AppleScript

The Comprehensive Guide to Scripting and Automation on Mac OS X, Second Edition

Hanaan Rosenthal

Apress®

AppleScript: The Comprehensive Guide to Scripting and Automation on Mac OS X, Second Edition

ISBN-13 (pbk): 978-1-59059-653-1

ISBN-10 (pbk): 1-59059-653-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Chris Mills
Technical Reviewer: Hamish Sanderson
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Dina Quan
Proofreader: Lori Bring
Indexer: John Collin
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.customflowsolutions.com and at http://www.apress.com in the Source Code/Download section.

CHAPTER 2

Scripting—From the Ground Up

OK, all you "been-scripting-for-a-year-and-think-you-can-script-the-New-York-City-traffic-light-system" people, listen up. Although this chapter covers fundamental AppleScript concepts, it was written also for you. Nothing is better than rereading the stuff you think is way behind you. Like any other complex subject, AppleScript has a lot of levels, and to get up to the next level, you sometimes have to start fresh and pretend you know nothing.

Script Concentrate: Just Add Water!

In this section, you'll write a script that contains some of the basic AppleScript constructs. Then you'll run the script and try to understand what happened line by line.

This script will be a bit longer than your usual "Hello World" script; however, once you sift through all the lines, you'll be on top of it! Here's what the script will do: It will get a random number and use it to create a name for a folder. Then it will create the folder in the Finder, name it, and finally delete it if the user clicks Yes in a dialog box.

This section will go over many big AppleScript issues. Don't get stressed if it's too much—just go with the flow, read the text, and do the exercises. All of the points covered here will be repeated in great detail throughout the book.

OK, let's start:

1. Open Script Editor. You can find it in the `AppleScript` folder of your `Applications` folder (unless the system administrator thought it was a game and deleted it in last night's software raid).

   The main window is divided into two parts: the top is the area where you can write your script, and the bottom is divided into three tabs that display various kinds of useful information. The three tabs are Description, Result, and Event Log. Both the Result tab and the Event Log tab are essential for locating problems and solving them. The toolbar at the top of the window contains buttons for basic AppleScript functions: Record, Stop, Run, and Compile. That's the tour for now; let's start scripting.

2. In the blank, new Script Editor window that should be staring you in the eye right now, type the following two-word command, as shown in Figure 2-1:

   ```
   random number
   ```

The text you typed should be formatted with the font Courier. This is because it has not been compiled yet. To compile your script (which will, amongst other things, check whether your syntax is correct), either press the Enter key or click the Compile button.

Once you compile the script, the font, and possibly the color of the text, changes. Suddenly AppleScript understands what you're saying. Well, it understand what you're saying given you speak to it properly, of course.

Figure 2-1. *The noncompiled script in the Script Editor window*

Take a second to understand what happens when you click the Compile button. Before that, you will save the "random number" script as text to the desktop. Saving the script as text does not require you to compile the script. To save the script as text, choose File ➤ Save As, and choose Text from the File Format pop-up menu. We'll return to the "random number" script later.

To better understand the difference between correct syntax that manages to compile and usable code that compiles and runs correctly, try the following:

1. Start yet another script by choosing File ➤ New, and enter the following:

   ```
   It's time to go
   ```

   When you try to compile this script, you get an error. Ignore the specific error, and just note that what you typed didn't adhere to the AppleScript syntax and also isn't defined in any scripting addition installed on your Mac. In other words, if you were in Greece and couldn't speak Greek but tried to say something you just read in the dictionary, the response of the Greek person you were trying to communicate with would be something along the lines of "Huh?"

2. Back to AppleScript now . . . let's delete the words *to go* to end up with this:

   ```
   It's time
   ```

Now the script compiles just fine. I bet you're all excited now—let's try to run it. Now what you said was actually a sentence in Greek, but with no real meaning. The Greek person will now smile at the poor foreigner and move on. Even if the script compiles, it still doesn't guarantee that it'll run to completion!

YOU'RE WELCOME TO ENTER, BUT, PLEASE, DO NOT RETURN

The Enter key on the Mac, to all you Windows migrants, is not the same as the Return key. You use the Return key to start a new line, and you use the Enter key for other tasks, such as compiling a script in Script Editor. For the most part, these keys will be marked Return and Enter. On extended keyboards, the Enter key is usually the right-most key on the keypad.

For the script to run successfully, it has to be a correctly written AppleScript code. Being able to compile a script does, however, ensure that you can save it as a compiled script. If you can't compile it (and there may be other reasons for that), you may have to save it as plain text until you figure out what went wrong.

OK, let's return to the "random number" script:

1. If you closed it, open it; if you want to start over, open a new script window, and enter **random number**.
2. Now run the script to see what happens.

As far as folders suddenly synchronizing, catalogs being created, or your iMac suddenly doing the macarena, I can't say that anything really happened. However, the script did run. Take a peek at the result area. If it's hidden, display the pane by dragging the horizontal divider line with the grab bar up until the pane at the bottom fills about a third of the window, and click the Result tab in the center, if not already selected.

The result area shows you the script's result. And since the result area shows the result of the last expression, the result is the result of that first line. What you get is a decimal number somewhere from 0 to 1 (such as 0.582275391438), which is the result of the `random number` command. In this context, `random number` is also an expression and therefore responsible for the result of the script line. Much like any other language, expressions and results are the building blocks of AppleScript.

Take, for instance, the following situation: My goal is to play squash, and here is how I go about it—I check what day it is, and the result of that expression is "Thursday." I can play on Thursday! I check the time; it's still early enough. I call my buddy, and he can play. Every single action I took had a result, which in some way led to me taking the next action. In this case, the positive result of each step led to the next step, and a negative result would have caused me to stop.

For that to happen, I need to store the results someplace. This "someplace" is another form of expression called a *variable*. A variable is a word you can invent. This word becomes the name of a container, which holds a value that you can use later in the script. A script without variables is like a bucket with a hole at the bottom. You can put a lot in it, but since nothing is retained, you can't utilize in the script information you were given at an earlier point.

With that said, let's put the random number you picked into a variable so you can refer to it later in the script:

1. Change the single line in the script to the following:

   ```
   set my_number to random number
   ```

   The `set` statement assigns the result of the expression, and in this case the `random number` command, into the `my_number` variable. Later in the script you will retrieve the value that is stored in that variable to name a folder.

2. For now, add the number 100 to the end of the line. This will return a random number from 0 to 100. The new line will now look like this:

   ```
   set my_number to random number 100
   ```

   The number that follows the `random number` command is a parameter. The `random number` command has other commands that you'll look at later. The number parameter that follows the command name makes the `random number` command return a whole number (also called an *integer*) from 0 to that number, instead of a decimal number from 0 to 1 (also referred to as a *real*).

3. To move to the next line, press Return, and then type the following:

   ```
   my_number as text
   ```

   This expression converts the number value stored in `my_number` to a text value, a process known as *coercion*. Since the result of the `random number` command is always a number, the new value assigned to the `my_number` variable is a number. This is all OK, but for this script you won't need that number for its numeric value but rather for including in a name of a folder. For that you need text.

   You might be asking yourself why AppleScript should care whether a value is a number or text. Well, AppleScript cares about the type of value far less than most other programming languages, which may make your first few scripts easier to swallow but can create some confusion later when AppleScript's coercion logic doesn't give you the exact conversion you expected. It's better to explicitly specify what you want the *datatype* to be. In fact, since you later attach that random number to the end of a folder name, which is a string, AppleScript will convert the number to text on its own. You will learn much more about that later, so hang on.

4. Now run the script. Before, the result was just a number; now the number appears in double quotes. Double quotes are the distinguishing characteristic of text literals (or, to use a more programming-like term, *strings*).

■ **Note** A *text literal* is a text value you enter directly into the script, and such text is always surrounded by double quotes. Notice step 6 in this exercise: there's text stored in the `my_number` variable, and then there's the text literal `"Folder Number "`. While the script runs, values in variables can be changed by the script, but literal values can't.

