AppleScript

Adam Goldstein

POGUE PRESS™
O'REILLY®

CHAPTER
5

Controlling Files

Most people have a love-hate relationship with their computer. Sure, your Mac is a great tool for when you want to edit images or video, send and receive email, or play Halo. But your computer also serves as a digital file cabinet: a place where you can create and store files, move them around, organize them in folders, trash them when they're no longer needed, and copy them to another disk or computer on the fly.

Files, of course, are nothing more than individual packages of information that you keep on your hard drive. But for all the filing tasks they perform, most computer users tend to handle files *manually:* drag this file here, create a new folder there, and so on. After a while, these mundane tasks are what make people start to hate their computers.

If you're sick of dealing with your files one at a time—and taking up half your day in the process—there's no better tool in your arsenal than AppleScript. By commanding the Finder, AppleScript lets you:

- Move all the files off your desktop in one fell swoop (page 90)
- Back up an important folder to a separate hard drive, just in case your computer dies (page 93)
- Rename all the files in a folder—without having to type their new names individually (page 113)

For these jobs and more, AppleScript can save you annoyance, tedium, and—most of all—time.

Note: The example scripts from this chapter can be found on the AppleScript Examples CD (see page 24 for instructions).

File Path Boot Camp

The one thing AppleScript *can't* save you from is the fact that files are essentially geeky things. Mac OS X's Unix heritage, while great news for programmers, also means that old Mac fans have to adapt to a few new file conventions—how to name files, what programs to open them with, and so on. Therefore, before you jump head-first into controlling files with AppleScript, there are a few things you should know:

- **In Mac OS X, files should always have a file extension.** A *file extension* is a short abbreviation added after a period in a file name. Microsoft Word files, for example, end in .doc, while sound files often end in .mp3 or .aiff.

  To Mac OS X (and Windows, for that matter) a file extension reveals what kind of information a file holds. In many cases, a file extension also tells Mac OS X which program should open a file: .doc files open in Microsoft Word, while .psd files open in Photoshop. (Of course, certain types of files can open in several different programs; .jpg files, for example, can open in just about any image-viewing program on the planet.)

Note: As a general rule, *folders* should not have file extensions. The exceptions are *bundles*–little folders that masquerade as files (page 34), like the .key files that Keynote produces.

To see what's inside in a bundle, Control-click the bundle in the Finder and select Show Package Contents from the shortcut menu. In the new window that appears, you can sift through the files that comprise the package, discovering, for example, that Keynote "files" are actually made up of dozens of smaller files.

UP TO SPEED

Path Notation

As described on page 77, a *path* is a Unix-esque way of describing where a file or folder resides on your hard drive. When you want to specify the lowest level of your hard drive, you simply specify the Unix path */* (a single forward slash). Similarly, when you want to refer to an item *inside* your hard drive, you must *begin* the item's path with a forward slash.

However, when specifying a path, folders must also *end* in a forward slash. That means the path to your Applications folder would be */Applications/* (the first slash to tell Mac OS X to look in your hard drive, and the last slash to tell Mac OS X that Applications refers to a folder).

When you refer to a *file,* however, you omit the trailing slash. The path to your Library → Fonts → Times New Roman file, therefore, would be */Library/Fonts/Times New Roman,* with slashes after Library and Fonts (since they're folders) but no slash after Times New Roman (since it's a file).

When you want to refer to your Home folder, you have two choices. You can specify the folder the normal way, by typing */Users/yourUsername/* (substituting your actual username for *yourUsername,* of course). Or you can use the convenient Unix shortcut (*~/*), which tells Mac OS X "substitute the actual path to my Home folder here".

If you want to refer to a file on a disk *besides* your startup disk, you have to begin your path with */Volumes/*. Just follow that with the name of the disk and another slash–like */Volumes/Backup Drive/* for a disk named Backup Drive–and the path now refers to your specified disk.

And something to note if you come from the Windows world: in places where you would have formerly used a backslash (\) in a path name–to identify folders, for example–use a forward-slash now. It's just one more instance of how Windows is, well, backward.

Although certain programs don't *require* file extensions, it's still a good idea to use them. That way, if you ever need to send a file to a Windows user, you won't get back an angry email asking you to *resend* the file with an extension so your recipient can actually open it.

- **A *path* is a string that tells you how to get to a certain file or folder.** Each item in path is separated by a forward-slash (/) in Mac OS X—a by-product of your computer's Unix heritage. That means the path to your Home → Desktop folder would be */Users/yourUsername/Desktop/*, while the path to your copy of TextEdit would be */Applications/TextEdit.app*.

  When you want to play with a path in AppleScript, you can use special type of information called a *POSIX* file. (POSIX is nerd lingo for "portable operating system interface," which basically means that file paths can be used anywhere, on any computer that supports the POSIX standard. To learn more about POSIX, you can read up on it online at *www.satimage.fr/software/en/file_paths.html*.) To get the path to your Desktop folder, for instance, you'd write the following, replacing *yourUsername* with your actual one-word username:

  ```
  POSIX file "/Users/yourUsername/Desktop/"
  ```

  Still, you'll find that most commands (like *choose file*, for presenting an Open dialog box [page 97]) use the *alias* type to refer to files. The *alias* format separates each folder in a path with a colon, rather than a forward slash. To get the *alias* to your Desktop folder, for example, you'd write this:

  ```
  alias ":Users:yourUsername:Desktop:"
  ```

Note: Of course, you should replace *yourUsername* with your actual username. If you don't know what your username is, you can look it up in System Preferences → Accounts; you'll find your username in the Short Name field.

- **You can open any file or folder with the *open* command directed at the Finder.** For example, to open Library → Desktop Pictures → Aqua Blue.jpg (the image that appears behind Mac OS X's login dialog box), you could write:

  ```
  tell application "Finder"
      activate --Bring the Finder forward
      open POSIX file "/Library/Desktop Pictures/Aqua Blue.jpg"
  end tell
  ```

Note: You'll notice a couple of oddities when you run this AppleScript. First off, the *open POSIX file* statement gets changed to *open file*. Then, all the forward slashes are converted to colons, and AppleScript inserts the name of your hard drive at the beginning of the path string. None of these changes affect what your code actually does; AppleScript just makes these changes so it understands what you're asking it to do.

If you prefer to write your code using the more common *alias* type, you could rewrite the previous script as follows:

```
tell application "Finder"
    activate
    open alias ":Library:Desktop Pictures:Aqua Blue.jpg"
end tell
```

Either way you write the script, the Aqua Blue image opens and shows up on your screen.

Displaying Folders

When you're working on a bunch of related documents at once, you might want to jump quickly to their folder in the Finder. Normally, of course, you'd switch to the Finder and navigate through your hard drive to get to the correct folder. Or perhaps, if you're a power user, you've already put the folder in the Finder's Sidebar for easy access. Either way, though, you have to switch to the Finder and open a new window, which is a massive waste of time.

