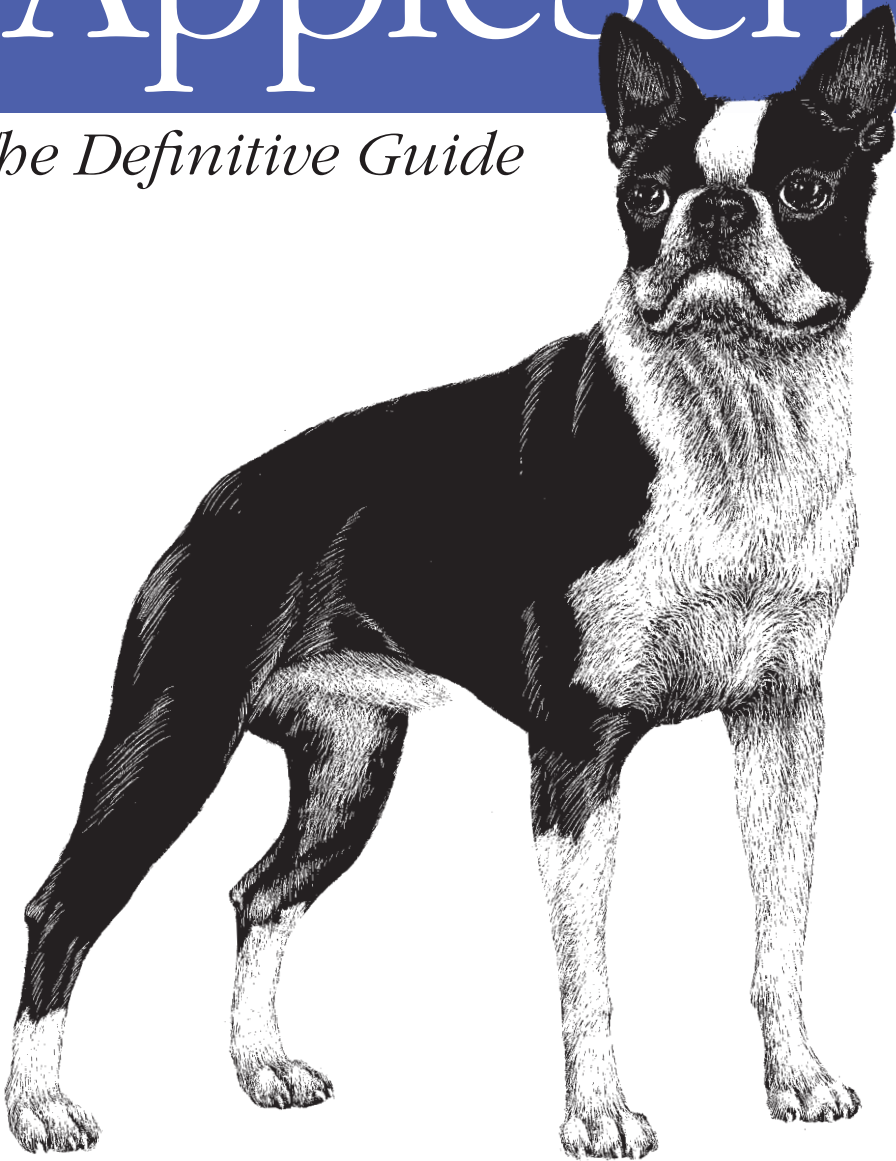


*Scripting and Automating Your Mac*

**2nd Edition**  
Covers Mac OS X Tiger

# AppleScript

*The Definitive Guide*



**O'REILLY®**

*Matt Neuburg*

# Writing Applications

You might wish to create a standalone application, perhaps as a way of distributing your script to other users in a form that's easy to use. AppleScript provides the simplest way in the world to write an application: just save your script as an applet. An applet is a true standalone application; it can even accept drag-and-drop of files and folders onto its icon, and (most surprising) it's scriptable. However, an applet has essentially no interface (except for `display dialog` and other user-interactive scripting addition commands). You can wrap your script in a full-fledged interface, with windows, buttons, text fields, menus, and similar bells and whistles, using AppleScript Studio, a free development environment that lets you create a Cocoa application even if the only programming language you know is AppleScript. Another use of AppleScript Studio is to wrap your script in the smaller interface of an Automator action; users can then customize it through its interface and link it to other actions to create their own workflows. This chapter also deals with some more advanced issues related to writing applications: how to get started adding scriptability to a Cocoa application, and how an AppleScript Studio application can communicate internally from Cocoa to AppleScript. (See also "Application" in Chapter 2.)

## Applets

An *applet* is a compiled script wrapped up in a simple standalone application shell (see "Applet and Droplet" in Chapter 3). To make a script into an applet, save it from a script editor application as an application instead of as a compiled script. You select this option in the Save As dialog. When you launch the resulting application (by double-clicking it in the Finder, for example), the script runs.

It is also possible to save a script as an application bundle. From the outside, this looks and works like an applet. Because it's a bundle, though, you can do things with it that you can't do with an old-style applet, such as storing extra resources inside it; for an example, see "Persistence," later in this chapter. Also, an application bundle can call scripting additions contained within itself; see "Loading

Scripting Additions” in Chapter 21. Keep in mind that this format is not compatible with systems earlier than Panther.

For applet formats, and for special applet behaviors when an application is missing as an applet starts up, see “Applet and Droplet” and “Application Missing When an Applet Launches” in Chapter 3, and “Using Terms From” in Chapter 19. Persistence works in applets; see “Persistence of Top-Level Entities” in Chapter 8. For the behavior of an applet when a runtime error occurs, see “Errors” in Chapter 19. When an applet runs, no decompilation takes place; for one way this can affect the behavior of your script, see “Raw Four-Letter Codes” in Chapter 20.

## Applet Options

When you elect to save a script as an applet, you are given some options that affect how the applet will behave:

### *Stay Open*

The normal behavior of an applet when started up is to run its script and then automatically quit. A stay-open applet does *not* automatically quit; it just sits there running, like any application. An applet has some built-in menus, and in a stay-open applet the user has time to access them; they include a Quit menu item, which the user can choose to quit the applet. If stay-open applet has already run its script, what’s the point of its staying open? For the answer, see “Applet Event Handlers,” later in this section.

### *Startup Screen*

(Also called Show Startup or Show Startup Screen; in older versions of Script Editor this option was reversed and was called Never Show Startup Screen.) The startup screen, if it is to be shown, is a kind of introductory splash screen displaying the script’s description when the applet is started up. In a script editor application, there is a text view into which a script’s description may be entered. The description is styled text, and the styling is maintained in the startup screen dialog. The splash screen also offers the choice to run the applet’s script or to quit without running it.

In a stay-open applet, where the user has time to access the applet’s menus, the user can toggle the startup screen option by choosing File → Use Startup Screen. Even in an ordinary applet, the user can compel the startup screen to appear by holding down the Control key as the applet starts up.

## Editing an Applet

A script saved as an applet is normally still legible and editable. The only way to prevent this is to save the applet as Run Only. Keep in mind that this means *even you* can’t edit the applet’s script; if you have no other copy of the script, you lose all ability to edit the applet’s script, forever.

Let's presume the applet is not run-only. How can its script be edited? Not by double-clicking the applet from the Finder, because that runs the applet. However, a script editor application can still open it (through its Open dialog, for example). In fact, you can even keep an applet script open for editing in a script editor application, save it without closing it, and then double-click it in the Finder to run it, as a way of testing while developing. But, for obvious reasons, you can't save an applet's script into the applet while the applet is actually running—well, you actually can, but the changed functionality won't be available until you quit the applet and start it up again.

Another way to edit an applet is to choose its Edit → Edit Script menu item. In a stay-open applet, that menu item can be chosen whenever the applet is idle. In non-stay-open applet, the menus are visible while the startup screen is displayed; thus, the user can start up the applet while holding down the Control key, to force the display of the startup screen, and then choose Edit → Edit Script.

## Applet Event Handlers

An applet script may contain certain event handlers (see “Event Handlers” in Chapter 9) which, if present, will be called automatically at specific moments in the lifetime of the applet:

### *run*

The run handler, whether implicit or explicit (see “The Run Handler” in Chapter 9), is called when the applet is started up, either by the user opening it from the Finder or by a script targeting it. (To start up an applet without calling its run handler, tell it to launch.)

### *reopen*

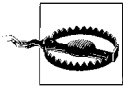
A reopen event handler, if present, is called when the already running applet is summoned to the front by such means as being double-clicked in the Finder or having its icon clicked in the Dock. Merely switching among applications with ⌘-Tab, or telling the applet to activate, does *not* send a reopen event.

### *idle*

An idle event handler, if present, is called as soon as the run handler finishes executing, and then again periodically while a stay-open applet is running. The value returned by the idle handler is a real number, representing how many seconds later the idle handler should be called again. A return value of 0, or any other value that can't be coerced to a positive real, is treated as 30.

### *quit*

A quit event handler, if present, is called when the applet is about to quit. If it is a stay-open applet, this might be because the user has chosen its Quit menu item; if not, it might be because the applet has been started up and its run handler has finished executing. If the quit handler wishes to permit the applet to quit, it must give the continue quit command.



An applet having a quit handler that does not give the continue quit command will appear to the user to be impossible to quit (except by force-quitting).

So, for example, here's an annoying little stay-open applet:

```
on run
    display dialog "Howdy!"
end run
on reopen
    display dialog "Get to Work!"
end reopen
on quit
    display dialog "Farewell!"
    continue quit
end quit
on idle
    beep
    activate
    display dialog "Get to Work!"
    return 1 * minutes
end idle
```

An applet is scriptable with respect to its event handlers; that is, you can tell an applet to run, reopen, idle, or quit, and it will execute the respective handler if it has one. If you tell an applet to quit and it has no quit handler, it simply quits. If you tell an applet to idle and it has no idle handler, or to reopen and it has no reopen handler, nothing happens (but the calling script may receive an error).

If an applet is not running, do not tell it to run or you'll confuse yourself. The run handler will be called *twice*—once because you targeted the applet, and again because you told it to run—and the idle handler will be called as well. To prevent this, tell the nonrunning applet to activate (the run handler will be called once, and the idle handler will be called).

Alternatively, you can tell the nonrunning applet to launch. In that case, neither the run handler nor the idle handler will be called, and no other event handler is called either; if you immediately tell the applet to quit, not even its quit handler is called. An applet launched in this way “comes to life” only if you send it some further Apple event (for example, tell it to launch and *then* tell it to run).

An idle handler should not be treated as ensuring a precise measure of time; the time interval returned is merely a request not to be called until after that interval has elapsed. (I am not entirely clear on what the time interval is measured *from*; experiments returning 0 seemed to suggest that it was measured from when the idle handler was last called, not from when it last returned, but this didn't seem to be true for other values.) If your goal is to run a script at certain times or intervals, you might be happier using a timer utility to handle this for you (see “Timers, Hooks, Attachability, Observability” in Chapter 26).

A question arises of how to interrupt a time-consuming applet. (This problem, and its solution, were suggested to me by Paul Berkowitz.) Suppose the run handler takes a long time and the user wishes to stop it and quit. The user can press Command- (period), which raises an error (-128, “User canceled”) that stops the run handler dead in its tracks, but the user might not know this. The Quit menu item doesn’t work; if the user chooses it, the applet doesn’t quit—if it has a quit handler, the quit handler is called, but when the quit handler says `continue quit`, the applet goes right back to executing the run handler where it left off! (I regard this as a bug.) There is also the question of how you can make sure that any required clean-up actions are performed as the applet quits. The best strategy is probably something like this:

```
global shouldQuit
global didCleanup
on run
    set shouldQuit to false
    set didCleanup to false
    try
        -- lengthy operation goes here
        repeat with x from 1 to 10
            if shouldQuit then error
            say (x as string)
            delay 5
        end repeat
    on error
        tell me to quit
    end try
end run
on quit
    if not didCleanup then
        -- cleanup operation goes here
        say "cleaning up"
        set didCleanup to true
    end if
    set shouldQuit to true
    continue quit
end quit
```

When the user chooses Quit from the menu bar, our quit handler is called, but when we say `continue quit` we will merely resume the run handler, so we also set a global indicating that the user is trying to quit. The resumed run handler notices this and deliberately errors out, and we catch the error and call our own quit handler, and quit in good order. We would perform our cleanup operations twice in this case, but that is prevented by another global. It’s all very ingenious, but it’s messy, and ultimately not very satisfactory. If the user chooses Quit from the applet’s Dock menu, our applet’s quit menu handler isn’t called (another bug), and if the user force-quits our applet then of course no cleanup is performed.

## Droplets

A *droplet* is an applet onto whose icon the user can drop Finder items (files and folders). Internally, it is simply an applet with an open event handler:

*open*

An open event handler, if present, will be called when items are dropped in the Finder onto the droplet's icon. It should take one parameter, through which your code will receive a list of aliases to the items dropped.

