

## Chapter 1

# A Cannonball Dive into the Scripting Pool

### *In This Chapter*

- ▶ Writing a simple script
- ▶ Writing a more complex script
- ▶ Running a script

**I**n this chapter, you find out how easy scripting is. The scripting basics set the stage and provide a context for the detailed techniques you master in the rest of the book.

## *Writing Your First Script*

Lots of things that seem tough aren't. Nonetheless, many people tend to worry about how hard something is until doing it for the first time. In this section, I banish any scripting worries you may have and show you how to write your first script. After you see how easy writing a script is, you can easily put the rest of the book in perspective. **Remember:** Writing a script is just telling the computer what to do in writing rather than by using a mouse.

### **1. Find and open the AppleScript folder on your hard drive.**

The AppleScript folder is usually stored inside your Applications folder — that's your hard drive:Applications:Applescript:, shown in Figure 1-1.

### **2. From within the AppleScript folder, launch the Script Editor by double-clicking its icon.**

The Script Editor application opens on-screen.

## 12 Part I: Getting Started

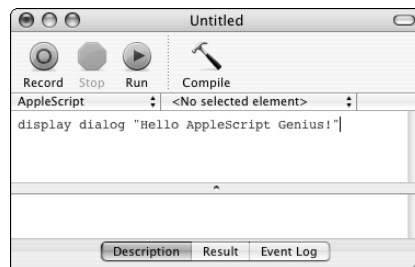


**Figure 1-1:**  
The Script Editor icon inside the AppleScript folder.

### 3. Type the following line into the script window.

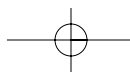
```
display dialog "Hello AppleScript Genius!"
```

The text appears just like in a word processor, as shown in Figure 1-2.



**Figure 1-2:**  
Typing a simple script.

Even though you may have never used AppleScript before, I bet you have a good idea what this script does, don't you? That's because AppleScript reads a lot like English. So when you run into some complex script, just take a deep breath and say, "It's like English. It's like English." If you want to see how the script works in detail, just keep reading. By the way, if the fonts and sizes look different in your version, don't panic. You can customize them, which I discuss in Chapter 3.



While AppleScript looks like English you'll find that sometimes the syntax (grammar) doesn't sound right. Because computers aren't particularly smart, scripts often require what civilized societies call atrociously bad grammar.

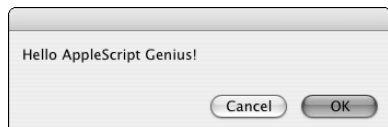
## *The first, and in this case only, script line*

Take a closer look at your first attempt at AppleScript writing:

```
display dialog "Hello AppleScript Genius!"
```

The `display dialog` part is an AppleScript command that tells AppleScript to put up a fairly standard — albeit highly customizable, as you see in the next example — dialog. The quote — called a *string* in AppleScript — that follows the `display dialog` command is the message that appears in the dialog. Figure 1-3 shows the dialog that results from AppleScript running this script line.

**Figure 1-3:**  
Welcome to  
the world of  
scripting!



After you type the script, click the Compile button in the Script Editor, which *compiles* your script — compiling is converting the AppleScript you write into a language the Mac understands. When you click the Compile button — it looks like a handy little claw hammer — AppleScript reads what you typed and checks to make sure the script is understandable. If you made a mistake — not that you, the customer, ever does, mind you — you get a little dialog telling you there was a problem. If you get such a message, just retype the script from scratch.



When you get more experienced, you can figure out what part of a line may be causing a problem, so you don't have to retype a whole line.

If AppleScript understands what you type, it reformats the script slightly, changing the styles of some words — `display dialog` is purple before compiling and blue afterwards, for example. You can find out how to select these styles in Chapter 3.

## 14 Part I: Getting Started



Just because AppleScript understands a script doesn't mean the script does what you want it to. If you mistakenly tell the script to delete all your files, but you enter the script correctly so that AppleScript understands it, AppleScript acknowledges that it understands your script even though you told it the wrong thing to do. AppleScript doesn't know what you want; only what you type.



**compile:** This term doesn't refer to building an evidentiary base so that you can sue your neighbor for letting his dog eat your geraniums. When you click the Compile button, AppleScript compiles your script into something that the Mac can understand. Computers are nice, friendly creatures, but they're pretty deficient in the language arts. They understand 0s and 1s. In the early days of computers, people had to speak the computer's language in order for the computer to understand what the heck they wanted. That's why old programmers — those over 30 — like to speak in terms of hex numbers — base 16, not witchcraft — or binary numbers such as 0 or 1. The closest most people get to that these days is Windows. (Just kidding.) Anyway, when AppleScript compiles your script, it makes a version of it, which you don't see, in a language your friendly but linguistically challenged Mac can handle. Unfortunately, AppleScript can't make the Mac understand what you mean when you type something incorrectly just by raising the speaker volume.