Why go through all those steps when you can get AppleScript to do it for you? Using AppleScript, you can save a folder-opening script as an application (page 33) and place the script on the Dock for easy access. From then on, all you'll need to do is click the script's icon in the Dock, and a Finder window pops open and takes you right to the folder you want.

"But wait," you say, "I could just put the folder's *icon* on the Dock, no script required." You are, of course, correct—and your method is what most people use for accessing commonly used folders. The trouble is, when you click a folder's icon on the Dock, you never know where the folder's window will open onscreen, or whether it'll be in List, Column, or Icon view. Plus, a folder icon on the Dock can open only one *specific* folder, whereas a script can open multiple folders at once—like your Music and Pictures folders—as shown the following example:

```
tell application "Finder"
    activate
    open the folder "Users:yourUsername:Music"
    open the folder "Users:yourUsername:Pictures"
end tell
```

Again, just save this script as an application (page 33), and then drag the script's icon to your Dock. From then on, you'll be just one click away from opening two folders at once.

Tip: Of course, if you have *more* than two folders you'd like to open simultaneously, you can insert extra *open* commands in the previous script for those folders as well.

Opening a Folder with AppleScript (Reprise)

As you've seen on pages 23 and 78, there's more than one way to open a folder from AppleScript. If you want to open your Applications folder, for example, you'd have five separate choices:

```
tell application "Finder"
    activate
    make new Finder window to alias "Macintosh HD:Applications:" --Option 1
    make new Finder window to POSIX file "/Applications/"        --Option 2
    open alias "Macintosh HD:Applications:"                      --Option 3
    open POSIX file "/Applications/"                             --Option 4
    open the folder "Applications" of the startup disk           --Option 5
end tell
```

If you don't have any Finder windows open and you use one of these commands, the same thing would happen: a new Finder window would appear, taking you right to the Applications folder.

GEM IN THE ROUGH

AppleScript Shortcuts

After using AppleScript for a while, you might be wondering what exactly the word *the* does. The answer? Nothing. Using *the* in your scripts just makes them easier for humans to read–it makes no difference to AppleScript. You can prove it to yourself by running this script, for example:

```
tell application "Finder"
    open the the the folder "Applications"
of startup disk
end tell
```

The fact that you have three *the's* in a row makes no difference–AppleScript ignores them all.

That's not the only word you can omit in AppleScript, though. If you're writing a series of nested statements (like *if* [page 47], *tell* [page 43], or *repeat* [page 71]) for commanding a program, you can omit the second half of the *end* commands, and AppleScript fills them in for you automatically when you compile the script. For instance, you could write this script:

```
(* This script creates a bunch of new
folders in your Home folder; well, 15 of
them at least. *)
tell application "Finder"
    repeat 15 times
        if (count every folder in home) ¬
            is less than 15 then
            make new folder at home
        end
    end
end
```

When you compile or run this script (page 24), the last three lines are automatically expanded to include the correct commands (*end if, end repeat,* and *end tell*). You can even shorten the word *application* to *app* (on the fourth line), and AppleScript expands it automatically.

Finally, you can replace the nerdy-sounding word *of* with the more English-like *'s*. For instance, the following script would work just as well as the one shown at the top of this sidebar:

```
tell application "Finder"
    open the startup disk's folder ¬
        "Applications"
    (*Note the apostrophe-S instead of
    the word "of"*)
end tell
```

However, if you *already* have your Applications folder open, the commands behave differently:

- **Options 1 or 2** create *new* windows, each of which drops you off in the Applications folder. Option 1 employs the *alias* data type to do so (page 78), while Option 2 uses the new-age *POSIX file* data type (page 77). The *make* command, used in both Options, is described in detail on page 61.
- **Options 3, 4, or 5** simply bring the existing Applications folder to the front, without opening a new Finder window. Option 3 uses the aforementioned *alias* type, Option 4 uses the *POSIX* file type, and Option 5 uses neither (it simply tells the Finder which disk to look in). That's how *open* works.

The difference between these approaches is pretty small, of course, but it's important to understand: the *make* command always creates a new copy of something (in this case, a window), while the *open* command opens a new copy only if one doesn't already exist.

Changing a Finder Window's View

Why stop with just *opening* a folder when you can change the Finder window's view, too? If you've been using Mac OS X for more than a few days, you probably already know that the Finder has three viewing options (available at the top of the View menu), each of which provides a different, potentially timesaving way of looking at your files:

- **Icon view** shows you the icons for each item in a folder (Figure 5-1, top). That way, if you're browsing a folder full of Photoshop images, for example, you can find the particular image you want just by glancing at its icon.
- **List view** organizes the items in a folder alphabetically, by the dates they were created, or by just about any other criterion you want (Figure 5-1, middle). As an added benefit, list view shows more files in the same space than Icon view does.
- **Column view** gives you a hierarchical view of your hard drive, showing you the order of folders that contain the item you're looking at (Figure 5-1, bottom). This is the most compact way of looking through a folder, so it's worth using if you need to locate a file in a hurry.

But the fun doesn't stop there; each Finder view also has its own *options.* For example, you can change a window's background color from the default white to some other color—or post a picture behind a Finder window. Simply open the Finder's dictionary (page 44) and navigate to the *Finder window* entry (Figure 5-2).

As you can see, there are several useful properties you can set for Finder windows. If you want your script to automatically open the Applications folder in Column view, for example, you could modify your script like this:

```
tell application "Finder"
    activate
    open the folder "Applications" of the startup disk
```

```
        set the current view of the front Finder window to column view
    end tell
```

Now when you run your script, the Applications folder comes to the front and then quickly switches into Column view.

Figure 5-1:
The three ways of looking at your Library → Desktop Pictures folder. Top: Icon view (⌘-1) shows only four items in a window this small, but their icons are quite large. Use this view if you have bad vision.

Middle: List view (⌘-2) shows nine items in a window this small, and lets you sort the items however you'd like. Click the Date Created column to sort your files and folders from newest to oldest, for example, or click the Size column to sort the items from biggest to smallest. (Click either column a second time to reverse the sorting order.)

Bottom: Column view (⌘-3) shows 10 items in a window this small–the most of any of the three views. As a nice side effect, Column view also lets you see where the folder you're looking at is stored (in the left columns).

Tip: If you'd prefer, you can replace *column view* with *icon view* or *list view* to suit your file-viewing tastes.