If a droplet is started up by double-clicking it from the Finder, then its run handler is executed and its open handler is not. But if it is started up by dropping items on it in the Finder, then it's the other way around: its open handler is executed and its run handler is not. Once a droplet is running (assuming it is a stay-open droplet), the open handler can be executed by dropping items onto the droplet's icon in the Finder. The open handler is also scriptable, using the open command, whose parameter should be a list of aliases.

In this simple example, the droplet reports how many folders were dropped on its icon:

```
on open what
  set total to 0
  repeat with f in what
    if folder of (info for f size no) then set total to total + 1
  end repeat
  display dialog (total as string) & " folder(s)"
end open
```

## Persistence

Persistence of top-level entities (see Chapter 8) works in an applet. The script is resaved when the applet quits, maintaining the state of its top-level environment.

So, for example, the following modification to the previous example would cause an applet to report the count of folders that had *ever* been dropped on it, not just the count of folders dropped on it at this moment:

```
property total : 0
on open what
  repeat with f in what
    if folder of (info for f size no) then set total to total + 1
  end repeat
  display dialog (total as string) & " folder(s)"
end open
```

On the other hand, this persistence naturally ends as soon as the applet's script is edited. If you're still developing an applet, or are likely to edit it further for any reason, you might like a way to store data persistently with no chance of losing it. The

application bundle format supplies a solution. An application bundle looks, and behaves in the Finder, just like an applet, but is in reality a folder. We can perform persistent data storage in a separate file inside the bundle; the user won't see this separate file, and its data will persist even when we edit the applet's main script.

To illustrate, let's return to the example in "Data Storage" in Chapter 8. This code is just the same as in that example, except that we now assume we are an application bundle, and the opening lines have been changed to store the data inside the bundle:

```
set thePath to (path to resource "applet.icns") as string
set text item delimiters to ":"
set thePath to ((text items 1 thru -2 of thePath) as string) & ":myPrefs.scpt"
script myPrefs
  property favoriteColor : ""
end script
try
  set myPrefs to load script file thePath
on error
  set favoriteColor of myPrefs to text returned of -
    (display dialog "Favorite Color:" default answer -
      "" buttons {"OK"} default button "OK")
  store script myPrefs in file thePath replacing yes
end try
display dialog "Your favorite color is " & favoriteColor of myPrefs
```

The first three lines of the script are a rather elaborate song-and-dance to obtain the pathname of the bundle's *Resources* directory. We could hardcode the path from the top of the bundle, like this:

```
set thePath to (path to me as string) & "Contents:Resources:myPrefs.scpt"
```

Instead, we use the `path to resource` command. But this command has an odd functionality hole: we can obtain the pathname of a resource file inside the *Resources* directory, but not the pathname of the directory itself. So we start with a resource file we know is present and work our way up the folder hierarchy and back down again.

## Applet Scriptability

We have seen already that an applet is scriptable with respect to its event handlers: you can tell an applet to run, open, reopen, idle, or quit, and the corresponding event handler will be called. An applet is scriptable also with respect to user handlers. You call them like ordinary handlers. For example, suppose we have a stay-open applet *howdy* whose script goes like this:

```
on sayHowdy(toWhom)
  activate
  display dialog "Howdy, " & toWhom
end sayHowdy
```

Then we can say in another script:

```
tell application "howdy"
    sayHowdy("Matt")
end tell
```

The value is returned as one would expect; here, the calling script receives the value {button returned: "OK"} if user presses the OK button.

The reason this is possible is that AppleScript's mechanism for calling a user handler in a script object is extended to work even when you're talking to an applet. The call is translated into a special event, the `Call•subroutine` command ('`ascr\psbr'`). This is a sort of meta-command; its parameters are the name of the handler you're calling and the parameters you're passing. At the other end, the target applet unpacks the handler call and performs it within its script. Thus it's just as if you said `sayHowdy("Matt")` from inside the applet's script. If the script defines such a handler, the result comes back as the reply. If it doesn't, you get a "Can't continue" error message. (This error message is familiar from when you *accidentally* use this same mechanism to send a user handler call to an ordinary scriptable application, which, as we have seen in Chapter 11 and elsewhere, is all too easy to do.)

An applet is also scriptable with respect to its other top-level entities. If an applet has a top-level property `x`, you can get and set its `x`. And if an applet defines a top-level script object `s`, you can refer to `s`, and you can get and set *its* properties.

In short, an applet behaves like a script object, whose top-level entities you can access in the normal way (see "Top-Level Entities" in Chapter 8). You might think of it as a stored script object that you can target without loading it, and that is automatically persistent without your storing it.

Of course, this means that when you're scripting an applet, no special dictionary is involved. An applet doesn't have a dictionary, and it doesn't need one, any more than a script object needs a dictionary. Therefore an applet has no way to publish information about what top-level entities it contains. The user who wishes to script the applet must know in some other way how to do so (by reading the applet's script, or through some other documentation).

## AppleScript Studio

*AppleScript Studio* is a free development environment from Apple allowing you to write Cocoa applications using the AppleScript language. It would require an entire book to discuss AppleScript Studio adequately, so in this section I'll just explain what AppleScript Studio is and how it works, and talk about how you might go about learning it more fully; I'll also provide a simple hands-on example of AppleScript Studio in action.

## Cocoa and AppleScript Studio

AppleScript Studio is Cocoa. The precise sense in which I mean this will be clearer in a moment, but it's a simple truth on the face of it, and it means that to understand what AppleScript Studio is, you need to know what Cocoa is.

*Cocoa* is a massive application framework included as part of Mac OS X. This framework knows how to do all the things that an application might typically wish to do. For example, it can put up windows, in which it can display many different kinds of interface elements for interacting with the user, such as buttons and text fields and sliders and tables and so forth. It also provides very strong text and graphics capabilities. Cocoa is a remarkably well-constructed application framework, striking an excellent balance between power and flexibility; with Cocoa, it's easy to write a simple standard application quite quickly, while at the same time the framework usually provides enough leeway so that the programmer can fully customize the application's behavior if desired. The presence of Cocoa as part of Mac OS X makes it much easier for programmers to write sophisticated, powerful, Mac OS X–native applications, while at the same time such applications often require relatively little code, because so much of the code that does the work resides in the framework.

AppleScript Studio is Apple's way of letting you, the AppleScript programmer, take advantage of the Cocoa application framework without having to learn a different programming language. The “native” Cocoa programming language is Objective-C, and to use Cocoa fully, you would want to learn that language. But as an AppleScript programmer, you might already have written a working script. You don't want to rewrite its functionality in some other language; you want to enhance your script with a more sophisticated user interface than AppleScript alone can provide. AppleScript Studio can let you do this. Think of it as a way to leverage an existing script into a Cocoa application, a way to wrap a Cocoa interface around AppleScript functionality with relative ease.

AppleScript Studio is not, however, a way to take *full* advantage of the power of Cocoa. By this I mean that you should not expect AppleScript Studio to let you do *everything* that Objective-C/Cocoa would let you do. If that's what you want, learn Objective-C! AppleScript Studio gives you access to a limited portion of Cocoa's power; that portion is usually enough to let you write a satisfactory application pretty quickly and easily, provided that your needs are fairly simple and your ambitions don't get out of hand. You should not feel disappointed about the fact that AppleScript Studio exposes to the AppleScript programmer only a simplified fraction of Cocoa's abilities. Simple is good. After all, Cocoa is very big, and can require years to learn fully. AppleScript Studio, on the other hand, is relatively tractable.

Let me explain in a bit more technical depth how AppleScript Studio exposes Cocoa to the AppleScript programmer. You'll need a sense of this in order to use AppleScript Studio effectively in any case. Cocoa is an application framework, so it operates through messages that travel back and forth between *its* code (the code inside

the framework) and *your* code (the code that you actually write, whether you're writing in Objective-C, AppleScript, or whatever). Cocoa is like a gigantic lock, and your code must be structured as a key that fits that lock; you have to write code that will slot into Cocoa's expectations of how an application should work. Thus, even though you can't see inside Cocoa, you do need to know what messages to send to Cocoa and what messages Cocoa will send you, so that its code and your code can work together properly. These messages come under some basic headings:

#### *Built-in methods*

The interface elements, such as windows and menus and buttons and text fields, as well as the application as a whole, are prepared to respond to certain messages that you can send. For example, you might like to tell a button to change its title, or ask a text field what the user has typed in it, or tell a window to close, or ask the application whether it is frontmost. You can do these things because the entity to whom you're speaking defines an appropriate message. A button "knows" how to change its title and defines a way for you to tell it to do so; a text field "knows" how to report what the user has typed in it; and so forth. These predefined messages that you can send are the *built-in methods*.

#### *Action messages*

Certain interface elements have a single special behavior when the user performs a certain special operation on them. In the case of a button, that operation is pushing the button. In the case of a text field, it's typing Return within the text field. This is the interface element's *action*, and when the action occurs, your code can receive an *action message* so that it can respond.

#### *Lifetime notifications*

Cocoa will, if you wish, notify your code of things that routinely take place over the lifetime of an application. For example, you might like your code to be called when your application first starts up, when it comes to the front, or when it is about to quit; or you might like to be informed when the user selects a line in a table or types a character in a text field. These are the *lifetime notifications*, and again, they exist so that your code has a chance to respond to what's happening.

#### *Delegation queries*

Cocoa will sometimes offer your code the opportunity to intervene in its behavior. For example, suppose that the user clicks the mouse to select a line in a table. Normally that line would just be selected, but perhaps your code might have some reason for making it unselectable at that moment. Cocoa can delegate the responsibility for this sort of decision to your code, querying your code as to whether it should proceed normally; if you elect to receive such *delegation queries*, your code must give a definite answer as to how Cocoa should proceed.

Thus there are two kinds of messages in Cocoa—those you send to Cocoa (the built-in methods) and those that Cocoa will send to you (the action messages, lifetime notifications, and delegation queries). Furthermore, your code is essentially idle until

something triggers some part of it. An application does nothing unless it is somehow told to do it. And the only way your code can be told to do anything is through a message that Cocoa sends it. In other words, *all* your code will run only in response to action messages, lifetime notifications, and delegations queries; so your code must be structured, not independently, but as responses to messages from Cocoa.

The architecture I've just described does not perfectly match how AppleScript code works. So to make it match, the Apple folks have interposed a kind of interpreter between Cocoa and your AppleScript code. The interpreter's job is to take care of all the differences between Cocoa and AppleScript, so that you don't have to worry about them. Cocoa sends out a delegation query or a notification message; the interpreter receives this and turns it into AppleScript and routes an appropriate event handler call to your script. Your AppleScript code obtains references to interface elements and targets them with commands, and gets and sets their properties; the interpreter turns this into Objective-C and calls the corresponding built-in method of the relevant Cocoa object. This interpreted linkage between AppleScript and Cocoa is often referred to as a *bridge*, and we say that the Apple folks have bridged AppleScript to (certain parts of) Cocoa.

The first thing to do in learning AppleScript Studio is to consider learning more about Cocoa. After all, AppleScript Studio *is* Cocoa, and even if you'd rather not learn any Objective-C, it can be really helpful to have some familiarity with the location of the Cocoa documentation on your hard drive, and perhaps to read a couple of good introductory Cocoa books. (My favorite is Aaron Hillegass, *Cocoa Programming for Mac OS X* [Addison-Wesley, 2004], 2nd ed.) Furthermore, it's possible that you might end up wanting to learn some Objective-C after all. You can easily find your programming desires thwarted when you encounter an area where AppleScript is *not* bridged to Cocoa. There are four solutions when this happens:

#### *Give up*

If you restrain your desires, your desires can't be thwarted. If something you want to do isn't bridged, stop wanting to do it and confine yourself to what is bridged. This is not an ignoble way out; all programming involves a trade-off of time and effort, and it may be that simplifying your needs is the wisest course.

#### *Use call method*

You can call directly from AppleScript into an Objective-C method with the `call` method command (I'll illustrate its use later on).

#### *Make a hybrid*

It's perfectly possible for some of your code to be written in AppleScript, in a script, and for other parts of your code to be written in Objective-C, in a custom class. In effect, such an application is a hybrid of Objective-C/Cocoa and AppleScript Studio.

#### *Give up, the other way*

Sometimes AppleScript in Cocoa is a square peg in a round hole. Consider learning Objective-C and writing your application in Objective-C/Cocoa. You can

still use AppleScript from within Objective-C (see “Application” in Chapter 2). It may seem paradoxical, but *not* using AppleScript Studio can sometimes be the wisest development strategy.

## The Pieces of AppleScript Studio

AppleScript Studio is like Los Angeles: it isn’t actually anywhere. It isn’t a thing or a place; it’s many tools and resources, used in a certain way. Let’s talk about those tools and resources.

### *The developer tools*

The developer tools, collectively known as the Xcode Tools, are on the Tiger DVD, but they are an optional installation. If you don’t install them, you won’t have any of the AppleScript Studio tools and resources on your hard disk. So install them! An even better approach, as the version on your copy of the Tiger DVD may be outdated, is to obtain the latest version from Apple. First you must join the Apple Developer Connection; the “online” membership level is free (<http://developer.apple.com/membership/online.html>). Then you’ll be able to log in and download the Xcode Tools.