The last line, again, needs to be put in a variable for later use. What you'll do now, however, is just put it back in the same variable, which is OK if you don't need the original value. If you did want to use the value later in the script, you would want to store it in a variable whose name tells you what the value is, such as `user_age`, `current_score`, `area_code`, and so on.

■ **Note** Recycling variables is the practice of using a variable for one purpose in the script and then changing the purpose later. Some variable identifiers, such as `counter` or `flag`, are natural candidates for recycling. You need them for a short period of time, and then you use them somewhere else for a different assignment.

---

5. Keep the first line of the script, but change the last line you wrote to this:

   ```
   set my_number to my_number as text
   ```

6. Now you can build the string that will be used to name the folder—type the following new line at the end of the script:

   ```
   set folder_name to "Folder number " & my_number
   ```

   Your script should now look like the one in Figure 2-2.

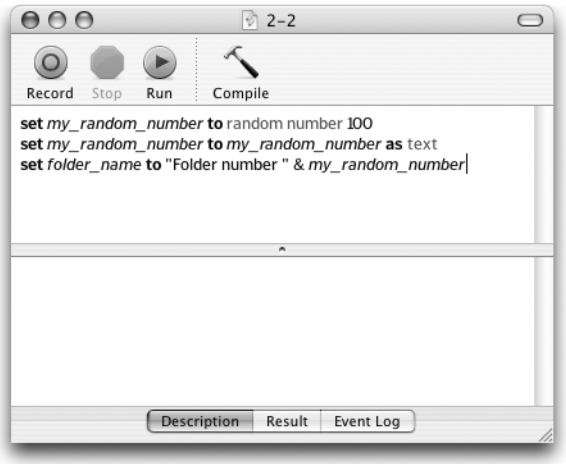

Figure 2-2. *The script so far*

The last statement joined together two values—in this case strings—to produce a new one, a process called *concatenation*. The & symbol, when put between two strings, returns the two strings as a single string. This is an example of an expression, one that includes the concatenation operator.

7. Run the script to see that the result is something like "Folder number 35" (assuming that the random number is 35).

8. Add the following line at the bottom of your script, which will speak directly to the Finder:

```
tell application "Finder" to make new folder at desktop
```

This statement performs two tasks: First, it invokes a command, which is a direct order to make something real happen—to create a folder on the desktop. Second, it also returns an AppleScript result—a pointer to the new folder. This pointer to an item on your disk isn't text or a number, and it is most likely going to be unusable for other applications in its current form. However, it will give you a way to refer to that new folder you just made.

9. Add a fifth line, as follows, to grab that Finder object reference by putting the result of the previous line into a variable:

```
set my_folder to the result
```

■ **Note** The `result` variable is a variable used by AppleScript to store the result of the last command that ran. This value of the result, if needed by the scripter, is usually assigned to another variable as well, making the `result` variable itself redundant.

---

10. Run the script, and see what happens—you should get a result along the lines of the following:

```
folder "untitled folder" of folder "Desktop" of folder¬
 "hanaan" of folder "Users" of startup disk
```

This is what's called a *reference*, which is how you can point to a file when scripting the Finder and other applications. Although this will work now, as soon as you run the line that changes the name of the folder, this reference is useless. It's a bit like telling someone whom you've never met before that you are going to be the guy with the beard and then going and shaving it off! You are still you, but the description you gave is no good.

11. To get a more lasting hold on your file, you can use the `alias` data type. Change the last line to this:

```
set my_folder to the result as alias
```

12. Now run the script again, and you should see the following returned:

```
alias "Macintosh HD:Users:hanaan:Desktop:untitled folder:"
```

Any subsequent time you run the script, the resulting folder's name will be `untitled folder 2`, `untitled folder 3`, and so on. This name difference will be reflected in the result of the script as well. This unique way to point to a file works much like aliases in the Finder: it knows what file you're talking about even if it has been moved or renamed! This will come in handy.

13. Let's talk to the Finder again—add the following sixth line:

```
tell application "Finder" to set the name of my_folder to folder_name
```

What you have done here is to use the two variables you collected earlier in the script, `folder_name` and `my_folder`, to rename the folder you created.

14. Now enter the following line—this seventh line will create a dialog box to ask the user whether they want the folder deleted:

```
display dialog ("Delete new folder "& folder_name & "?") buttons {"Yes", "No"}
```

The `display dialog` command has a few optional parameters. You will use the command's direct parameter to denote the dialog box's text and its `buttons` parameter to specify the button labels. By default a dialog box has two buttons: OK and Cancel.

■ **Note** You put the `"Delete new folder "& folder_name &"?"` operation in parentheses in this exercise. Although you didn't have to, it keeps your code a little more human readable. In other cases, using parentheses will change the outcome of an operation. Also note that the result of the dialog box is a record. We won't get into records now, but for now you just need to know that you can find out what button the user clicked from that record and then use that information elsewhere in your code.

---

15. The next few lines look inside the dialog box result to determine whether the user clicked Yes or No and act accordingly. Add these lines to the bottom of the script:

```
if button returned of result is "Yes" then
   tell application "Finder" to delete my_folder
end if
```

Notice that the alias value stored in the variable `my_folder` knows what folder you mean even though the folder's name was changed earlier. The script should now look like the one in Figure 2-3.

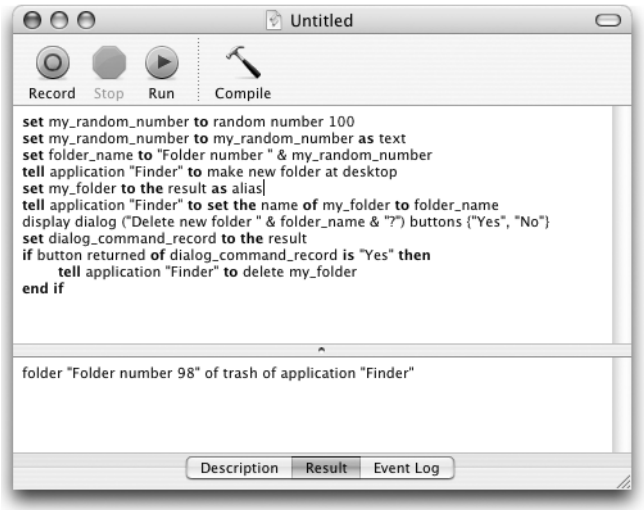

Figure 2-3. *The script so far*

More Results with Less Talk

Let's discuss again the way AppleScript relates to expressions and results. You know that almost every line has a result that can be captured. This is because most lines contain expressions that return results.

A statement can contain a single expression or multiple expressions. For instance, you can write the following:

```
set x to 3 + 4
set y to x * 5
set z to y - 37
set final_number to z / 15
```

or you can make it a more math-like operation and write this:

```
set final_number to (((3 + 4) * 5) - 37) / 15
```

You can perform the same operation by using the `result` variable instead of creating your own variables:

```
3 + 4
result * 5
result - 37
set final_number to result / 15
```

More on Variables

Variables are, in practical terms, the memory of AppleScript. Anytime you want to figure something out and use it later, you assign your conclusions to a variable. You can later retrieve that value, and use it elsewhere in the script, simply by mentioning the variable.

Naming variables logically can save you a lot of frustration later. The variable name is called an *identifier*.

These are the basic rules for naming variables:

- They must start with an alphabetic character or underscore.
- They can contain alphabetic characters (a–z, A–Z), digits (0–9), and underscores (_).

■ **Note** Identifiers are case insensitive; however, once you use a variable in a script, AppleScript will remember how you typed it, and anywhere else that you use it, AppleScript will change the case of the characters in the variable to match the same pattern you used the first time.

Unlike many other programming languages, AppleScript allows you to assign any type of data to any variable. In some languages, when you create a variable, you also tell it what type of data it will hold. In AppleScript, you can create a variable, assign a text value to it, and later replace that value with a number. It makes no difference.

Let's try some variable assignment exercises:

1. Open a new Script Editor window, and type the following:

   ```
   set the_city to "Providence"
   ```

   Since the word `Providence` is in quotes, AppleScript knows it's literal text or a string, not an identifier. The variable `the_city` now has the string value `Providence` assigned to it.