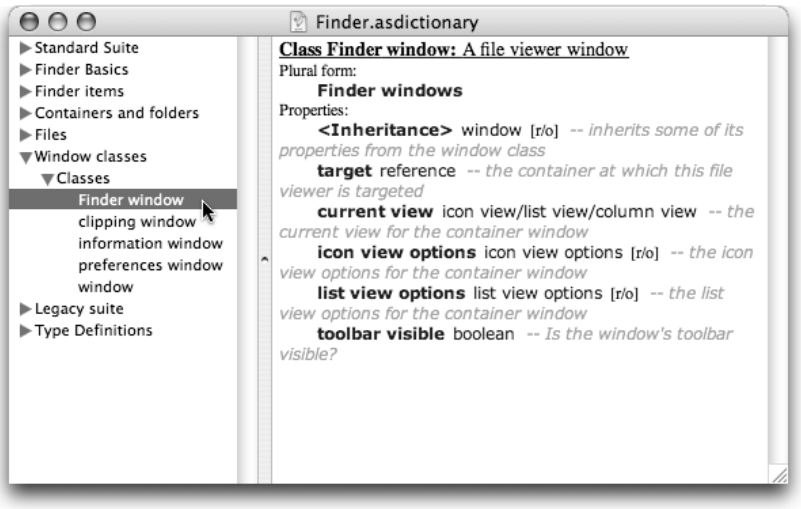


Figure 5-2: *In AppleScript, the properties of an object are all the attributes it has. Here you can see the properties of a Finder window: its current view, whether its toolbar is visible, and so on. Options marked with an [r/o] label are generally read only—that is, you can ask AppleScript for their values, but you can't change them.*

More Finder Window Settings

As you can see from the Finder's dictionary, there are a good number of extra properties you can set for Finder windows. You might have noticed with some puzzlement, however, that there's a big bold *<Inheritance>* label inside the entry for *Finder window*.

Your first instinct might be to assume that this property is off-limits to you—after all, it's got the *[r/o]* label, which usually means that you can't change the setting. As it turns out, however, inheritance is a powerful tool in AppleScript that puts even more control at your fingertips.

When you see the *<Inheritance>* label in a dictionary, look at the word immediately to its right. In the entry for *Finder window*, for instance, you'll see *window* next to *<Inheritance>*. That means that a Finder window, along with all its own properties, *also* has all the properties of a regular, everyday AppleScript *window*.

So what's that mean to you when you're up late at night writing scripts? It tells you to look in the dictionary entry for *window* in addition to the entry for *Finder window* (Figure 5-3), essentially doubling the number of commands you can send to a Finder window.

Armed with this information, you can add an extra command to your script:

```
tell application "Finder"
    activate
```

```
    open the folder "Applications" of the startup disk
    set the current view of the front Finder window to column view
    --Minimize the window to the Dock.
    set the collapsed of the front Finder window to true
end tell
```

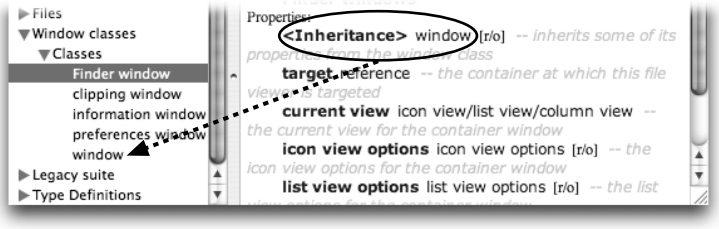


Figure 5-3: *Since Finder windows inherit from plain old windows, you should read the entry for window, too. You'll come across several additional properties you can control: where the window is onscreen (the position property), whether the window is big on the screen (the zoomed property), and even whether the window is minimized to the Dock (the collapsed property).*

Working with More than One Window

The current script is great for showing your Applications folder, but it won't save you that much time; you can always just click once on the desktop and use a keyboard shortcut (Shift-⌘-A) to launch the Applications folder instead.

You *really* start saving time when your script opens more than one folder. That way, you can have a quick way to view your Applications, Documents, Music, and Movies folders, for example—all with a single click.

UP TO SPEED

Inheritance

Inheritance is such a simple word. It means you get something for nothing, and when it comes to scripting, you can't get any better than that.

AppleScript's system of *inheritance* is pretty confusing at first, especially if you've never programmed before. The key to understanding it is seeing how similar it is to the real world, where some things have properties of others.

Say you have a Subaru. Now, your Subaru has properties that are different from most other cars: it uses four-wheel drive, for example. However, your Subaru also has properties in *common* with other cars: it has tires, a steering wheel, and brakes (you hope).

Think of it like this: your Subaru is a specific kind of car. In AppleScript, you'd explain the relationship like this: "*Subaru* inherits from *car.*" Your Subaru has all the properties of a car, plus some extras of its own.

Keep in mind, however, that inheritance is a one-way street. While every Subaru has the properties of a generic car, *not* every generic car has the properties of a Subaru.

Now, applied to AppleScript, this whole scheme just means that an object can use all the properties of the object marked *<Inheritance>*, but not the other way around. That's why you can use the properties from a *window* in your script that controls a *Finder window*, but you wouldn't be able to use the properties from a *Finder window* in a script that controlled a generic *window.*

One approach to opening multiple folders from your script is to simply copy and paste the existing commands repeatedly. (Then you'd insert the names of the folders as appropriate.) Using this method, your final script would look something like this:

```
tell application "Finder"
    activate
    open the folder "Applications" of the startup disk
    set the current view of the front Finder window to column view
    set the collapsed of the front Finder window to true
    open the folder "Documents" of home
    set the current view of the front Finder window to column view
    set the collapsed of the front Finder window to true
    open the folder "Music" of home
    set the current view of the front Finder window to column view
    set the collapsed of the front Finder window to true
    open the folder "Movies" of home
    set the current view of the front Finder window to column view
    set the collapsed of the front Finder window to true
end tell
```

POWER USERS' CLINIC

Moving a Window

Normally, moving a Finder window to a more convenient place on your screen is easy: you just drag any gray area of the window, and the rest of the window follows. This simple task, however, masks the fact that moving a window *precisely* where you want it is an exceptionally difficult task. In fact, getting a window to the exact top-left corner of your screen—so you can see everything on the right side of your monitor—is harder than writing a computer book.

That's why AppleScript's window-placing features are so convenient. By setting a window's *position* property, you can send the window anywhere on (or off) the screen. Here's how:

```
tell application "Finder"
    activate
    open the folder "Applications" ¬
        of the startup disk
    set the current view of the front ¬
        window to column view
    --Move the window where you want it:
    set the position of the front window ¬
        to {200, 100}
end tell
```

In that script, you place the Applications window 200 pixels from the left edge of the screen and 100 pixels from the top edge of the screen—a good position if you want to leave room for additional windows at the *bottom* of your screen.