### *Interface Builder*

Interface Builder, an application located (after you’ve installed the developer tools) in */Developer/Applications*, is where you’ll design your application’s interface. It’s worth perusing the Interface Builder Help early in the game.

### *Xcode*

The Xcode application (not to be confused with the Xcode Tools as a whole) is also in */Developer/Applications*. This is where you create and work with a *project* (the collection of files that will be combined to create your application); it’s where you write your code, it’s where you access the files that constitute the project, and it’s where you’ll ask to have your code turned into a real application (called *building* the project). What makes your project an AppleScript Studio application is that you specify “AppleScript Application” when you create the project.

The fact that you design your interface in one application but edit your code and build the application in another is a tricky aspect of the Cocoa development experience, and takes some getting used to. As a beginner, you should work on only one project at a time; while doing so, keep both Interface Builder and Xcode running but hide whichever you’re not using at that moment.

### *Tutorial*

There is a hands-on tutorial at */Developer/ADC Reference Library/documentation/AppleScript/Conceptual/StudioBuildingApps/*. I have reservations about its appropriateness (it employs some techniques I regard as inadvisable), but it’s probably worth going through it, if only as a way of becoming familiar with Xcode and Interface Builder.

## Reference

The most important AppleScript Studio documentation is the reference document at */Developer/ADC Reference Library/documentation/AppleScript/Reference/StudioReference/*. You should skim through this at the outset and then expect to refer to it constantly while working.

## The dictionary

The terminology for the repertory of things you can say in an AppleScript Studio application is defined in a dictionary. (You don't have to target any application in your AppleScript Studio code in order to gain access to this terminology; it is made automatically available.) This dictionary appears in your project as *AppleScriptKit.sdef*; double-click it to view it in a dictionary display. This is essentially the same information as in the reference, but not as well presented, so you're less likely to use it.

## Examples

There are many AppleScript Studio examples located in */Developer/Examples/AppleScript Studio*. They explore and demonstrate most of the important aspects of using AppleScript Studio to drive the user interface; it's very worthwhile to study them.

## Cocoa documentation

There are good links to help you find your way into Cocoa, Objective-C, and Cocoa documentation in an introductory document at */Developer/ADC Reference Library/referencelibrary/GettingStarted/GS\_Cocoa/*. Furthermore, each page about an AppleScript class in the AppleScript Studio reference document is cross-linked to the page about the corresponding Cocoa class; the discussion of the Cocoa class is well worth consulting, as it often explains things better, and frequently has links to even more useful pages on general topics about how things work in Cocoa.

## AppleScript Studio Example

As a tutorial example, let's return to the code presented earlier for searching the TidBITS online archive (in Chapter 25). Recall what it does. We allow the user to enter search terms. We use `curl` to submit those terms to the TidBITS search engine. The reply is a page of HTML listing the pages found. We use Perl to parse the HTML, and present the results to the user as a list. If the user double-clicks a listed article, we present the corresponding page in the web browser. The purpose of the tutorial is to illustrate AppleScript Studio development by wrapping this script in a nice interface. We'll have two windows, a Search window where the user enters search terms and a Results window where the results will be listed.

Begin by creating the project. Start up Xcode. Choose File → New Project and select "AppleScript Application"; in the dialog that appears, name the project *SearchTidBITS* and finish creating it.

We must embed the Perl script into the bundle of the built application. This must be done by the build process, so the Perl script must be incorporated into the project. Select the Resources folder on the left side of the project window and choose Project → Add Files. In the Open File dialog, find and select the Perl script, which is called *parseHTML.pl*. In the next dialog, check the box at the top which asks whether you want to copy the file into the project, and click the Add button.

Now we'll design the interface. In the project window, double-click *MainMenu.nib*. Interface Builder will open. I'm going to skip some details ("drag this interface element from this palette onto this window, resize it, make these settings in the Attributes pane of the Inspector," and so forth) and simply show you the interface design we're going to construct. It consists of two windows and some changes to the menu bar:

- The Search window contains an NSForm displaying three fields the user can search on (text, title, and author) and a Search button, along with a spinning progress indicator to provide feedback while we're talking to the Internet (Figure 27-1).



Figure 27-1. Search window

- The Results window contains a single-column table for displaying the titles of the found articles, along with some explanatory text telling the user what to do (Figure 27-2).
- In the menu bar, I remove the File menu and replace it with a Search menu consisting of a single New menu item; also, I add a Close menu item to the Window menu. Finally, I run through the menu items of the application menu and the Help menu and replace the placeholder "NewApplication" with "SearchTidBITS," the name of our application (Figure 27-3).

Now comes the really interesting part. As you'll recall, the Cocoa framework defines certain action messages, lifetime notifications, and delegation queries that can be sent to your code. In Interface Builder, we must specify which of these messages we want to receive, with respect to each element of our interface as well as

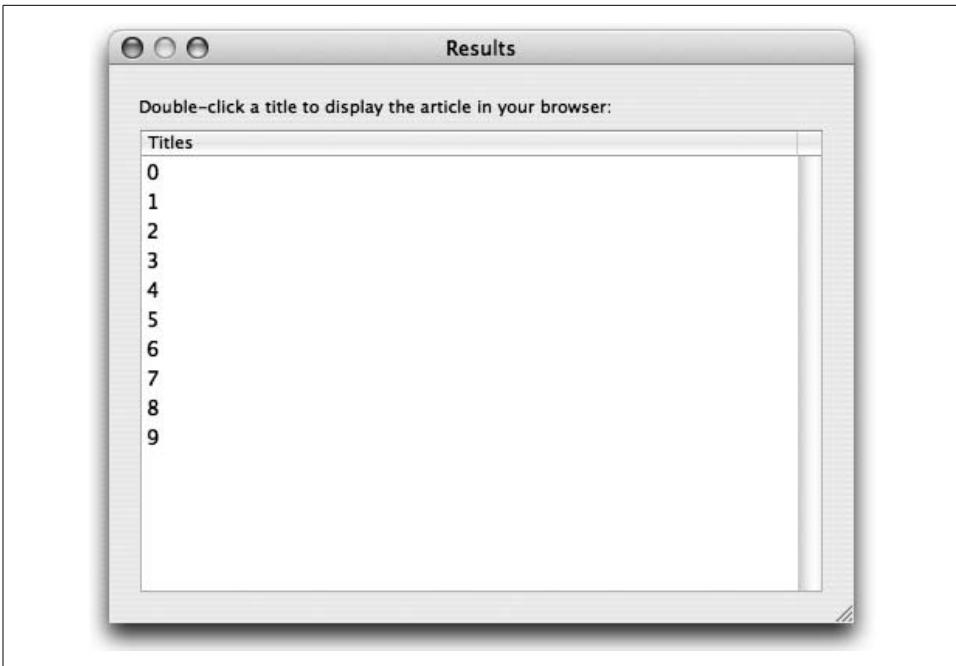


Figure 27-2. Results window



Figure 27-3. The menu bar

the application as a whole. (Remember, we *must* elect to receive some of these messages or our code will never run.) At the same time, we should also give at least some of our interface elements AppleScript names, so that our AppleScript code can refer to them. All of this is done in the AppleScript pane of the Inspector window.

Figure 27-4 shows the process. I've selected the Search window (not shown) and now I focus my attention on the Inspector and its AppleScript pane. In the Name field I've entered an AppleScript name, "search"; this will allow our code to refer to this window by name, as window "search". In the list of event handlers, I've checked the "will open" checkbox; this means that our code will receive a will open event at the appropriate moment in the lifetime of this window (namely, when it is about to open). Somewhat confusingly, it is not sufficient to check this checkbox; it is also necessary to check the Script checkbox specifying the file *SearchTidBITS.applescript*, so that AppleScript Studio knows what script to send the will open event to (even though our project has in fact only one script).

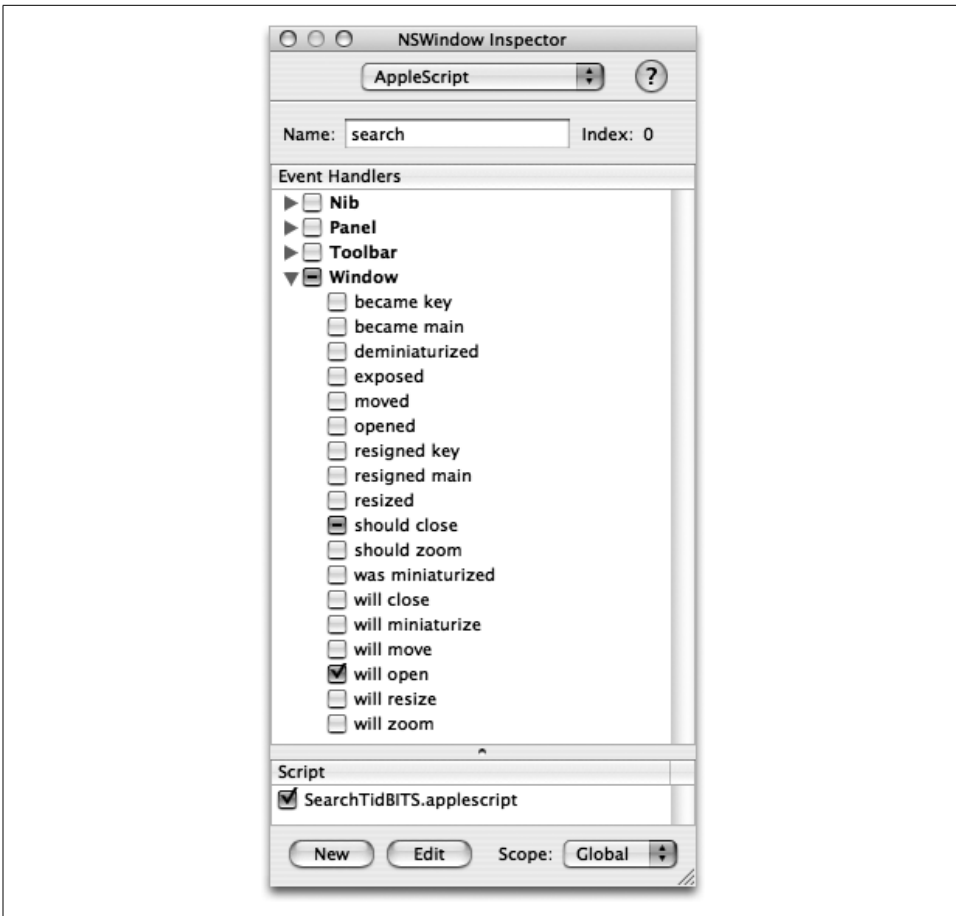


Figure 27-4. AppleScript Inspector for the Search window

Here are the interface items of our project and the treatment I apply to each:

#### *File's Owner*

This icon in the main Interface Builder window represents the application as a whole. I want to know when the application has launched so that I can perform some initializations, so I check `Application: launched` (a lifetime notification).

#### *Search → New menu item*

I want to know when the user chooses this menu item, so I check `Menu: choose menu item` (the menu item's action message).

#### *Window → Close menu item*

I want the frontmost window to close when the user chooses this menu item. I could check `Menu: choose menu item` for this menu item as well, but the knowledge of how to close a window is already built into Cocoa, so there is no need to

involve AppleScript or any code at all. Instead, I form a Cocoa connection. To do so, I Control-drag from this menu item to the First Responder icon in the main Interface Builder window, and connect it to the First Responder's `performClose:` method.

#### *Search window*

I name this window `search`. I want to know when this window is about to open, because I want to make some interface adjustments; so I select `Window: will open` (a lifetime notification).

#### *Search window: the Search button*

I want to know when the user clicks this button, so I select `Action: clicked` (the button's action message).

#### *Results window*

I name this window `results`. I want to know when the user tries to close this window, so I check `Window: should close` (a delegation query).

#### *Results window: the table view*

I want to know when the user double-clicks a row of the table view, so I select `Action: double clicked`.

This completes our use of Interface Builder, so we save our work and switch to Xcode. Find the *SearchTidBITS.applescript* file in the Xcode project window and open it. Templates for the event handlers that we specified in Interface Builder have already been created:

```
on will open theObject
    (*Add your script here.*)
end will open

on launched theObject
    (*Add your script here.*)
end launched

on choose menu item theObject
    (*Add your script here.*)
end choose menu item

on clicked theObject
    (*Add your script here.*)
end clicked

on double clicked theObject
    (*Add your script here.*)
end double clicked

on should close theObject
    (*Add your script here.*)
end should close
```

Now we'll write our script's code, filling in these event handlers and adding any further user handlers of our own. Let's tour the final code a little at a time. In Example 27-1 we declare some top-level globals. We also write code for the launched handler that will be called right after the application has started up; this is the best place for general initializations, which is just what we use it for here, locating the Perl file within our application's bundle and retaining its POSIX pathname as a global.

