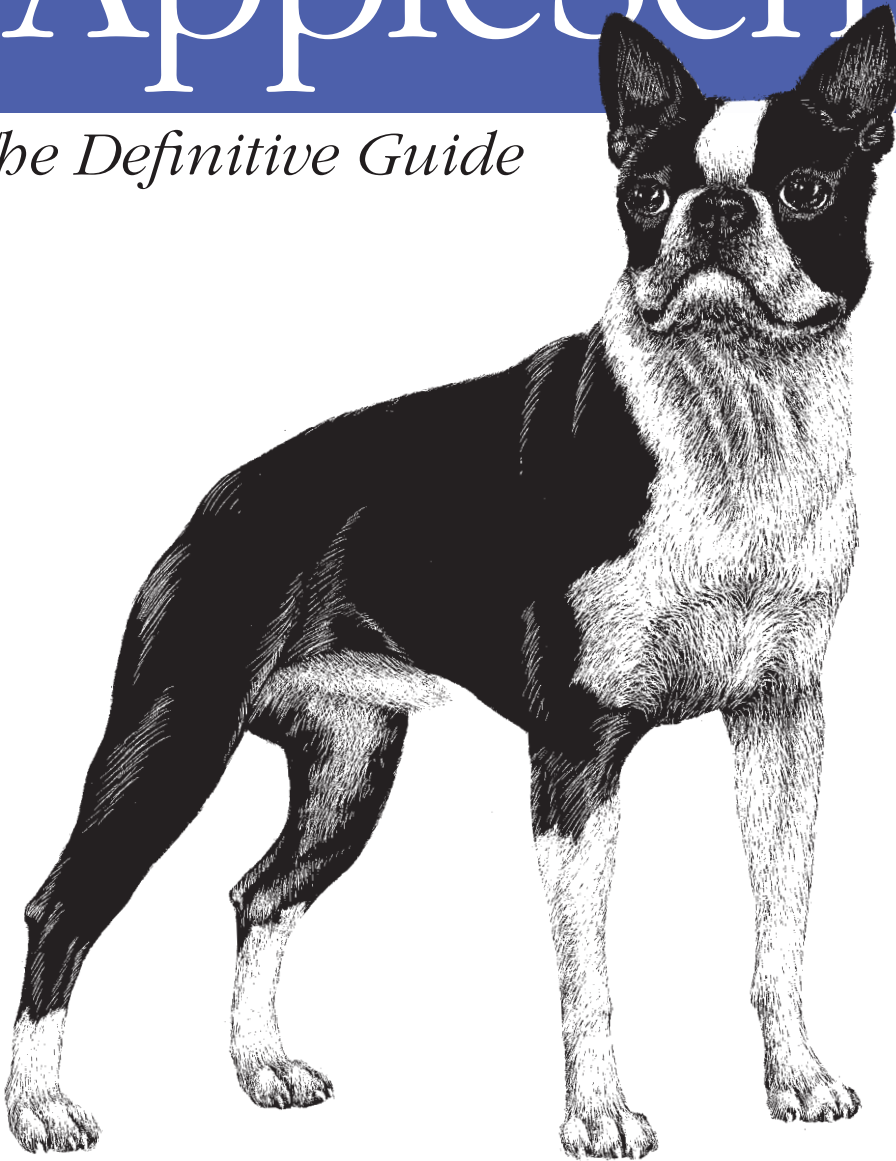


Scripting and Automating Your Mac

2nd Edition
Covers Mac OS X Tiger

AppleScript

The Definitive Guide



O'REILLY®

Matt Neuburg

Scope

By *scope* is meant the ability of one region of code to see variables in another region of code. The rules of scope for AppleScript are remarkably involved—as involved of those of any computer language I know. They are a perennial source of pitfalls for beginning and experienced programmers alike. If you don't understand the rules of scope, or (even worse) if you try to ignore them, you'll find your scripts mysteriously going wrong in confusing ways. This chapter discusses the rules of scope, along with a powerful advanced scope-related feature of AppleScript—closures.

Regions of Scope

Every AppleScript program has regions of scope. Some regions of scope 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. In other words, the regions of scope are *nested*. The top level of the script is a script object and is a region of scope. Any other regions of scope are inside this, and are created by the presence of *scope blocks*: handler definitions and script object definitions. (See Chapter 6.)