2. Type another line:

   ```
   set the_state to "Rhode Island"
   ```

3. Now you can do something with these variables. Type a third and fourth line:

   ```
   set my_greeting to "Hello, I live in " & the_city& ", " & the_state
   display dialog my_greeting
   ```

   Figure 2-4 shows the script so far.

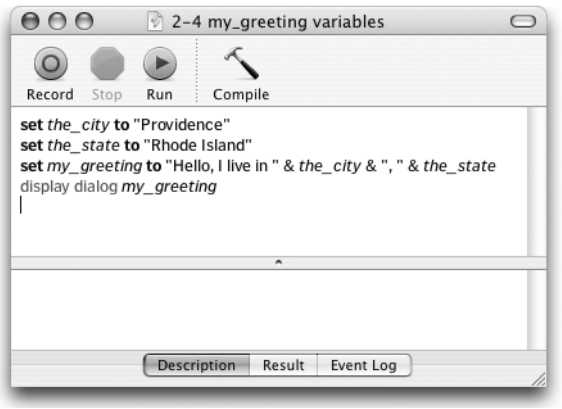

Figure 2-4. *The two variables are assembled into a string that's assigned to a third variable.*

You use the & character to join, or concatenate, the different strings—both the ones that are specified literally and the strings that are stored in the variables. The way in which AppleScript evaluates the concatenation expression is by retrieving the value each variable contains and then passing these values as operands to the concatenation operator (&), which joins the two values to create a new one; the full expression's final result is assigned to a new variable, `my_greeting`.

4. When you run the script, the dialog box displays the final greeting, as shown in Figure 2-5.

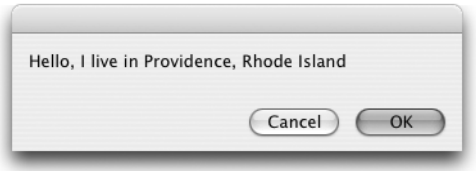

Figure 2-5. *The script's resulting dialog box*

Variables have a scope in which they are visible. If you try to use a variable outside of that scope, what you have is actually two variables that happen to use the same name. Imagine having a co-worker named George; as long as you work at that office, the name George is used to describe your co-worker. If you move to another company and work in another office, the name George may have no meaning at all if no one named George works there. It could also be that there's another George there, so calling George means the wrong person will come. However, if you return to the original office, the name George will again have the old meaning, and the original George still answers to it.

For now, you're not moving to another company, so you have nothing to worry about. But later I will discuss in great detail the implications of using variables in handlers, global and local variables, and properties. As long as a script includes only the `run` handler, you can be sure that using a certain variable name always refers to the same variable.

Values Come in Many Classes

Value classes in AppleScript define the ways in which information is represented. Information may appear as a number, it may be text, it may be either true or false, or it may be a reference to an object somewhere. The value class is what tells AppleScript how to treat the value when it is used in the script. For instance, the value 12 can be a number you use in the script in a math operation, or you can use it as part of a text string, such as "Expire in 12 days." In either case, the value is 12, but in the first case it's a number, specifically an integer (whole number), and in the second case it's two characters: the character 1 followed by the character 2.

The basic value classes are very logical, as you will see in the following sections.

Text

Also called a *string*, text is just that: a bunch of characters strung together. A *string literal* is the written representation of a piece of data, as used in source code. In other words, a string literal is text you actually type in your script; string literals always have double quotes on either side. Here are some string literals:

```
"A"
"My oh my!"
"IT IS SO ST&@%*"
"75"
```

Your scripts can also obtain string values from other sources. Take, for instance, the following statement:

```
set the_time to time string of (current date)
```

In this line of code, the value of the `time string` property of the `date` object returned by the `current date` command gets stored in the `the_time` variable. Although the value stored in the `the_time` variable is a string, its value is assigned while the script is running and therefore isn't shown in the code.

Let's look at the last literal string in the previous examples: `"75"`. Notice that even though you have the number 75, since it is in quotes, it is a string.

To concatenate strings, you use the & symbol. For instance, the result of `"to" & "day"` is `"today"`.

Other types of strings exist, and chief among them is Unicode text, which I will discuss in Chapter 3.

Number

Numbers come in two flavors: integer and real. An *integer* is always a whole round number, such as 1, 52, or 100,000. A real can be either whole or decimal but always has a decimal point. Here are some real numbers: 0.5, 100.1, 0.003, and 20.0.

When performing math operations, you can mix and match reals and integers; however, with the exception of the `div` operator, the result will always be a real:

```
8.5 + 70 = 78.5
1.75 + 1.25 = 3.00
```

Boolean

A Boolean is one of the most often used types of values. A Boolean value can be either true or false:

```
3 = 5 --> false
"BIG" is "big" --> true
disk "Macintosh HD" exists --> true
```

Tell Me About It

The `tell` statement is one of the structures you use to get the attention of the object to which you want to direct commands. Imagine being a fly on the wall in the dean's office at a college. The dean has a helper named AppleScript, who's in charge of making sure the dean's commands get to the right place.

The first command the dean gives is, "Bring me the report by tomorrow." The helper looks at him baffled; something is missing!

Oh, the dean finally gets it, "Tell the student in the second seat in the first row of Ms. Steinberg's class to bring me the report by tomorrow." That's better.

Unless you're using statements that contain only AppleScript commands and objects, you should use the `tell` statement to direct AppleScript to the right object. Simply throwing the command out there or directing it to the wrong object will most likely generate an error.

For instance, if you happen to be scripting Adobe InDesign and you want to get the font of some text, you use the `tell` statement to direct AppleScript to the text frame that is found on the page that is found in a specific document. If you tell the page to get the name of the font, you will get an error.

The Many Faces of tell

This section provides a few variations of how you can use the `tell` statement. They all work the same, but some are better than others in different situations. The main difference between the different ways of using the `tell` statement is in readability versus the number of lines you use, so the decision is left up to you.

So, what are the different ways to `tell`?

Let's return to the dean example. The dean used a long sentence that included the target object (the student) and the command in a single line: "Tell the student in the second seat in the first row of Ms. Steinberg's class to bring me the report by tomorrow."

From that statement, you can start to understand the school's object model, which is essential if it is a scriptable application. The model is as follows: the main object is a school, this object contains the classroom objects, and each classroom has a teacher. You can refer to each classroom by the teacher's name! Also, each classroom has seats arranged in rows, and each seat has a student. Each student has a name, age, and other useful school-related properties. In this case, the main school object contains only one object, but it can contain other objects, such as labs, janitors, gyms, and so on.

Now that you understand the object model a bit better, you realize that the dean could have also said to his helper, AppleScript: "Tell Ms. Steinberg's class to tell the first row to tell the student in the second seat that the report is due . . . ."

Notice how things got reversed? Now, instead of starting from the last object, you start at the top and go down: tell the class, followed by the row, followed by the seat. Before it was the seat that is in the row that is in the class.

How about a real example? Open Script Editor, and enter the script shown in Figure 2-6.

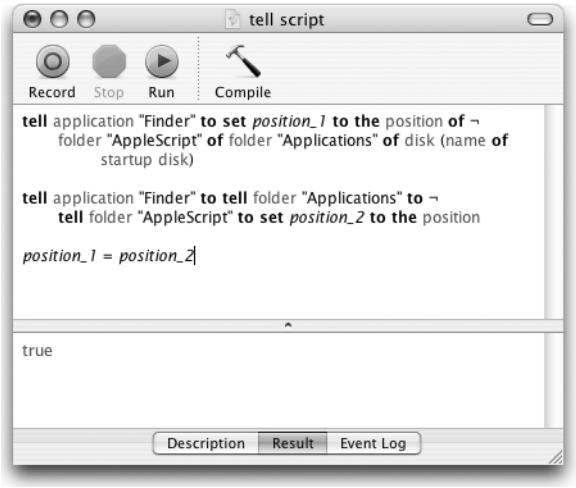

Figure 2-6. *Referencing the same object in two ways*

Look at the two ways you provide a reference to the same folder object. In the first example, you wrote a single, complete reference to the object you want: `folder "AppleScript" of folder "Applications" of startup disk`.