*Example 27-1. Globals and launched handler*

```
global perlScriptPath, L1, L2, textSought, titleSought, authorSought

on launched theObject
    local f
    set f to (path to resource "parseHTML.pl")
    set perlScriptPath to quoted form of (POSIX path of f)
end launched
```

Example 27-2 shows the will open handler. Observe that these event handlers have a parameter, `theObject`; this contains a reference to the interface element with which this action message, notification, or delegation query is associated. In this case, we know that only the Search window is set to deliver a will open notification, so there is no need to bother checking `theObject`'s identity; we just blithely assume that it's the Search window.

*Example 27-2. The will open handler*

```
on will open theObject
    tell theObject
        call method "setDisplayedWhenStopped:" of progress indicator 1 with parameter 0
    end tell
end will open
```

The code illustrates the use of `call method` to form a manual bridge from AppleScript to Objective-C. We want to make sure that the progress indicator is invisible when not spinning. This setting is available through a built-in Cocoa method, but no AppleScript property is bridged to it, so we cross the bridge ourselves; in effect, we form the equivalent of the following line of Objective-C code:

```
[theProgressIndicator setDisplayedWhenStopped: NO];
```

Example 27-3 shows the choose menu item event handler, which is the action message from the Search → New menu item. We hide the Results window and show the Search window, emptying the form cells and selecting the first cell, ready for the user to enter text. Once again, AppleScript is not bridged to a built-in Cocoa method that we want to call, so we use the `call method` command to call it directly.

*Example 27-3. The choose menu item handler*

```
on choose menu item theObject
    hide window "results"
    tell window "search"
        tell matrix 1
            set content of cell 1 to ""
            set content of cell 2 to ""
            set content of cell 3 to ""
            call method "selectTextAtIndex:" of it with parameter 0
        end tell
    show
end tell
end choose menu item
```

Example 27-4 shows the clicked handler. This is the Search button's action message, and is the heart of our application. To make the code clearer, I've broken the functionality out into some ancillary user handlers. We start by initializing our globals based on what's in the Search window form; then, after a sanity check, we call the next user handler, `doTheSearch`.

*Example 27-4. The clicked handler and an associated utility*

```
on clicked theObject
    tell matrix 1 of window "search"
        set textSought to my urlEncode(content of cell 1)
        set titleSought to my urlEncode(content of cell 2)
        set authorSought to my urlEncode(content of cell 3)
    end tell
    if length of textSought < 5 and length of titleSought < 5 and -
        length of authorSought < 5 then
        beep
        return
    end if
    doTheSearch()
end clicked

on urlEncode(what)
    set text item delimiters to "+"
    return (words of what) as string
end urlEncode
```

Example 27-5 is a utility handler, `feedbackBusy`, purely for manipulating the interface to provide some user feedback. We're going to be talking to the Internet by way of `curl`, and while we're doing this, nothing will be happening. The user might think that the application is idle or broken. Therefore we spin the progress indicator and disable the Search button to give the user a sense that the application is busy and that he should keep his hands off while it does whatever it's doing. The handler is called with a boolean parameter telling whether to begin or end this feedback. Once again

there's a use of `call` method to make up for a deficiency in the bridging: here, we force the window to update its display so that the disabled or enabled Search button will look disabled or enabled.

*Example 27-5. The `feedbackBusy` handler*

```
on feedbackBusy(yn)
    tell window "search"
        if yn then
            set enabled of button 1 to false
            start progress indicator 1
        else
            set enabled of button 1 to true
            stop progress indicator 1
        end if
        call method "display" of it
    end tell
end feedbackBusy
```

Example 27-6 shows the `doTheSearch` handler. This should seem familiar, being nearly unchanged from the code in Chapter 25. The main differences are:

- The `post` argument now incorporates values from the three different form fields the user is allowed to fill out.
- We ask for 2,000 articles instead of 20. The reason is that we're hoping to capture the titles of all the found articles. The search was originally constructed so that its results could be displayed in a web page, where you're supposed to find the first 20 results, then ask for another page showing the next 20, and so forth. I originally thought of trying to emulate this in our application. But then it struck me that we've got this nice scrolling table view to play with, and displaying a large number of titles is no problem, so we may as well gather lots of them in one search and be done with it.
- Interface feedback is provided to the user through calls to `feedbackBusy` before and after the call to `curl`.
- The call to `curl` now has a few more parameters—for example, we provide some timeout values, because the TidBITS search server can be rather slow.
- The intermediary file is now located in the temporary items directory, where the user won't see it (it will be deleted automatically when the user logs out).
- Error handling has been added. It's primitive—if there's a problem, we beep—but this is enough to prevent any mysterious error messages from appearing before the user's eyes. The problem will usually be either that no results were obtained from the search or that the search was never run because we couldn't connect to the server; in real life it might be nice to distinguish these cases and to provide nice error messages, but this is left as an exercise to the reader (meaning that I was too lazy to do it myself).

*Example 27-6. The doTheSearch handler*

```
on doTheSearch()
    local d, f, r
    set d to "'-response=TBSearch.lasso&-token.srch=TBAAdv"
    set d to d & "&Article+HTML=" & textSought
    set d to d & "&Article+Author=" & authorSought
    set d to d & "&Article+Title=" & titleSought
    set d to d & "&-operator"
    set d to d & "=eq&RawIssueNum=&-operator>equals&ArticleDate"
    set d to d & "=&-sortField=ArticleDate&-sortOrder=descending"
    set d to d & "&-maxRecords=2000&-nothing=MSExplorerHack&-nothing"
    set d to d & "=Start+Search' "
    set u to "http://db.tidbits.com/TBSrchAdv.lasso"
    set f to (POSIX path of (path to temporary items)) & "tempTidBITS"
    feedbackBusy(true)
    try
        do shell script -
            "curl -s --connect-timeout 25 -m 120 -d " & d & " -o " & f & " " & u
        set r to do shell script ("perl " & perlScriptPath & " " & f)
        feedbackBusy(false)
        set L to paragraphs of r
        set half to (count L) / 2
        set L1 to items 1 thru half of L
        set L2 to items (half + 1) thru -1 of L
        displayResults()
    on error
        feedbackBusy(false)
        beep
    end try
end doTheSearch
```

If the `doTheSearch` handler doesn't error out, it calls `displayResults` to populate the Results window and present it to the user. Example 27-7 shows the `displayResults` handler, along with two event handlers connected with the Results window. The double clicked event handler responds when the user double-clicks a line of the table of article titles: the corresponding URL is sent to the web browser for display. The should close handler works around a bug in Tiger's version of AppleScript Studio (at least I think it's a bug): a window that might be shown again later must not be closed, as if it *is* shown again later it will malfunction. Therefore when the user attempts to close the Results window we prevent it (by returning false) and hide the window instead; it looks to the user as if the window has closed, but it hasn't.

*Example 27-7. The displayResults handler*

```
on displayResults()
    tell table view 1 of scroll view 1 of window "results"
        set contents to L2
    end tell
    show window "results"
end displayResults
```

Example 27-7. The `displayResults` handler (continued)

```
on double clicked theObject
    try
        open location (item (clicked row of theObject) of L1)
    end try
end double clicked

on should close theObject
    hide theObject
    return false
end should close
```

This completes the development of our application. To test it, choose **Build** → **Build and Run** in Xcode. (Figure 27-5 shows the running application in action; we've performed a search for articles by our favorite author mentioning our favorite subject, and the first article found is being displayed in a web browser in the background.) To prepare your application for public release, choose **Project** → **Set Active Build Configuration** → **Release**; then choose **Build** → **Clean All Targets** and then **Build** → **Build**. The result is a more compact application that will run on other users' machines, with the script saved as run-only to hide it from prying eyes.



Figure 27-5. SearchTidBITS in action

## Automator Actions

An Automator action (see “Automator” in Chapter 2) is an excellent way to wrap some AppleScript code with a lightweight interface. An Automator action is not a standalone application; rather, it is hosted by Automator (or by some other environment that can run Automator workflows). An action typically has no windows or menus; rather, a single pane (technically, an `NSView`) appears as the action’s interface within Automator, and optionally can appear when the workflow runs, as a way of supplying the script with parameters. This isn’t much interface, but in many cases it will be just enough. The script also receives as a parameter the output values from the previous action in the workflow. An Automator action thus gives the end user more power and flexibility than a pure script: the end user can position the action within a larger workflow, and can set options in its interface, effectively repurposing the script and customizing its behavior without seeing or editing its code (and without having to know any AppleScript). To write an Automator action is not difficult, and takes only a few minutes; and it can be done using AppleScript Studio.

Here’s a hands-on tutorial illustrating the process of writing an Automator action. (See */Developer/ADC Reference Library/documentation/AppleApplications/Conceptual/AutomatorConcepts* for Apple’s full documentation.) For our example, we’ll write an action that accepts file aliases and encodes those files as MP3s using `lame` (<http://lame.sourceforge.net>). It is assumed that `lame` is installed in its default location, which is `/usr/local/bin`. In our action’s interface, we’ll provide an option for selecting a preset and a bitrate. MP3 encoding is a time-consuming activity, so our implementation will script the Terminal (rather than calling `do shell script`) so the user can see some feedback as the encoding proceeds.

Start up Xcode. Choose `File → New Project` and select `AppleScript Automator Action`; name the project *LAME Encode*. When the project window appears, choose `File → New File` to create an `AppleScript Text File` called *ui.applescript* to be added to the project. The reason is that we wish to write some code that will interact with the action’s interface, and this code must not go into *main.applescript*, the only script supplied by default.

Now let’s create our action’s interface. In the project window, double-click *main.nib* to start up Interface Builder. The View window displays the pane in which the Action’s interface is to appear. Start by removing the placeholder text (“UI elements go here”). Apple’s interface guidelines for Automator actions specify that interface elements should be Small size rather than Regular, and that the `NSView` should have 10-pixel margins. They also suggest using space economically, and in particular they ask that a popup menu should be preferred to radio buttons.

Figure 27-6 shows our action’s interface. The top row contains the interface elements the user can set. The popup menu contains the names of the most important presets: its items are “insane,” “extreme,” “standard,” “medium,” a menu item separator, then “studio,” “cd,” “hifi,” “tape,” and “mw-us.” The second row contains a text field that we’ll use to show the user the results of the current settings in the top row. We won’t implement VBR bitrates, so the bitrate text field will be disabled unless the “cbr” checkbox is checked. The height of the `NSView` has been reduced as much as possible; vertical space is at a premium when the user is constructing a workflow in Automator, so we don’t want to waste any.

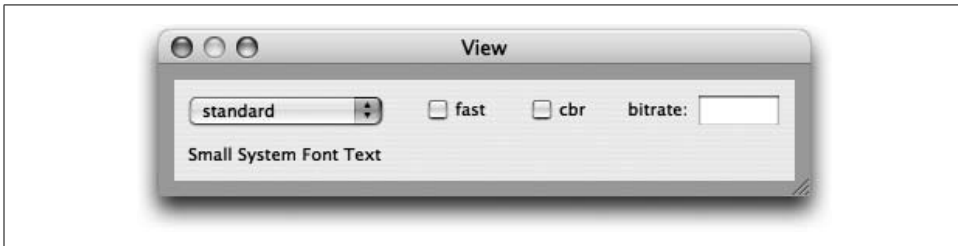


Figure 27-6. The action’s interface

Just as in the AppleScript Studio tutorial, earlier in this chapter, we now give our interface elements AppleScript names and arrange for notification when the user changes one of them. The names are `namePopup`, `fastSwitch`, `cbrSwitch`, `cbrText`, and `example`. The notifications must all be sent to `ui.applescript`; they are the popup menu’s action, the `clicked` of the two checkboxes, and the bitrate text field’s action and `changed`, along with the `NSView`’s `update` parameters and `awake from nib`.

We can use Cocoa bindings to tie our interface items to parameter values that our `main.applescript` code will receive. That’s why the `NSObjectController` called “Parameters” is present in the main window. The use of bindings isn’t compulsory, but in this case we can save ourselves a bit of coding by binding the “cbr” checkbox’s value to a key “cbrSwitch” and the bitrate text field’s value to a key “cbrText.” We can now immediately bind the bitrate text field’s `enabled` to the “cbr” checkbox; thus, the text field will be enabled or disabled automatically as the user checks or unchecks the check box. Also, later on we’ll be able to supply a default value for the bitrate text field without writing any code. (Figure 27-7 shows the bindings for the bitrate text field as displayed in the inspector after creating them.)