In Example 10-1, I've sketched a sample script with scope blocks nested in various combinations. At every point where code can go, I've put a comment distinguishing that region of scope by number. So, scope 1 is the top level of the script itself (its implicit run handler); scope 2 is the code within `handlerOne`, a handler defined at the top level of the script; scope 3 is the code within `scriptOne`, a script object defined within `handlerOne`; and so on.

Example 10-1. Regions of scope

```
-- scope 1
on handlerOne()
  -- scope 2
  script scriptOne
```

Example 10-1. Regions of scope (continued)

```
-- scope 3
end script
-- scope 2
end handlerOne
-- scope 1
script scriptTwo
-- scope 5
on handlerTwo()
-- scope 6
end handlerTwo
-- scope 5
script scriptThree
-- scope 7
end script
-- scope 5
end script
-- scope 1
```

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. To understand scope is to know what kinds of variable can belong to a given region of scope and in what other regions each variable is in scope.

Note that visibility is visibility and access is access. If code can see a variable in any manner, it can access it, which means it can both get and set that variable. In the examples in this chapter I will mostly use getting rather than setting, because that's the simplest way to detect when code *can't* see a variable (the attempt to fetch the value of an undefined variable results in a runtime error—see Chapter 7).

Kinds of Variable

With regard to scope, we must distinguish four kinds of variable:

Top-level entities

Top-level entities are script properties declared, and script objects and handlers defined, at the top level of a script or script object. (See “Top-Level Entities” in Chapter 8.)

Explicit locals

An explicit local is a variable announced by a local declaration, which looks like this:

```
local x
```

Explicit globals

An explicit global is a variable announced by a global declaration, which looks like this:

```
global x
```

Undeclared variables

An undeclared variable is none of the above. It's just a name that your code suddenly starts using. This is legal, but when you do it, AppleScript assigns the variable a scope, in ways that may surprise you.

The scope of a variable, and what kind of variable it is (these amount to the same thing), are determined at compile time.

Scope of Top-Level Entities

A top-level entity is visible in the scope where it is defined, and at all deeper levels nested in that scope subsequent to where it is defined.

Let's take, for example, a script property. The script property `x` here, declared at the top level of the script as a whole, is also in scope inside a script object and a handler:

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

If a top-level entity is in scope in a script object, a script object nested at a deeper level may declare a top-level entity with the same name. This deeper name will *over-shadow* the first entity's name within that deeper scope. Thus in this example there is never the slightest ambiguity as to what is meant by `x` at any point:

```
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
```

Regions of scope outside a script object cannot see that script object's top-level entities. But if they can see a name referring to that script object, they can ask to access that script's top-level entities, as explained in "Top-Level Entities" in Chapter 8. In the same way, a region of scope at a deeper level can access an overshadowed name:

```
script scriptOne
  property x : 10
  script inner
    property x : 20
    display dialog scriptOne's x -- 10
  end script
end script
script scriptTwo
  display dialog scriptOne's x -- 10
  display dialog x -- error: The variable x is not defined
end script
run scriptOne's inner
run scriptTwo
```

In that example, `scriptOne` is a top-level entity of the top-level script (that's where it's defined). For that reason (and because the name `scriptOne` is not overshadowed), `script inner`, which is nested two levels down within the top-level script, can see `scriptOne`. The name `x` is overshadowed within `inner`, so `inner` cannot see `scriptOne`'s property `x`. But because it can see `scriptOne`, it can ask `scriptOne` for access to it. For the same reason, code inside `scriptTwo` can see `scriptOne`. It is outside `scriptOne`, so it can't see `scriptOne`'s property `x`. But since it can see `scriptOne`, it can ask `scriptOne` for access to it.

A curious fact is that a top-level entity is visible at its own level to code even before it is defined. That's because the initialization takes place during compilation, before any code runs. So, for example, a script object:

```
run myScript -- Howdy
script myScript
  display dialog "Howdy"
end script
```

A handler:

```
myHandler() -- Howdy
on myHandler()
  display dialog "Howdy"
end myHandler
```

And even a property:

```
display dialog x -- Howdy
property x : "Howdy"
```

This sort of thing, however, especially with a property, is generally regarded as confusing and poor style. Also, code at a deeper nested level can see a higher-level entity only after that entity has been defined. This doesn't work:

```
run myScript
script myScript
  display dialog myOtherScript's x -- error: The variable myOtherScript is not defined
end script
script myOtherScript
  property x : "Howdy"
end script
```

