

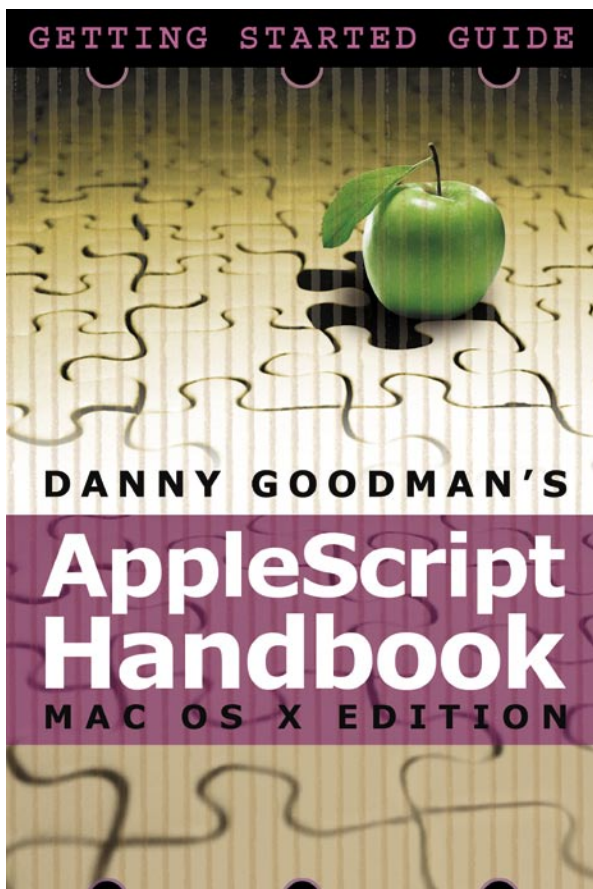
GETTING STARTED GUIDE



DANNY GOODMAN'S

AppleScript Handbook

MAC OS X EDITION



\$14.95 USD ISBN: 0-9744344-2-6

Version 1.0 - [Check for Updates](#)



SpiderWorks

For more great ebooks, order online
at <http://www.spiderworks.com>



Table of Contents

About this eBook	4	Additional Script Editor Windows	40
How to Use this eBook	5	Script Editor Advanced Features	42
Introduction to the Third Edition	6	Compiling (Checking Syntax)	42
Installing the Companion Software	8	Making a Statement	46
Chapter 1: The Tools You Need Are Free	9	Commandments	47
Where Stuff Is	9	Telling an Object What To Do	47
AppleScript Studio	12	Common Actions	48
The Script Menu and Scripts Folders	12	Where the Wording Comes From	49
Chapter 2: How to Learn AppleScript	16	Chapter 5: A Crash Course in Programming Fundamentals	52
Gradus ad Scriptum	17	Statements	53
If You've Never Programmed Before	19	Commands and Objects	53
If You've Done a Little Programming Before	20	Working with Information	56
If You've Programmed a Lot Before	21	Variables	57
If You've Programmed in HyperTalk Before	25	Expressions and Evaluation	58
Chapter 3: Your First AppleScript Script	27	Operators	60
Scriptable Applications	27	Chapter 6: Issuing Commands and Getting Results	62
Recordable Applications	28	Commands Provoke Action from Something	62
Attachable Applications	29	Where the Words Are	66
Script Editor	29	More About Parameters	68
Deciding What To Script	34	Getting Results	70
Recording Your First Script	34	"Built-in" Commands	71
Chapter 4: Writing AppleScript Scripts—An Overview	38	Traditional AppleScript Commands	73
Script Editor—The Bottom Pane	38	Chapter 7: Scripting Addition Commands and Dictionaries	92
		System Commands	94
		Finder Commands	113
		String Commands	127
		Numeric Commands	132

