

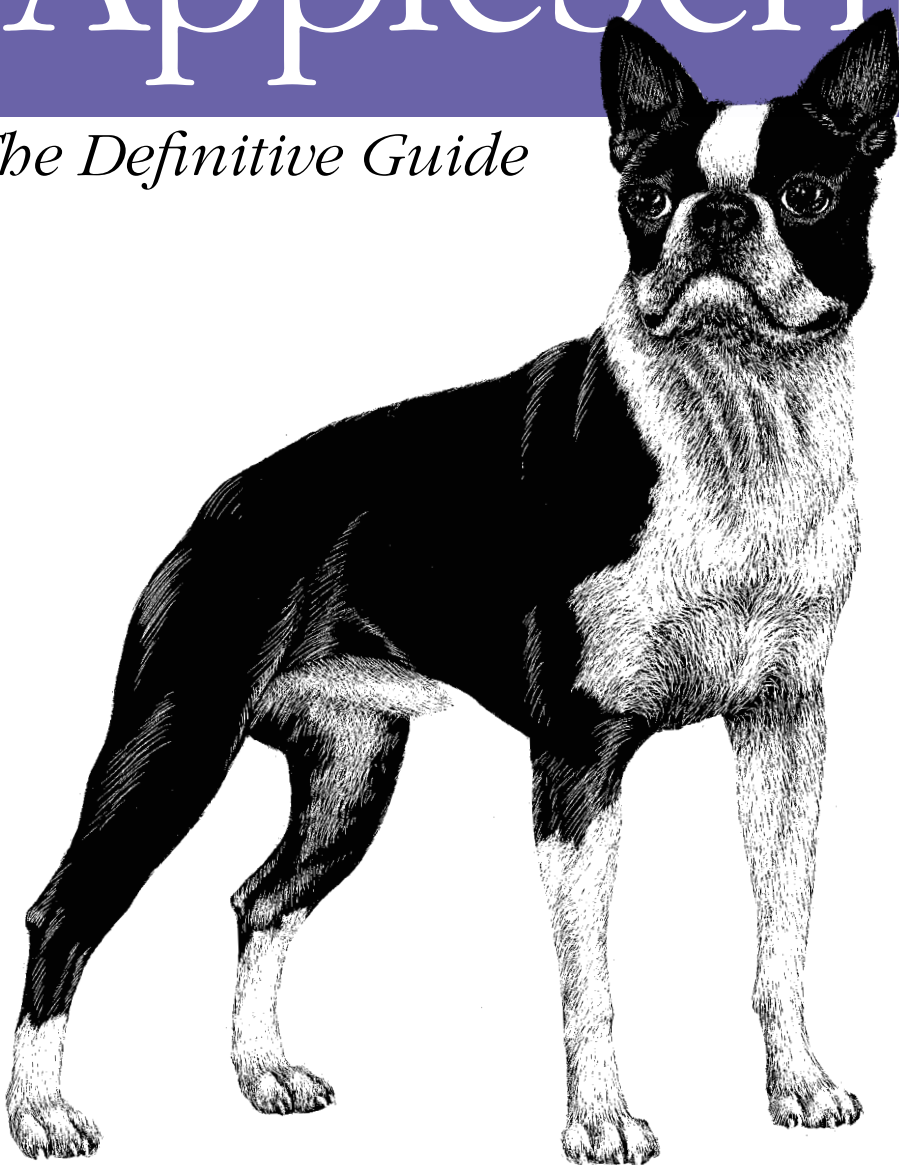


Developer
Connection
Recommended Title

Covers
Mac OS X Panther

AppleScript

The Definitive Guide



O'REILLY®

Matt Neuburg

This chapter describes the rules for declaration, typing, initialization, naming, scoping, and lifetime of variables in the AppleScript language.

A *variable* is a binding between a name and a value. You can think of it as a shoebox with a label on it, into which something is placed for storage. The shoebox's label is the variable's name; what's inside the shoebox is the variable's value. For example, when we say:

```
set x to 5
```

it is as if we had a shoebox labeled “x” into which we place the number 5.

Assignment and Retrieval

To *assign* a value to a variable is to put something into the shoebox. If the variable already has a value, that value is replaced. Assignment is performed with one of two commands: `set` or `copy`, described here.

set

Syntax

```
set variableName to value
```

Description

Assigns *value* to *variableName*.

Example

```
set x to 5
```

There is a synonym using the word `returning` instead of `set`, with the parameters in reverse order, like this: `5 returning x`. But I have never seen this used.

copy

Syntax

`copy value to variableName`

Description

Assigns *value* to *variableName*.

Example

```
copy 5 to x
```

An abbreviation for copy is put; an abbreviation for to is into. Thus you could type `put 5 into x`, although it would still come out as `copy 5 to x`. This is doubtless to accommodate HyperCard users, who were habituated to this syntax.

In these expressions, *variableName* can optionally be a list of variable names, allowing multiple assignments in one command. The *value*, too, will then be a list—a list of the values to be assigned. The first item in the *value* list is assigned to the first item in the *variableName* list, the second to the second, and so forth. If the *value* list is longer than the *variableName* list, the extra values are not assigned to anything; if the *value* list is shorter than the *variableName* list, there is a runtime error. This remarkably elegant feature is probably under-utilized by beginners. (For a parallel construction involving assignment to a record, see “Record” in Chapter 13.) For example:

```
set {x, y, z} to {1, 2, 3}
z -- 3, and can you guess what x and y are?
```

It sounds from their descriptions as if `set` and `copy` must be completely interchangeable. In most cases, they are; but with regard to four types of value—lists, records, dates, and script objects—they are not. This point will be covered in subsequent chapters. For other datatypes, you may use whichever command you prefer; I prefer `set`.