This completes the design of the interface, so save and quit Interface Builder. Back in Xcode, now comes the touchiest part of the process—editing the `Info.plist` file. Starting with Xcode 2.1 this task is has been made somewhat easier, because it can be

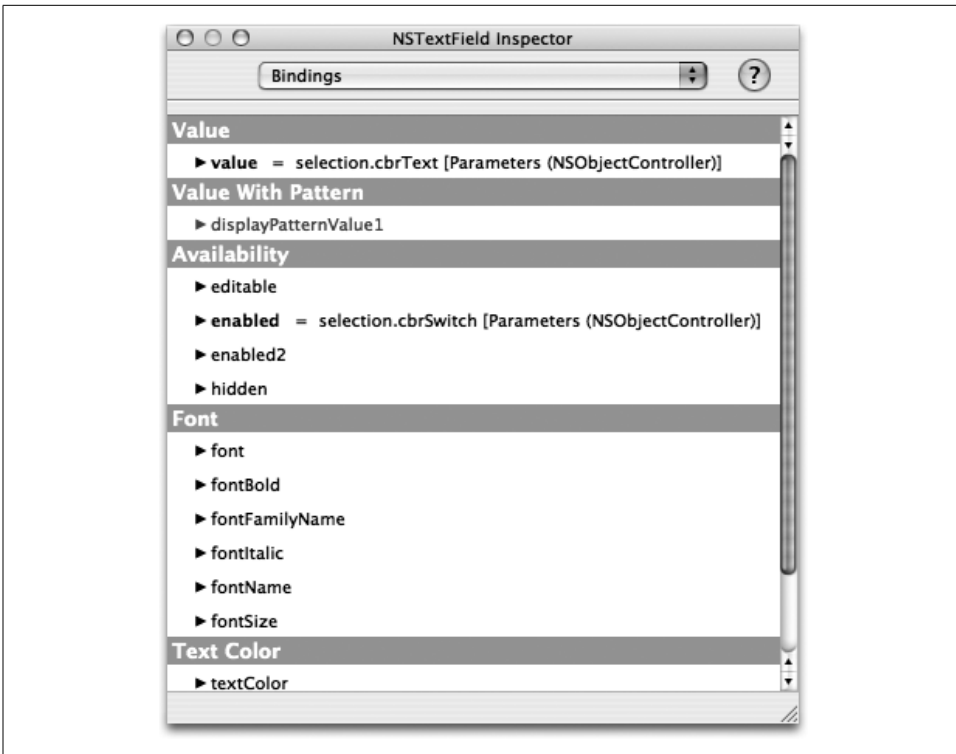


Figure 27-7. Binding an interface element

done mostly in the target’s info window. Choose Project → Edit Active Project to bring up the info window, and switch to the Properties pane; in the lower half is the Collection popup menu, and the idea is to choose each of its items in turn and edit, as necessary, the members of that collection. Here are the settings I’ve elected to use:

Collection	Setting	Value
General	Action name	LAME Encode
General	Application	iTunes
General	Category	iTunes
General	Icon name	iTunes
Parameters	cbrSwitch	Type boolean; value 0
Parameters	cbrText	Type string; value 128
Parameters	exampleText	Type string; value "dummy"
Description	Summary	Encodes files to MP3.

Also, in the General collection, check “Can show when run” at the bottom of the window.

Observe the use of the Parameters settings to supply part of our interface with initial values by way of the bound keys; so, for example, the bitrate field will initially contain “128” because we say so here. It is crucial that the name and spelling for the keys, such as `cbrText`, should match exactly between Interface Builder (where the interface item is bound to that key) and here (where the key’s initial value is given).

You should be wondering what the `exampleText` parameter is for, as we have no interface element bound to this key. This parameter is part of a trick that will be used later to communicate between our *ui.applescript* and *main.applescript* files, so read on and all will be made clear.

So much for the info window, but unfortunately we still need to edit *Info.plist* a bit further by hand. Select *Info.plist* in the project window and choose File → Open With Finder to bring up the file for editing in Property List Editor. (Under no circumstances should you attempt to edit this file by hand as a text file, as you will invariably make a mistake and render the file invalid as a property list.) Open each triangle and select and delete each line that we’ve left as a default comment (such as `AMDInput` and most of the other entries under `AMDescription`). When you’re done with all that, save the file and quit Property List Editor. But, alas, we are *still* not done. We must also edit the localised versions of these settings, in the file called *InfoPlist.strings*. Open this file as a text file and delete everything except the `CFBundleName` and the `AMName`, and save it.

We are now ready, at long last, to write our code. Start with *ui.applescript*. Remember, the purpose of this script is to interact with our action’s interface. The code is almost identical to what we’d put in a standalone application built with AppleScript Studio. I’ll present the code a bit at a time.

Example 27-8 shows start of the code. The very first thing is to capture a reference to the `NSView` that contains the interface elements so that the rest of our code can refer to them. (Unlike our earlier standalone application example, we have no globally available named window, such as `window "search"`, on which to base a reference to an interface element.) We have set the `NSView` to send us an `awake from nib` notification; this notification is guaranteed to arrive very early when our action’s interface loads, and since `theObject` in this case is the `NSView` itself, we simply copy it to a property. To this and all the other action messages and notifications, we respond by calling our `updateInterface` handler; thus our interface will be “live,” updating itself whenever the user does anything.

*Example 27-8. Keeping the interface updated*

```
property theView : missing value
```

```
on awake from nib theObject
    set theView to theObject
    updateInterface()
end awake from nib
```

*Example 27-8. Keeping the interface updated (continued)*

```
on action theObject
    updateInterface()
end action

on clicked theObject
    updateInterface()
end clicked

on changed theObject
    updateInterface()
end changed
```

Example 27-9 shows the `updateInterface` handler. The idea is to construct the `lame` command based on what the user has selected and typed in the interface, displaying this command in the `example` text field in the action's interface. To keep things simple, I have omitted to perform certain validations; in particular, the user can enter anything in the `bitrate` text field. (If the value is an unreasonable number, we use it anyway, and if it's not a number, we treat it as 128.)

*Example 27-9. Constructing the lame command*

```
on updateInterface()
    tell theView
        set s to title of popup button 1
        set s1 to "lame --preset "
        if (state of button "fastSwitch") is 1 then
            if state of button "cbrSwitch" is 0 then
                if s is in {"medium", "standard", "extreme"} then
                    set s1 to s1 & "fast "
                end if
            end if
        end if
        if (state of button "cbrSwitch") is 1 then
            try
                set s to "cbr " & (content of text field "cbrText" as integer)
            on error
                set s to "cbr 128"
            end try
        end if
        set content of text field "example" to s1 & s
    end tell
end updateInterface
```

Now comes the clever part (Example 27-10). We want *main.applescript* to receive, among its parameters, the updated value of the `example` text field, as this is the `lame` command to be sent along to the Terminal. These parameters are supplied as a record whose items are based on the parameters listed in *Info.plist*. That's why we created the `exampleText` parameter in *Info.plist*—to make it appear in the parameters

record. In addition, we have implemented the update parameters event handler, whose purpose is exactly to give us a chance to modify the values in the parameters record. This event handler will be called just before the workflow runs; the record containing the parameters arrives under the name `theParameters`. So we update the interface one last time, and then we modify the `exampleText` item of the `Parameters` and hand back the modified record.

*Example 27-10. Passing a parameter to main.applescript*

```
on update parameters theObject parameters theParameters
    updateInterface()
    set |exampleText| of theParameters to (content of text field "example" of theView)
    return theParameters
end update parameters
```

Now we are ready for *main.applescript* (Example 27-11). This script contains just one event handler, the run handler, which has been created for us. It takes two parameters: `input`, which is the output from the previous step in the workflow, and `parameters`, which contains the bound values from the interface along with the modifications made in our update parameters handler. We assume that `input` is a list of aliases; we convert these aliases to POSIX pathnames. We extract the `exampleText` item from the `parameters` record; this is the `lame` command the user wants us to perform. Now we form a shell command that will loop through each POSIX pathname in turn and hand it to our `lame` command. (The shell is assumed to be `bash`.) We send this shell command to the Terminal for execution. The Terminal will execute this command asynchronously (we have specified that are ignoring application responses), so our handler will end immediately; we must return something to serve as our action's output, even though the action produces no meaningful output, so on the principle of doing least harm we return the same list we received as `input`.

*Example 27-11. The action's main script*

```
on run {input, parameters}
    set myInput to (input as list)
    set L to {}
    repeat with anAlias in input
        set end of L to quoted form of POSIX path of anAlias
    end repeat
    set text item delimiters to space
    set theFiles to L as string
    set s to "arr=(" & theFiles & "); "
    set s to s & "for i in \"${arr[@]}\"; "
    set s to s & "do echo /usr/local/bin/" & |exampleText| of parameters & space
    set s to s & "\"$i\" \"${i%\\.}.mp3\"; done"
    ignoring application responses
        tell application "Terminal" to do script s
    end ignoring
    return input
end run
```

The really cool part is that, because we have allowed it in *Info.plist*, the user can now choose “Show Action When Run” to display our action’s interface in a workflow at runtime (Figure 27-8).



Figure 27-8. Choosing to show the action’s interface at runtime

This gives the user maximum flexibility. Let’s say, for example, that the user creates a workflow consisting of just our LAME Encode action, with “Show Action When Run” checked, and saves it as a Finder plug-in. This means that in the Finder this workflow will appear as a contextual menu item. The user can select some sound files in the Finder and, using the contextual menu, run this workflow. Our action’s interface will appear as a dialog in the Finder! (See Figure 27-9.) The user can then specify the desired settings and continue with the workflow; the files will be converted to MP3 format, with feedback in the Terminal. This illustrates my point about an Automator action having just enough interface.



Figure 27-9. Our Automator action at work

# Cocoa Scripting

Adding scriptability to an application that you write has been, in the past, not a task for the faint of heart. An 'aete'-format dictionary is difficult to create and maintain. On the programming side, the system needs to be able to call into your application when an Apple event arrives, so as your application starts up it must register the appropriate functions with the Apple Event Manager, and when an Apple event arrives, your code must parse it (no mean feat, especially if it involves a reference to an object in your application) and respond appropriately.

For this reason, programmers have often relied on sample code and application frameworks for assistance in making an application scriptable. There was some concern among programmers, therefore, when Mac OS X first emerged, over how it would be possible to take advantage of the Cocoa application framework and make an application scriptable at the same time. Since those early days, support for scriptability has gradually been folded into Cocoa; this is called *Cocoa scripting*. Cocoa scripting is still not perfect, but at least it has passed its infancy, and in Tiger it is easier than ever, thanks in part to the introduction of the sdef-based dictionary.

Thus, if you're a Cocoa programmer, Cocoa scripting in Tiger is a good way to start adding scriptability to your application. Getting started is the hardest part, though, for several reasons:

## *Multiple workplaces*

You have to coordinate the sdef dictionary with your code. If you make a mistake in either of them, or if you cause one of them not to match the other, some aspect of scriptability can fail mysteriously.

## *Scattered documentation*

The documentation is copious, but it's scattered in many different places, and elementary tasks and common problems are often not explained clearly. Also, Cocoa scripting uses "key-value coding," which means that often there is no way to look up a troublesome method in the documentation (because it *isn't* documented, except as a kind of template).

## *Tiresome testing*

Testing is tedious and difficult. Basically, you have to test by scripting your application; you must think of everything an end user might say with AppleScript to your application, and see whether your application responds coherently.

To help you get started with Cocoa scripting, here's a tutorial that adds the rudiments of scriptability to an existing Cocoa application, a little bit at a time. In order to make the example useful, this scriptability will include elements, properties, an

enumeration, and a command. I'll assume you're using Tiger for development, and our example application will be scriptable on Tiger only; once you've achieved Tiger scriptability, it is possible to extend your scriptability to work on earlier systems (I'll talk about how to do that in the next section, "AppleScript Studio Scriptability").

Our Cocoa application, which is called Pairs, is very simple. We have a Person class, and we can create multiple instances of it. A Person has a name. Two Person instances can be paired, and this works like a monogamous marriage: once a Person is united to another, it can't be united to any other Person. The application presumably has some sort of interface, but I'm not going to concern myself with that. We assume that the application's basic functionality is working, and that it initializes itself on startup into some useful state—for example, it creates two initial Persons, named "Jack" and "Jill"—and now we want to go back and make it scriptable.

Here's the structure of the application before we start adding scriptability. The main controller class, instantiated in the nib, is MyObject. Here is its interface:

```
@interface MyObject : NSObject
{
    NSMutableArray* persons;
    NSMutableArray* pairs;
    // outlets go here
}
// method declarations go here
@end
```

The persons mutable array is made up of Person objects; each Person instance, as it is created, is added to this array. Here is the interface for Person:

```
@class Pair;
@interface Person : NSObject {
    NSString* name;
    Pair* pair;
}
- (NSString *)name;
- (void)setName:(NSString *)aName;
- (Pair *)pair;
- (void)setPair:(Pair *)aPair;
// other method declarations go here
@end
```

As you can see, we have accessors for our name instance variable. We also have a pair instance variable in the Person class, plus there is a pairs array in MyObject. It happens that the rest of our implementation is as follows. A Pair has two Person pointers, called person1 and person2. To pair two Persons, we make a new Pair object, add it to the pairs array, and point its two Person pointers at the two Persons; we also point the pair pointer of each of these paired Persons at this Pair object. Thus, a Pair

and its Persons are double-linked; a Person knows it is paired because its `pair` instance variable isn't nil, and it can find the Person to which it is paired by looking at its Pair's Persons and finding the one different from itself.

The question of whether this way of implementing pairings is a particularly good one is beside the point. What's important is that we do *not* intend to expose this to the end user. You don't have to show the user everything that goes on behind the scenes! The end user will be thinking in terms of persons, not pairs, and we want our scripting interface to match the user's conceptual thought processes, not to reveal our backstage implementation.