In the second example, you have a series of nested `tell` blocks, each describing part of the reference. As AppleScript evaluates these `tell` blocks, it gradually assembles these partial references into a single, complete reference to the object you want, which again is this: `folder "AppleScript" of folder "Applications" of startup disk`.

To prove that both examples identify the same folder object, run the two examples. The first one opens the `AppleScript` folder in your `Applications` folder, and the second one closes it again.

Objects You Can Tell Things To

Let's return to the main topic here, the `tell` statement. Let's look at some objects that like to be referred to with the `tell` statement.

The main type of object is `application`, an object representing a scriptable application. Anytime you want to send commands to an application, you have to direct them to an `application` object that identifies that application. For convenience, a `tell` block is normally used to identify that `application` object as the target for one or more commands within that `tell` block. Later you will also see how you can load script objects into variables and that the value contained in the variable will become an object that likes to be told things.

Telling in Blocks

The dean scratches his head. He realized that two more students from that same class owe him different reports. Now, when instructing his trusted helper, AppleScript, he may not want to write out the entire reference to a student each time he wants to send that student a command. Instead, he can start with this: "While you're in Ms. Steinberg's class, tell the student in seat 4 of row 2, the student in seat 1 of row 6, and the student in seat 2 of row 3 that they have reports due."

See, when writing scripts, many times you will want to direct more than one command to the same object (or group of objects). What you won't want to do is write the entire object reference (`file 1 of folder 4 of folder Applications`) each time. Instead, you can use a `tell` block.

Here is how: in a `tell` block, you start by identifying the object or objects you want to affect, then you apply one or more commands that will affect these objects, and you finish with the line `end tell`.

For instance, in the Finder example, you have the folder's position. What if you want to get the modification date of the folder as well? Here is how you can do that:

1. You start with the `tell` block. Enter the lines shown in Figure 2-7, and click the Compile button. The second line starts with a double hyphen. This turns it into a comment, and AppleScript ignores it.

Figure 2-7. *A* `tell` *block*

2. Now add the lines shown in Figure 2-8. Notice how each level of the `tell` block starts with the word `tell`, the `tell` block ends with the word `end`, and all the lines in between are indented for readability.

Figure 2-8. *Adding more levels to the* `tell` *block*

3. Now you add statements that collect data from the items inside the folder, as well as from the folder object's own properties. The three lines are as follows:

```
set file_list to name of every item
set time_folder_was_created to creation date
set time_folder_was_modified to modification date
```

Figure 2-9 shows the final script.

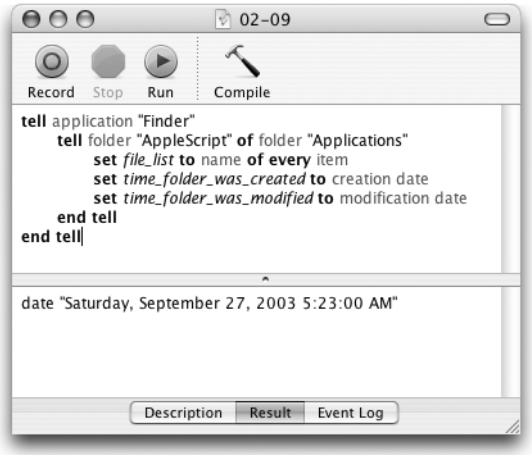

Figure 2-9. *The final script with the* `tell` *blocks*

When you run the script, the result shows only the last line's result; however, the rest of the data you collected is safe and sound in the variables `file_list`, `time_folder_was_created`, and `time_folder_was_changed`. Here are typical values:

```
file_list -->
{"AppleScript Utility.app", "Example Scripts"¬
   , "Folder Actions Setup.app", "Script Editor.app"}
time_folder_was_created --> date "Sunday, March 20, 2005 11:24:53 PM"
time_folder_was_changed --> date "Monday, October 31, 2005 7:23:42 PM"
```

More on Script Editor

One of the AppleScript improvements OS X 10.3 provided was a major update of Script Editor; it has many nice new features that are covered throughout the book. It has a customizable Aqua toolbar that gives you access to the main features and easy access to results, the event log, and their respective histories.

Event Log

You can find the event log by selecting the Event Log tab at the bottom of the Script Editor window. The event log records the commands that AppleScript sends to applications and the result returned for each. In Figure 2-10 you can see the previous script and the log it created when running.

Figure 2-10. *The response of the event log*

Result History

The result history feature keeps track of all script results in chronological order.

Choose Window ➤ Result History, and click the clock-shaped History button. Figure 2-11 shows the Result History window.

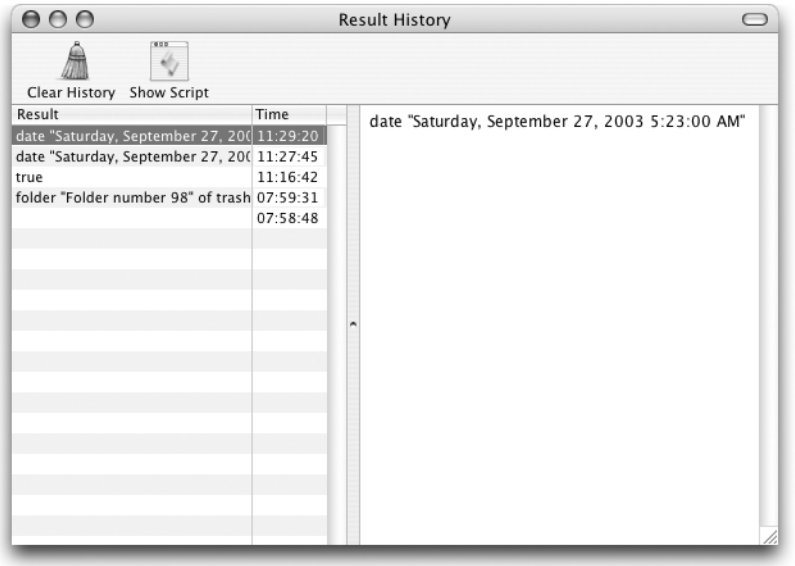

Figure 2-11. *I ran the "random number" script multiple times, and the result history feature captured them all.*

Script Description

Clicking the Description tab at the bottom of the script window in Script Editor reveals a large text field. You can write a description for your script and format it using the Format menu. Change the type size, font, and color as you like.

You can have that formatted text appear in a dialog box every time your script application launches. To display the dialog box at launch, check Startup Screen when you save the script as an application.

Scriptability

Script Editor itself is scriptable. AppleScript can tell Script Editor to create scripts; manipulate text; compile, execute, and save script documents; and more. For example, run the following script; it will create a new script document. Name it My Script, and then compile it and execute it:

```
tell application "Script Editor"
  set my_script to make new document with properties {name:"My Script"}
  set text of my_script to "say date string of (current date)"
  compile my_script
  execute my_script
end tell
```

Contextual Menu

Try Ctrl-clicking the script window to see the contextual menu that will pop up. The bottom of the menu includes folder menus from which you can choose useful scripts that will do everything from inserting `tell` statements, repeat loops, and conditional statements to adding error handlers. You can find the scripts in the menus in this folder: `/Library/Scripts/Script Editor Scripts/`.

Recording Scripts

Here's recording in a nutshell: you click the Record button in Script Editor, you do some stuff in another application, and everything you do is recorded in the AppleScript language.

OK, if that's so easy, why do you need to write a script ever again? Good question.

Although recording is nice, most applications aren't recordable. The recordability feature is rare, since application developers have to put in a lot of extra work to add it.

When you record scripts, you get computer-generated AppleScript code. It doesn't contain any repeat loops, handlers, conditional statements, or easy-to-manage `tell` blocks. For the most part, recording scripts is great if you can't figure out how to script a specific aspect of an application. For instance, in OS X 10.2, the Finder wasn't yet scriptable; in OS X 10.3 and later, the Finder is at least partially scriptable, which can make your life a little easier.