A handler appears to be an exception, but it isn't really:

```
run myScript
script myScript
  myHandler()
end script
on myHandler()
  display dialog "Howdy"
end myHandler
```

That worked, but not because code inside `myScript` can see `myHandler`. It can't see it—`myHandler` is not in scope here—but it can call it! That's because of the rules of message-passing, which have nothing to do with the rules of scope. (See “Inheritance” in Chapter 8.) The handler call `myHandler()` in `myScript` reaches `myHandler` at the top level because the top level is `myScript`'s parent. To show that code inside `myScript` can't see `myHandler`, consider the same script without the parentheses after `myHandler`, so that the code is not calling `myHandler` but merely mentioning it:

```
run myScript
script myScript
  myHandler -- error: The variable myHandler is not defined
end script
on myHandler()
  display dialog "Howdy"
end myHandler
```

Note that a script object defined *not* at the top level of a script object (or script) is *not* a top-level entity. Basically, it's an implicit local (see “Scope of Undeclared Variables,” later in this chapter), which means that such a definition must precede any reference to itself, and that it can't be seen outside the scope where it is defined. Interestingly, when you define a script object at the top level of a handler, the rule about the definition preceding a reference to itself is enforced by the compiler (with a mysterious error message):

```
on myHandler()
  run myScript
  script myScript -- compile-time error: The myScript property is specified more than once
    display dialog "Howdy"
  end script
end myHandler
myHandler()
```

Scope of Locals

An explicit local is a variable whose name appears in a local declaration. A local declaration can declare several variables at once, delimited by comma:

```
local a
local b, c, d
```

A local variable is visible only within the very same region of scope where it is declared—not outside it, and not at a deeper level. (But there’s an exception, which we’ll come to in a moment.)

Local variables completely disambiguate a name within their own scope, and local variables in other scopes can have the same name without conflict. Suppose a script object starts like this:

```
script myScript
  local x
```

The local declaration for `x` means that from now on when code in this script object’s scope says `x` it means this local `x` and no other. What’s more, 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 -- error: The variable x is not defined
end script
run myScript
```

Observe how completely different this is from what would have happened if `x` had been a top-level property. Here, there is a variable called `x` and it is defined, but it is declared local and therefore is visible only within its own scope. 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. (I’ll explain later just what `x` AppleScript takes the `x` of `display dialog x` to be.)

A local declaration overshadows the visibility of a top-level entity from a higher level:

```
property x : 5
script myScript
  local x
  display dialog x -- error: The variable x is not defined
end script
run myScript
```

But this overshadowing affects only the scope of the local declaration—not a deeper scope:

```
property x : 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. Even though `myScript` overshadows the top-level property `x` with a local `x` declaration, this has no effect on `myHandler`, which still sees the top-level property `x`. (This makes sense, because it can't see `myScript`'s local `x`.) When `myHandler` sets `x` to 10, that is the same `x` as at the top level. When `myScript` sets `x` to 20, that's its local `x`, and the value displayed in the last line is unaffected.

In a handler, the variable names used for parameters in the definition of the handler are local within the handler. Naturally, a handler may also declare further locals:

```
on myHandler(what)
  local x
  set x to 7
end myHandler
myHandler("howdy")
display dialog what -- error: The variable what is not defined
display dialog x -- error: The variable x is not defined
```

Now we come to the great exception: a script object defined in a handler can see the handler's local variables. To a script object in a handler, the handler's locals are like top-level entities: they are visible to the script object and to scopes nested within it. (We took advantage of this rule in “Power Handler Tricks” in Chapter 9.) Thus:

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

I think the reason for this rule is that a handler can contain a nested scope (a script object) but has no top-level entities. A handler can't declare a property, so without this rule, it would have no encapsulated way to expose values to a nested script

object. Similarly, without this rule, code in a script object in a handler would be unable to see a script object earlier in the same handler:

```
on h()
  script s
    display dialog "howdy"
  end script
  script ss
    run s
  end script
  run ss
end h
h() -- howdy
```

Scope of Globals

An explicit global is a variable whose name appears in a global declaration. A global declaration can declare several variables at once, delimited by comma:

```
global a
global b, c, d
```

Globals are rather complicated. I'll divide the discussion into two parts: the downward effect of a global declaration, and its upward effect.