So how should our scripting interface look to the AppleScript programmer? Clearly there needs to be a person class, and a person should have a name property. There can be multiple persons, so there should be a persons element. This is not a document-based application, so the only coherent location for the persons element is at the top level of the object model—that is, it will be an element of the application class.

Now we'll create our `sdef`-format dictionary and add it to the project. The best way to make the dictionary is with the wonderful Sdef Editor application (see Appendix C for this and other Cocoa scripting resources). Let's call the dictionary `pairs.sdef`. Then to make our application scriptable through this dictionary, we must add the following lines to our project's `Info.plist` file:

```
<key>NSAppleScriptEnabled</key>
<string>YES</string>
<key>OSAScriptingDefinition</key>
<string>pairs.sdef</string>
```

The first step in creating an `sdef` is to give it whatever common commands we intend to implement. For example, we want the user to be able to ask how many persons there are, using the `count` command. This won't be possible all by itself; we have to include the `count` command in the dictionary. Common commands can be found in the Standard Suite (see "Suites" in Chapter 20), which you can access in Sdef Editor by choosing `File → Open Standard Suite → NSCoreSuite`. The idea here is to put `NSCoreSuite` into the dictionary and then immediately remove from it everything we don't need; in this case, the remaining commands will be just `count`, `delete`, `exists`, and `make`—the bare minimum needed for working with a collection of persons. (There is no need to include `get` and `set`, because they are short-circuited, and we don't need an entry for `quit` because every application can do that.)

Now we make a new suite, which I'll call the Pairs Suite. I like to move the application class into this, and I'll simplify the application class, leaving just the `name`, `frontmost`, and `version` properties, which are implemented automatically. Now we can add the person class with its `name` property, and give the application class a person element.

I assume you can figure out how to work with Sdef Editor, so let's focus on the text version of the result. I'll present it in two parts. First we have the automatically generated Standard Suite:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dictionary SYSTEM "file://localhost/System/Library/DTDs/sdef.dtd">
<dictionary title="Pairs Dictionary">
  <suite name="Standard Suite" code="????">
    description="Common classes and commands for most applications.">
      <cocoa name="NSCoreSuite"/>
      <command name="count" code="corecnte" description="Return the number of
        elements of a particular class within an object.">
        <cocoa class="NSCountCommand"/>
        <direct-parameter description="the object whose elements are to be
          counted" type="specifier"/>
        <parameter name="each" code="kocl" description="The class of objects to
          be counted." type="type" optional="yes">
          <cocoa key="ObjectClass"/>
        </parameter>
        <result description="the number of elements" type="integer"/>
      </command>
      <command name="delete" code="coredelo" description="Delete an object.">
        <cocoa class="NSDeleteCommand"/>
        <direct-parameter description="the object to delete" type="specifier"/>
      </command>
      <command name="exists" code="coredoex">
        description="Verify if an object exists.">
        <cocoa class="NSExistsCommand"/>
        <direct-parameter description="the object in question"
          type="specifier"/>
        <result description="true if it exists, false if not" type="boolean"/>
      </command>
      <command name="make" code="corecrel" description="Make a new object.">
        <cocoa name="Create" class="NSCreateCommand"/>
        <parameter name="new" code="kocl"
          description="The class of the new object." type="type">
          <cocoa key="ObjectClass"/>
        </parameter>
        <parameter name="at" code="insh" description="The location at which to
          insert the object." type="location specifier" optional="yes">
          <cocoa key="Location"/>
        </parameter>
        <parameter name="with data" code="data" description="The initial data
          for the object." type="any" optional="yes">
          <cocoa key="ObjectData"/>
        </parameter>
        <parameter name="with properties" code="prdt" description="The initial
          values for properties of the object." type="record" optional="yes">
          <cocoa key="KeyDictionary"/>
        </parameter>
        <result description="to the new object" type="specifier"/>
      </command>
    </suite>
  </dictionary>

```

Next comes the Pairs Suite. The application’s name, frontmost, and version properties were generated automatically; the application’s person element, and the person class itself, were added by you. The material you’ve actually had to create is shown here in bold type:

```
<suite name="Pairs Suite" code="pAIR" description="The Pairs suite">
  <class name="application" code="capp" description="An application's top
    level scripting object.">
    <cocoa class="NSApplication"/>
    <element description="The persons." type="person">
      <cocoa key="persons"/>
    </element>
    <property name="name" code="pname"
      description="The name of the application." type="text" access="r"/>
    <property name="frontmost" code="pif" description="Is this the
      frontmost (active) application?" type="boolean" access="r">
      <cocoa key="isActive"/>
    </property>
    <property name="version" code="vers"
      description="The version of the application." type="text" access="r"/>
  </class>
  <class name="person" code="pRSN" description="A person." plural="persons">
    <cocoa class="Person"/>
    <property name="name" code="pname"
      description="The person's name." type="text">
      <cocoa key="name"/>
    </property>
  </class>
</suite>
</dictionary>
```

Even though only a dozen lines were added by you, there’s lots of room to go wrong here. (I speak from experience.) The following points are worth emphasizing:

*You must make up some four-letter codes.*

The codes 'pAIR' and 'pRSN' are arbitrary. It is crucial, however, that they not overlap with existing four-letter codes. One way to feel confident of this is to use some capital letters, as we’ve done here; Apple reserves to itself all four-letter codes consisting entirely of lowercase letters.

*You must use existing four-letter codes.*

The name property of the person class has code 'pname'. This is a standard property, already defined in AppleScript, and it is essential to get the code right.

*You must match Cocoa keys with your code.*

The Cocoa key for the person element of the application class is “persons”; that’s because persons is the name of the instance variable in MyObject through which the collection of Person objects is accessed. The Cocoa key for the person class is “Person”; that’s because Person is the name of the Cocoa class implementing this AppleScript class. The Cocoa key for the person class’s name property is

“name”; that’s because `name` is the name of the instance variable in `Person` representing the `name` property. A mistake here—even a discrepancy in capitalization between the Objective-C code, on the one hand, and the Cocoa key in the `sdef`, on the other—will cause scriptability to fail.

The reason Cocoa keys in the `sdef` are so crucial is that Cocoa scripting uses key-value coding to find its way through your code. *Key-value coding* (or *KVC*) is an informal protocol that takes advantage of Objective-C’s dynamism and introspection. It uses a string as a key to hunt for names among your instance variables and methods. The object model is navigated by way of a path leading down from the application class. At every step of a path, your classes must be KVC-compliant (meaning that the right instance variables or methods are present) or things won’t work.

In our application, so far, there is just one simple little path. The scriptability framework will start with the application class. It has a `person` element whose Cocoa key is “`persons`.” So the framework looks in `MyObject` to see if it is KVC-compliant with respect to the key “`persons`.” Is it? Yes, because it has an instance variable named `persons`. That instance variable is an `NSMutableArray`; that’s a built-in class which is itself KVC-compliant. The contents of this `NSMutableArray` are `Person` objects; that fits with the Cocoa key for this class, which says that everything in `persons` should be a `Person`. And `Person` is KVC-compliant with respect to “`name`,” because it has a `name` accessor method and a `setName:` accessor method.

We build our application and run it, and point Script Editor at it, to test our scriptability—and it doesn’t work! For example, we say:

```
tell application "Pairs"
    get person 1
end tell
```

and we get an error message in the console:

```
[<NSApplication 0x314220> valueForKey:]: this class is not key value coding-compliant for the key persons
```

The reason is simple: the very first step in the path is incorrectly set up. As our `sdef` says, the application class’s Cocoa key is “`NSApplication`.” But we want the path to start in `MyObject`, not in `NSApplication`. We must not change the Cocoa key for the application class; rather, we need a way to tell the scriptability framework to jump from `NSApplication` to `MyObject` as it descends the path. One very simple way to do this is to make `MyObject` the *delegate* of `NSApplication`; you can specify this by a connection in the nib. We must also write some code in `MyObject` announcing that it, as the application delegate, implements certain keys:

```
- (BOOL)application:(NSApplication *)sender delegateHandlesKey:(NSString *)key {
    if ([key isEqualToString:@"persons"]) return YES;
    return NO;
}
```

We add that code to MyObject, which is now also NSApplication's delegate. We build and run the application, and we test it in Script Editor; lo and behold, it works!

```
tell application "Pairs"
  count persons -- 2
  name of person 1 -- "Jack"
  name of person 2 -- "Jill"
  name of every person -- {"Jack", "Jill"}
  name of every person whose name ends with "k" -- {"Jack"}
  exists person "Jack" -- true
  exists person "Matt" -- false
  delete person "Jack"
  count persons -- 1
  get name of person 1 -- "Jill"
  set name of person 1 to "Mannie"
  name of every person -- {"Mannie"}
end tell
```

This shows the advantage of starting with an application framework. We've added to an existing application no more than a couple of lines of code and a dozen lines of dictionary, and presto, we're scripting our application. We can get and set a property; we can count elements, delete an element, and test by property for the existence of an element; we can even use a boolean test specifier.

Having achieved this initial intoxicating success, we should consider some improvements and refactoring before proceeding any further:

#### *Better accessors*

So far, our code lends itself more or less by accident to KVC. For example, access to the person element is possible only because there happens to be an instance variable called `persons` in MyObject. We should implement our accessors in a more deliberate fashion, in accordance with the expectations of key-value coding and the scriptability framework.

#### *Separate accessors*

It will be wise to separate the scriptability accessors from the programming accessors. For example, when the user changes the `name` property of a person, the scriptability framework is using the `setName:` accessor, which is exactly the same method our own Objective-C code would use to change the `name` instance variable of a Person. But our code might need to respond differently depending on who is calling; our code should be able to do things that a user should not be able to do through AppleScript (think of a read-only property). We should nominate a different Cocoa key in the dictionary and create a different set of scriptability framework accessors.

#### *Separate code*

As long as we're going to have separate accessors, we might want to separate the code that responds to scripting from the code that implements our application's internal functionality. A common architecture is to implement scriptability as an Objective-C category on the existing classes.

### *Add checks and error handling*

At present, our application is very open to the user's commands; for example, a script can delete a person, give two persons the same name, or create a person with no name. We will want to close some of these doors and return a runtime error message to the script when the user tries to do something we disapprove of.

### *Implement objectSpecifier*

Every AppleScript class that our application declares should have in its corresponding Objective-C class an implementation of the `objectSpecifier` method. This is what allows an object reference such as person "Matt" of application "Pairs" to be returned to a script when the user says something like `get person 1` or `make new person`. Without an `objectSpecifier` implementation, the script will receive a meaningless reference.

Here's code that illustrates these points. First, I've changed two of the Cocoa keys in the dictionary: the person element of the application class now has Cocoa key "personsArray," and the name property of the person class now has Cocoa key "personName." These changes will allow our code to respond separately to the messages sent by the scriptability framework. I've moved all the scriptability code into its own file, where it is implemented through categories on the existing classes. I'll present it a piece at a time. First we have a general utility routine implemented as a category on `NSObject`, because every scriptable class will need it:

```
@implementation NSObject (MNscriptability)

- (void) returnError:(int)n string:(NSString*)s {
    NSScriptCommand* c = [NSScriptCommand currentCommand];
    [c setScriptErrorNumber:n];
    if (s) [c setScriptErrorString:s];
}
@end
```

Observe how to return an error to AppleScript: you fetch the pending command and assign it an error number and, optionally, an error message to accompany it. (See Figure 3-1 in Chapter 3; the system holds out the incoming Apple event to your application like an envelope, from which you read the message and into which you insert any response, whether it's a result or an error.)

Next, we have the category on `Person`:

```
@implementation Person (MNscriptability)

- (NSScriptObjectSpecifier *)objectSpecifier {
    NSScriptClassDescription* appDesc
        = (NSScriptClassDescription*)[NSApp classDescription];
    return [[[NSNameSpecifier alloc]
        initWithContainerClassDescription:appDesc
        containerSpecifier:nil
        key:@"personsArray"
        name:[self name]] autorelease];
}
```