To test it, start a new script window in Script Editor, and click the Record button in the toolbar. Then, create new windows, move them, move files around, and then look at Script Editor to see the recorded actions. As you will see, some basic actions do not get recorded, such as closing windows and moving and duplicating files.

Spaces Don't Count

Extra spaces and tabs will be cleaned up when you check syntax, so don't bother with them. Also, indentation will happen automatically. You can, for readability's sake, leave some blank lines here and there. AppleScript will leave those alone.

To add text that you want the script to ignore, precede it with a double hyphen (`--`). Any text starting after the double hyphen, all the way to the end of the line, will compile as an inline comment.

Adding comments is essential if you or someone else returns to some code written a couple of months ago and wants to figure out how the code works with the aim of adding functionality or fixing a bug. It is amazing how difficult it can be to decipher even your own scripts after a while, after you can no longer remember what the different parts of the script are supposed to do.

Another reason to comment scripts is that when you're creating scripts that are part of a large system, these comments will be a part of your technical specifications. Clients take well to scripters who comment their scripts.

You can comment out whole blocks of text as well. You do that by starting the block with `(*` and ending it with `*)`. Here's an example:

```
(* The following statements identify the files
that are a part of the job handled by the script at the time. *)
```

This type of comment is great for longer explanations you want to add to your scripts. Adding a sort of executive summary at the start of the script is a great way to explain what the code does and how it does it.

■ **Note** The time you spend writing comments will pay for itself many times over when it comes time for you to change the code, especially if some time has passed since you first wrote it.

The Application Scripting Dictionary

The application dictionary describes all the commands and object classes defined by an application. A dictionary is your first stop when discovering how to script a specific application. You can tell whether an application is scriptable or not just by checking to see whether it has a dictionary.

You can view the dictionary for any scriptable application in Script Editor by choosing File ➤ Open Dictionary and then choosing an application from the application list. The dictionary of the selected application will be displayed in the dictionary window. Figure 2-12 shows the dictionary of the Finder.

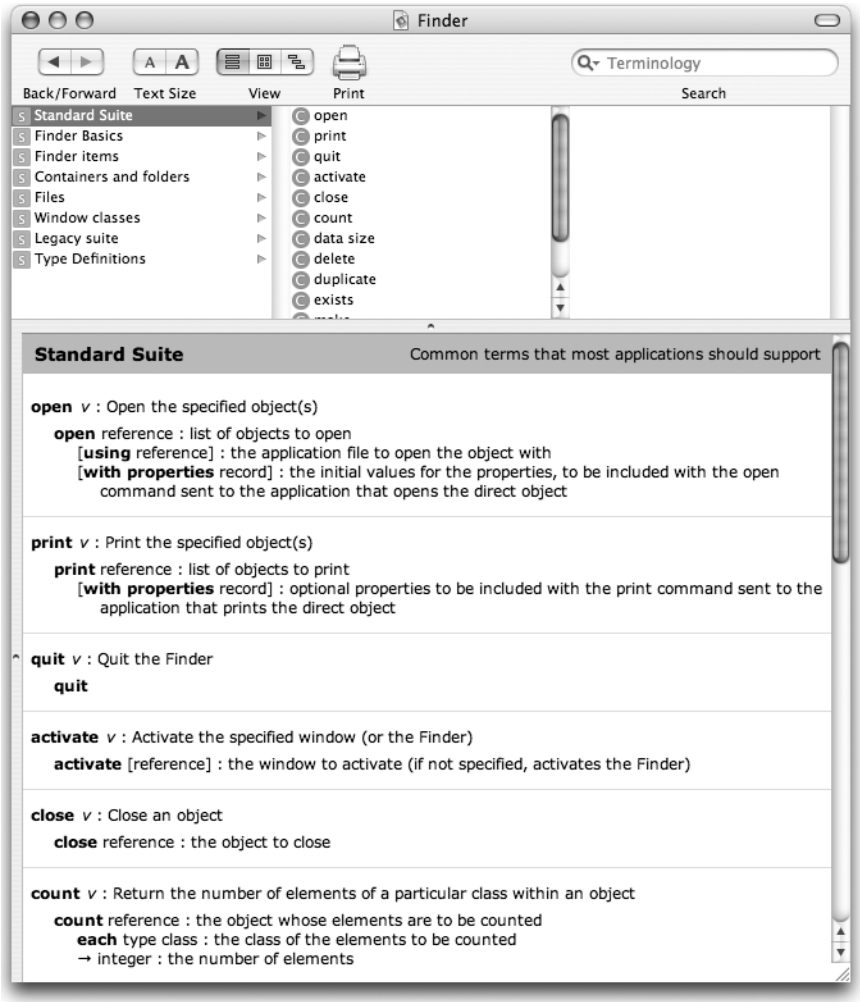

Figure 2-12. *The Finder's dictionary as shown in Script Editor*

■ **Note** You can open an application's dictionary in Script Editor in two other ways. You can drag the application's icon and drop it on the Script Editor application's icon. Alternatively, in the Library panel in Script Editor (Window ➤ Library), you can click the plus button in the Library window's toolbar to add the application to the list, then select the application in the list, and finally click the Dictionary button in the toolbar to display that application's dictionary.

As you can see in Figure 2-12, an application's dictionary is segmented into suites. The suites are organized in a logical way in order to help you find the information you need.

In each suite you will find definitions of both classes and commands—or sometimes just one or the other.

Note that the classes and commands are organized in suites so that the scripter will have an easier time browsing them. The suites don't play any actual role in how scripting actually works.

Classes in the Dictionary

Each object type, or class, is listed in the dictionary under the most fitting suite or is sometimes spread over more than one suite. This listing includes specific details about that class.

Besides the class's name and description, you will find two main types of information for every class. *Elements* are the first type of information. Under the "Elements" heading are the potential elements a class could have once it is an actual object in the application. For instance, among the elements of the `Folder` class are `folder`, `file`, `alias file`, `clipping`, and so on. Disks do not appear as elements of folders since a folder can never contain a disk.

The other type of information listed for a given class is that class's *properties*. Properties can be read using the `get` command and can be changed using the `set` command. Not all properties can be set, however. Properties that are denoted with "r/o" are read-only and therefore can't be changed, only read.

■ **Note** There's much to know about classes and commands that is not explained here. Please read on; it will all make sense in time.

Commands in the Dictionary

The commands in the dictionary appear along with a brief description, which is followed by the command's syntax, which also includes the parameters. The optional parameters appear in square brackets.

The following is an example taken from the Finder's dictionary. The example shows the listing of the `update` command:

```
update: Update the display of the specified object(s)
  to match their on-disk representation
update  reference  -- the item to update
 [necessity  boolean]  -- only update if necessary (i.e. a finder window is open).
     default is false
 [registering applications  boolean]  -- register applications. default is true
```

The description of the `update` command is great. You read it, and you have no doubt about what the command does. The required parameter for the command is the reference to the object you want to update.

The optional parameters are `necessity`, which is explained well, and `registering applications`, which . . . well, I have no idea what it's good for or what it does. The dictionary author either got a bit lazy or simply didn't know the answer either.

At Their Mercy

Although dictionaries are invaluable for any scripter, they are sometimes badly written and incomplete. They don't necessarily lack commands or classes, but many times dictionaries are written as an afterthought; dictionary authors sometimes just do not account for the immense difference a good dictionary can make to the scripter who tries to automate that application. The amount of detail and level of clarity invested in dictionaries are solely up to the dictionary authors, and the range of acceptable detail is high. For instance, when the properties are listed for a specific class, the dictionary usually mentions something in regard to the value that the property accepts. Some properties accept only a list, string, or number; some can accept only specific tokens defined by that application, such as the `owner privileges` property of the `item` class in the Finder. The properties can be `read only`, `read write`, `write only`, or `none`. Although the Finder's dictionary does list these options, some dictionaries lack basic information—something that can make scripting frustrating.