Keep in mind when writing scripts like this, however, that the menu bar is 83 pixels tall. Also, a Finder window needs a 5 pixel "barrier" from the left edge of the screen, so you can see the entire window. Therefore, if you wanted to place a window at *exactly* the top-left pixel on the screen, you would have to substitute the following command in the bold part of the previous script:

```
set the position of the front window ¬
    to {5, 83}
```

For a more detailed explanation of Mac OS X's odd window-positioning system, see page 124.

This kind of approach, unfortunately, has several downsides:

- **It repeats a lot of commands.** That means your script is longer than it has to be, which can become a problem if you start writing scripts that are hundreds of lines long.
- **It's annoying to use.** Each time you paste the commands again, you have to go back and change the portion of the command that needs modifying. Again, if you're working with a long script, this can be quite a nuisance.
- **If you want to change a command, you have to change it in multiple places.** For instance, if you wanted to change *column view* to *list view,* you'd have to do it four times.

Luckily, there's a solution to these problems: breaking the redundant commands into a separate portion of your script, known as a *subroutine.*

Subroutines

In AppleScript (and in other programming languages), a *subroutine* is a section of code that's meant to be used over and over again. Rather than having to retype a big block of code, a subroutine lets you write that code just once, assign a name *to the code*, and then simply use the subroutine's *name* whenever you want to run the corresponding code. Think of it like using a kitchen appliance: no matter what time of day it is, you can expect the "coffeemaker" appliance to behave the same way. Similarly, no matter what part of your script you run a subroutine from, you can expect the subroutine to run the exact same lines of code.

You can do anything you want in a subroutine: tally the points from a sports game, connect to a Web site, or anything else you're likely to do more than once in your script. (And incidentally, subroutines are sometimes called *handlers,* too.)

Now that you know what subroutines can do, it's time to put them to use in simplifying your Finder window script.

Defining a subroutine

In general, a subroutine looks something like this:

```
on subroutineName( <any variables being passed into the subroutine> )
    --Any commands you want in the subroutine go here
end subroutineName
```

You'll notice that, unlike *repeat, if,* and *tell* statements, subroutines begin with the keyword *on.* That's your way of telling AppleScript, "Hey! There's a subroutine here, and whenever I refer to this name, please run the lines of code that follow."

Note: Some people use the word *to* instead of *on* to introduce their subroutines. Either way is valid.

Running a subroutine

Calling a subroutine from your script couldn't be easier. (*Calling* is just geek-speak for "running the code contained in a subroutine.") You simply type the name of the subroutine, followed by parentheses, like this:

```
subroutineName( <any variables you want to pass into the subroutine> )
```

On the other hand, if your code is inside a *tell* statement, you have to preface your subroutine with the word *my*. That's your way of telling AppleScript, "I know I'm targeting a particular program with my script, but take a break for a second and run my *personal* subroutine, will ya?"

Variables in subroutines

When defining a subroutine, you don't have to put anything between the parentheses on the subroutine's first line. If you *do* choose to put variable names there, however, you'll need to use the same number of values when you *call* the subroutine from your script.

In other words, if you defined your subroutine like this:

```
on displayGreater(a, b) --Note that there are two variables
    if a > b then
        set theResult to a
    else
        set theResult to b
    end if
    display dialog "The greater number is: " & theResult
end displayGreater
```

you'd have to run the subroutine from your code like this:

```
displayGreater(10, 88) --You must provide two values
```

On the other hand, your subroutines don't *have* to accept variables at all. AppleScript is perfectly happy to run a subroutine defined like this, for example:

```
on displayPi( )
    display dialog pi
end displayPi
```

In that case, since the subroutine doesn't expect any values (indicated by the lack of variables between its parentheses), you'd run this subroutine with this simple command:

```
displayPi( ) --Display a dialog box showing pi
```

Note: *pi* is a special AppleScript keyword, a namesake of the famous irrational number.

Writing the appropriate subroutine

Now that you know how to write and call subroutines, you can eliminate the redundant portions of your Finder-window script (page 84). First, add this subroutine to the bottom of that script:

```
on columnAndMinimize( )
    tell application "Finder"
        set the current view of the front Finder window to column view
        set the collapsed of the front Finder window to true
    end tell
end columnAndMinimize
```

In writing this subroutine, you've isolated your script's redundant code. Now you can *erase* your script's redundant code by calling the *columnAndMinimize* subroutine as follows:

```
tell application "Finder"
    activate
    open the folder "Applications" of the startup disk
    my columnAndMinimize( )
    open the folder "Documents" of home
    my columnAndMinimize( )
    open the folder "Music" of home
    my columnAndMinimize( )
    open the folder "Movies" of home
    my columnAndMinimize( )
end tell

--Here's the subroutine:
on columnAndMinimize ( )
    tell application "Finder"
        set the current view of the front Finder window to column view
        set the collapsed of the front Finder window to true
    end tell
end columnAndMinimize
```

Note: Remember, you must use the word *my* before these subroutine-running commands because you're using those commands inside a *tell* statement.

Now, with this final script, you've not only saved several lines of code, but you've also made it easier to change your script's behavior in the future. If you ever want to modify the code now, it's simply a matter of modifying the subroutine *once,* rather than modifying four separate lines. Here are a few possible tweaks:

- To keep all your newly opened windows from minimizing, delete *set the collapsed of the front Finder window to true* from your subroutine.

- If you'd rather your new windows appear in List view or Icon view (page 80), replace *column view* with either *list view* or *icon view* in your subroutine.
- Finally, if you don't care what view your new windows use, just delete the entire *set the current view of the front Finder window to [whatever]* line from your subroutine.

Moving Files Around

Being able to display your files is useful, but it's only half of what the Finder can do. The other half, of course, is to *move* your files—putting them in a new folder, deleting them, and so on. With AppleScript, you can automate these actions and more.

Transferring Items from One Folder to Another

The simplest action you can perform on a file is dragging it from one folder (or disk) to another. With AppleScript, it's also one of the simplest actions you can control in the Finder.

The key to this trick is the *move* command. It's part of the Finder's Standard Suite (page 45), so it's a pretty common command. And, as the Finder's dictionary explains, the *move* command follows this simple structure:

```
tell application "Finder"
    move someItem to somePlace
end tell
```

Tip: In this example, *someItem* can be either a single item or a list of items. That makes the *move* command perfect for transferring whole clusters of files (or folders) in a single step. (See page 105 for the low-down on lists.)

And also, if you use the *move* command to transfer files from one *disk* to another, AppleScript (like the Finder) assumes that you mean to *copy* the files. If you want, you can then use the *delete* command (page 94) to erase the files from the original disk.

Now, if you're the sort of person who always saves downloads and email attachments to your desktop, you've probably noticed your desktop getting pretty full. You might have so many icons that you've had to shrink them down (View → Show View Options). Or perhaps your desktop is *so* cluttered that icons have started overlapping each other, obscuring all the important stuff you keep there.