All you have to do to run your script is to click the Run button. AppleScript displays the dialog (refer to Figure 1-3).

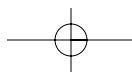
You can display different messages by changing the text in your script that appears between the two quotation marks. Pretty straightforward stuff, isn't it?

## Writing a Second (And Even Better) Script

After you accomplished a basic script, you're ready to take on a fairly complex script. You soon see that, because scripts look a lot like English, you can understand what they do. You may spend some time — and read the next part of this book — before you can write your own scripts, but you'll find that you don't need to be intimidated by scripts. With a little effort, you can understand what the script is doing even if you can't write the script yourself.



One of the best ways to increase productivity is to start with a script someone else has written and modify it. You can find various Web sites — see Chapter 29 for a list of resources — that have example scripts online. In addition, OS X comes with a bunch of scripts installed in the same folder as the Script Editor (Applications:Applescript:Example Scripts). Even if you're not 100 percent sure of why a script works in a certain way, you can understand enough



## Chapter 1: A Cannonball Dive into the Scripting Pool

# 15

to modify it to do what you want. This method of scripting saves you time and effort.

This next script is a slightly more sophisticated version of the first “Hello AppleScript” script.



One thing to watch out for in this sample script is the funny `~` character — called a soft return in the trade — which tells AppleScript to treat the stuff on the next line as though it’s part of the current line. It’s a carriage return for you but a nothing for AppleScript when it reads the line. You don’t need to use the `~` symbol; you can just use wide windows or the scroll bar. I’m using it in this book so that long script lines don’t require a 15-inch page. You get the `~` by pressing Option+Return. You only need to use Option+Return in Script Editors for versions of OS prior to 10.3.

```
set user_stuff to display dialog ~
    "My name is AppleScript. What's yours?" default answer ~
    "Who Knows" buttons {"Buzz off", "Hi"} default button "Hi"
set button_name to button returned of user_stuff
if button_name is "Hi" then
    set your_name to text returned of user_stuff
    display dialog "Welcome to AppleScript, soon to be Dr. " &
        your_name & ~
        ", expert AppleScript genius." buttons {"Howdy"}
        default button "Howdy"
else
    display dialog "Sorry you don't want to be friends."
        buttons {"For sure"} ~
        default button "For sure"
end if
```

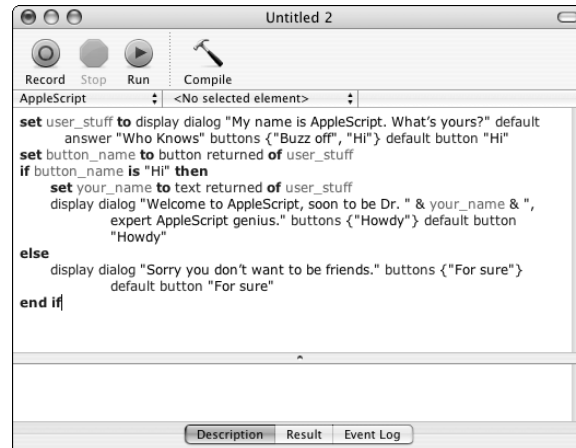
Enter the script into the Script Editor window, making sure you either press the Return key at the end of each script line or place a `~` character (Option Return) where you want the line to break before its “real” end. (Turn to the “Writing Your First Script” section if you need a refresher on entering scripts.) When you’re done, it looks like Figure 1-4 after you click the Compile button. If you get an error message telling you that AppleScript doesn’t understand something, compare what you typed with the preceding script. One common mistake is to press Return rather than Option Return inside a script line. You can check for that mistake by making sure that the funny `~` characters in your script are in the same places as in the sample script.



If you put an `~` inside a quote — a *string* for you technical types — everything works, but the `~` shows up when the text displays, which probably isn’t what you want. So avoid putting an `~` inside a quote for aesthetic reasons.

I bet by just reading through this script, you have a fairly good idea of what it does. If you’re not 100 percent sure, take a stroll through the following sections.