There is no simple assignment operator, such as equals sign (=). You cannot, for example, perform an assignment like this:

```
x = 5
```

That is a comparison, and returns a boolean result revealing whether `x` is already 5. The fact that such code is legal (and therefore does not cause a compile-time error) but is not an assignment (as any mildly experienced programmer would expect) is a frequent cause of bugs in my scripts. See “The “English-likeness” Monster” in Chapter 5.

To *retrieve* the value of a variable (or *fetch* the value, or *use* the value, or whatever you want to call it), simply use the variable’s name in code. As with most computer languages, there is no problem retrieving from and assigning to the same variable in a single statement:

```
set x to x + 1
```

There is no circularity, because first the value of `x` is retrieved, and 1 is added to that value; then the result of this addition is assigned to `x`, replacing the original value.

The result of a line consisting of just the name of a variable is the value of that variable. So, for example:

```
set x to 5
x
```

The result of that script is 5. This can be useful when you want to employ the implicit result of a script as a way of testing or debugging (see “Implicit Result” in Chapter 6).

It is possible to retrieve a variable’s value by using the `get` command:

```
set x to (get x) + 1
```

But no one ever talks this way in real life, and as far as I know this use of `get` with a variable adds nothing. However, `get` with an object reference is another matter; see “Get” in Chapter 10.

Declaration and Definition of Variables

There is no requirement in AppleScript that variables be declared explicitly. The rule is basically that if you use a word that AppleScript doesn’t understand, the word is assumed to be the name of a variable. The following code, as a complete script, will compile just fine:

```
set x to x + 1
```

Definition

The code in that last example, as a complete script, will compile, but it won’t run; at runtime, it generates an error. That’s because `x` has never been assigned a value. The error message reads: “The variable `x` is not defined.” The problem is *not* that the variable `x` has never been declared! There is no need to declare it. AppleScript understands (or assumes) that `x` is supposed to be a variable. Nor is the problem that you are trying to assign to it. The problem is that you are trying to fetch its value, and it has no value. An AppleScript variable is not defined until you first give it a value explicitly. To continue our shoebox analogy, there is no “`x`” shoebox to fetch the contents of, because you’ve never put anything into it.

This code both compiles and runs:

```
set x to 5
set x to x + 1
```

During execution of the first line, AppleScript observes that you’re putting something into the “`x`” shoebox, but there is no such shoebox as yet. No problem;

AppleScript creates the shoebox, labels it “x”, and puts 5 into it. Now the second line runs fine, because there is a shoebox “x” from which to fetch a value.

Once a variable has been defined in the course of running a script, it generally stays defined until its scope finishes executing, as discussed later in the chapter. There is no command explicitly letting you “undefine” a variable or assign the “undefined” value to it. However, you can undefine a variable by assigning to it the result of a command that has no result. This is typically an accident: you were expecting a command to return a value, but it doesn’t. Code for doing it on purpose appears under “Returned Value” in Chapter 8.

There is no way to ask whether a variable’s value is defined; all you can do is fetch its value and see if you get an error. It would then be up to your code to handle this error (“Errors” in Chapter 12); otherwise your script will simply stop running at that point.

Initialization

A variable is *initialized* (given its first value) when you explicitly assign it its first value. There is no auto-initialization of variables in AppleScript, and there is no special syntax for initializing variables. A variable is undefined until you assign it a value; at that moment it is defined and initialized—the variable now exists, it has a value, and it is possible to fetch that value.

The exception is a *script property*. A script property is a kind of global variable, and it is declared and initialized in the same line of code. Script properties have some other interesting features, which are discussed later in this chapter.

Typing

A variable in AppleScript has *no fixed type*. By this I mean simply that it is permissible to assign any variable any value, any time. The following code is legal:

```
set x to 5
set x to 5.2
set x to "hello"
set x to string
set x to {"fee", "fie", "fo", "fum"}
set x to (path to current user folder)
```

In that code, x becomes successively an integer, a real, a string, a class, a list, and an alias. A defined variable (one that has a value) has a type, called its *class*; this is simply the class (datatype) of its current value, and it changes if a value of a different class is assigned to it.

The various built-in datatypes, and the ways in which AppleScript lets you coerce implicitly and explicitly from one to another, are discussed later in this book (Chapters 13, 14, and 15).

Explicit Declaration

Although it is not *necessary* to declare a variable, it is *possible* to declare a variable, and there are three ways to do so:

- As a script property:

```
property x : 5
set x to x + 1
```

- As a global:

```
global x
set x to 5
set x to x + 1
```

- As a local:

```
local x
set x to 5
set x to x + 1
```

The meanings of these declarations are explained later in this chapter.



It is almost always best to declare your variables. Your code will be easier to understand, and the scoping rules are simpler and clearer, if all your variables are declared. Unfortunately there is no way to have AppleScript warn you when a variable is not declared.

Variable Names

The name of a variable must begin with a letter or underscore and must consist entirely of alphanumeric characters or underscore. So a variable name must begin with a character in the character set [a-zA-Z_] and must consist entirely of characters in the character set [a-zA-Z0-9_].

Variable names are *case-insensitive at compile time*. That means the following code will compile and run:

```
set myVar to 5
set myvar to myvar + 1
```

AppleScript assumes that `myvar` in the second line is the same variable as `myVar` in the first line. Furthermore, as a reflection of this assumption, AppleScript rewrites the variable names after compilation so that their case matches the first usage of the name:

```
set myVar to 5
set myVar to myVar + 1
```

This suggests a trick that can help you spot undeclared variables: in your declarations, use an uppercase letter somewhere in every variable name; elsewhere, never use an uppercase letter in a variable name. Then, after compilation, any variable

name without an uppercase letter must be an undeclared variable. For example, here's some code that I typed following these rules, after compilation:

```
local myVar
set myVar to 5
set mybar to myVar + 1
```

In that code I have accidentally created and set the value of an unwanted variable `mybar` in the last line. I meant to say `myvar`, but I mistyped it. This won't cause AppleScript to generate any error, and the script will misbehave. The chances that I will spot my mistake are increased by my use of the case trick.