Another fact you can't tell by looking at a dictionary is which classes of objects accept which commands. For instance, the Finder dictionary contains the class `window` and the command `duplicate`, but that doesn't mean you can duplicate a window. How commands and classes interact is information you have to find in other places such as books, Internet lists, sample scripts that come with the application, or just good ol', late-night, trial-and-error experimentation.

The Application Object Model

Perhaps the most compelling part of application scriptability is its object model.

The idea of an object model is that one object can contain a number of other objects, which can themselves contain other objects, and so on. An object that contains other objects is known as a *container*, and the objects it contains are described as being *elements* of that container. The number of elements can be anywhere from zero up.

This type of hierarchy is called a *containment hierarchy*, since it describes which objects contain which elements and which objects contain which elements.

Let's look at InDesign's object model as an example: the InDesign `application` object contains zero or more `document` elements. (As you know, InDesign allows you to have any number of documents open at the same time.) In turn, each `document` object contains its own child elements such as spreads, layers, and so on. Spreads can have many pages, and each page has text frames, guides, and so on.

Like in some other applications, in InDesign you can ask for the object model in reverse as well; the parent of a specific text frame may be a specific page, and so on.

Objects and Class Inheritance

Class inheritance is not the same as containment hierarchy. Inheritance tells you how and where some kinds (or classes) of objects share similarities with other kinds of objects. For example, in the Finder a document file and an application file have many features in common: both have a name, a size, and a creation date (amongst other things). They also have a few differences: opening an application file launches that application, whereas opening a document file opens that document in an application. But overall, objects of these two classes are pretty similar. Other kinds of objects may share few, if any, common characteristics; for example, in Apple iPhoto a photo serves a completely

different purpose than an album does: one is an image, and the other is a container for holding images. Understanding these sorts of relationships is important if you're going to script applications effectively.

To show you how closely different classes of objects are related and what features and abilities they have in common, each application defines an inheritance hierarchy. For example, to show that the `document file` class is related to the `application file` class, the Finder defines an extra class, `file`, that lists all the features they have in common. The `document file` and `application file` classes both then declare `file` as their parent class. This tells you two facts about `document file` and `application file` objects:

- They are all "files" of one kind or another.
- They include (*inherit*) all the attributes declared by the `file` class, as well as any additional attributes declared by their own classes.

In fact, the Finder's inheritance hierarchy goes even further than this because the `file` class itself inherits many of its attributes from the `item` class. The `item` class is also inherited by the `container` class, which defines all the attributes shared by its two child classes: `disk` and `folder`. Figure 2-13 shows the Finder's `item` class inheritance hierarchy diagram.

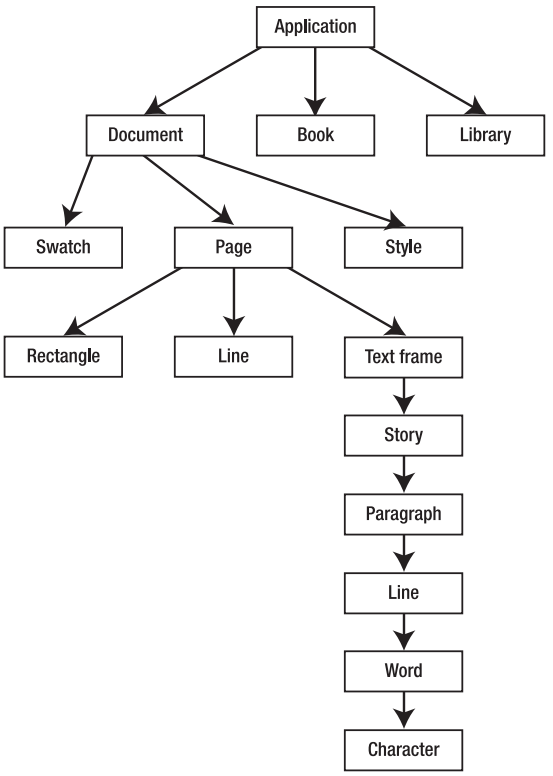

Figure 2-13. *The Finder's* `item` *class inheritance hierarchy diagram taken from Script Debugger 3*

■ **Note** *Superclass* and *subclass* are common synonyms for *parent class* and *child class*, respectively.

Once you've worked out what the Finder's inheritance hierarchy looks like, you can deduce which classes of objects are most similar in nature and will therefore respond to commands in similar, if not identical, ways. You can also determine exactly what properties and elements an object of a particular class possesses when examining the application dictionary. For example, given an object of class `document file`, you'd work your way up its inheritance chain (`document file` ➤ `document file` ➤ `item`), "adding up" all the attributes you find in each.

For example, you can think of dogs as a class. The `dog` class has a few subclasses: `terrier`, `herding dog`, `poodle`, and so on. The `dog` class is also part of another class: `mammal`; this makes `mammal` the `dog` class's superclass. The `mammal` class by itself is a subclass of the class `living-being`. That last class, `living-being`, is the top superclass; any subclass of the `living-being` class by default inherits all of its properties. For instance, all living beings are born, then they live, and then they die. That is true for mammals, dogs, and terriers. However, every member of the `dog` class walks on four legs, which is not true for all members of the `mammal` class, which is is the `dog` class's superclass.

Classes and Commands

Classes also share commands with their subclasses. For instance, the Finder's `item` class understands commands such as `duplicate` and `delete`. That ability to understand these commands also passes to all of the `item` class subclasses such as `file`, `folder`, and so on.

Classes may be a bit intimidating in the beginning, and you may be asking yourself just how essential is it to get that info down. Well, classes are important, and although you should be grateful they exist and they make life more organized, don't kill yourself trying to understand them right now. You can take care of a dog just fine without dwelling on the evolutionary reasons for the existence of its tail.

What Makes an Object What It Is? A Look at Object Properties

Properties are where the relationship between a class of objects and the objects themselves becomes clear. While a class defines all the properties that all objects of that class have, the objects themselves have something the class doesn't: *values*. For example, in the Finder you have the `file` class. That class has a property called `creator type`. The class, however, is not an actual file; it's only the definition of what a `file` object's structure looks like. On the other hand, a file on the hard disk is a real file, and the `creator type` property of the object that represents that file actually has a value, which tells you the `creator type` of that particular file; for example, a value of `ttxt` would tell you the file was a TextEdit document.

Object properties are much like people properties. We all share the same set of properties in the class `human`; it's just that the values are different for each of us. For instance, we all have a `height` property. Each one of us has a `weight` property that we can give to the nurse at our annual checkup. This number is recorded, and then you are told whether you need to lose some weight! In AppleScript, object properties are used in similar ways. Let's look at the files in the Finder window. In the script, you can pinpoint the file you want to deal with and inspect its properties. You figure out that the size property of the file is 2MB. You write that in your script in a variable and later use it to do something useful.

You can see which object properties AppleScript has access to in two ways. One is the application's AppleScript dictionary, and the other is getting the value of an object's `properties` property, if it's available.

When you browse the dictionary in OS X 10.4's Script Editor, you start by clicking a suite name at the left column of the top section, which should reveal all the classes and/or commands defined in that suite. From there, you can click any items from the middle column; classes are marked with the letter *C* in a purple square background, and commands are marked with a *C* in a blue circle. After selecting one class (or Shift-clicking to select several), you will then see the elements and properties of the classes you clicked, one after the other in the bottom section of the dictionary viewer. (Remember, *class* is another word for *object type*, so a class defines all the properties and elements of all the objects that belong to that class.)

Figure 2-14 shows the FileMaker Pro 7 dictionary in Script Editor.