## 16 Part I: Getting Started



**Figure 1-4:**  
Your second  
script after  
it's compiled.

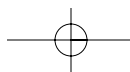
### *Line 1: Displaying a dialog*

```
set user_stuff to display dialog
    "My name is AppleScript. What's yours?" default answer
    "Who Knows" buttons {"Buzz off", "Hi"} default button "Hi"
```

The first thing to remember is that this command is all one line as far as AppleScript is concerned because even though it is on three lines, the Script Editor autowraps the line. This line is just a slightly more complicated version of the `display dialog` command. Ignore the `set user_stuff` to for a moment. The first item after the `display dialog` is a quote — "My name is AppleScript. What's yours?" — which displays in the dialog just as the "Hello AppleScript Genius!" line displays in the `display dialog` script (from the "Writing Your First Script" section earlier in this chapter).

The `default answer` phrase tells AppleScript that you want the user to enter a response of some sort. The "Who Knows" quote that follows `default answer` displays to the user as the default value is used if the user doesn't enter anything.

The `buttons` word tells AppleScript to put two buttons in the dialog: one labeled "Buzz off" and the other labeled "Hi." Just as `default answer` uses a quote as input, `buttons` uses a list — which I discuss in Chapter 4 — that is just two curly brackets with some items in the middle separated by commas. If you want to be creative, you can have three buttons by just adding another item to the list. The `default button` phrase tells AppleScript to make the Hi button the default button. In the Mac OS X world, the default button is the one filled with blue. A default button acts as though it's clicked if you press the Return or Enter key. If you want to see what the dialog looks like, jump ahead to Figure 1-6.



## Chapter 1: A Cannonball Dive into the Scripting Pool

# 17

Bet you didn't forget about the `set user_stuff` to part, did you? Bet you thought I did, though. First thing to realize is that when AppleScript sees something like

```
set x to y
```

it sets the value of `x` to whatever the value of `y` is. So the line

```
set x to 2
```

sets the value of the variable `x` to 2. (See Chapter 5 for more information on variables.) Variables in AppleScript are just named places to store stuff such as numbers and words that you want to use later on in a script.

Okay, so the script is trying to set the value of the variable `user_stuff`. But how do you stuff a dialog into a variable? The answer is that many AppleScript commands return a value. Here's the way it works after you click the Run button:

1. AppleScript displays the dialog shown later in Figure 1-6.
2. The user enters a value and clicks one of the buttons.
3. The `display dialog` command builds a value (or *result*) that it returns.
4. AppleScript puts that value into the script where the `display dialog` is, so the line looks like

```
set user_stuff to the_result
```

5. AppleScript sets the value of `user_stuff` to whatever `display dialog` returned.

The next question that's probably running, uncontrolled, through your mind is what in the heck does that `display dialog` command return? Well, stay tuned for the next thrilling episode . . . oops, wrong medium. The answer is that `display dialog` returns a list of things that have to do with what the user did.



Actually, `display dialog` returns a record. The difference between “list” and “record” is that the items in a record have names while those in a list don't. For more on the differences, take a gander at Chapter 4 or just read it if you don't have a goose.

If the user clicks the Hi button the record looks like this:

```
{text returned:"Who Knows", button returned:"Hi"}
```

A record — which I discuss in detail in Chapter 4 — is defined by those weird curly brackets. Each item in this record is separated from the other by a comma. A record item is made up of two pieces separated by a colon. The first piece is a name by which you can access the value, and the second piece is the value. The next line shows you how to access items in a record by name.

# 18

**Part I: Getting Started**

## *Line 2: Accessing the user's choice*

```
set button_name to button returned of user_stuff
```

This line sets the value of the local `button_name` variable to the value associated with the name `button` returned in the variable `user_stuff`. Remember that the value of `user_stuff` looks like this:

```
{text returned:"Who Knows", button returned:"Hi"}
```

So if the user clicks the Hi button, AppleScript does the following:

1. Figures out what value in `user_stuff` is associated with the `button` returned.
2. Sets the value of the `button_name` variable to that value.

If there is no `button` returned, or if you misspell it when typing it in, you get an error message and AppleScript tells you there is a problem.