Once a script has been compiled for the first time, its variable names are remembered as they appear at that moment. (Recall that AppleScript has a memory. See “Maintenance of State” in Chapter 4.) Suppose you compile this script:

```
local avariable
set avariable to 7
```

You then change your mind and decide to use the inner capitalization trick, so you edit the script to give the variable names inner capitalization:

```
local aVariable
set aVariable to 7
```

When you compile, you find your efforts are in vain; AppleScript removes the inner capitalization!

```
local avariable
set avariable to 7
```

The reason is that when AppleScript first saw the variable name `avariable`—the first occurrence during the first compilation of the script—it had no capitalization, and that's how the name is remembered from then on.

What's more, this rule washes over to other scripts that you edit during the same session! To see this, start up your script editor program and compile this script:

```
set myvar to 7
```

Now open a new, different window and compile this script:

```
local myVar
set myVar to 7
```

Your variable names are changed in this second script! It ends up looking like this:

```
local myvar
set myvar to 7
```

This bizarre behavior is caused by the combination of two facts: variable names are remembered at global level in the AppleScript scripting component, and there is just one AppleScript scripting component instance per script editor session. (You did reread “Maintenance of State” in Chapter 4, didn't you?) This instance is shared by all the scripts you compile during that session, so the variable names in one script

affect the variable names in another. This phenomenon persists until you quit the Script Editor program. Two different applications don't share the same AppleScript scripting component instance, though, so your variable names in Script Editor do not affect your variable names in Script Debugger at the same moment.

You can force an illegal variable name to be legal by surrounding it with vertical bars, also known as “pipes” (|). So, for example:

```
set |1| to 2
if |1| is 2 then
    display dialog "The laws of logic are suspended."
end if
```

The laws of logic aren't really suspended; 1 and 2 have not become the same number. A variable named “1” has been assigned the value 2, that's all. This device is good also for variable names in languages other than English:

```
set |monZéro| to 0
```

or for spaces in a variable name:

```
set |my big long variable name with spaces| to 7
```

A variable name surrounded by pipes is *case-sensitive*. This script will compile, but it won't run:

```
set |MyVar| to 5
set |MyVar| to |Myvar| + 1 -- error
```

The reason is that |Myvar| is not the same variable as |MyVar| and has never been given a value, so its value can't be fetched. AppleScript will not touch the case of names in pipes after compilation.

A variable name surrounded by pipes may include a backslash as an “escape” character. The legal escape expressions in this context are \n, \r, \t, \, and \\..

The real meaning of pipes is to tell AppleScript to suspend its compile-time parsing rules and turn what's inside the pipes into a token. The main reason this is genuinely useful is to avoid a conflict between a token name and a reserved word. For example:

```
set |is| to "ought"
```

You couldn't do that without the pipes, because `is` is a reserved word, a part of the AppleScript language.

Now, you might say: “So what? I'll never need to worry about that; I just won't use any names that conflict with reserved words.” But even though *you* might not use such names, some entity with which you need to communicate might do so. For example, some unwary developer could use a reserved word as part of the vocabulary defined by a scriptable application (even though they're not supposed to), and you would then need pipes in order to use that word to talk to the application. (See “Resolution Difficulties” in Chapter 19.)

Scoping of Variables

The notion of *scoping* has to do with where an entity is visible. Your code consists of regions; these regions are each continuous, and some of them may be inside others, but they do not partially intersect—given two regions, either one is entirely inside the other or they are completely distinct. The region in which a variable is visible is called its *scope*. A variable that is visible at a certain point is said to be *in scope* at that point.

Scoping of variables in AppleScript is extraordinarily complicated (in my opinion). It's also very important to understand, so don't skip this section.

How Scoping Is Meaningful

Before we can talk about the scoping of variables in particular, you must understand the basic principles of AppleScript scoping in general. These are:

- The top level of all scope is the script as a whole.
- The regions of scope are handlers and script objects (and the top level).
- A script object may contain a handler. A handler may contain a script object. A script object may contain a script object. But a handler may not (directly) contain a handler.

Let's start with the first rule. Your script as a whole is itself the ultimate region of scope, containing everything else. So, let's say that your script consists of just the following code:

```
set x to 7
```

In terms of scope, where are we when this code executes? This is all the code there is, and we're not in a handler or a script object, so we are at the top level of all scope, the script as a whole.

Now I will illustrate the second and third rules, even though I have not yet explained rigorously what a script object or a handler is; all you need to know is that in code a script object is a block declared by the word `script` and a handler is a block declared by the word `on`. This code, then, is legal:

```
on handlerOne()  
    script scriptOne  
    end script  
end handlerOne  
script scriptTwo  
    on handlerTwo()  
    end handlerTwo  
    script scriptThree  
    end script  
end script
```

Everything you see in that code is inside the top-level script. Within the top-level script are handlerOne and scriptTwo. Within handlerOne is scriptOne. Within scriptTwo are handlerTwo and scriptThree.

Along with the top level, these handlers and script objects are the regions of scope. Each region starts with the declaration of the handler or script object, and ends with the corresponding end line. Into this code may be inserted further code, and every line of this further code is in some definite region of scope.

The question we want to answer, then, is how variables in these various regions of scope are visible to code within other regions of scope.

Explicit Locals

An *explicit local* is a variable declared with the keyword `local`. It is legal to declare more than one variable local in the same command, by separating them with commas. So:

```
local x
local y, z
```

In general, an explicit local is visible *only within the scope where it is declared*. (There is one exception, which I'll mention in a moment.)

Different local variables in different scopes can thus have the same name without trampling on one another. Suppose a script object starts like this:

```
script myScript
  local x
```

The moment that local declaration for `x` is encountered, it means that from now on when code in this script object's scope says `x` it means this local `x` and no other. Other scopes may declare their own local `x`, they may declare a global `x`, they may bang the floor and have a temper tantrum, but they absolutely will not be able to have any effect upon `myScript`'s `x`, nor will anything `myScript` does with its `x` have any effect upon them.