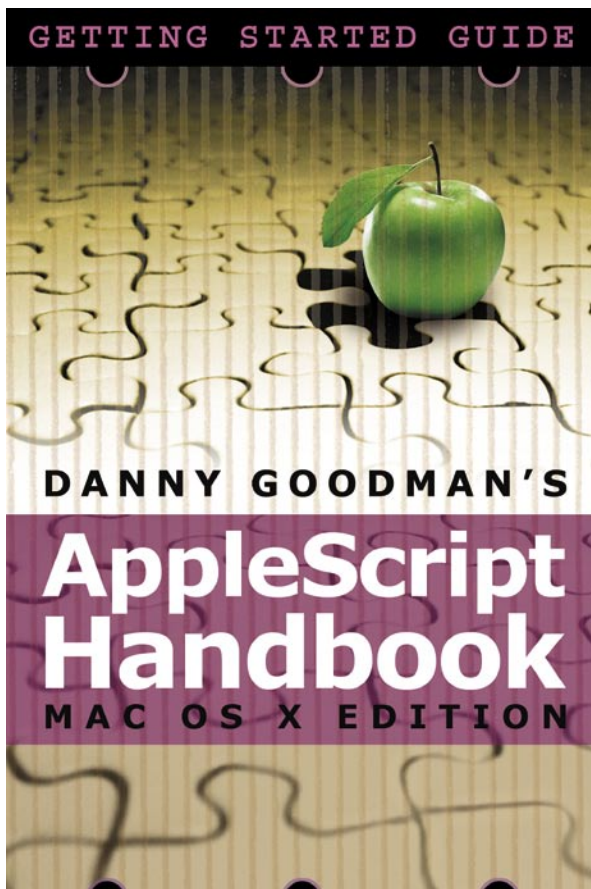


Table of Contents Continued

Script Commands	135	File Specification Class	228
User-Interface Commands	143	Integer Class	229
File Commands	154	International Text Class	230
Debug Commands	167	List Class	230
Date/Time Commands	167	Number Class	237
Clipboard Commands	169	Real Class	237
Folder Action Commands	174	Record Class	239
Internet Commands	176	Reference Class	241
Understanding Application Dictionaries	177	RGB Color Class	243
		String Class	243
		Styled Clipboard Text Class	248
		Styled Text Class	249
		Text Class & Unicode Text Class	249
		Unit Type Classes	250
Chapter 8: Describing Objects—References and Properties	182		
Objects In Real Life...	182		
...Translated to Computer	183		
References	184		
Container Syntax	185		
Default Object References	186		
Object Properties	187		
Object Reference Syntax	188		
Chapter 9: Working with Data—Values, Variables, and Expressions	214	Chapter 10: Going with the Flow (or Not): Control Structures	252
Kinds of Values	214	Shop Til You Drop	252
Variables and Value Classes	215	AppleScript Flow Control	253
Coercing Value Classes	216	Tell Statements	254
Coercion Caveats	218	If-Then Constructions	256
Value Class Details	219	Repeat Statements	259
Boolean Class	219	Timeout Flow Control	268
Class Class	221	Other Control Statements	270
Constant Class	222		
Data Class	223	Chapter 11: AppleScript Operators	271
Date Class	223	Four Operator Types	271
Working with Dates and Times	224	Integers, Reals, and Operators	276
		Strings and Operators	277
		Lists and Operators	278
		Records and Operators	280
		Booleans and Operators	281
		Coercing Values—the As Operator	283