No matter what the issue, AppleScript can help you clean up your desktop. With one fell swoop, a script can sweep up all the files and folders there, and stash them somewhere less intrusive.

Tip: You can use the *move* command for any number of other jobs, too: transferring sounds from an old folder to the Mac OS X–standard Music folder; moving downloaded files to a special Just Downloaded folder; or even sending files to another computer on your network (page 199).

Creating the destination folder

Right out of the box, your computer comes with one Desktop folder—the one you see right now. That won't help much, though, when you want to clear your desktop and sweep all those files into *another* folder.

Thankfully, you can create a new folder to hold everything from your old desktop. (In fact, if you're feeling especially creative, you could even name the folder Old Desktop.) Here's how:

```
tell application "Finder"
    make new folder at home with properties {name:"Old Desktop"}
end tell
```

As you've seen on page 61, the *make* command lets you create a new item (such as a document or folder) straight from AppleScript. The *at* option lets you specify where to create it (in this case, your Home folder). Plus, when you add the *with properties* option, you can specify various extra settings for the new item you create. (In this case, the *name* property gives the new folder a name, which is *Old Desktop*.)

After the *with properties* option, the settings you specify have to follow a special structure: they have to be surrounded by curly brackets, and each setting's name has to be followed by a colon and the value you want to use for it. For instance, to create a folder with the name Old Desktop, you have to use the property *{name: "Old Desktop"}*.

Tip: You can specify as many settings as you want inside the brackets. A more involved *make* command, for instance, could go something like this:

```
make new folder at home with properties ¬
    {name:"Old Desktop", comment:"Old Desktop files and folders"}
```

That creates an Old Desktop folder in your Home folder, but it also adds a *comment* to the folder. Then, at some point in the future, you could see your comment by selecting the folder in the Finder and choosing File → Get Info (⌘-I).

Eliminating the "already an item with that name" error

When you first run your script, it'll silently create a new folder named Old Desktop in your Home folder. The trouble is, if you already *have* an Old Desktop folder when you run the script, you'll see the error message shown in Figure 5-4.

The key to avoiding this problem is placing an *if* statement around the folder-creating command. That way, if the script discovers that there's already an Old Desktop folder, AppleScript just skips over the command for creating the folder and move on to the next command.

To add the *if* statement, just modify the script by adding the lines you see here in bold:

```
tell application "Finder"
    if not (the folder "Old Desktop" of home exists) then
```

```
        make new folder at home with properties {name:"Old Desktop"}
    end if
end tell
```

Figure 5-4: *AppleScript isn't very graceful about handling errors with the make command: it stops your entire script and presents a dialog box. Luckily, you can spare it the trouble by enclosing your make command in an if statement.*

The new *if* statement looks in your Home folder to see whether you have an Old Desktop folder there already (the *exists* part). If you don't have an Old Desktop folder, the script runs the next command (*make new folder*) and creates one. Now you'll never get that annoying dialog box when you run your script in the future.

Moving the files and folders

Now that your script can tell whether or not there's an Old Desktop folder, you can get down to business: moving the files and folders from your desktop into your Old Desktop folder. Luckily, this is just a matter of adding two *move* commands to your script:

```
tell application "Finder"
    if not (the folder "Old Desktop" of home exists) then
        make new folder at home with properties {name:"Old Desktop"}
    end if
    move every file of the desktop to the folder "Old Desktop" of home
    move every folder of the desktop to the folder "Old Desktop" of home
end tell
```

These *move* commands, as you could probably guess, take all the files and folders that sit on your desktop and deposit them into your Home → Old Desktop folder. In a single click of the Run button, your desktop's clean.

Tip: For the ultimate in convenience, add this script to your Script Menu (page 15). Then you can run it from any program, whenever you want. This can come in really handy when, for example, you're snapping a bunch of screenshots (with either Shift-⌘-3 for a partial-screen picture or Shift-⌘-4 for a full-screen picture) and you want to quickly move the images off your desktop (which is where Mac OS X saves them by default). It's a great trick for Mac book authors, especially.

Backing Up Files

Talk to any computer expert, and you'll be told the same thing: backing up your files is not an option, it's a *must*! Unless you're interested in joining the millions of people who've lost essential files, you should back up your files regularly.

There are plenty of choices for backing up your files; it's only a matter of picking the one with the features and price you like. Here are a few of the options:

- Commercial programs like Dantz's **Retrospect Desktop** (*www.dantz.com/en/products/mac_desktop/index.dtml;* $130) let you automatically back up important files at intervals you specify.

UP TO SPEED

Boolean Values

One of the cornerstones of all programming languages is the *Boolean* type. This simple kind of information has only two possible states: *true* and *false*. That makes it perfect for simple operations, such as determining whether something exists.

You can set Boolean variables just like any other variable:

```
set finderShouldQuit to true
set scriptDone to false
```

Boolean values also have special *operators* (keywords) you can use. If you ever took a logic course in high school, you'll instantly recognize the three basic operators: *and, or,* and *not.* These keywords let you combine two or more Boolean values in one command, doubling their power. For instance, operators let you check whether two conditions are *both* met in your script, or whether *either* is met. Here's how:

- If both sides of an *and* operator are true, it produces *true*. Otherwise, it produces *false*. For example:

```
display dialog (true and false)
    --That displays "false"
display dialog (true and true)
    --That displays "true"
```

- An *or* operator, on the other hand, produces *true* if either side is true. The only time it produces a *false* is if both sides are false. For example:

```
display dialog (true or false)
    --That displays "true"
display dialog (false or false)
    --That displays "false"
```

- The *not* operator works with only a single value (either *true* or *false*), and produces its opposite; for example:

```
display dialog (not true)
    --That displays "false"
display dialog (not false)
    --That displays "true"
```

You can use any of these operators in your *if* statements as well. That's why the script for creating the Old Desktop folder works: it checks to see if the Old Desktop folder already exists, and then applies a *not* operator to the result. That means that if the folder *doesn't* exist, the *if* statement is run; if the folder *does* exist, the *if* statement isn't run.

- Apple's own **Backup 2.0** program is a perk for using the .Mac service ($99 per year). You can use it to automatically back up files onto a CD or DVD, to your iPod, or even to your iDisk. See *www.mac.com* for more details about Backup 2.0, a mac.com email address, and all the other benefits of a .Mac membership.
- Shareware programs like Mike Bombich's **Carbon Copy Cloner** (*www.bombich.com/software/ccc.html;* $5) can back up your entire hard drive to another disk, quickly and easily. If you're looking for an inexpensive, simple backup solution, Carbon Copy Cloner is the perfect tool.
- The **Finder** (free, included with Mac OS X) can be used to back up files, too. Unfortunately, every file or folder you want to back up has to be copied—manually—by *you*. Don't use this method if you have lots of important files to back up; it'll take hours.