Here's an example of a local variable in action:

```
local x
set x to 5
script myScript
  display dialog x
end script
run myScript -- error
```

(The code inside a script object does not run when the script object is defined. To run the code inside a script object, you tell that script object to run. This is formally explained in Chapter 9. So in this example, first we define `myScript`, then we run it.)

That code compiles, but it won't run; it stops with a runtime error at the `display dialog x` command, objecting that `x` is not defined. There is a variable called `x` and it

is defined, but it is declared local and therefore is visible only within its own scope. In this case, that scope is the top-level script. The `display dialog x` command is in a different scope, that of the script object `myScript`. Therefore AppleScript takes this to be a different `x`, and this different `x` has never been assigned a value.

Now let's do it the other way round:

```
on myHandler()  
    local x  
    set x to 5  
end myHandler  
myHandler()  
display dialog x -- error
```

(The code inside a handler does not run when the handler is defined. To run the code inside a handler, you say its name followed by parentheses. This is formally explained in Chapter 8. So in this example, first we define `myHandler`, then we run it.)

This code stops with a runtime error at the `display dialog x` command, objecting that `x` is not defined. There is a variable `x` that has been defined, but that happened inside the scope of the handler `myHandler`, where this `x` was declared local. The `display dialog x` command is in a different scope, namely the top level. Therefore AppleScript takes this to be a different `x`, and this different `x` has never been assigned a value.

There is, however, this one great exception to the rule about the scope of local variables: *a script object defined in a handler can see the handler's local variables*. For example:

```
on myHandler()  
    local x  
    set x to 5  
    script myScript  
        display dialog x  
    end script  
    run myScript  
end myHandler  
myHandler() -- 5
```

This remarkable exception to the local scoping rule will permit us to pass a handler as a parameter to another handler and call it (“Handlers as Parameters” in Chapter 8). We will also combine it elegantly with a handler's ability to return a script object (“Script Object as Handler Result” in Chapter 9).

Global Declarations: The Downward Effect

Our next topic will be variables declared with the keyword `global`. It is legal to declare more than one variable global in the same command, by separating them with a comma. So:

```
global x  
global y, z
```

For clarity, I'm going to discuss the effect of a global declaration in two stages. First I'm going to explain what I call the *downward effect* of a global declaration. By this I mean the effect a global declaration has on code in the same scope as the declaration, or a deeper scope within that scope.

Here's the rule. A variable declared global is visible *subsequently in the same scope* as the declaration, and within all handlers and scripts defined subsequently in the same scope, *to an infinite depth*.

For example, this code runs:

```
global x
set x to 5
on myHandler()
    display dialog x
end myHandler
myHandler() -- 5
```

The variable `x` is declared global; it is then visible downward into the scope of `myHandler`, because `myHandler` is defined subsequently in the same scope as `x`. (Do you see why I call this the downward effect of the declaration?)

In that example, I proved that code inside `myHandler` could see `x` by having that code fetch its value. But such code can equally well set its value:

```
global x
on myHandler()
    set x to 5
end myHandler
myHandler()
display dialog x -- 5
```

That's important. Code exposed to a variable by a global declaration is very powerful. The code gains full access to the variable.

Let's prove that the downward visibility created by a global declaration operates to a greater depth (it operates, as I said earlier, "to an infinite depth," but I don't see how to prove that):

```
global x
on myHandler()
    script myScript
        on mySecondHandler()
            set x to 5
        end mySecondHandler
    end script
    myScript's mySecondHandler()
end myHandler
myHandler()
display dialog x -- 5
```

The line where `x` is given a value appears in a handler within a script object within a handler. Nevertheless that line can see the variable `x` declared global in the first line, and the script runs successfully.

Now let's concentrate on the word "subsequently." A variable not declared global until after the definition of a script object or handler cannot be seen by code within that script object or handler. The following code does not work:

```
on myHandler()  
  script myScript  
    on mySecondHandler()  
      set x to 5  
    end mySecondHandler  
  end script  
  myScript's mySecondHandler()  
end myHandler  
global x  
myHandler()  
display dialog x -- error
```

The variable `x` was not declared global until after the definition of the handler `myHandler`. Therefore code within `myHandler` cannot see `x`. The line `set x to 5` was executed, but it failed to set the global variable `x`'s value; therefore when we reach the last line, the global variable `x` has no value and a runtime error occurs.

(You might now be wondering: "Okay, the line `set x to 5` didn't set the value of the global variable `x`, but it didn't cause a runtime error either; so what did it do?" I'll get to that, I promise.)

Naturally, any downward scope may shield itself from the downward effect of a global declaration at a higher level, simply by declaring the same variable name as local for its own scope. For example:

```
global x  
set x to 5  
script myScript  
  local x  
  on myHandler()  
    set x to 10  
  end myHandler  
  myHandler()  
  set x to 20  
end script  
run myScript  
display dialog x -- 10 (not 20)
```

The dialog displays 10, not 20. The global `x` declaration in the first line has a downward effect on `myScript` and, within it, on `myHandler`. But `myScript` then shields itself from this effect with a local `x` declaration. The variable `x` declared global in the first line is thus visible at top level and at the third level in `myHandler`, but at the second level in `myScript` it is not visible, and the `x` referred to there is a different `x`. When `myHandler` sets `x` to 10, that is the same `x` as at the top level. When `myScript` then sets `x` to 20, that's a different `x` and this has no effect on the value displayed in the last line.

Global Declarations: The Upward Effect

We are now ready to talk about what a global variable really is. In the previous section, in talking about the downward effect of a global declaration, I was talking about just the declaration; a global declaration can appear anywhere. But now we're talking about the actual variable named by a global declaration—a global variable. The rule is: *a global variable actually exists at the top level.*

However, it turns out that a global variable, even though it exists at top level, *cannot automatically be seen in every scope.* In order for a global variable to be seen in a scope other than top level, it must be explicitly declared.