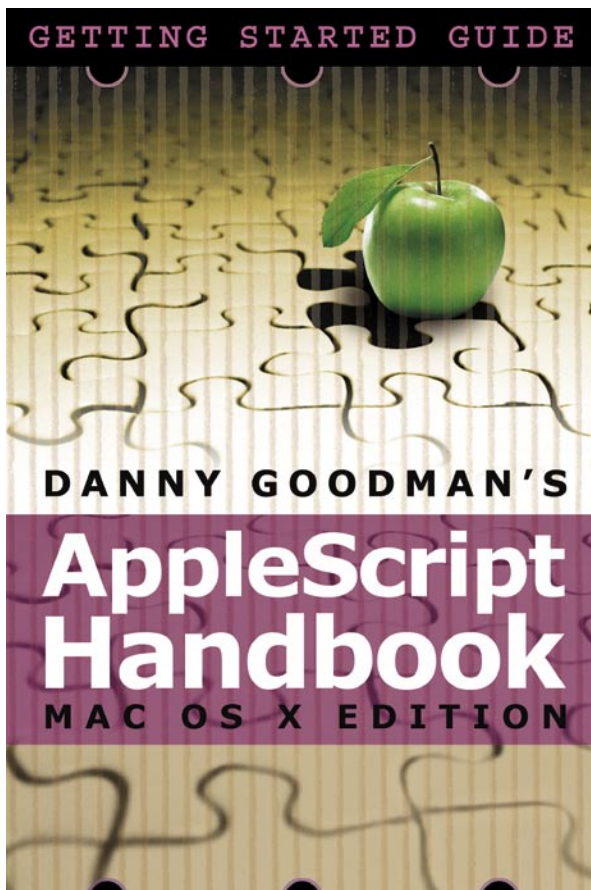


Table of Contents Continued

String Comparison Aids	286	Storing Libraries	331
Operator (and Reference) Precedence	288	Handlers in Attachable Applications	332
Chapter 12: Error Checking in Scripts	291	Chapter 15: Script Properties, Objects, and "Agents"	334
Why We Forget to Error Check	291	Script Properties	334
Anticipating Errors	293	Script Objects	338
AppleScript Errors	293	Advanced Object-Oriented Techniques	342
Trapping Cancel Buttons	295	Delegation	343
Purposely Generating Errors	296	Creating Droplets	345
Error Numbers and Messages	298	Agents	348
Chapter 13: Debugging Scripts	300	Chapter 16: Scripting Third-Party Applications	352
Script Editor Setup	301	Knowing the Program	352
Compile and Execution Errors	302	Approaching a New Program	353
Using The Result Pane	303		
Display Dialog	304	Appendix A: AppleScript Quick Reference	364
Event Log Pane	304		
Aural Clues: Beeps and Speech	309	Appendix B: ASCII Table	384
Try Statements	309		
"Commenting Out" Lines	310	License Agreement	388
A Debugging Demonstration	311		
Chapter 14: Using Subroutines, Handlers, and Script Libraries	316		
Subroutines and Scripts	316		
Subroutine Concerns	317		
Subroutine Scope	319		
Subroutine Definitions—Two Types	320		
Subroutine Parameters—By Value and By Reference	327		
Subroutine Variables	328		
Recursion	330		
Turning Subroutines into Libraries	330		



About this eBook

The Author

Danny Goodman is the author of numerous critically acclaimed and best-selling books, including *The Complete HyperCard Handbook*, *JavaScript Bible*, *Dynamic HTML: The Definitive Reference*, *JavaScript & DHTML Cookbook*, and *Spam Wars*. He is a renowned authority and expert teacher of computer scripting languages and has been deciphering high-tech for non-geeks since the late 1970s.

Visit Danny Goodman's official web site at <http://www.dannyg.com>



Author Acknowledgments

Although I have not forgotten the immense help the original AppleScript team was to my first foray into the technology in the early 1990s, I wish to acknowledge here those who contributed their efforts to this third edition. Thanks to Dave Mark and Dave Wooldridge for providing a publishing platform that got me off my butt to tackle this long-overdue update. David Fugate was gracious enough to handle the paperwork details that could have otherwise derailed this effort. And my deep appreciation goes to Ben Waldie who provided invaluable technical assistance.

Publisher Credits

Cover Design: **Mark Dame** and **Dave Wooldridge**

Interior Page Design: **Robin Williams**

PDF Production: **Dave Wooldridge**



How to Use this eBook

On-Screen Viewing

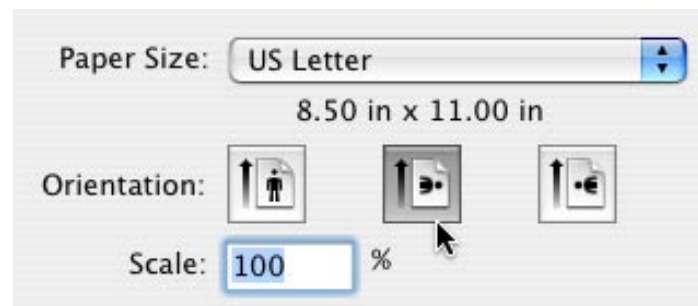
We recommend using Adobe Acrobat or the [free Adobe Reader](#) to view this ebook. Apple Preview and other third-party PDF viewers may also work, but many of them do not support the latest PDF features. For best results, use Adobe Acrobat/Reader.