```

- (NSString *)personName {return name;}

- (void)setPersonName:(NSString *)aName {
    if ([[NSScriptCommand currentCommand] isKindOfClass: [NSCreateCommand class]])
        [self setName:aName];
    else
        [master scripterWantsToChangeName:aName of:self];
}
@end

```

The implementation of `objectSpecifier` allows proper object references to be returned to AppleScript. We must specify the object’s container, which in this case is the application class, and we must provide a key (“`personsArray`”) matching the Cocoa key in the dictionary for how this element is accessed from that container.

Next we have the scriptability accessors for the name property, now keyed through “`personName`.” A tricky architectural difficulty arises immediately, illustrating why it’s so hard to get started with Cocoa scripting.

In good object-oriented programming, objects are assigned appropriate tasks. Some objects are just data (“model”); other objects control that data (“controller”). In my application, a `Person` in `MyObject`’s `persons` array is just data; it is `MyObject` that should be responsible for creating and validating a `Person`. But key-value coding slams into your existing application like a sudden side wind, ignoring your architecture and surprising your code. When the user tries to change the name of an existing person, it is `Person`’s `setPersonName:` that is called, even though it is `MyObject` that should decide whether the new name is valid. Accordingly, I’ve given `Person` a `master` instance variable pointing at its creator, which in this case is `MyObject`; when the user asks to change a person’s name, the request is shuttled off to `MyObject`, which will decide the suitability of the requested change and comply if appropriate.

But it gets worse. We have an additional problem when the user says `make new person`, because at that moment the scriptability framework creates a `Person` object by calling `alloc` and `init` directly on our `Person` class; any designated initializer is ignored, and `MyObject` doesn’t have a chance to perform initializations or pass judgment. There is no easy way to prevent this (such as saying to the framework, “When you want to create a `Person`, call such-and-such a method”). Furthermore, if the user’s command also says `with properties {name:"whatever"}`, `setPersonName:` is called to set the new name. This puts our code in a quandary; there is no `master`, so there is no one to judge the suitability of the new name.

Fortunately, if the user is creating this person (a condition for which we can test, as the code demonstrates), the scriptability framework will send the resulting `Person` object to `MyObject` anyway, for insertion into the `persons` collection. So we set the name as requested, because `MyObject` will eventually get a chance to pass judgment on this proposed new person—and initialize it properly.

Now let's talk about the category on MyObject. Here's the first part of it:

```
@implementation MyObject (MNScriptability)

- (BOOL)application:(NSApplication *)sender delegateHandlesKey:(NSString *)key {
    if ([key isEqualToString:@"personsArray"]) return YES;
    return NO;
}

- (unsigned int)countOfPersonsArray {
    return [persons count];
}

- (Person *)objectInPersonsArrayAtIndex:(unsigned int)i {
    return [persons objectAtIndex:i];
}
```

First, we have our same old `application:delegateHandlesKey:` method. Next, we've implemented our own access to the `persons` array through the Cocoa key "personsArray"; we will report its size and return an object in it, so that the scriptability framework never gets its hands on the array directly.

Here's more of the MyObject category:

```
- (BOOL) canGivePerson:(Person*)p name:(NSString*)name {
    if (!name || [name isEqualToString:@""]) {
        [self returnError:errorOSACantAssign
            string:@"Can't give person empty name."];
        return NO;
    }
    if ([self existsPersonWithName: name]) {
        [self returnError:errorOSACantAssign
            string:@"Can't give person same name as existing person."];
        return NO;
    }
    return YES;
}

- (void) scripterWantsToChangeName:(NSString*)n of:(Person*)p {
    if ([n isEqualToString: [p name]]) return; // nothing to do
    if (![self canGivePerson:p name:n]) return;
    [p setName: n];
}

- (void)insertObject:(Person *)p inPersonsArrayAtIndex:(unsigned int)index {
    if (![self canGivePerson:p name:[p name]]) return;
    [p setMaster: self];
    [persons insertObject:p atIndex:index];
}

- (void)insertInPersonsArray:(Person *)p {
    if (![self canGivePerson:p name:[p name]]) return;
    [p setMaster: self];
    [persons addObject:p];
}
```

First we have a general name-checking routine. If the user wants to assign a person a name, either as part of creating that person or altering the name of an existing person, we report an error to AppleScript if the name is the empty string or matches that of

an existing person. Then we have the method that will be called by Person if the user tries to change the name of an existing person: either we return an error or we comply by setting the name. Finally, we have two methods that may be called when the user says make new person; I don't actually know which is called when, but it appears they are both needed for KVC-compliance so I've implemented both.

This is indicative of another difficulty that besets the new scriptability programmer. There is no straightforward documentation stating directly what methods will be sought and called, and when, and in what order, and what the scriptability framework will do if it fails to find a method it's looking for (will it default to a different method, or will it throw an error declaring your class not KVC-compliant, or will it return a mysterious error to the script, or what?). So you can never be quite sure what you need to implement and what method the framework may decide to call at any moment. Here you can see me working around this difficulty by implementing the same functionality twice. And I do the same thing in the last part of the MyObject category:

```
-(void)removeObjectFromPersonsArrayAtIndex:(unsigned int)index {
    [self returnError:OSAMessageNotUnderstood string:nil];
}
-(void)removeFromPersonsArrayAtIndex:(unsigned int)index {
    [self returnError:OSAMessageNotUnderstood string:nil];
}
@end
```

That code is to prevent the user from deleting a person. I've implemented two methods that do the same thing because the framework seems to complain on different occasions if I fail to implement either one, and I don't know why; lacking clear documentation, it seems easiest to fall back on a double implementation and move on.

With all of that in place, the previous test script still works perfectly. In addition, our application can now successfully return an object reference; and it now responds coherently to the user's attempts to do things we don't permit:

```
tell application "Pairs"
  get name of every person -- {"Jack", "Jill"}
  delete person "Jack"
  -- error: Pairs got an error: person "Jack" doesn't understand the delete message
  make new person
  -- error: Pairs got an error: Can't give person empty name
  make new person at end with properties {name:"Moe"}
  -- person "Moe" of application "Pairs"
  set name of person "Jill" to "Mannie"
  set name of person 1 to "Mannie"
  -- error: Pairs got an error: Can't give person same name as existing person
  make new person with properties {name:"Mannie"}
  -- error: Pairs got an error: Can't give person same name as existing person
end tell
```

Now let's add some more features. Let's give a person an additional property: gender, which is either male or female. This is simply to illustrate how you implement an

enumeration. The Person class will need an instance variable, `gender`, whose value is an `int`, and we should probably add accessors `gender` and `setGender`. To define the enumeration and its enumerators for AppleScript, you want something in the dictionary like this:

```
<enumeration name="genders" code="gEND" description="A gender." inline="2">
  <enumerator name="male" code="gMAL" description="Male gender."/>
  <enumerator name="female" code="gFEM" description="Female gender."/>
</enumeration>
```

Back in our Objective-C code, we define the same enumeration, like this:

```
enum {
    MALE='gMAL',
    FEMALE='gFEM'
} genders;
```

The match between the four-letter codes in the dictionary and our Objective-C code is crucial. Now, in the dictionary, we add the property to the person class:

```
<property name="gender" code="gNdR" description="The person's gender.">
  <cocoa key="personGender"/>
</property>
```

(Observe that I do not make the common mistake of giving the property the same four-letter code as the class.) The Cocoa key “`personGender`” means that in our Person class the accessors `personGender` and `setPersonGender`: will be called. Implementation of these in the category on Person is straightforward; in my implementation I’ve allowed (and indeed required) the user to supply a gender when creating a person, but I’ve made it illegal for the user to change the gender of an existing person.

Now let’s implement a verb. Let’s call this pair, and we’ll have it apply to two person objects. One will be the direct object; the other will appear after a parameter, to:

```
pair person 1 to person 2
```

Verbs (commands) are implemented in two different ways in Cocoa scripting. If a verb basically applies to a single object, it can appear in the Objective-C code as a method in the class that corresponds to the class of that object. This is called *object-first dispatch*. (The other way of implementing a command, *verb-first dispatch*, is demonstrated later in this chapter.) Having defined the command in the dictionary, you then specify in the dictionary every class that can serve as the direct object to this command. So the dictionary will contain this definition of the command:

```
<command name="pair" code="pAiRpAiR">
  <direct-parameter description="One person." type="person"/>
  <parameter name="to" code="othR" description="The other person." type="person">
    <cocoa key="otherPerson"/>
  </parameter>
</command>
```

And in our person class, the dictionary now contains the following:

```

<responds-to name="pair">
  <cocoa method="scripterSaysPair:"/>
</responds-to>

```

What this means is that when the user invokes the `pair` command, a message `scripterSaysPair:` will be sent to the `Person` object who represents the direct object of the command. The parameter to this method is an `NSScriptCommand` object whose `evaluatedArguments` method yields an `NSDictionary` containing the command's additional parameters, accessible through their Cocoa keys; in this case, there is just one additional parameter, and its key will be "otherPerson." So now we can implement `scripterSaysPair:` in our category on the `Person` class:

```

- (void)scripterSaysPair:(NSScriptCommand*)command {
    Person* p1 = [command evaluatedReceivers];
    Person* p2 = [[command evaluatedArguments] valueForKey:@"otherPerson"];
    if (self != p1 || self == p2) {
        [self returnError:errOSACantAssign string:@"Invalid pairing."];
        return;
    }
    [master scripterWantsToPair:p1 with:p2];
}

```

After an error check, the command is passed on to `MyObject` for processing. The routine in `MyObject` (not shown here) does some more error-checking (making sure neither of the person objects is already paired) and then does whatever it usually does to pair two `Persons`. I use this architecture in order to distribute responsibilities appropriately; a `Person` can look to see whether the `pair` command makes basic sense, but it is `MyObject`, as master of the persons collection, who decides whether two `Persons` can be paired and, if so, pairs them.

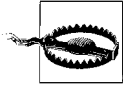
It would be nice to complete the picture by adding a read-only boolean `paired` property to the person class, stating whether the person has been paired, along with a read-only `partner` property that returns a reference to the other person in the pair. These correspond to no instance variable of the `Person` class, which just goes to show that a sensible scripting interface needn't be like the underlying implementation.

```

- (id) personPartner {
    Pair* myPair = [self pair];
    if (!myPair) return [NSNull null];
    return ([myPair person1] == self ? [myPair person2] : [myPair person1]);
}
- (void) setPersonPartner:(id)newPartner {
    [self returnError:errOSACantAssign string:@"Partner property is read-only."];
}
- (BOOL) personPaired {
    return ([self pair] != nil);
}
- (void) setPersonPaired:(BOOL)newPaired {
    [self returnError:errOSACantAssign string:@"Paired property is read-only."];
}

```

The `personPartner` method returns an `NSNull` instance if the person hasn't been paired; that's how you cause `missing value` to be returned to AppleScript. Finally, observe that you must implement setter accessors even though the dictionary marks these properties as read-only.



When the user requests an object that doesn't exist (a person whose index number is too large, for example), an ugly and obnoxious “`NSReceiverScriptError`” runtime error message is returned by the scriptability framework. Various workarounds have been proposed, and I have elsewhere tentatively proposed one of my own, but the truth is that this is a serious flaw in the underlying framework and needs to be corrected at that level.

## AppleScript Studio Scriptability

It is natural to wonder whether an AppleScript Studio application is scriptable. The news here is something of a mixed bag.

AppleScript programmers who are accustomed to writing applets, which are inherently scriptable, will be disappointed to learn that AppleScript Studio applications are not scriptable in quite the same easy way. The mere presence of a top-level entity in an applet's script makes the applet scriptable with respect to that entity, but no such thing is true of an AppleScript Studio application. So, for example, you cannot simply tell our `SearchTidBITS` application to `displayResults()` (see Example 27-7). The problem is that an AppleScript Studio application is not merely an application shell wrapped around a script; it's a true Cocoa application. So your message isn't magically routed to the correct script, because in the way stands the entire mechanism of a Cocoa application.

On the other hand, an AppleScript Studio application is scriptable with respect to the entire *AppleScriptKit.sdef* dictionary, which is actually visible to users though a script editor application as if it were your application's own dictionary. This means that whatever built-in commands you can give from within the code of an AppleScript Studio application, a user can give from outside it. For example:

```
tell application "SearchTidBITS"
    activate
    tell window "search"
        tell matrix 1
            set content of cell 1 to "AppleScript"
            set content of cell 3 to "Matt Neuburg"
        end tell
        tell button 1 to perform action
    end tell
end tell
```

That's exactly the same as if the user had typed values into two of the text fields and then pressed the Search button! Initially this may sound exciting, but most AppleScript Studio programmers ultimately regret that things work this way, for the following reasons:

*It's messy.*

The user who looks at your AppleScript Studio application's dictionary sees the entire confusing *AppleScriptKit.sdef* dictionary, which says nothing as to your application's purpose or what the user can appropriately do when scripting it.

*It's overly free.*

The user can do things to your application that a user really shouldn't be able to do. To give a simple example, if an interface element is tied to a script, that script is available through that interface element; the user can get its script, and even worse, can set its script. For example:

```
tell application "SearchTidBITS"
    script s
    end script
    set the script of button 1 of window "search" to s
end tell
```

That code disables the Search button; its functionality has been replaced, and the only way to restore it is to quit the application and start it up again.

*It's incomplete.*

The user can manipulate the interface, but can't call any event handlers. You'll notice that in the earlier example it was possible to press the Search button programmatically because there is a `perform` action command, but it is impossible to tell the application to clicked:

```
tell application "SearchTidBITS"
    clicked button 1 of window "search"
    -- error: SearchTidBITS got an error: NSReceiversCantHandleCommandScriptError
end tell
```

You might wonder, as you can get the script of an interface element, whether it might be possible to route a message to that script. This is a clever idea, and at first it looks promising:

```
tell application "SearchTidBITS"
    set s to (get script of button 1 of window "search")
    tell s
        urlEncode("hi there") -- "hi+there"
    end tell
end tell
```

It turns out, however, that when you get the script of an interface element, it's a copy. The real script object, the one that the interface element is actually using at that moment, was copied and loaded when the application started up; what you've

got is a different script object, completely unbound from its proper context in the running application—for example, its top-level entity values are not the same as the current top-level entity values of the real script. (This is a serious problem for AppleScript Studio programmers as well, because it complicates communication between scripts and makes the reliable storage of true globals rather an elaborate exercise.)

Thus an AppleScript Studio application is automatically scriptable, but only in a messy, disordered way. On the other hand, because an AppleScript Studio application is a Cocoa application, you might wonder whether you can add customized scriptability to your AppleScript Studio application through Cocoa scripting. You can, although there are two major shortcomings to this approach:

### *Big dictionary*

You can add your own suite and your own terms to the dictionary, but you can't at the same time suppress the default AppleScript Studio dictionary. The user might not even notice your custom scriptability amid the vast wash of confusing and useless information.

### *Communication between languages*

It's obvious how AppleScript can talk to Objective-C classes and instances (using `call method`), but it is far from clear how Objective-C can talk to AppleScript. This is essentially the same problem raised a moment ago; your Objective-C code can't get direct access to the AppleScript script objects that are currently loaded and functioning as the scripts attached to the interface.

With those caveats, it is possible to add Cocoa scripting to an AppleScript Studio application. The procedure is straightforward, except for one thing: you can't use the `sdef` dictionary format to implement your scriptability. To put it technically, if you add the `OSAScriptingDefinition` key to your *Info.plist*, AppleScript Studio itself will break and your application will be stripped of its functionality. Therefore you must implement scriptability the old way, with a resource file. This is not such a terrible thing, as it's what you would have to do in order to implement scriptability for a pre-Tiger application anyway. And besides, you can *develop* your scriptability using an `sdef` file; you simply can't *implement* it with an `sdef` file in the built application. Thus, as the application is built, you must transform your `sdef` file into a different format, one that is compatible with earlier systems; and it happens that there's a Unix tool, `sdp`, that makes this easy to do.

To illustrate, we'll add some basic custom scriptability to the SearchTidBITS application developed earlier in this chapter with AppleScript Studio. The first step is to whip out the Sdef Editor application and create the `sdef` file. Here it is (for brevity, descriptions are omitted):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dictionary SYSTEM "file://localhost/System/Library/DTDs/sdef.dtd">
<dictionary title="Dictionary">
  <suite name="SearchTidBITS Suite" code="sTBs">
```

```

<class name="application" code="capp" plural="applications"
inherits="ASKApplicationSuite.NSApplication">
  <cocoa class="NSApplication"/>
  <property name="search text" code="sTbT" description="" type="text">
    <cocoa key="searchText"/>
  </property>
  <property name="search title" code="sTbI" description="" type="text">
    <cocoa key="searchTitle"/>
  </property>
  <property name="search author" code="sTBa" description="" type="text">
    <cocoa key="searchAuthor"/>
  </property>
</class>
<command name="do search" code="sTIDdSRC" description="">
  <cocoa class="MyScriptCommand"/>
</command>
</suite>
</dictionary>

```

The most important thing here is to hook the application class in this file to the existing AppleScript Studio application class. The lines in boldface are absolutely crucial. Get one of these values wrong and the integration between Cocoa scripting and AppleScript Studio will fail. The four-letter code must be 'capp'; the inheritance must specify “ASKApplicationSuite.NSApplication” as the superclass; and the Cocoa class must be “NSApplication.”

Save the sdef file as *searchTidBITS.sdef*. Open the *SearchTidBITS* project in Xcode and import the sdef file into the project; elect to copy it into the project folder but do *not* add it to the target. We want this file to live in the project folder for development purposes, but we do not want it to be copied into the built application.

What we do want copied into the built application is a resource file containing the dictionary, along with *scriptSuite* and *scriptTerminology* files to implement Cocoa scriptability (see “Dictionary Formats” in Chapter 3). To arrange for this to happen automatically, choose Project → New Build Phase → New Shell Script Build Phase. The info window for the run script phase will appear; in the script field, enter this Unix code:

```

/usr/bin/sdp -fast -o "$BUILT_PRODUCTS_DIR/$FULL_PRODUCT_NAME/Contents/Resources"
"$SOURCE_ROOT/searchTidBITS.sdef"

```

When you build and run the application, you’ll find that it runs normally; if you examine its dictionary in a script editor application, you’ll find that the SearchTidBITS Suite is present and that there are three new application properties (search text, search title, and search author) and a new command (do search).

Now let’s add implementation code. We’ll need a place to put it, so we’ll create a new Cocoa class. (I assume you know how to do this, so my instructions will be very abbreviated.) Open *MainMenu.nib* and, in Interface Builder, create a new NSObject subclass called MyObject, instantiate it, and make the instance the application delegate by making the Cocoa connection between the File’s Owner and MyObject. Now

save MyObject into the project. Save, and quit Interface Builder. Back in Xcode, here's the implementation code for MyObject:

```
@implementation MyObject

- (BOOL) application: (id) sender delegateHandlesKey:(NSString*) key {
    NSLog(@"handles key? %@", key);
    if ([key isEqualToString: @"searchText"])
        return YES;
    if ([key isEqualToString: @"searchTitle"])
        return YES;
    if ([key isEqualToString: @"searchAuthor"])
        return YES;
    return NO;
}

- (NSAppleEventDescriptor*) doAS: (NSString*) s {
    NSAppleScript* as = [[NSAppleScript alloc] initWithSource:s];
    NSAppleEventDescriptor* d = [as executeAndReturnError:nil];
    [as release];
    return d;
}

- (NSString*) searchText {
    NSString* s = @"tell current application "
        @"to get content of cell 1 of matrix 1 of window \"search\"";
    return [[self doAS:s] stringValue];
}

- (void) setSearchText: (NSString*) t {
    NSString* s = [NSString stringWithFormat: @"tell current application "
        @"to set content of cell 1 of matrix 1 of window \"search\" "
        @"to \"%@\"", t];
    [self doAS:s];
}

// ... and so on ...
@end
```

The accessors for “searchTitle” and “searchAuthor” are omitted for brevity; you should be able to write them easily. (They are exactly the same as the accessors for “searchText” except for the names, and except for the cell numbers, which are 2 and 3 respectively.)

This implementation works around the problem of communicating from Objective-C code to AppleScript code by not even trying to do so. Instead, we communicate with the interface. We know that our AppleScript Studio application is scriptable through the native AppleScript Studio commands, so we use them directly, just as we do in our AppleScript code, to drive the interface. We can do this readily; the current application is SearchTidBITS itself, so we are sending a message to ourselves. But this is still a skanky solution: instead of sending a message from one region of code to another, we are using the interface as a kind of drop box. We can't tell our AppleScript code to set its internal textSought, titleSought, and authorSought globals, so we content ourselves with leaving the corresponding values in the interface, where the AppleScript code will find them later.

So now let's tell the AppleScript code to find them, by implementing the `doSearch` command. This command doesn't take a direct object, so we'll implement it using verb-first dispatch. (See the earlier section "Cocoa Scripting" for the other way of implementing a command, object-first dispatch.) It works like this: in the `sdef`, you declare a Cocoa class representing your command; in your project, you create an `NSScriptCommand` subclass with the same name. We've declared in the `sdef` that our class is called `MyScriptCommand`, so now we create that class. In Xcode, choose `File` → `New File`, making the new file an Objective-C class and calling it `MyScriptCommand`. In the header file, make `MyScriptCommand` a subclass of `NSScriptCommand`. In the implementation file, enter this code:

```
@implementation MyScriptCommand

- (NSAppleEventDescriptor*) doAS: (NSString*) s {
    NSAppleScript* as = [[NSAppleScript alloc] initWithSource:s];
    NSAppleEventDescriptor* d = [as executeAndReturnError:nil];
    [as release];
    return d;
}

- (id) performDefaultImplementation {
    NSString* s = @"tell current application "
        @" to perform action of button 1 of window \"search\"";
    [self doAS:s];
    return nil;
}

@end
```

In our override of the `performDefaultImplementation` method, we get to implement our own functionality for this command. Once more we have not tried to solve the problem of communicating from Objective-C to AppleScript; instead, we have again used the interface as a medium of indirect communication. We can't call the `clicked` handler directly, so instead we effectively press the `Search` button, using an AppleScript Studio command that permits us to do so, and this triggers the `clicked` handler already tied to that button.

Even though we can't send messages from Objective-C to AppleScript, perhaps we can improve the way we send messages to the interface. At present we are forming a script as text on the fly and then compiling and executing it. This way is very slow, uses a lot of unnecessary overhead, and may be justly charged with a certain fragility. To give just one example, if the user says `set search text to` with a value that contains a quote character, our `setSearchText:` method will break, because of the blithe way it constructs a literal string. That's clearly a bug.

I will conclude, therefore, by demonstrating a more elegant architecture that uses a compiled script as an intermediary (a "trampoline"). We will call into this compiled script with an Apple event, and the compiled script will send an Apple event to our interface. Apple events are fast, and running a compiled script is fast; it's compiling text on the fly that's slow. And this approach will be immune to the bug with strings

containing a quote, because we will never “unpack” an AppleScript string—we will pass it along directly to the compiled script, and we know that this will work because the string must have been a valid AppleScript string to start with (or we could never have received it in the first place).

Start with a script encapsulating the AppleScript code we’re already using to get and set the contents of a form cell in the Search window:

```
on setCell(n, s)
    tell current application
        using terms from application "Automator"
            set content of cell n of matrix 1 of window "search" to s
        end using terms from
    end tell
end setCell
on getCell(n)
    tell current application
        using terms from application "Automator"
            get content of cell n of matrix 1 of window "search"
        end using terms from
    end tell
end getCell
```

The terms blocks, targeting Automator, are a trick: in order for the central lines to compile, we must resolve them with respect to AppleScript Studio’s terminology; Automator’s dictionary contains this terminology.

Now compile the script (at which point Automator’s task is done, because it is never actually targeted), and save it as *trampoline.sct*. Add it to the SearchTidBITS project so that it will be copied into the built application bundle.

Because we might be using this script any time the user targets the SearchTidBITS application, we’ll save time by loading it once and for all into an `NSAppleScript*` instance variable (called `trampoline`) as our application starts up:

```
- (void) awakeFromNib {
    NSString* path = [[NSBundle mainBundle] pathForResource:@"trampoline"
        ofType:@"sct"];
    NSURL* url = [NSURL fileURLWithPath:path];
    trampoline = [[NSAppleScript alloc] initWithContentsOfURL:url error:nil];
}
```

When the user wants to get or set any of the properties search text or search title or search author, we will call the corresponding handler of *trampoline.sct*, along with appropriate parameter values. We know how to execute a script as a whole in Cocoa using `NSAppleScript`, but how do we call a particular handler, and how do we pass it parameters? The solution is the very same mechanism by which scriptability of user handlers in an applet is implemented (see “Applet Scriptability,” earlier in this chapter)—the `'ascr\psbr'` Apple event. We must form this Apple event more or

less manually, but it isn't hard to do. Here's how. Observe that for the sake of brevity and clarity I've issued a `#define` equating "Desc" to the lengthy term "NSAppleEventDescriptor" which would otherwise clutter up the code.

```
#define Desc NSAppleEventDescriptor
- (Desc*) callSub:(NSString*)handler params:(Desc*)firstParam, ... {
    Desc* list = [Desc listDescriptor];
    int i=0; va_list ppp; va_start(ppp, firstParam);
    Desc* aParam = firstParam;
    while(aParam) {
        [list insertDescriptor:aParam atIndex:++i];
        aParam = va_arg(ppp, Desc*);
    }
    Desc* h = [Desc descriptorWithString:[handler lowercaseString]];
    Desc* ae = [Desc appleEventWithEventClass:'ascr' eventId:'psbr'
        targetDescriptor:[Desc nullDescriptor]
        returnID:kAutoGenerateReturnID
        transactionID:kAnyTransactionID];
    [ae setParamDescriptor:h forKeyword:'snam'];
    [ae setParamDescriptor:list forKeyword:keyDirectObject];
    return ae;
}
- (void) setCell: (int) n toString: (NSString*) s {
    Desc* dn = [Desc descriptorWithInt32:n];
    Desc* ds = [Desc descriptorWithString:s];
    [trampoline executeAppleEvent:
        [self callSub:@"setCell" params:dn, ds, nil] error:nil];
}
- (id) getCell: (int) n{
    Desc* dn = [Desc descriptorWithInt32:n];
    return [[trampoline executeAppleEvent:
        [self callSub:@"getCell" params:dn, nil] error:nil] stringValue];
}
- (NSString*) searchText {
    return [self getCell: 1];
}
- (void) setSearchText: (NSString*) t {
    [self setCell: 1 toString: t];
}
// ...and so on...
```

The `callSub:` routine is a general utility for helping to form the 'ascr\psbr' Apple event. It takes as its parameters the name of the AppleScript handler you want to call, followed by a nil-terminated series of AppleScript parameter values. Each AppleScript parameter value must have previously been embedded into an NSAppleEventDescriptor of the proper type, but this is not usually difficult to do; the `setCell:` and `getCell:` methods exemplify the technique, and show how to call `callSub:`.