There are two ways for a global variable to be declared so as to be visible in a scope other than top level. One we have already seen: the top level may declare it, which makes it visible subsequently downwards. The other way is for the deeper scope to ask to see the global variable; it does this with a global declaration in its own scope.

The upward effect of declaring a variable global is to *identify the declared variable with the top-level global variable of the same name.* This top-level global variable may or may not exist already; if it doesn't exist, we may say that the declaration also creates it. (But the global variable doesn't *really* exist until it is defined by being assigned a value.)

To illustrate:

```
on setX()  
  global x  
  set x to 5  
end setX  
on getX()  
  global x  
  display dialog x  
end getX  
setX()  
getX() -- 5
```

When `setX` runs, its declaration of `x` as global creates the global variable `x` at top level (because no such global variable exists previously), and identifies this `x` with that `x`; thus, in setting the value of `x`, it sets the global's value. Then when `getX` runs, its declaration of `x` as global identifies this `x` with the now existing global variable `x` at top level. Thus, when it displays the value of `x`, it is the global's value that it displays. Both `setX` and `getX` have access to the very same global variable `x`; and they have that access because they each asked for it, with a global declaration.

It goes without saying (said he, saying it) that such a global declaration, by virtue of its downward effects, also gives to all downward scopes the same powers over a top-level global that it gives its own scope (unless, of course, they deliberately shield themselves from this power by means of a local declaration, as in the example at the end of the previous section). For example, this code works:

```
script myScript  
  global x
```

```

        on myHandler()
            set x to 5
        end myHandler
    myHandler()
end script
on getX()
    global x
    display dialog x
end
run myScript
getX() -- 5

```

By the time we come to the last line, the script object `myScript` has run, and its handler `myHandler` has executed the line `set x to 5`. Because there has been a higher-level global declaration of `x`, this `x` is that global `x`. (That’s the downward effect of the global declaration.) But this `x` is also the top-level global `x`. (That’s the upward effect of the global declaration.) Therefore, even though it contains no global declaration within itself, `myHandler` is able to set the value of the top-level global `x`. Then when `getX` runs, since it starts with a global declaration for `x`, it sees the same top-level global variable, and displays its value.

Undeclared Variables

We come now to the question of undeclared variables. Say that at compile time, a variable name is encountered, and this name has not been declared for this scope—that is, it has not been declared as local in this same scope, and it has not been declared as global in the same or a higher scope. Now, a variable can only be local or global. *What will AppleScript do?*

It turns out that there are two different answers, depending on where the code occurs—at the top level, or elsewhere.

Undeclared variables at top level

Code at the top level of the script is special. (Technically, what I mean here is “code at the top level of the run handler.” See “The Run Handler” in Chapter 8.) Code at the top level of the script doesn’t need a global declaration in order to create, set, or see a global variable. An undeclared variable name at top level is *treated as global*. But lacking the explicit global declaration, it *lacks the downward effect* that an explicit global declaration would have. We may call such a variable an *implicit global*.

So, for example:

```

set x to 5
on getX()
    global x
    display dialog x
end getX
getX() -- 5

```

In the first line, the previously undeclared `x` was created as a top-level global, implicitly, and given a value. When `getX` runs, it is able to access that value with a global declaration, and can display it.

Now let's do the converse:

```
on setX()  
  global x  
  set x to 5  
end setX  
setX()  
display dialog x -- 5
```

By the time we get to the last line, `setX` has run; it has created a top-level global variable `x`, and has set its value. In the last line, the previously undeclared `x` spoken of at top level is an implicit global, so it is that same top-level global variable `x`. Thus the value 5 is displayed.

But this code fails with a runtime error, unless the first line is uncommented:

```
-- global x  
set x to 5  
on getX()  
  display dialog x  
end getX  
getX() -- error, unless you restore the global x declaration
```

The presence of the `global x` declaration has a downward effect, enabling code inside the `getX` handler to see `x` without a `global x` declaration of its own. Without such an explicit `global x` declaration anywhere, `x` is still global, but code in `getX` can't see it.

It is perfectly possible for code at the top level to shut off its own access to a global variable, just as any other scope may do, by declaring a local variable of the same name. For example:

```
local x  
set x to 5  
on setX()  
  global x  
  set x to 10  
end setX  
setX()  
display dialog x -- 5
```

This displays 5, not 10. That's because there has been a declaration that the `x` spoken of at top level is a local. It's true that `setX` set a top-level global to 10, but that's a different variable! The top-level code isn't accessing that variable; it closed off its access to it, through the local declaration.

Undeclared variables not at top level

An undeclared variable name *not* at top level is treated *exactly as if that variable had been declared local in its scope*. We may call such a variable an *implicit local*.

For example:

```
set x to 5
script myScript
  set x to 10
end script
run myScript
display dialog x -- 5
```

The dialog displays 5. You should now understand why. Each `set x` line creates a different `x`—the first creates an implicit global at top level, the second creates an implicit local in its own scope. Thus the `x` that is set to 10 is a different `x` from the `x` that was set to 5; it is the `x` that was set to 5 that is displayed in the last line. There was no runtime error; no one tried to access the value of an undefined variable. But `myScript`'s `x` was wasted; it was set to 10 and then was immediately destroyed as it went out of scope, with no effect on anything else in the code.

Declare your variables

Sounds like “eat your vegetables,” doesn't it? Well, it should. Each motto is good advice, no matter how unpalatable it may seem at first. I strongly advise you (once again) to declare all your variables—even your locals.

Now that you understand what happens when variables are not declared, you can imagine the sorts of confusion that can arise. If you let yourself become lulled into a false sense of security by the fact that there's no need to declare your variables, then you can be surprised when some other scope tramples them.