To jump directly to a specific page, click on a topic from either the Table of Contents on the first page or from the PDF Bookmarks. In Adobe Reader, the PDF Bookmarks can be accessed by clicking on the Bookmarks tab on the left side of the screen. In Apple Preview, the PDF Bookmarks are located in a drawer (Command-T to open).

If your mouse cursor turns into a hand icon when hovering over some text, that indicates the text is a hyperlink. Table of Contents links jump to a specific page within the ebook when clicked. Text links that begin with “http” or “ftp” will attempt to access an external web site or FTP server when clicked (requires an Internet connection).

Printing

Since SpiderWorks ebooks utilize a unique horizontal page layout for optimal on-screen viewing, you should choose the “Landscape” setting (in Page Setup) to print pages sideways on standard 8.5” x 11” paper. If the Orientation option does not label the choices as “Portrait” and “Landscape”, then choose the visual icon of the letter “A” or person’s head printed sideways on the page (see example below).





Introduction to the Third Edition

The previous edition of this book reached store shelves over ten years ago. In the computer and computer book worlds, that's an eternity. Or two.

Eventually, however, the technology passed the book by, and the title went out of print at the original publisher, Random House. Despite the fact that parts of the book became outdated as early as the release of Mac OS 8, my loyal readers continued to recommend it to others who wanted to learn AppleScript. The demand was so great that I arranged to get the second edition reprinted through a print-on-demand publisher.

In the meantime, my writing focus had shifted to Web-related scripting technologies, particularly JavaScript and Dynamic HTML, where I wrote three different titles spread across eight editions since 1996. I also devoted more than a year of research and writing to the subject of spam in a recent book titled *Spam Wars*. All through that period, in the back of my mind, I hoped one day to return to AppleScript and bring the old book up to date. Although I had written a few scripts in the intervening years for my

own use, I had not been monitoring the changes to AppleScript from one OS release to the next. I knew that to do the right job on an update, I'd have to operate in catch-up mode, and practically teach myself again, just as I did for the very first edition. This "to-do" list item nagged at my conscience for years.

Along came an opportunity that had a lot going for it. If I updated the core portions of the book, it could be published initially in electronic form and offered at a price much lower than the typically high computer book prices. As I got underway, I was pleasantly surprised at the positive improvements to AppleScript features delivered with recent Mac OS X releases. Macs even shipped with numerous scriptable applications, including the ever-popular iTunes.

As often happens with a book revision, the job turned out to be far more intensive than I first imagined. Part of that came from my penchant for demanding to understand the roots of things before I write about them. I had to acquaint myself with



Introduction to the Third Edition

deeper aspects of Mac OS X programming than I had expected. In the process, I also wrote a fairly sophisticated AppleScript application that performs some vital daily data analysis and Web page content updates for one of my Web sites, involving Fetch and BBEdit, all triggered by a shell script on my Mac. The AppleScript buzz was back.

Thus, this update to my *AppleScript Handbook* became a labor of love. I was glad to volunteer my time to this project with the hope of inspiring a new generation of beginning and intermediate AppleScript users with contemporary commentary and examples that actually worked in today's environment.

I still believe that AppleScript is an outstanding technology that puts the Macintosh far ahead of the alternatives. As a mature technology with oodles of support from the developer community, AppleScript can inspire any thoughtful and creative Mac user to do great things.



Installing the Companion Software

Requirements

The collection of companion scripts and data files from this book are contained in a download called *HandbookScripts.sit*, which can be downloaded from the SpiderWorks Customer Download Center at <http://www.spiderworks.com/extras/>. To login, you will need your Customer Username and Password that was listed in your order confirmation e-mail.

This book assumes that you are running Mac OS X 10.3 (Panther) or later. Scripts have not been tested thoroughly on earlier versions of Mac OS, although some will work on Mac OS 9 and even earlier.

Most of the examples either operate independently of applications or use applications that come pre-installed with Mac OS X 10.3 or later. The exception is an early example that relies on the text editor application called BBEdit. If you don't have BBEdit installed on your computer, follow this link to download a fully-functional, trial version: <http://www.barebones.com/products/bbedit/index.shtml>

Software Installation

Once you have downloaded and decompressed *HandbookScripts.sit* (using Stuffit Expander), you will see a directory called *Handbook Scripts*. Nested inside that directory are further subdirectories labeled for the chapter to which the example files apply. Not all chapters have example files in the *Handbook Scripts* collection. Move the *Handbook Scripts* directory to a convenient location on your hard disk from which you can open the files with Script Editor and other applications.