## *Line 3: Responding to the user's choice*

```
if button_name is "Hi" then
```

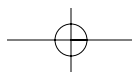
This line introduces a couple of concepts. The first is fairly clear. The script is checking to see if the value of the `button_name` variable is `Hi`. If it is, what then? Well, the script does something. But what? Figuring out what the script does requires you to know the full form of an `if` statement. A full-blown, ultra spiffy `if` looks like this:

```
if something then
    some script lines (A)
else
    some other script lines (B)
end if
```

Before trying to go through this statement in detail, its English equivalent is something like this:

```
if something is true then execute the script lines labeled A.
if something is false then execute the script lines labeled
    B.
```

What is truth? While some confused people think that's a toughie in real life, in AppleScript, the truth is fairly easy. Things like `2 < 5` — `2` is less than `5` — are true while things like `"Missing"` is the same as `"Hi"` are false.



An `if` statement checks some value or expression and sees if it's true. If it is, then the `if` statement executes the script lines you associated with that condition. If there's an `else` clause and the value or expression being tested is false, the script lines associated with the `else` clause execute. You use the `if` statements in your scripts to make decisions based on user actions or other events that you can't predict when you write the script.

If the item right after the `if` statement evaluates to `true`, the script lines between the `if` line and the `else` line execute by AppleScript. If the item right after the `if` statement evaluates to `false` and there is an `else` statement, the lines between the `else` statement and the `end if` statement execute. (By the way, if you read Chapter 10, then you're sure to become a master `if` scripter.)

### Line 4: Filling a variable

```
set your_name to text returned of user_stuff
```

This line only executes if the value of the `button_name` variable is `Hi`. If the user clicks the `Hi` button, the script assumes that he or she entered a name in the edit field replacing the default answer of `Who Knows`. Not surprisingly, the `text returned` element of the record stores the value in that edit field — as in “the text returned by this dialog.” Line 4 just takes the value of the text that the user enters and puts it into the variable called `user_stuff`.



Give variables names you understand. Even though you may do a little bit more typing, you'll find that you save time if you give variables names that reflect their contents. Check out Chapter 5 for tons of info on naming variables, but trust me, I know what I'm doing when I say that variable names that mean something save you time — time that you can use to earn money to buy my next book!

### Line 5: Greeting the user

```
display dialog "Welcome to AppleScript, soon to be Dr. " &  
    your_name &  
    ", expert AppleScript genius." buttons {"Howdy"} default  
    button "Howdy"
```

Like the fourth line, this line only executes if the user clicks the `Hi` button because that's the only time that the `button_name` variable has the value of `Hi`. By now you're probably an expert on the `display dialog` command (see the “Writing Your First Scrip” section if you're not), so I don't go over the

## 20 Part I: Getting Started

---

obvious things; however, you may be wondering about that funny & character. Is it a typo? Is it a printing error? Is it an alien being spying on you? The actual explanation is much more mundane. The & just *appends* — joins together — one quote to another. For example, when AppleScript executes this line

```
set a_sentence to "This is the first " & " sentence here."
```

the value of the `a_sentence` variable is set to

```
"This is the first sentence here."
```

If either or both of the values on either side of the & are variables, AppleScript replaces the variable with its value and then appends it. So this script

```
set part_1 to "To be or not to be "  
set part_2 to "that is the question."  
set line_1 to part_1 & part_2
```

sets the value of `line_1` to “To be or not to be that is the question.”

Using this character allows you to show the user a dialog personalized with his or her own name. Feel free to skip ahead for a second to look at Figure 1-6.

### *Line 6: Examining alternatives*

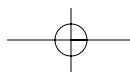
```
else
```

This line just serves to separate the script lines that execute if the value of `button_name` is `Hi` from the script lines that only execute if the value of the variable `button_name` isn't `Hi`.

### *Line 7: Macs do have feelings!*

```
display dialog "Sorry you don't want to be friends." buttons  
    {"For sure"}  
default button "For sure"
```

Another fun-filled `display dialog` command. Nothing exciting here, but note that even if you have only one button, it doesn't automatically become the default button. You have to use the `default button` option to make one of the dialog buttons the default if you use the `buttons` option. If you don't do that, the dialog doesn't have any default button.



## Line 8: Ending the if

```
end if
```

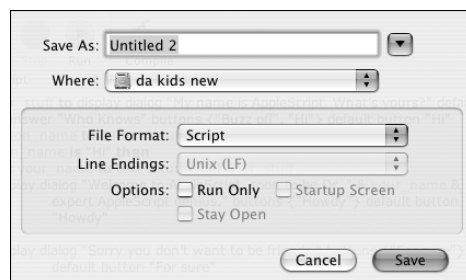
This line just marks the end of the `if` statement and the end of the script lines that execute if the user doesn't click the Hi button.

Now that you understand what the script does, you need to save the script. By some freakish coincidence, how to save a script is the next topic.