For example, imagine that your script starts like this:

```
set x to 5
```

That's top-level code, so you've just implicitly declared `x` a global. This means that any other handler or script object anywhere in this script can access your `x` and change its value, just by saying this:

```
global x
```

This other code may not have intended to trample on your `x`; perhaps it was trying to establish a global of its own, possibly in order to communicate its value to code at a lower level. But the damage is done, because of the upward effects of a global declaration. And to think you could have prevented this, just by declaring your `x` local to start with.

In the case of script objects the problem is particularly insidious, because it is possible to run a script object whose code you can't see, by loading it from a compiled script file on disk. There will be more about that later (Chapter 9), but here's a quick example:

```
set x to 5
run (load script alias "myHardDrive:aScriptFile.scpt")
display dialog x
```

The frightening fact is that `x` could now be anything! If the script in the file `aScriptFile.scpt` happens for any reason to declare global `x`, it can freely change the value of your `x`.

The converse is also true. Pretend you have a large script and that this code occurs somewhere within it:

```
on myHandler()  
    set x to 5  
end
```

Is `x` a local or a global here? You don't know! It depends upon the context. If `x` has previously been declared global at a higher level, this `x` is global (by the downward effect of that declaration). If not, this `x` is local. But it is intolerable that you should have to look elsewhere to learn the scope of `x` within `myHandler`! All you have to do is explicitly declare it global or local, right here in `myHandler`, and then you'll know for sure.

Free Variables

An entity defined outside a handler or script object but globally visible within it, and not overshadowed by a declaration of the same name, is called a *free variable* with respect to that handler or script object. For example:

```
global x, y, z  
script myScript  
    property x : 1  
    local y  
    yy  
    z  
end script
```

Within `myScript`, `x` is explicitly defined as a property (as explained later in this chapter in “Script Properties”) and `y` is explicitly defined as a local, so neither is a free variable. The variable `yy` isn't explicitly defined within `myScript`, but it isn't defined outside it either, so it is an implicit local and not a free variable. But the variable `z` is globally visible within `myScript` (from the global declaration at the start of the code), and the name is not redeclared within `myScript`, so `z` within `myScript` is a free variable, and is identified with the global `z` declared in the first line.

A free variable takes its value within the handler or script object *at the time the code runs*, not at the time the handler or script object is defined. For example:

```
set x to 5  
on myHandler()  
    global x  
    display dialog x  
end myHandler  
set x to 10  
myHandler() -- 10 (not 5)
```

The dialog displays 10, not 5. It doesn't matter that `x` had been set to 5 when `myHandler` was defined; it only matters what its value is when the code inside `myHandler` actually runs. By that time, `x` has been set to 10.

It is important here (as we saw earlier) that `x` has been declared global in code that appears *before* the code where `myHandler` speaks of it. Free variables' values are determined at runtime, but the identification of a variable as a free variable, and its association with some particular globally visible variable, is performed during compilation (and AppleScript's compilation is single-pass, remember). This way of resolving the meaning of free variable names is called *lexical scoping*.

For example:

```
set x to 5
on myHandler()
    display dialog x
end myHandler
global x
myHandler() -- error
```

It's true that `x` is declared global before (temporally) `myHandler` runs. But that's not good enough. We must declare `x` global before (physically) `myHandler` speaks of it; otherwise, `myHandler`'s `x` isn't the global `x`, and the code fails with a runtime error because `myHandler`'s `x` isn't defined.

Redeclaration of Locals and Globals

It is a compile-time error to redeclare an implicit global as local:

```
set x to 5
local x -- compile-time error
```

It is a compile-time error to redeclare an implicit local as global:

```
on getX()
    display dialog x
    global x -- compile-time error
end getX
```

It is a compile-time error to redeclare as local a variable declared global in the same scope (except at top level):

```
on getX()
    global x
    local x -- compile-time error
end getX
```

It is a compile-time error to redeclare as global a variable declared local in the same scope (except at top level):

```
on getX()
    local x
    global x -- compile-time error
end getX
```

At top level, it is *not* an error to declare a variable local and then declare it global in the same scope. But it doesn't have any effect within the top-level scope either. For example:

```
local x
global x
set x to 5
on setX()
    set x to 10
end setX
on getX()
    display dialog x
end getX
setX()
getX() -- 10
display dialog x -- 5
```

Once *x* is declared global, both *setX* and *getX* have automatic access to a top-level global variable *x*. But the code in the top level does not have such access. There, *x* has already been declared local; nothing can change this. Once a local, always a local. The *x* that is set to 5, and that is displayed at the end, is this local *x*, which is different from the global *x*.

At top level, it is *not* an error to declare a variable global and then declare it local in the same scope. But access to the global variable is lost in the top-level scope. For example:

```
global x
set x to 5
local x
on getX()
    display dialog x
end getX
getX() -- 5
display dialog x -- error
```

After the first two lines, there is a top-level global variable *x* and its value is 5, and code at a deeper level can access it; the subsequent local declaration has no effect on this fact, even though it precedes the definition of the deeper-level code. But the top-level code has lost its access to this global variable, and can never recover it.

Script Properties

A *script property* (often just called a *property*) is a script-level global variable with initialization. A script property must be declared, and an initial value must be supplied as part of the declaration. The syntax is:

```
property propertyName : initialValue
```

For example:

```
property x : 5
```

The abbreviation for property is `prop`.

A property declaration can appear only at top level or at the top level of a script object. For example:

```
property x : 5
script myScript
  property y : 10
  -- other stuff
end script
-- other stuff
```

A property is a variable, so its value can be set and fetched in the normal way. For example:

```
property x : 10
display dialog x -- 10
set x to 5
display dialog x -- 5
```

Scoping of Properties

A property is a kind of global variable, and a property declaration has the *same downward effect as a global declaration*:

```
property x : 10
script myScript
  display dialog x
end script
on myHandler()
  display dialog x
end myHandler
run myScript -- 10
myHandler() -- 10
```

Both `myScript` and `myHandler` can see the property `x`, because the property declaration works like a global declaration with respect to its downward effects.