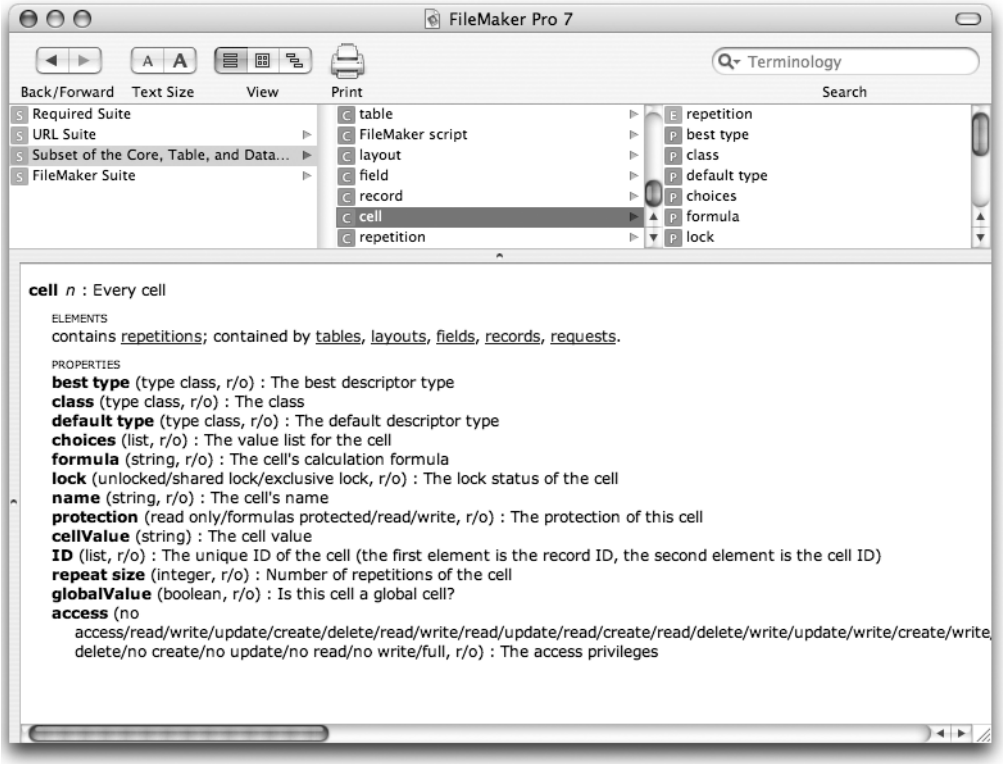

Figure 2-14. *The FileMaker Pro dictionary shown in Script Editor. The* `cell` *class is selected in the top outline portion, and all the* `cell` *class properties appear in the lower area.*

The advantage of using the dictionary to examine a class's properties is that the dictionary often comes with helpful hints. As you can see in Figure 2-14, the FileMaker Pro 7 dictionary has a nice explanation after each property. This can be an invaluable source of knowledge, especially because it is not always obvious what type or range of value the property will agree to use. After every property in the dictionary, either the data type or the actual values you can use are specified. For example, the FileMaker Pro 7 dictionary shows you that the `lock` property of the `cell` class can have one of three values: `unlocked`, `shared lock`, or `exclusive lock`.

Get Properties

Getting the values of an object's properties using the command `get properties` is a bit different. Not only do you get the list of property names, but you also get the value of all or most properties for a specific object.

Let's try to get the properties for a single file in the Finder. You will first need to identify the file whose properties you want to get. To do that, you will use the `choose folder` command, which returns an alias value identifying the folder. After putting that value in a variable, you will ask the Finder to get you the properties of that folder. Create a new script window, and enter the four lines shown in Figure 2-15.

Figure 2-15. *The script and the resulting properties of a chosen folder*

The result of the script is a record that contains about 30 properties associated with the `folder` class and their values that apply to the actual folder you chose.

Although the `get properties of folder the_alias` statement seems like its own little command, it is not. The command is actually the verb `get`, and `properties` is a reference to the property named `properties`. Some object properties return a number, such as `size`, and some return text, such as `name`. The `properties` property returns a record that includes most of the other properties. Looking at the folder properties you got (Figure 2-15), you can see that each property label/property value pair is separated by commas and that the value is separated from the label by a colon. Here is part of it:

```
{name:"first book script", index:3, displayed name:"first book script"
name extension:"", extension hidden:false, ...
```

You can easily tell that the value of the property `name` is `"first book script"`, the `index` is `3`, `extension hidden` is false, and so on.

The `properties` property exists in most objects in many applications, so it's almost a sure bet that you will get the results you want if you try it. All you have to do is make sure you have a valid object reference (see the "How to Talk to Objects So They Listen" section later in this chapter).

Read-Only

As you look at a class's properties in the dictionary, some properties will have "r/o" written next to them. These properties are read-only, which means you can ask to see their value, but you can't change it.

Although initially it appears to be a questionable restriction, some properties were simply not meant to be tampered with or are naturally unchangeable. Take the application Address Book, for example. When you create a new `person` entry, that `person` automatically gets a `creation date` value assigned to it. Although being able to use AppleScript to get the creation date of a person is nice, it would not make any sense if you could change that date; it would no longer be the creation date, but just any date.

If you try to change the value of a read-only property, as in the script shown in Figure 2-16, you will get an error, as shown in Figure 2-17.

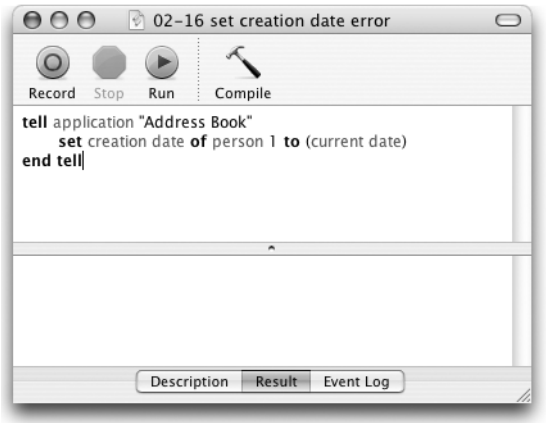

Figure 2-16. *An unsuccessful attempt at changing a read-only property*

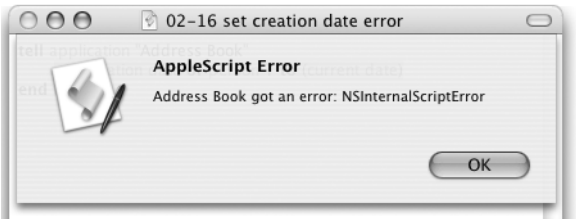

Figure 2-17. *Trying to set the* `creation date` *property of* `person` *generates an error*

How to Talk to Objects So They Listen

As you already know, in order to have your commands obeyed, you must address the recipient correctly. In AppleScript, you have many ways to address objects that will make them eligible to receive and obey your script's commands.

These ways of pointing to an object are called *reference forms*. Mastering reference forms is crucial to the success of your script. If you can't point to the right object using the appropriate reference form, your script will not function as expected, if at all. The coming sections will help you understand reference forms and how to use them.

Let's return for a minute to the dean at the made-up school. The dean's job today is to assign the extracurricular duties to students. Before naming the duties that need to be carried out (the commands), he needs to identify the students who will perform the tasks. Based on what you know so far, he can give tasks to each student individually: tell the student in seat 4 of row 8 of Mr. Popokovich's class that his duty is dishwashing at the cafeteria. Or, in other words: the duty of the student of seat 4 of row 8 of Mr. Popokovich's class is dishwashing. Although this is all good, the poor dean will realize soon enough that in order to name all the students and duties for the day, he will need to spend the entire afternoon assigning duties! He needs a better system.

Index Reference Form

The *index* reference form uses an integer to reference objects. To use the index reference form, you type the element's name followed by the position of the specific element you want. For instance, to get the first file of the second folder, you can type `file 1 of folder 2`. The index reference form uses the order of the elements in the way the application defines that order. In the Finder, for the most part, the files are arranged alphabetically. In page-layout applications, the order of page elements is determined based on their stacking order; the topmost item is always item 1. The order of some objects, such as pages, is easy to determine.

The index reference form is particularly convenient for referring to the frontmost document, since `document` elements are always ordered from front to back. Thus, to address commands to the front document, just direct them to `document 1` (for example, by wrapping them in a `tell document 1` block).