Global Declarations: The Downward Effect

The downward effect of a global declaration is very much like a top-level entity declaration. A variable declared global is visible in the scope where it is defined, subsequent to the point where it is defined, and at all deeper levels within that scope, subsequent to the point where it is defined.

For example:

```
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`.

But the following code does not work:

```
set x to 5
on myHandler()
  display dialog x
end myHandler
global x
set x to 10
myHandler() -- error: The variable x is not defined
```

We keep setting `x` like mad, but to no avail; the global declaration doesn't come until after the handler definition, so code inside the handler can't see `x`.

Just as with a top-level entity, a global's downward effect is overshadowed by a local declaration of the same variable name at a deeper level, and this overshadowing affects only the scope of the local declaration—not a deeper scope.

Global Declarations: The Upward Effect

The upward effect of a global declaration of a variable is to identify that variable with a variable at the top level of the script with the same name (creating this variable if it does not already exist). This means that separate regions of scope can share a variable simply by declaring it global:

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

That example demonstrates that the two handlers are sharing the same variable `x`. Now let's demonstrate that this `x` is being instantiated at the top level:

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

The fact that the last line works, rather than generating a runtime error complaining that `x` is not defined, shows the call to `setX` has defined `x` at the top level.

The status of a global created in this way is interesting. First, observe that it is not automatically visible downwards, as it would be if a global declaration (or a property declaration) were present at the top level. This doesn't work:

```
on setX()
  global x
  set x to 5
end setX
on getX()
  display dialog x
end getX
setX()
getX() -- error: The variable x is not defined
```

Even though `x` is a global defined at the top level, as we proved in the previous example, yet now `getX` can't see it. There is no global `x` declaration at a higher level than `getX`, so there is no downward effect to endow `getX` with the ability to see `x`. And `getX` itself now contains no global `x` declaration, so in effect it has failed to put in a request of its own to see `x`.

Global as static

A local is not persistent—it goes out of scope and is destroyed each time its scope finishes executing—but a global is. So a global is one way to implement a static variable (that is, a variable that will persist between executions of the local scope):

```
on incrementOrReport(report)
  global ct
  if report then return ct
  try
    set ct to ct + 1
  on error
    set ct to 1
  end try
end incrementOrReport
incrementOrReport(false)
incrementOrReport(false)
incrementOrReport(false)
incrementOrReport(true) -- 3
```

However, that is really poor programming style; a script object and a property would let you do this in a far more encapsulated way.

Global as script property

A global is, in fact, persistent between script executions, just like a script property. This is a little tricky to prove, because we need a way to initialize the global only just in case it isn't already defined; to do this, I'll have to use some syntax we haven't discussed yet (see "Errors" in Chapter 19):

```
on setX()
  global x
  try
    set x to x + 1
  on error
    set x to 5
  end try
end setX
on getX()
  global x
  display dialog x
end getX
setX()
getX() -- 5, then 6, and so on
```

Indeed, a global isn't merely *like* a script property; it *is* a script property. Again, I have to be a little tricky to prove it to you:

```
on setX()
  global x
  set x to x + 1
end setX
on getX()
  global x
  display dialog x
end getX
setX()
getX() -- 6, then 7, and so on
property x : 5
```

In that example, I deliberately postponed the property declaration until after all the other code, to show that it isn't the property declaration itself that is making *x* visible in lower scopes (remember, a top-level entity is visible to nested scopes only after its declaration). Thus it must be the global declarations within *setX* and *getX* that are identifying *x* with the script property *x* at the top level.

Here's another way to show that a global is a script property: I'll use a global to give the anonymous top-level script a name, and then I'll speak of another global as a property of the top-level script. This is a rewrite of a device from "The Implicit Parent Chain" in Chapter 8, using globals instead of properties at the top level:

```
global topLevel, x
set topLevel to me
set x to 5
script s
  property x : 10
  display dialog topLevel's x
end script
run s -- 5
```

Observe that a global declaration lets a scope region refer to an entity defined at the top level of the script, even when its name is overshadowed by a top-level entity at a deeper level:

```
global x
set x to 10
script outer
  property x : 20
  script inner
    global x
    display dialog x
  end script
end script
run outer's inner -- 10
```

Without the *global x* declaration in *inner*, *inner's x* would be *outer's* property *x*, and the script would display 20. This script works the same way if the top-level *x* is declared as a property, not a global; they are the same thing. Thus we have another

solution to the problem of how to speak of the top-level entities of a script as a whole even though it is anonymous (see “The Implicit Parent Chain” in Chapter 8).

The double effect of global declarations

In working with explicit globals, remember that every global declaration has both the downward and upward effect at the same time:

```
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
```

In `myScript`, the `global x` declaration does two things: it creates the top-level global `x` (upward effect), and it gives `myHandler` access to it (downward effect). Thus `myHandler` is able to set the value of the top-level global that will later be seen and displayed by `getX`.

Scope of Undeclared Variables

In AppleScript, you do not have to declare variables. When you use a name that, by the preceding rules of scope, is not an existing variable, AppleScript does not complain; rather, it creates the variable for you. How it does this depends upon the location of the code that uses the nonexistent variable name:

Code at the top level

The variable is created as a global. There is no explicit global declaration, so there is no downward effect, but other scopes can see this variable through a global declaration. I call this an *implicit global*.

Code not at the top level

The variable is created as a local. I call this an *implicit local*.

Let’s illustrate an implicit global first:

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

The first line never said explicitly that `x` should be a global. But it clearly is one, since when `getX` comes along and asks to see a global called `x`, the `x` created in the first line is what it sees. (Incidentally, you can move the “set `x` to 5” line to after the `getX` handler definition and the script will still work; the important thing is that the global `x` be defined by the time `getX` runs, not necessarily before `getX` itself is defined.)

Incidentally, a variable created implicitly in a script’s top-level explicit run handler is an implicit global as well, just as if you’d declared it at the absolute top level:

```
on run
  set howdy to "Howdy"
  sayHowdy() -- Howdy
end run
on sayHowdy()
  global howdy
  display dialog howdy
end sayHowdy
```

That’s a very odd rule, as in no other respect (I believe) is an explicit run handler treated like the absolute top level; usually, it’s treated like a handler, in which case the implicit variable would be local. This shows the extremes the language goes to in order to cope with undeclared variables.

Now for an implicit local:

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

Clearly we are now talking about two different variables called `x`. We have already established that the top-level `x` is global; but `getX` is evidently not seeing it. It is setting to 10, and displaying, some other `x`; its actions are having no effect on the top-level `x`, as the last line proves. So `getX`’s `x` must be local to `getX`. To prove that it is truly local and not some sort of magic downward global within `getX`, we need another test:

```
script s
  set x to 10
  on myHandler()
    display dialog x
  end myHandler
  myHandler()
end script
run s -- error: The variable x is not defined
```

But if we reverse the structure—a script object in a handler, instead of a handler in a script object—then it works:

```
on getX()
    set x to 10
    script s
        display dialog x
    end script
    run s
end getX
getX() -- 10
```

That isn't because `x` is not local; it's because of the special exceptional rule that a script object in a handler can see the handler's locals.

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 to declare all your variables—even (especially!) your locals. Unfortunately, AppleScript gives you no help with this; you can't make it warn you when a variable is undeclared. Rather, it silently declares it for you. This behavior, doubtless intended to make programming in AppleScript easier, is in fact the source of more mistakes and confusion among AppleScript programmers than any other feature of the language.

The trouble is that if you are lulled into a false sense of security because 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, or to function as persistent storage between calls. But the damage is done, just because of the upward effects of a global declaration. And to think that you could have prevented this by declaring your top-level `x` local to start with.

The converse is also true. Pretend that 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 search elsewhere to learn the scope of `x` within `myHandler`!

A particularly tricky problem is presented by script objects loaded from a compiled script file on disk (see Chapter 8). For example:

```
set f to (path to desktop as string) & "lib.scpt"
run (load script alias f)
x -- "howdy"
```

Hokey smokes! How is that possible? Where did this variable `x` come from? It came from the script in the file `lib.scpt`, which simply says this:

