. Each of the general functions implemented in the Sketchpad system abstracts, in some sense, some common property of pictures independent of the specific subject matter of the pictures themselves.
>
> Ivan Sutherland,
> *Sketchpad: a Man-Machine Graphical Communication System*, 1963


Components in Sketchpad

Sketchpad was a great influence on Douglas Engelbart who developed a research program around a vision of using computers interactively to enhance human intellect. In what has become known as "the mother of all demos", Engelbart and his colleagues demonstrated a networked, graphical, interactive computing system to the general public for the first time in 1968. He and his colleague, Bill English, also invented the computer mouse and gave it its name.

Sketchpad was also a major influence on Alan Kay in developing object-oriented programming. The first language to include built-in support for objects was the Simula programming language, developed in Norway in the 1960s by Kristen Nygaard and Ole Johan Dahl. Simula was designed to be a language for implementing simulations. It provided mechanisms for packaging data and procedures, and for implementing subclasses using inheritance.

Alan Kay

In 1966, Alan Kay entered graduate school at the University of Utah, where Ivan Sutherland was then a professor. Here's how he describes his first assignment:

> Head whirling, I found my desk. ON it was a pile of tapes and listings, and a note: "This is the Algol for the 1108. It doesn't work. Please make it work." The latest graduate student gets the latest dirty task. The documentation was incomprehensible. Supposedly, this was the Case-Western Reserve 1107 Algol—but it had been doctored to make a language called Simula; the documentation read like Norwegian transliterated into English, which in fact it was. There were uses of words like activity and process that didn't seem to coincide with normal English usage.
>
> Finally, another graduate student and I unrolled the program listing 80 feet down the hall and crawled over it yelling discoveries to each other. The weirdest part was the storage allocator, which did not obey a stack discipline as was usual for Algol. A few days later, that provided the clue. What Simula was allocating were structures very much like the instances of Sketchpad. There wee descriptions that acted like masters and they could create instances, each of which was an independent entity. What Sketchpad called masters and instances, Simula called activities and processes. Moreover, Simula was a procedural language for controlling Sketchpad-like objects, thus having considerably more flexibility than constraints (though at some cost in elegance).
>
> This was the big hit, and I've not been the same since... For the first time I thought of the whole as the entire computer and wondered why anyone would want to divide it up into weaker things called data structures and procedures. Why not divide it up into little computers, as time sharing was starting to? But not in dozens. Why not thousands of them, each simulating a useful structure?
>
> Alan Kay, *The Early History of Smalltalk*, 1993

Alan Kay went on to design the language Smalltalk, which became the first widely used object-oriented language. Smalltalk was developed as part of a project at XEROX's Palo Alto Research Center to develop a hand-held computer that could be used as a learning environment by children.

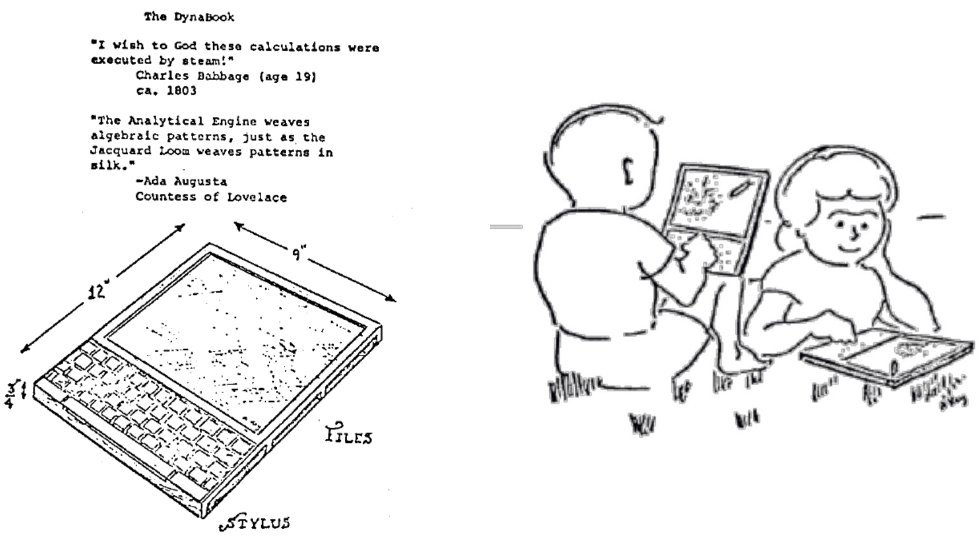
Don't worry about what anybody else is going to do. The best way to predict the future is to invent it. Really smart people with reasonable funding can do just about anything that doesn't violate too many of Newton's Laws!
Alan Kay

In Smalltalk, *everything* is an object, and all computation is done by sending messages to objects. For example, in Smalltalk one computes (+ 1 2) by sending the message + 2 to the object 1. Here is Smalltalk code for implementing a counter object:

```
class name counter
  instance variable names count
  new count <— 0
  next count <— count + 1
  current ^ count
```

The *new* method is a constructor similar to *make-counter*. The *next* method replaces the value of the instance variable *count* with the result of sending the message + 1 to the *count* object.

Nearly all widely-used languages today provide built-in support for some



The DynaBook

"I wish to God these calculations were executed by steam!"
Charles Babbage (age 19)
ca. 1803

"The Analytical Engine weaves algebraic patterns, just as the Jacquard Loom weaves patterns in silk."
-Ada Augusta
Countess of Lovelace

Figure 11.4. Images from Alan Kay, *A Personal Computer for Children of All Ages*, 1972.

form of object-oriented programming. For example, here is how a counter object could be defined in Python:

```
class counter:
  def __init__(self): self._count = 0
  def rest(self): self._count = 0
  def next(self): self._count = self._count + 1
  def current(self): return self._count
```

The constructor is named *__init__*. Similarly to the object system we developed for Scheme, each method takes the *self* object as its parameter.

11.4 Summary

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.