The problem with all these solutions, of course, is that they either cost money or are too time-consuming to use regularly. That's why AppleScript is a great alternative: you can customize the files you want it to back up, and it's completely free.

The duplicate Command

The *move* command is for transporting an item from one folder to another. The *duplicate* command, on the other hand, is for *copying* an item from one folder to another. The original file stays untouched, and an exact copy of that file is goes anywhere you specify. That location can be another folder, another partition of your hard drive, or another drive altogether (including a USB thumb drive, an external FireWire drive, or your iPod).

The way you use the *duplicate* command is very similar to the way you use the *move* command:

```
tell application "Finder"
    duplicate someItem to somePlace with replacing
end tell
```

Here's how the command breaks down:

- *duplicate* is the command directed at the Finder, to tell it to copy something.
- Everything that follows the *duplicate* command goes by the order of "what you want to duplicate" followed by "where you will save that duplicated copy." You replace the *someItem* variable with the name(s) of the files and/or folders you want to duplicate. Likewise, you replace *somePlace* with the name of the folder or disk where you want those duplicate copies to be saved.
- The *with replacing* bit tells the Finder to erase any older revisions of your files in the backup folder and replace them with the newer version. That way, you won't be stuck with all your month-old backups; you'll just have the newest versions of your files backed up.

Note, however, that the *with replacing* option considers only one thing: file names. If two files have the same name, the file that you're duplicating *always* replaces the one that's already there—even if the file that's already there is bigger, newer, and shinier than the one that you're duplicating.

Note: The *with replacing* option is case insensitive. If you duplicate myllamas.txt to a folder that already has MyLlamas.txt, for instance, Mac OS X considers them the same name, so it would replace the existing file (MyLlamas.txt) with the new file (myllamas.txt).

With that information in hand, you can write a simple backup subroutine, as shown here:

```
on backupFolderToDisk(startFolder, targetDisk)
    tell application "Finder"
        duplicate every file of startFolder to disk targetDisk with replacing
        duplicate every folder of startFolder to disk targetDisk ¬
            with replacing
    end tell
end backupFolderToDisk
```

Say you're a doctor and you want to back up your Patients folder to an external FireWire drive called Medical Backup. While you're at it, you'd also like to back up everything in your Home → Documents folder, just in case your hard drive gets damaged on the flight to your next medical convention.

The good news is that you already have a subroutine for backing up files to a separate disk, so you're halfway to a working script. The bad news is that your script doesn't actually *run* your subroutine anywhere, so your essential files never get copied over to your external drive.

To fix this, you just have to call your existing subroutine from elsewhere in your script. The new subroutine-calling lines (shown next in bold) are what actually tell AppleScript "Please run my backup commands":

```
backupFolderToDisk("Patients", "Medical Backup")
--Replace yourUsername below with your actual username
backupFolderToDisk("Macintosh:Users:yourUsername:Desktop:","Medical Backup")

--Here's the previous subroutine:
on backupFolderToDisk(startFolder, targetDisk)
    tell application "Finder"
        duplicate every file of startFolder to disk targetDisk with replacing
        duplicate every folder of startFolder to disk targetDisk with
replacing
    end tell
end backupFolderToDisk
```

Each time you call the *backupFolderToDisk* subroutine, the Finder whirs into action and copies your requested files to the backup disk (Figure 5-5).

Tip: Be sure to replace "Patients," "Medical Backup," and so on with the actual folders and backup disk you want to use.

And if you'd like to back up additional folders, just insert extra *backupFolderToDisk* calls at the top of your script.

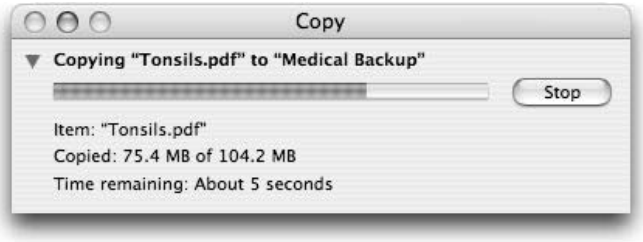


Figure 5-5:
This same window shows up whether you're copying files yourself in the Finder or having AppleScript do the copying for you.

When it's this easy, you have no excuse not to back up your Mac.

Tip: To make it even *easier,* you can schedule your backup script to run on certain days of the week. Page 261 has the details.

Deleting Files

So far, you've moved and copied files in the Finder from one place to another. There are some times, though, when you just want to get rid of a file—and AppleScript can do that too.

The key here is AppleScript's *delete* command. It works just like *move* or *duplicate,* except you don't have to specify where the deleted files should go (AppleScript automatically knows that deleted files should go in the Trash). Thus, a typical *delete* command would look something like this:

```
tell application "Finder"
    delete the file "Chihuahuas.doc" of the desktop
end tell
```

When you run this command—substituting the name of the actual file you want to delete, of course—you hear a satisfying *clunch* as the Finder wads up your file and

deposits it in the Trash can. If you don't hear this sound effect, three things could be wrong:

- **You've muted your speakers.** The fix: press the volume-up key or increase the volume in your System Preferences → Sound → Output tab.
- **You've turned off Mac OS X's sound effects.** To turn them back on, visit System Preferences → Sound → Sound Effects tab and turn on "Play user interface sound effects."
- **You don't have a Chihuahuas.doc file on your desktop.** Either get one, or replace *Chihuahuas.doc of the desktop* with the name of the file you want to delete.

An Example: Clearing Out Safari's Icon Cache

If you use Safari for a few weeks, visiting hundreds or thousands of Web sites, you'll probably notice a significant slowdown each time you load a page. That's caused, in part, by Safari's gigantic database of *favicons*—those little icons you see in Safari's Address bar (Figure 5-6).

Figure 5-6: *The Apple icon, shown here, is the favicon that goes along with Apple's Web site. If you add this site to your bookmarks (Bookmarks → Add Bookmark), the specialized icon will show up in the Bookmarks menu too.*

If you delete Safari's icon cache (which is stored in your Home → Library → Safari → Icons folder), you can give Safari a significant speed boost—and save a few megabytes of space while you're at it. Here's a script to automate the process:

```
tell application "Finder"
    delete folder "Icons" of folder "Safari" of folder "Library" of home
    display dialog "Would you like to empty the trash now?"
    (* If you click Cancel in the dialog box, the script ends here.
       Otherwise, it continues to the next line *)
    empty the trash
end tell
```

When you run the script, the Finder drops the Icons folder in the Trash, and then presents the dialog boxes shown in Figure 5-7.