You are now ready to begin your AppleScript journey.



Chapter 1 The Tools You Need Are Free

As long as you have Mac OS X installed on your Mac, you already have the basic tools you will use to learn and become productive with AppleScript. This chapter describes the various files you'll be using and tells you where to find them. I'll also introduce you to the Script menu, which gives you ready access to scripts that you add to your system.

Where Stuff Is

Various AppleScript resources are a bit scattered around your hard disk. That's not uncommon in Mac OS X, where the shared system and personal user directories have lots of similarly-named and similarly-organized files. Shared directories and their contents are available to all users who have login privileges for a particular Macintosh computer, whereas the contents of directories within a user's home directory are private to that user.

If, like most Mac users, you are a single-user Macintosh owner, then the division between system and user resources seems unnecessary. Your Mac automatically logs you into your user area when you start up, and you immediately have access to both areas. Your programs and documents end up seeing a composite of system and user resources. On the other hand, if you share your Mac with other users, your composite world consists of the contents of your user directory and only those system-wide resources made sharable by the system's administrator.



Chapter 1: The Tools You Need Are Free

The bottom line, then, is to be aware that you may have to look in either the system or user set of directories to find a particular file and set up your working environment. When I talk about paths to shared resources, I'll refer to them in the following manner: *HD/Folder/Folder/...*, where *HD* would be your startup disk's name. For items in the private part of the hard drive, I'll refer to path names in the form *[user]/Folder/Folder/...* where *[user]* is the name of *your* user directory. With that in mind, let's look at the key items you should know about as you begin learning AppleScript.

Script Editor

The tool that you will use more than any other is an application called Script Editor, located in your Mac OS X *HD/Applications/AppleScript* folder (see Figure 1.1). You'll use this application to write and edit scripts as well as explore the scriptability of other applications on your hard drive. Each script you write is saved as a separate file on the hard drive, just as you're accustomed doing when editing text or graphic files with other programs. If you intend to spend a fair amount of time with Script Editor, I recommend dragging its icon to your Dock so that you don't have to dig around your Applications folder to find it each time you want to write or look at a script.



Figure 1.1 The Script Editor program icon

Script Menu Installer/Uninstaller

Also inside the *HD/Applications/AppleScript* folder (see Figure 1.2) are two applications that install and uninstall a Script menu on the right end of the Macintosh menubar. Having this menu available is a real time-saver for scripters and anyone who uses scripts frequently. Double-click the Install Script Menu application to display the menu, which appears as a little scroll icon in the menubar. I'll have more to say about this menu and how to take advantage of it later in this chapter.

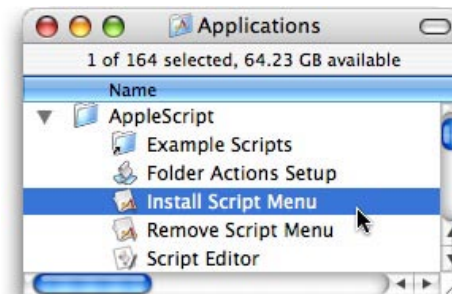


Figure 1.2 Inside the *HD/Applications/AppleScript* folder



Chapter 1:
**The Tools You
Need Are Free**

Folder Actions Setup

One other default application delivered inside the *HD/Applications/AppleScript* folder is called Folder Actions Setup. Double-clicking this application icon presents the same Folder Actions Setup window you see if you choose Configure Folder Actions from the Finder's contextual menu (the menu viewed when you ctrl-click on the Desktop). Folder actions are scripts that you can associate with any folder on your hard drive. One of the example Folder Action scripts presents an alert dialog box that notifies you when a file has been dropped into a folder (e.g., a networked user drops a file into your shared drop folder). The dialog contains a button that lets you command the folder to open its window and highlight the item that had been dropped onto the folder. The Folder Actions Setup window lets you view and manage the settings of all Folder Action scripts that you have turned on (and may be quite scattered) throughout your hard drive. You'll learn more about Folder Action scripts in Chapters 7 and 15.