The index reference form works from the back as well by specifying negative numbers. The last item is also item –1, one before last is item –2, and so on.

Another way to use the index reference form is with number words such as `first`, `second`, and `third`, like this: `third application file of last folder`.

Range Reference Forms

The *range* reference form allows you to reference an entire range of elements, instead of one element at a time. You do that by specifying the position of the first element in the range you want, followed by the word `thru`, and then followed by the position of the last element in the range: `pages 3 thru 6`, `files 1 thru 10`, and so on.

Ranges are often used when scripting different aspects of text. The following script extracts a portion of a file's name:

```
tell application "TextEdit"
   set miniaturized of (windows 2 thru -1) to true
end tell
```

This last script will collapse all TextEdit windows other than the front one. Note that the range reference takes place on the second line: `(windows 2 thru -1)`.

Object Names

Many application objects contain a `name` property. The `name` property of the object makes it easier to refer to that specific object. Sometimes it is up to you to name it in the first place; in other instances, such as working with filenames or document names, the objects are already named, and all you have to do is refer to the object by name in your script.

The following example creates a new Script Editor document with a name and then refers to the document and to the window by that name:

```
tell application "Script Editor"
  make new document with properties {name:"Great Script"}
  set contents of document "Great Script" to¬
   "tell application \"finder\" to activate"
  compile document "Great Script"
  set bounds of window "Great Script" to {0, 0, 400, 500}
end tell
```

May I See Some ID, Please?

Although names are useful, some applications provide an additional identifier for some or all of their objects: a *unique ID*.

Unlike naming objects, an object's unique ID is assigned by the application and is a read-only property, which means you can't change it; you can only look at it. That aspect of a unique ID is what makes it so useful: although the name of the object can be changed by either a script or in the user interface, an ID is created automatically when the object is created and stays with the object until it is deleted. On top of that, an object's ID is a read-only property and has no way of being seen through the graphical user interface.

Unique IDs are essential reference tools, since unlike the `name` and `index` properties, they are truly unique. You have to be aware that the value of the `id` property for a particular object may change the next time you launch the application. Different applications may use different ID formats; for example, in InDesign object IDs are a three-digit integer, or more. In Address Book, a person's ID may look like this: `03CEBE56-CA2B-11D6-8B0C-003065F93D88:ABPerson`.

Applications that assign IDs to objects are InDesign, QuarkXPress, FileMaker Pro, Address Book, iCal, BBEdit, and many more. These applications, for the most part, don't assign IDs to objects: Apple Mail, Adobe Illustrator, Adobe Photoshop, Apple Safari, and Microsoft Excel.

■ **Note** Cocoa-based applications will assign an ID to the objects of the `window` class, since this feature is part of the Cocoa framework.

Here's an example of using the `id` property of an element. The following script will create a new person in Address Book, extract that person object's `id` property to a variable, and then use the `id` property to change the `person` object:

```
tell application "Address Book"
  set new_guy to make new person with properties {first name:"Pat"}
  set new_guy_id to id of new_guy
  -- new_guy_id value is something like:
  -- 6C244865-51CC-43F5-A563-ECA2861707FE:ABPerson
  tell person id new_guy_id
     set title to "Ms."
  end tell
end tell
```

every . . . whose

The `whose` clause, also known as the *by-test reference form*, is one of the most powerful programming constructs in the AppleScript language. It gives you the ability to identify elements whose property and/or element values meet certain criteria. Let's break it down. Whenever you examine any group of objects, some of them are similar in different ways. If a folder contains 50 files, 20 of them may be TIFF files, 10 may be aliases, 8 may be larger than 10MB, and some may be older than a month. On an InDesign page, you may have 30 page items: 10 of them may be lines, 22 may have a stroke of 1 point, and 20 may have a blue fill. Notice that I named properties that are unrelated; in the Finder, the same file can be more than 10MB, can be less than a month old, and may or may not be a TIFF file. This way of grouping objects allows you to easily and logically refer to just the objects you want to affect.

The dean, for instance, could have a thought: "I can simply say, 'Every student whose grade average is C or lower has dishwashing duty today.'" Ouch! That is way too mean, but it would have worked.

In some applications, however, you can use an unlimited combination of `whose` clauses and commands. Let's look at a few examples:

```
tell application "InDesign"
  tell page 1 of document 1
     delete every text frame whose contents is ""
  end tell
end tell
```

The entire set you reference here is the set of text frames on page 1. Using the `whose` clause, you can single out the text frames that contain no text and delete them as follows:

```
set documents_folder to path to documents folder from user domain
tell application "Finder"
  duplicate (every file of documents_folder whose size is greater than (2¬
    * 1000 * 1000))
end tell
```

Here, the set of objects is all the files in the user's `Documents` folder. The script wants to duplicate only those files larger then 2MB:

```
tell application "FileMaker Pro"
  tell table  "guests" of database "party"
    set young_kids to cell "name" of every record whose cell "age" is less than 12
  end tell
end tell
```

This script lists the names of all the party guests younger than 12.

When you work with an application that doesn't support the `whose` clause, you have to settle for slower, clumsier repeat loops, performing each test yourself.

Here's another example:

```
tell application "Address Book"
  get name of every person where (email_address is in (value of every email))
end tell
```

This example searches for a person by giving an e-mail address. Notice that e-mails are stored as elements of the `person` object, not a property, so you can't simply write `every person whose email is email_address`. Fortunately, Address Book is pretty clever at interpreting even fairly complex `whose` clause tests, so you can ask it to look through each `email` element's `value` property to see whether the desired address is in any of them. Had this powerful `whose` clause not been available,

you would have had to use a pair of nested AppleScript loops to search through each `email` element of each `person` object one at a time—a rather more laborious and far slower solution than getting Address Book to do all the hard work for you.

Relatives

A *relative reference* identifies an element that is positioned before or after another element. For instance, the reference `paragraph after paragraph 3 of document 1 of application "TextEdit"` requires that `paragraph 3 of document 1 of application "TextEdit"` will be a valid reference in order for it to be valid itself. If there's no paragraph 3, then you can't use the reference `paragraph after paragraph 3` or `paragraph before paragraph 3`.

The following example selects the photo that follows the currently selected photo:

```
tell application "iPhoto"
   try
      set current_selection to item 1 of (get selection)
      select (photo after current_selection)
   end try
end tell
```

■ **Note** The `try` statement was added to trap the error iPhoto generates in case no photos are selected.

Ordinal Reference Forms

Ordinal reference forms allow you to reference certain elements in a natural way.

All Elements in a Collection

In many situations, you will want a reference to all the elements of a particular object, such as all paragraphs on a TextEdit document, all files of a folder in the Finder, or all pages of an InDesign document.

To indicate a reference to all elements, you can use the optional term `every`. Here are some examples that use a reference to all elements of an object:

```
tell application "Address Book"
   set people_list to people -- people is the plural of the person object
   --or:
   set people_list to every person
end tell
```

Here are some other examples:

```
tell application "Finder"
   tell startup disk
      set file_names to name of files of startup disk
      --or:
      set file_names to name of every file of startup disk
   end tell
end tell
```

first, last, and middle

As it sounds, these reference forms identify a single element based on its position in the collection of elements. Here are some examples:

```
tell application "Adobe InDesign CS2"
   delete first page of active document
end tell
```

Here are some examples in string manipulation:

```
middle word of "Application automation rocks!" --> "automation"
```

Power Wrap-Up

In this chapter, you discovered Script Editor and used it to write your first AppleScript scripts. You saw how to create values and assign them to variables. You used a few of AppleScript's built-in commands, including the very useful `display dialog`. You discovered how to send commands to scriptable applications and how to refer to and manipulate objects within those applications. And, finally, you discovered how to define commands of your own.

This was a real whirlwind tour of some of the most interesting and useful areas of AppleScript and application scripting, and it was a lot to master in such a short period of time. But don't worry—you'll slowly work through each of these topics, and a few more besides, in the coming chapters. But right now, just take a deep breath—or ten—and then give yourself a great big pat on the back. Congratulations, you're now officially an AppleScripter!