The only other time you'd really use the *delete* command is when you have a folder that needs to be emptied regularly (like your Decade-old Home Videos folder, for example). Besides that, there's not much use for *delete*, since you can always achieve one-time erasures by choosing File → Move to Trash (or by pressing ⌘-Delete) in the Finder.

Figure 5-7:
Top: Your script shows you a dialog box asking whether it should continue, thereby emptying the Trash. Click Cancel to stop the script in its tracks.

Bottom: If you click OK, the Finder displays a progress bar as it erases everything in the Trash. That includes other things you've recently deleted–not just the Icon folder–so make sure that's OK with you before you proceed.

GEM IN THE ROUGH

The Reveal Command

As you go about your script-writing business, there might come a time when you want to show a file or folder in the Finder. After you've copied a file from one folder to another, for instance, you might want to display the new location for the file.

AppleScript makes this job easy using the *reveal* command. It works just like the File → Show Song File command in iTunes–that is, it shows you the folder that contains a given file in the Finder. Try this to select your copy of TextEdit in the Applications folder:

```
tell application "Finder"
    activate
    reveal the file "TextEdit.app" ¬
        of the folder "Applications" ¬
        of the startup disk
end tell
```

The *reveal* command is useful for showing folders, too. For example, you could highlight your Home folder (inside the Users folder) with this command:

```
tell application "Finder"
    activate
    reveal home
end tell
```

Picking a File from a Dialog Box

Back on page 60, you learned how to use the *display dialog* command to present information onscreen, and how to give feedback to your scripts while they're running. The trouble with that command, though, is that you can't choose a *file* with it. And when you're using Finder commands, you often *want* to choose a file for your script to work with.

That's where the *choose file* command comes in. Rather than having to specify an actual file name in your script, *choose file* uses Mac OS X's standard Open dialog box, letting you pick the precise file you want to work on (Figure 5-8). That way you can choose a different file for your script to operate on each time it runs.

Figure 5-8: *When you use the choose file command, you see the same Open dialog box that you see in other Mac OS X programs. There's only one difference: the choose file dialog box shows normally hidden files (like .DS_Store), too. There are some benefits to this feature: you can see all the Unix configuration files that Mac OS X uses, for example. However, if it bothers you to have all those hidden files clogging up your Open dialog box, just add the without invisibles option to the end of your choose file command.*

In its purest form, the *choose file* command can occupy a line all by itself—displaying an Open dialog box but doing absolutely nothing else:

```
choose file
```

Of course, it won't do much good just to display an Open dialog box on the screen; the *real* power comes when your script can figure out what file you chose. AppleScript makes this easy, too:

```
set selectedFile to (choose file)
(*The selectedFile variable now stores an "alias" [page 77] of the file you
chose*)
```

```
tell application "Finder"
    open selectedFile
end tell
```

When run, this script presents an Open dialog box, and then opens whatever file you chose. It's not going to win any programming awards, but it's a start.

Tip: If you'd like to pick out a folder instead of a file, use the *choose folder* command. For more information on these commands, check out the Standard Additions dictionary (page 50).

Showing When a File was Created

Admit it: you've got folders that you haven't cleaned out in months—or maybe even years. You've let your junk accumulate, putting off the day you have to sort through it. Now, using AppleScript, you can finally tell how long it's been sitting around, so you can brag to your similarly procrastinatory friends.

When you script the Finder, you have access to the *modification date* property for everything on your hard drive. To figure out when a file was modified, therefore, you simply have to tell AppleScript *which* file you want the information for. The *choose file* command provides the perfect opportunity to enlighten AppleScript as to your file of interest:

```
set selectedFile to (choose file)
tell application "Finder"
    set modDate to the modification date of selectedFile
end tell
display dialog "That file was last modified on: " & modDate
```

Still, this script isn't perfect. For one thing, it doesn't give you any perspective, like how many months ago the file was modified. Instead, it just tells you the *date* the file was modified, which isn't as easy to interpret at a glance.

One of AppleScript's nice features, though, is that you can subtract one date from another. It's a great way to figure out how long ago a file was modified—in days, months, or even years. Simply edit your script like this:

```
--Part 1:
set selectedFile to (choose file)
tell application "Finder"
    set modDate to the modification date of selectedFile
    --Part 2:
    set curDate to the current date
    --Part 3:
    if (the year of modDate) ≠ (the year of curDate) then
        set ageInYears to (the year of curDate) - (the year of modDate)
        display dialog "The file was changed " & ageInYears & " years ago."
    --Part 4:
    else
```

```
            if (the month of modDate) ≠ (the month of curDate) then
                set ageInMonths to (the month of curDate) - (the month ¬
                    of modDate)
                display dialog "The file was changed " & ageInMonths & ¬
                    "months ago."
            --Part 5
            else
                if (the day of modDate) ≠ (the day of curDate) then
                    set ageInDays to (the day of curDate) - (the day of modDate)
                    display dialog "The file is " & ageInDays & "days old."
                else
                    display dialog "The file was changed today."
                end if
            end if
        end if
    end tell
```

Note: You can type the ≠ symbol by pressing Option-=. Or, if you'd prefer, you can substitute the plain-English phrase *is not equal to* in place of the ≠ symbol.

This is the most involved script you've written so far. At first it looks pretty complicated, but it actually works fairly simply:

- **Part 1:** The script presents an Open dialog box, and sets *modDate* to the date the selected file was modified.

Tip: If you want to check when a file was created rather than when it was modified, use the *creation date* property instead of *modification date*.

- **Part 2:** The current date, as you're running the script, goes into the *curDate* variable.
- **Part 3:** The script checks if the year the file was modified is the same as the current year. If they're *not* the same, the script sets the *ageInYears* variable to the difference between the two years—and a dialog box tells you how many years ago the file was modified.

  On the other hand, if the file *was* modified this year, the script proceeds to the next part.
- **Part 4:** Now the script checks if the *month* the file was modified is the same as the current month. If they're different, the script shows a dialog box with the difference in months. Otherwise, if they're the same month, the script proceeds to the next part.
- **Part 5:** If the script has gotten this far, you know that the file was modified this month of this year. The only thing left to check, then, is whether the file was last modified *today*.

If it wasn't, the script calculates how many days ago the file was modified. It then displays that information in a dialog box.

On the other hand, if the file *was* last modified today, the script presents a dialog box informing you of that. At this point, every possibility has been covered, and the script ends.