Scripts Folders

The Script menu gets its list of scripts from two primary sources. Both the root hard drive and user directories have *Library/Scripts* folders, whose contents are joined together to produce the list of scripts directly accessible through the Script menu. The *HD/Library/Scripts* folder contains scripts that all users of the Macintosh machine can share, while the *[user]/Library/Scripts* folder contain scripts that

are available only to the currently logged-in user. The alias named Example Scripts in the AppleScript folder (Figure 1.2) links to the shared *HD/Library/Scripts* folder, where a number of pre-written scripts are ready for immediate use or can be customized once you learn more about AppleScript. Folder Action scripts are grouped together in specially-named folders within the Scripts folder. There is more to managing the contents of these Scripts folders, which I'll get to shortly.

Scripting Additions Folder

A handful of files are delivered in the *HD/System/Library/Scripting Additions* folder. These files are extensions to the basic AppleScript functionality built into Mac OS X. These extensions are written not in AppleScript but other programming languages, and are designed to provide extra native powers to your scripts. One of these scripting additions files, called Standard Additions, will become a vital part of your AppleScript experience. You will learn how to explore these extensions in Chapter 7.



Chapter 1:
**The Tools You
Need Are Free**

AppleScript Studio

As you will come to appreciate beginning in Chapter 2, one of the conceptual difficulties in learning AppleScript is that when you run a typical script, there is no obvious user interface or anything like an application window visible while the script runs. If the script needs to communicate with the script's user, you'll include an occasional dialog box or file chooser, but that's about it. If you want to create a visible application window, you can migrate your development to another Apple product, called AppleScript Studio. This tool is part of a much larger set of developer tools—called Xcode—that you can download for free after registering with the Apple Developer Connection program (<http://developer.apple.com>).

The Xcode tool is one that many professional Apple developers use to create applications. Known as an integrated development environment (IDE), Xcode has its own learning curve, separate from whatever programming language you use to build applications. But it does provide a simple interface builder that greatly simplifies the process of building your applications interface using various buttons, fields, lists, and so on. If you are new to programming, I strongly recommend first getting to know the basics of AppleScript using the simpler Script Editor tool, as this book does. Once you have a good foundation of the language, you can then more easily bring those skills to AppleScript Studio.

The Script Menu and Scripts Folders

Mastering the Script menu and *Scripts* folders is an important step before you begin working with AppleScript. With hard disks these days holding tens, if not hundreds, of gigabytes of data, remembering where you leave stuff is not always easy. Fortunately, the Script menu and its supporting *Scripts* folders can help you organize and locate scripts that are still under development or scripts that you intend to use frequently.

Figure 1-3 shows the Script menu listing all of the default sample scripts that come in Mac OS X (version 10.3, Panther) plus two scripts I've added to the list. The way script items appear in the Script menu depends on how you divide scripts between the shared and user instances of the *Scripts* folders described earlier in this chapter. If you design scripts that are useful only inside one application, you can further organize your *Scripts* folders to display such scripts only when that application is the frontmost application.



Chapter 1:
**The Tools You
Need Are Free**

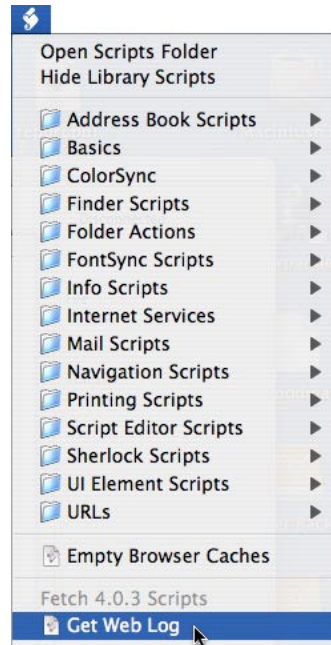


Figure 1.3 *The Script menu*

Listings of scripts and script folders (starting below the topmost divider line) derive their names directly from the script file and folder names inside the *Library/Scripts* folders. Changes you make to those names in the Finder are immediately reflected in the Script menu listings. Any folder you create in the *Scripts* folder becomes a hierarchical menu item within the Script menu. You can nest additional folders, with each new level creating one deeper level of hierarchical menu.