## Saving Your Work

AppleScript is a pretty stable tool. You have to work pretty hard to come up with a script that can crash your Mac, especially in OS X. However, crashes can happen, usually when you're working with other applications, so you need to save your scripts once in a while. It turns out that you have several options for saving a script, all of which I discuss in Chapter 3, but for right now, you can just choose `File`⇨`Save` from the Script Editor main menu. You get a fairly standard Mac Save dialog, as shown in Figure 1-5.

**Figure 1-5:**  
The  
standard  
AppleScript  
Save dialog.



You can just ignore the File Format pop-up menu at the bottom because the default selection (`Script`) is what you want. If your script has a problem, though (you get an error message when you click the `Compile` button), AppleScript can't save your script as anything other than `Text`. To find out what to do when that happens, see Chapter 18. For right now, though, if you get an error, compare what you typed to what's in Figure 1-4 and correct any differences, no matter how insignificant they may seem.

After you save your script, you're ready to run it.

## 22 Part I: Getting Started



### Just because AppleScript understands doesn't mean the script is right

Never forget that computers don't think. They just follow instructions. So just because AppleScript can understand what you type doesn't mean that a script does what you want. If you have a script that's supposed to delete all files containing the word "old" but you accidentally type the string "new" into the script, AppleScript understands what you typed and cheerfully deletes all files containing the word "new," even though that script deletes ten years of work.

In addition, some types of errors don't show up until you actually try to run a script. Suppose your

script is supposed to move files from one place to another, and you put in a nice way to pick the files to move. AppleScript has no way of determining that the files you pick in the future, when you run the script, are available. As a result, you could run the script and get an error even though AppleScript "understood" what you said.

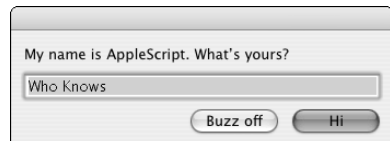
The bottom line is that just because you don't get an error message when you click the Compile button doesn't mean your script does what you want it to do.

## Running Your Script

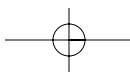
Running or executing a script just means that AppleScript follows the instructions you wrote down.

After AppleScript says your script is A-okay, you can run the script by clicking the Run button on the left side of the Script Editor. So go ahead and click the Run button for the script you wrote in the previous section. The first thing you see is the window in Figure 1-6.

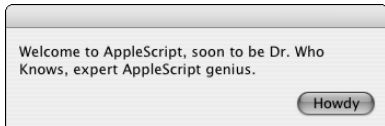
**Figure 1-6:**  
Starting  
a conver-  
sation.



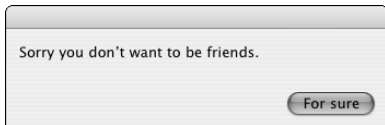
Because two buttons are in this dialog, the script has to deal with two possible user responses. That's why the `if` statement is in the script. Of course, if you want the script to respond the same way to the user pushing either button, you can, but I'm hard pressed to figure out a response equally appropriate for both "Hi" and "Buzz off." Because the script checks what button the user clicks, when the user clicks the Hi button the user sees the dialog in Figure 1-7, while if the user clicks the Buzz Off button, the dialog in Figure 1-8 appears.



**Figure 1-7:**  
The script is  
friendly if  
you are.



**Figure 1-8:**  
You've hurt  
its feelings,  
you wretch!

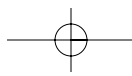


I originally arranged it so that if you hit the Buzz Off button, your Mac automatically transferred your life savings to my Swiss bank account, but Wiley told me that would raise its insurance rates. I told them that the book told you that they weren't liable for anything the scripts do, but for some reason, the folks at Wiley still weren't enthused. So my attempt to impoverish unfriendly people was temporarily stymied.

While this script doesn't do very much, it does illustrate how easy you can build a script whose behavior is tailored to user inputs. Much more complex user input is implemented by just asking the user more questions. But most of all, if you enter the script, you now are halfway to being a proficient scripter because you know the basic mechanics of how to use the Script Editor. You find a lot more detail in Chapter 3, but right now, you know enough to enter, save, and run scripts. All that's left is to familiarize yourself with a bit of the AppleScript language, which is a lot like English, and you're all set.



Oh yeah, one other thing: While I ran these scripts from inside the Script Editor, you can save scripts as applications that you can run by double-clicking their icons, just like any other application. You see how to do that in Chapter 3.



# 24 Part I: Getting Started

---