As you'll surely notice, the script is full of nested *if* statements, which makes it hard to read. Luckily, AppleScript lets you merge an *else* statement (on one line) with a subsequent *if* statement (on the following line), creating an *else if* statement. Here's what the previous script would look like if you linked your *else* and *if* statements in that way:

```
set selectedFile to (choose file)
tell application "Finder"
    set modDate to the modification date of selectedFile
    set curDate to the current date
    if (the year of modDate) ≠ (the year of curDate) then
        set ageInYears to (the year of curDate) - (the year of modDate)
        display dialog "The file was changed " & ageInYears & " years ago."
    else if (the month of modDate) ≠ (the month of curDate) then
        set ageInMonths to (the month of curDate) - (the month of modDate)
        display dialog "The file was changed " & ageInMonths & "months ago."
    else if (the day of modDate) ≠ (the day of curDate) then
        set ageInDays to (the day of curDate) - (the day of modDate)
        display dialog "The file is" & ageInDays & "days old."
    else
        display dialog "The file was changed today."
    end if
end tell
```

Now the script is much easier to read, and still works exactly the same way.

Note: People use *else if* statements for all sorts of other tasks, too. For example, you can check the file format of a document in this way: *if* it's a Word document…*else if* it's a PowerPoint document…*else if* it's an Excel file, and so on.

You can also use *else if* statements to react to the magnitude of something: *if* there are less than 5 files on the Desktop, leave the files alone…*else if* there are between 5 and 20 files on the Desktop, copy them to a different folder…*else if* there are more than 20 files on the desktop, delete them, for example.

Saving Files

The last piece of the Mac OS X file puzzle is saving documents you already have open. AppleScript makes this simple: the *save* command works the same way as choosing File → Save.

Tip: Similarly, the *save as* command works the same way as choosing File → Save As (that is, when you provide a file path to the *save as* command, Mac OS X saves a *copy* of your current file in the location you specify).

Unfortunately, this trick works only in certain programs (TextEdit, Microsoft Word, and Safari, for example). To see if a particular program supports AppleScript-based saves, open the program's dictionary and see if the dictionary's Standard Suite (page 45) includes the *save* command.

POWER USERS' CLINIC

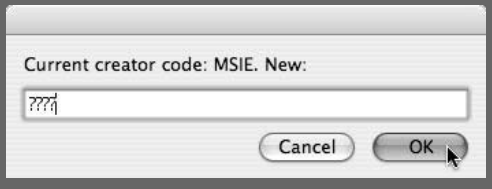
You Can't Judge a File by its Extension

Back in the days of Mac OS 9, before you had to put file extensions at the end of documents' names, your Mac knew what kind of files you had by their *type* and *creator codes*. The *type code* told your Mac what kind of information was stored in a file. For example, if a file's type code was "TEXT," that meant the file was just plain text, while a type code of "APPL" meant the file was an application.

The *creator code,* on the other hand, told your Mac which program produced a file. A file created with Photoshop would use the creator code "8BIM," while one created with AppleWorks would use "BOBO"—go figure.

The importance of type and creator codes in Mac OS X is reduced, but they're still around. In fact, if a file has a type and creator code, they *override* any settings for which program should open the file. That's why the help files that come with Photoshop won't open in your default Web browser: Adobe has set their creator code to "MSIE," so they'll always open in Internet Explorer.

Thankfully, AppleScript lets you modify type and creator codes–or get rid of them completely. The following script can help you do it:




```
set selectedFile to (choose file)
(* Get the file's current type and creator
codes: *)
tell application "Finder"
    set fType to the file type of ¬
        selectedFile
    set cType to the creator type of ¬
        selectedFile
end tell
(* Get the new type and creator codes you
want to use: *)
set newF to text returned of ¬
    (display dialog "Current type code: " ¬
    & fType & ". New:" default answer "")
set newC to text returned of ¬
    (display dialog "Current creator ¬
    code: " & cType & ". New:" default ¬
    answer "")
--Set the new type and creator codes:
tell application "Finder"
    set the file type of ¬
        selectedFile to newF
    set the creator type of ¬
        selectedFile to newC
end tell
```

If you'd like to banish the type and creator code from a file–so that Mac OS X will judge the file by its extension–enter "????" for both codes (or, if you want the details of erasing such codes, see *http://daringfireball.net/2004/02/setting_empty_file_and_creator_types*). If you'd rather just replace the existing codes with new ones, check out *http://kb.indiana.edu/data/aemh.html* for a list of type and creator codes for different programs and files.

An Example: Saving in TextEdit

As the epitome of Mac-ness, TextEdit does support the *save* command. Thus, if you needed a script to save your current TextEdit document, this one would work well:

```
tell application "TextEdit"
    activate
    save the front document
end tell
```

If you haven't saved your current TextEdit document yet, a Save dialog box appears. (The Save dialog box, as you know, lets you specify a name for the document and where you want it stored.)

Forgoing the Dialog Box

Of course, you could always save your files *without* using AppleScript. The real benefit of scripting Save operations, though, is that you can completely bypass the Save dialog box, saving you several precious seconds:

```
tell application "TextEdit"
    activate
    save the front document in ":Users:yourUsername:Desktop:kiwi.rtf"
    --Remember to replace yourUsername with your actual one-word username
end tell
```

The *in* option lets you specify the colon-separated path of the file into which you want to save your document (in this example, the *kiwi.rtf* file on your desktop). If there's a particular file you often save—say, your personal home page—you might find it useful to run the previous script whenever you want to save a *new* version of the document in the *old* location.

Note: **Make sure you specify a file, not a folder!** If you put a path to a folder after the *in* option, AppleScript overwrites that folder completely. And if that folder were your desktop, the script would instantly trash your Desktop folder and every file *inside* it, leaving your desktop files as mere memories. Here's what to avoid:

```
--DO NOT RUN THIS SCRIPT!!
tell application "TextEdit"
    activate
    --Wanna erase your Desktop? Here's how:
    save the front document in ¬
        "Macintosh HD:Users:yourUsername:Desktop:"
(* Since you specify the path to your Desktop, Mac OS X overwrites the
Desktop...permanently! *)
end tell
```

Instead, specify the actual *file* you want to save the document into. (You can tell you're specifying a file because it won't end in a colon.)

Saving All Documents at Once

There's one more timesaving trick to the *save* command: saving all your documents in one step. This can come in handy for those times when you have multiple files open and want to quickly save all of them—without switching to each document window individually and hitting ⌘-S. Use this script to get the job done:

```
tell application "TextEdit"
    activate
    save every document
end tell
```

When you run this script, TextEdit does the same thing it would do if you chose File → Save All. The nice thing about this script, though, is that it works in many programs that don't even *have* a Save All command, such as Microsoft Word, Safari, and even Script Editor itself. Simply change the *tell* statement to reflect the program you want to command, and run the script again.

For example, here's how that script would look if you wanted to use it with Word:

```
tell application "Microsoft Word"
    activate
    save every document
end tell
```

Now that you know how to open, move, copy, and save documents automatically, you can call yourself a true AppleScript filephile.