The Script menu, like the one shown in Figure 1.3, is divided into as many as four sections. In the topmost

section are two menu commands that operate on different *Scripts* folders. **Open Scripts Folder** causes the Finder to display the contents of the *[user]/Library/Scripts* folder. Despite the similar naming between the system and user folder structures, the **Hide Library Scripts** command removes from the Script menu all items only in the *HD/Library/Scripts* folder, not the user equivalent. When shared scripts are hidden from view, the menu item changes to **Show Library Scripts** to allow you to put the shared scripts back into the menu (see Figure 1.4).

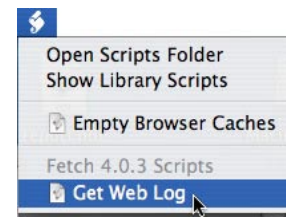


Figure 1.4 *The Script menu with library scripts hidden*

With shared library scripts displayed in the second section of the Script menu (Figure 1.3), the third section displays scripts and script folders placed into the *[user]/Library/Scripts* folder. As with the library scripts, folders become hierarchical menu items within the user library area.

One surprise about the user *Scripts* folder is that you can add scripts here that appear in the menu only while a particular application is the frontmost application on the Desktop. This feature can help



Chapter 1:

The Tools You Need Are Free

reduce clutter in your Script menu if you design many scripts to be used in concert with specific applications. Figures 1.3 and 1.4 were captured while Fetch 4.0.3 was the frontmost application. I had set up the user *Scripts* directory structure so that the Get Web Log script appears in the menu only while I'm working in Fetch.

To generate these application-specific Script menu listings, begin by creating a new empty folder called *Applications* inside the *Scripts* folder. Then, inside the *Applications* folder, create one more folder for each application for which you have written scripts. Name each application's folder identically to the name of the application, including version number or any other symbols that may be in the file name. To guarantee a perfect match for complicated application names, simply select the application icon and **Edit>Copy**. Then **Edit>Paste** the name into the newly created folder's name. Figure 1.5 shows the structure of the user *Scripts* folder responsible for the menu listings in Figures 1.3 and 1.4.

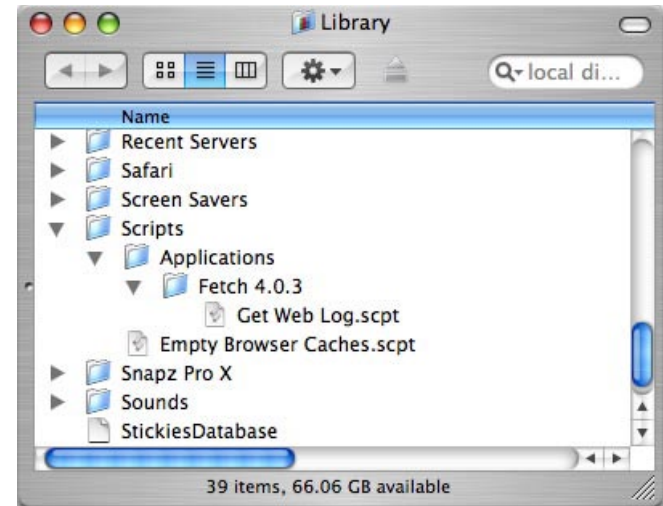


Figure 1.5 The *[user]/Library/Scripts* folder with application-specific scripts

Note that inside the *[user]/Library/Scripts* folder, the folder names *Applications* and *Folder Action Scripts* do not appear in the Script menu. These names are reserved for their special purposes.

Although you will develop your own working directory strategies as your AppleScript skills evolve, I recommend that you begin placing your scripting works-in-progress inside your *[user]/Library/Scripts* folder. I like this location because you can immediately reach the folder via the **Open Scripts Folder** item in the Script menu, no matter what you're doing on your Mac. Alternatively, you can store your scripts where you like and place Finder aliases to those scripts in your Scripts folder.

Let me also alert you to a possible point of Script



Chapter 1:

**The Tools You
Need Are Free**

menu confusion with some of Apple's own applications. The Address Book and Mail programs, for example, display their own iconic Script menus among the left-hand group of the application's menus. These Script menus simply duplicate the application-specific features of the main Script menu. So, yes, you are seeing double, but it's perfectly normal.