The big difference between a global variable and a script property is in the upward effect of their declaration. A property's scope is *confined to the script object where it is declared*. The property is automatically visible downwards, as if the property declaration had been a global declaration; but it is not automatically visible anywhere else. Different script objects may declare a property by the same name, and these properties will be separate variables.

For example:

```
property x : 5
script scriptOne
  property x : 10
  script scriptTwo
    property x : 20
    display dialog x
  end script
display dialog x
```

```

    run scriptTwo
end script
script scriptThree
    property x : 30
    display dialog x
end script
script scriptFour
    display dialog x
end script
display dialog x -- 5
run scriptOne -- 10, 20
run scriptThree -- 30
run scriptFour -- 5

```

Every property `x` in that code is a separate variable. Observe that this separateness would be impossible using global declarations, because each `global x` declaration at any level would refer to the very same top-level global variable. Of course, locals provide a similar separateness, but with locals you wouldn't get the downward effect of a property declaration (used by `scriptFour` to see the top-level `x`). Thus we see that the scoping effect of a property declaration is different from either a global declaration or a local declaration.

Furthermore a property, unlike a local, even in scopes where it isn't visible automatically, *is visible on demand wherever its script object is visible*. To speak of a property from outside the scope where it is visible automatically, you must employ a special syntax: either you use the `of` operator (or the apostrophe-ess operator) and the name of the script object, or you use the keyword `its` within a `tell` block addressed to the script object. You are then free both to fetch and to change the value of the property.

For example:

```

script myScript
    property x : 10
end script
on myHandler()
    set myScript's x to 20
end myHandler
display dialog x of myScript -- 10
myHandler()
display dialog myScript's x -- 20
tell myScript
    display dialog its x -- 20
end tell

```

A difficulty arises, though, when a script object has a property but wishes to speak of a top-level property of the same name:

```

property x : 5
script myScript
    property x : 10
    display dialog x -- but I want to speak of the top-level x
end script
run myScript -- 10, alas

```

The difficulty is that the top-level script is anonymous. Under normal circumstances, it may be referred to at any level as parent. For example:

```
property x : 5
script outerScript
  property x : 10
  script innerScript
    property x : 20
    display dialog parent's x
  end script
end script
run outerScript's innerScript -- 5 (not 10 or 20)
```

However, it is possible to subvert this by redefining a script object's parent. (This will be explained in “Inheritance” in Chapter 9.) For example:

```
property x : 5
script scriptOne
  property x : 10
end script
script scriptTwo
  property x : 20
  property parent : scriptOne
  display dialog parent's x
end script
run scriptTwo -- 10 (not 5)
```

To get around this, the surest method, and therefore the surest method for accessing a top-level property in general, is to give the top level a name. This is done by assigning the value *me* to a global variable or property at top level. (For the formal explanation of *me*, see “Me” in Chapter 10.) So, for example:

```
property topLevel : me
property x : 5
script scriptOne
  property x : 10
end script
script scriptTwo
  property x : 20
  property parent : scriptOne
  display dialog topLevel's x
end script
run scriptTwo -- 5
```

Top-Level Properties Are Globals

There is no difference between a top-level global variable and a top-level property (except that the property is initialized). I will pause a moment to let this sink in.

One consequence of this is that in all the examples in the previous section where I declared a top-level property, I could have used a top-level global instead. This

top-level global can be explicit (if I want the downward effects of the declaration) or implicit. For example, I'll just repeat the last example in a different guise:

```
global topLevel
script scriptOne
  property x : 10
end script
script scriptTwo
  property x : 20
  property parent : scriptOne
  display dialog topLevel's x
end script
set topLevel to me
set x to 5
run scriptTwo -- 5
```

In that version of the code, `topLevel` and `x` (at top level) are global variables, not properties. This changes essentially nothing. `scriptTwo` still speaks of `topLevel's x`, regardless. There is a declaration of `topLevel` as global, so that `scriptTwo` will be able to see it (the downward effect of the declaration), and it is assigned the value `me` so that `scriptTwo` can refer to the top level by name. `x` is an implicit global that comes into existence when it is set in the next-to-last line. The really interesting part of the example is this line:

```
display dialog topLevel's x
```

Here, `scriptTwo` can access the global variable `x` by referring to it in terms of the top level's name, just as if `x` were a property. And it can do this even though there has never been, and never will be, an explicit global declaration for `x`. This is because the compiler is satisfied by the specification `topLevel's x`; it knows just where to look for this `x`, and that's all the compiler wants to know. At runtime, by the time `scriptTwo` runs, the `x` in question has a value, and all is well.

The upward effect of a global declaration identifies the declared variable with a top-level property, just as it would with a top-level global variable. For example:

```
property x : 5
script outerScript
  property x : 10
  script innerScript
    global x
    display dialog x
  end script
end script
run outerScript's innerScript -- 5
```

The global declaration of `x` in `innerScript` identifies `x` in this scope with the top-level property of the same name. We thus have another way of jumping past the scope where `x` is 10 to see the `x` at top level.

Delayed Declaration of Properties

Because of the nature of AppleScript's one-pass compiler, a property declaration may appear anywhere in its scope, not just at the start. It still provides the initial value for the variable at the start of its scope, not merely from the point where the declaration appears.

So, for example, this script runs, and displays 10:

```
display dialog x
property x : 10
```

The property declaration is dealt with by the compiler, so before the script starts running `x` already exists and has the value 10. Thus the first line of the script works even though no definition of `x` precedes it. Even though it works, this is poor style and is to be discouraged.