```
set x to "howdy"
```

The `x` in `lib.scpt` is an implicit global, and when you load and run that script, the global infects your own script's namespace. This is extremely cool or extremely scary, depending on your point of view. It seems cool to be able to make a variable (which can equally be a script object or handler) magically appear, name and all, in your script. My own view, however, is that globals, implicit or explicit, are for this very reason to be avoided in scripts that will be loaded into other scripts. After all, if you didn't know that `lib.scpt` was going to do this, and your script had its own top-level `x`, then loading and running `lib.scpt` would make your script malfunction mysteriously.

The ideal way to manage scope, in my view, would be to declare everything. If you need a variable to be visible in this scope and deeper scopes, then declare a property. (In a handler, merely declaring a local will do.) Otherwise, declare a local. Use global declarations sparingly, only for special occasions when separate regions of scope need shared access to a value. If you follow these rules, then any undeclared variable must be a free variable (see the next section). Thus you always know what everything is. Unfortunately, as I've said, AppleScript doesn't help, so you can't easily and reliably follow these rules, even if you want to.

A local or global declaration, by the way, does not have to appear right at the top level of a scope block; it can be embedded in a control structure. However, it isn't an executable statement; it is seen and enforced by the compiler, not the runtime engine. In this example, 1 is not 2, so the local `x` declaration is never "executed," but the example shows that `x` is local anyway:

```
if 1 is 2 then
    local x
end if
set x to 7
on myHandler()
    global x
    display dialog x
end myHandler
myHandler() -- error: The variable x is not defined
```

That example is silly, but the ability to declare a variable within a control structure is not, because it encourages and helps you to (you guessed it) declare your variables. You could be deep in the middle of typing a complicated control structure when you

suddenly introduce a new variable, and you realize that you want to declare it. You don't have to hunt back through your code looking for a place to declare the variable at the top level; you can declare it where you are and get on with your work.

Free Variables

A variable 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. A free variable in AppleScript is identified with a top-level entity or global through the downward effect of its declaration. 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 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:

```
global x
set x to 5
on myHandler()
    display dialog x -- x is a free variable
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.

Free variables' *values* are determined at runtime; but the identification of the fact that a variable *is* a free variable, and its association with some particular globally visible variable, happens during compilation. This way of resolving the meaning of free variable names is called *lexical scoping*.

So, in the previous example, it is important that `x` is declared global before `myHandler` is defined. If you move the `global x` declaration to after the `myHandler` definition, the

script generates an error at runtime. Even if `x` is declared global before (temporally) `myHandler` runs, 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 when the code runs, `myHandler`'s `x` isn't defined. This is another reason why you should declare your variables, and not only declare them but declare them early, even though AppleScript doesn't require you to do so.

Here's an example with two top-level entities as free variables. One is a property, the other is a script object:

```
property x : 5
script myScript
    display dialog x -- x is a free variable
end script
script myOtherScript
    set x to 20 -- x is a free variable
    run myScript -- myScript is a free variable
end script
set x to 10
run myScript -- 10
run myOtherScript -- 20
```

Redeclaration of Variables

Redeclaration of variables is a nasty edge case, testing the limits of the language. Luckily, the compiler mostly stops you from doing it. But there is a great deal of legal nuttiness, especially among declarations at the top level of a script. Essentially AppleScript is hoist with its own petard: having ruled that variables can be declared implicitly, and that global variables exist at top level even when they are declared elsewhere, AppleScript is simply trying to cope coherently with the consequences.

Here are some examples of what happens when you redeclare a variable.

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

```
set x to 5
local x -- compile-time error: Can't declare x as both a local and global variable
```

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

```
on getX()
    display dialog x
    global x -- compile-time error: Can't declare x as both a local and global variable
end getX
```

By the same token, it is a compile-time error to declare a handler parameter as global within the handler:

```
on myHandler(what)
    global what -- compile-time error: Can't declare x as both a local and global variable
end
```

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

```
on getX()  
  global x  
  local x -- compile-time error: Can't declare x as both a local and global variable  
end getX
```

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

```
on getX()  
  local x  
  global x -- compile-time error: Can't declare x as both a local and global variable  
end getX
```

At the 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 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 the global variable is effectively overshadowed by the local 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: The variable x is not defined
```

After the first two lines, there is a 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 apparently lost its access to this global variable. It can recover this access, however, through the use of the keyword `my`:

```
global x
set x to 5
local x
set x to 10
x -- 10
my x -- 5
```

The same thing happens if a global is implicitly redeclared as local by a repeat block (see “Repeat With” in Chapter 19):

```
global x
set x to 10
repeat with x from 1 to 5
end repeat
x -- 5
my x -- 10
```

It is *not* a compile-time error to redeclare a top-level entity as a local or a local as a top-level entity, but access to the top-level entity is lost within that scope:

```
script myScript
  property x : 4
  display dialog x
  local x
  display dialog x
end script
run myScript -- 4, then error: The variable x is not defined
```

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 top-level entity in the same scope:

```
property x: 10
global x -- compile-time error: Can't declare x as both a local and global variable
```

It is *not* a compile-time error to do it the other way around, redeclaring a global as a top-level entity. This is merely taken as a delayed declaration of the top-level entity, 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.

Closures

A *closure* is one of those delightfully LISPy things that turns up in AppleScript. The subject is rather advanced, though, so don't feel you have to understand everything in this section at once.

It turns out that a script object may capture certain aspects of its context, maintaining this context even though the script object may run later in a different context. For example:

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

That is extremely odd. It violates the rule stated earlier (“Free Variables”) about the value of free variables being determined at runtime. By the time `myScript` runs, its free variable `x` has been set to 10, yet the dialog displays 20. The proximate cause turns out to be the mysterious line “set dummy to myScript.” If that line is removed, the dialog says 10, just as we expect. Yet it is hard to see what difference that one line can make. It's not as if we ever do anything with the variable `dummy`, after all. We simply assign to it and forget about it. So what's going on?

The rule appears to be that the mere act of assigning a script object variable to another variable—it can equally be a copy as a set—causes the script object to become a closure. A closure is a scope block *plus the values of its free variables at that moment*.

The example is structured in three parts so as to demonstrate the phenomenon fully. First, a property declaration precedes the script definition; this is how `x` inside the script definition becomes a free variable at compile time. Then, the script object is assigned to another variable; at that moment the closure is formed. The free variable has a different value by this time, so this is the value that gets frozen into the closure. Finally, the free variable's value is changed again and the script object is executed; but the script object was already turned into a closure, so the change in the value of top-level `x` has no effect on it.

If, at the start of that example, we substitute a global declaration for the property declaration, the example doesn't work: we don't get a closure. Rather, the dialog displays 10, the value of `x` at the moment the handler is executed—the normal, nonclosure behavior. So *only free variables whose value is supplied by a top-level entity can form a closure*.

Closures and Handlers

So far, we've generated a closure in accidental circumstances. In fact, this feels like a bug; the mere act of assigning a script object to a variable hardly seems to warrant freezing that same script object into a closure, and seems to be more a trap we might fall into, causing our script to misbehave mysteriously, than a feature we would use deliberately. Now, however, let's turn our attention to a situation where we might actually like to generate a closure: when returning a script object from a handler.

Sure enough, it works. A closure is generated at the time we run the handler and generate the script object. Whatever the value of the script object's free variables are at that moment, if that value comes from a top-level entity, that's the value they retain:

```
property x : 5
on h()
  script myScript
    display dialog x
  end script
  return myScript
end h
set x to 10
set s to h()
run s -- 10
set x to 20
run s -- 10
set x to 30
run s -- 10
```

As before, if we replace the property declaration in the first line with a global declaration, there's no closure, and the dialogs say 10, then 20, then 30. Similarly, if we put a global `x` declaration at the start of `h`, there's no closure. To get a closure, we need a top-level entity to come shining down from above into our script object's scope.

Because we're inside a handler, there's one more type of variable we need to consider. Remember, a script object inside a handler can see the handler's locals. So a handler's locals can be free variables in the script object. Can *they* generate a closure? Yes, they can:

```
on h()
  local x
  set x to "howdy"
  script myScript
    display dialog x
  end script
  return myScript
end h
set s to h()
run s -- howdy
```

A handler's parameter variables are locals. This means we can feed a parameter into a handler and capture it in a closure produced by the handler:

```
on scriptMaker(what)
  script myScript
    display dialog what
  end script
  return myScript
end scriptMaker
set s to scriptMaker("Hello")
run s -- Hello
```

Wait—do you recognize that example? It comes from “Handler and Script Object as Result” in Chapter 9, except that there's a line missing. Previously, we captured the handler's parameter in the script object by initializing a property to it:

```
property x : what
```

Now it turns out that, thanks to closures, there was no need for that line. Similarly, we could rewrite the `makeFilterer` example (from the same section) to use a closure, changing the name `criterion` to `crit` everywhere and eliminating this line:

```
property criterion : crit
```

I hope I'm communicating a sense of how marvelous closures are. A script like this shouldn't even be possible. The parameter `what` is local to the handler `scriptMaker`, and goes out of scope—ceases to exist—when `scriptMaker` finishes executing. Nothing in `myScript` explicitly copies or stores the value of this `what`. Yet in the last line, a copy of `myScript` is executed successfully in a context where there isn't even a name `what` in scope. That's because, mysteriously, invisibly, it brings along its own context where there *is* a name `what` in scope. That's a closure.

Closures and Stored Script Objects

Closures also come into play when a compiled script file is to be executed with `load script` or `run script` (see “Persistence of Top-Level Entities” in Chapter 8). There’s more to a compiled script file than meets the eye. The `store script` command saves a script object into a compiled script file, but it also saves that script object’s *context*, so that if the script object has free variables, it will still work. (You can’t see this context unless you use Script Debugger, but it’s there.) The `load script` command loads this context, and the `run script` command runs within it.

Consider what happens when we create a compiled script file like this:

```
set f to (path to desktop as string) & "myScript.scpt"
global greeting
set greeting to "Howdy"
property farewell : "Byebye"
script s
    display dialog greeting
    display dialog farewell
end script
store script s in file f replacing yes
```

Both `greeting` and `farewell` are free variables in the script object `s`. Only the two `display dialog` lines are being stored in a compiled script file, but that is not enough information to make sense of these variables. Therefore their context is stored in the file as well.



Actually, AppleScript makes no attempt to decide at this point what’s a free variable and what’s not. It just stores the script object’s *entire* context in the file. Thus, in this case, `f` ends up in the compiled script file as well. This makes no difference to how the stored script object will run, because the script object makes no reference to `f`. It might make a difference to you if you are strongly security-minded, though, as prying eyes with the proper tools can read the value of `f`. Also it can make the resulting compiled script file unnecessarily large. A trick to prevent `store script` from storing unwanted contextual material is to set the script object’s parent to AppleScript—but then of course no local context is stored at all (and the rest of the examples in this section will break). Storage of context is an all-or-nothing proposition.

With `run script`, the situation is simple. No script object is generated. The compiled script file runs within its stored context and that’s that:

```
set f to (path to desktop as string) & "myScript.scpt"
run script alias f -- Howdy, then Byebye
```

If we use `load script` to load the compiled script file into a script, things are more complicated. Now we have a script object, and that script object has a context within the current script as well as the context with which it was saved. How will those two contexts interact?

```

set f to (path to desktop as string) & "myScript.spt"
set s to load script alias f
global greeting
set greeting to "Bonjour"
property farewell : "Au revoir"
run s -- Bonjour, then Byebye

```

The first dialog displays the value of `greeting` from the new, current context. But the second dialog displays the value of `farewell` from the old, stored context. This result actually makes sense in light of what we already learned about closures. Recall that a free variable whose value is a top-level entity, such as a property, makes a closure; a free variable whose value is a global does not. So the value of `farewell`, which was stored in the compiled script file as a free variable whose value was a property, is unaffected when the compiled script is loaded into a new context. But in the case of `greeting`, things are different. This was a global originally, so it does not form a closure. The *fact that this is a free variable*, however, is remembered; and the variable remains free. When the compiled script file is loaded into our script, it looks to our script to supply a value for this free variable. The supplied value can be a top-level entity (usually a property); it can be a declared global, as in our example; it can even be an *implicit* global. This last fact is surprising, so I'll demonstrate:

```

set f to (path to desktop as string) & "myScript.spt"
set s to load script alias f
set greeting to "Bonjour"
property farewell : "Au revoir"
run s -- Bonjour, then Byebye

```

The point is that there must be *something* named `greeting` at a higher level to which the script object's free variable `greeting` can look to obtain a definition, or there will be a runtime error saying that `greeting` is not defined.

The same thing happens if there is a global declaration in the stored script object. Let's say we store our script object like this:

```

set f to (path to desktop as string) & "myScript.spt"
script s
  global greeting
  display dialog greeting
end script
store script s in file f replacing yes

```

When we load the stored script into a new context, there's no closure for `greeting`. But `greeting` is declared global, so we can supply a value:

```

set f to (path to desktop as string) & "myScript.spt"
set greeting to "Howdy"
run (load script alias f) -- Howdy

```