Redeclaration of Properties

It is *not* a compile-time error to redeclare a property as a local or a local as a property; but access to the property is lost within that scope. For example:

```
script myScript
  property x : 4
  display dialog x
  local x
  display dialog x
end script
run myScript -- 4, then error
```

The second attempt to display `x` fails because by that point `x` has been redeclared as local, and this local has no value. But the downward effect of the property declaration remains, so the property remains accessible at a deeper scope. Thus:

```
script myScript
  property x : 10
  local x
  set x to 20
  on myHandler()
    display dialog x
  end myHandler
  myHandler()
  display dialog x
end script
run myScript -- 10, then 20
```

It is a compile-time error to redeclare as global a variable declared as a property in the same scope:

```
property x: 10
global x -- compile-time error
```

It is *not* a compile-time error to do it the other way round, redeclaring a global as a property. This is merely taken as a delayed declaration of the property, and the global declaration has no effect. So:

```
global x
set x to 10
script myScript
  global x
  set x to 5
  property x : 20
  display dialog x
end script
run myScript -- 5
display dialog x -- 10
```

Within `myScript`, `x` is a property throughout; the global declaration inside `myScript` has no effect. The property `x` starts out with the value 20 before `myScript` runs, but `myScript` then sets it to 5, and this is the value that is displayed in the first dialog. The second dialog shows that the global `x` is unaffected. This code was written and executed on a closed course by a trained driver; please, do not attempt.

Lifetime of Variables

The *lifetime* of a variable means just what you think it means—how long the variable lives.

A local variable is born when it first is assigned a value, and dies when the scope in which it was born stops executing. A variable that behaves this way is sometimes called an *automatic* variable, because it comes into existence and goes out of existence automatically. For example:

```
on myHandler()
  local x
  set x to 5
  display dialog x
end myHandler
myHandler() -- 5
display dialog x -- error
```

Well, you already knew what would happen when that code runs. But what I'm saying now is something you can't see, and I can't quite prove, so you'll just have to believe me: by the time we get to the last line of that example, the local `x` inside `myHandler` isn't just unavailable, it's gone. It came into existence as `myHandler` was executing, and it went out of existence when `myHandler` finished executing.

A top-level entity other than a local variable is *persistent*. This means that its name and its value survives the execution of the script. This becomes interesting and relevant if you execute the same script twice.

Here's a simple example. Create this script in a script editor program and run it:

```
property x : 5
set x to x + 1
display dialog x -- 6
```

Now run it again, without doing anything else. Here's what happens:

```
property x : 5
set x to x + 1
display dialog x -- 7
```

This amazing result is possible because AppleScript has a memory. (You did reread “Maintenance of State” in Chapter 4, didn't you?) Your compiled script is in AppleScript's memory. After the script is executed, AppleScript retains the compiled script, and along with it, all the top-level entities that resulted from its execution. This includes properties, so after the first execution of the script, AppleScript is remembering that the script has a property `x` and that its value is 6. Thus when you run the script a second time and `x` is incremented, it becomes 7.

Now, at this point, you are saying: “But wait! I can see `x` being initialized to 5 right in the first line of the script. So what about that line? Are you saying that, the second time the script is executed, AppleScript is ignoring that line?” Well, it's really just a matter of what *initialize* means. It means to give a value to something that has no value. The second time the script is executed, `x` has a value already, remembered from the previous execution. So the property declaration has all the usual downward effects of a property declaration; but the initialization part of it has no effect, because `x` doesn't need initializing.

You can do the very same thing with a top-level global variable, but it's a little trickier because you need a way to initialize the global—to give it a value if it doesn't have one, but not if it does. Obviously this won't do:

```
set x to 5
set x to x + 1
display dialog x -- 6
```

You can execute that over and over, and it just displays 6 over and over. The problem is that `set` is not a mere initialization; it sets `x` regardless. Thus you keep resetting `x` to 5 each time you execute the script. This means that you probably don't believe me when I say that the value of `x` is being remembered between executions. To get around this, we have to use stealth (and some syntax that hasn't been discussed yet—try blocks are explained in Chapter 12):

```
try
    set x to x + 1
on error
    set x to 5
    set x to x + 1
end try
display dialog x -- 6, then 7, and so forth
```

Run that repeatedly. Now do you believe me?

A script object defined at top level is a top-level entity, so it and all script objects defined within it, and therefore all their properties, also persist after execution. For example:

```
script outerScript
  script innerScript
    property x : 5
    on increment()
      set x to x + 1
    end increment
  end script
  tell innerScript to increment()
  display dialog innerScript's x
end script
run outerScript -- 6, then 7, and so forth
```

Now, nothing lives forever, so just how long does all this persistence persist? Well, for one thing, it all comes to an end if you edit the script. That's because altering the script means that the script must be recompiled, and at that point the contents of the old compiled script, including the values of the top-level entities from the previous execution, are thrown away from AppleScript's memory. That's why throughout this section I've been telling you to execute the script multiple times without doing anything else.

The really, *really* surprising part, though, is that this persistence can survive the saving and reloading of the script in a script file. Unfortunately you can't see this if you save or open the script file using the current Script Editor. It doesn't work with Smile, either. Smile has another way of implementing persistence between sessions. So try it with the old version of the Script Editor (1.9), or with Script Debugger. Create and run this script several times:

```
property x : 5
set x to x + 1
display dialog x
```

Now save the script as a compiled script file, and quit, just to prove to yourself that AppleScript's own memory of the value of *x* is well and truly erased. Now open the compiled script file again and execute it. The incrementing of *x* picks up right where it left off previously.

This mechanism is not automatic. AppleScript itself has no way to enforce file-level persistence, because AppleScript itself doesn't deal in files. It is up the environment that's talking to the AppleScript scripting component, after it asks AppleScript to run the compiled script file, to save AppleScript's copy of the compiled script back into the compiled script file after execution. If it doesn't do this, then the compiled script file won't contain the new values, and the values won't persist. Fortunately most environments, including applets, as well as script runners such as the Script Menu or BBEdit's Scripts menu, are well-behaved in this regard. But as we've just seen, the current version of Script Editor is not. This inconsistency can be troublesome.