Now that you have an overview of what you'll be using and where it is, the next chapter offers some guidance about the best way to learn AppleScript.



Chapter 2 How to Learn AppleScript

To use AppleScript effectively, there is a fair amount you have to know. Fortunately, that knowledge can be accumulated gradually while putting AppleScript to work.

The reason there is so much to know is that the programming vocabulary and other concepts you will be using with AppleScript do not exist in a single, self-contained environment. As you'll learn in subsequent chapters, the core AppleScript language has only a handful of words in its vocabulary. The powers that let you control applications are actually contained *in the applications*, not in AppleScript system components. For example, AppleScript doesn't know anything about spreadsheet cells or page layouts. It's only when you use AppleScript to, say, automate the process of copying a cell range from a weekly budget Excel spreadsheet to a BBEdit activity report HTML page that you can access the vocabulary built into those programs.

Moreover, the AppleScript vocabulary of one program is not necessarily applicable in another program (although there may be similarities).

Spreadsheets have vocabulary entries for cells and ranges; word processors let us talk about words and paragraphs; iTunes uses the terminology of the track and play list.

What this all means is that there are several things to know—even if not in full depth—before it is possible to start making AppleScript work with applications:

- ▶ **AppleScript core language syntax.** The basic construction of statements that AppleScript can execute.
- ▶ **The applications.** Irrespective of AppleScript, it's important to know how to use the application(s) you wish to script. Just as a playwright must know the moral, psychological, and physical make up of a character before writing lines for him or her, so, too, must a scripter know a program's capabilities and user terminology before writing scripts that automate its processes.
- ▶ **The application's AppleScript syntax.** While the script editor provided with AppleScript (and any third-party editor) lets you display and print each scriptable application's AppleScript dictionary, the



Chapter 2:
**How to Learn
AppleScript**

dictionary goes only so far to explain intricacies of scripting the application. An important goal of this book is to help you interpret the dictionaries that come built into scriptable applications.

The ideal way for beginners to tackle this mountain of information is to do it in horizontal slices of difficulty. Rather than worrying about mastering every tiny bit of AppleScript syntax or an application's dictionary before writing your first line of AppleScript, you can work with the most common items first. Once you have a feel for the pieces you're working with, you can then begin poking deeper as needed.

Gradus ad Scriptum

Whatever your programming experience—even if it is zero—I recommend that you follow a sequence that I've found to be most helpful in getting a handle on AppleScript quickly. The rest of this book is organized according to this sequence.

Topics in most chapters beginning with Chapter 6 are rated by difficulty. The symbol I chose to represent each level is the pocket protector (hey, if we're scripting, we're programming!). A topic's difficulty will range from one to three according to the following scale:

beginner



intermediate



advanced



In your first time through a chapter, you can spend the most time on the beginner-rated items, and just glance through the others to gain an appreciation for what else is in the vicinity. Come back later to fill in your knowledge in more advanced subjects as you feel ready.

Once you are familiar with the basic vocabulary, you will be using this book more as a reference than as a tutorial. Therefore, I feel it is important to organize the content as you'll want it later. The pocket protector codes, however, will speed your learning because you won't be bogged down with mind-numbing details at first.



Chapter 2:
**How to Learn
AppleScript**

Here, then, is a proven sequence for learning AppleScript.

- 1) **Record scripts.** Some AppleScript-ready programs and the Finder (in Mac OS X version 10.3 and later) let users walk through operations manually while the programs send a journal of actions to the AppleScript script editor. In other words, you do what you normally do, while a perfect working script is assembled for you in the AppleScript editor.
- 2) **Master AppleScript's core language syntax.** It's vital to know how to construct statements that AppleScript can follow.
- 3) **Learn AppleScript's built-in commands.** This is a short list, which you'll see early during the mastery of the language syntax (after all, you need some words to work with while learning the syntax).
- 4) **Understand how scripting additions fill out AppleScript's core language.** You'll see how to use AppleScript and the standard scripting additions that come with it to perform key Finder- and system-level actions.
- 5) **Write self-contained scripts.** These are scripts that perform meaningful actions on their own, without calling up any other applications.
- 6) **Start writing scripts for one application at a time.** You'll start to get a feel for accessing a program's scripting dictionary and manipulating information of the same kind.
- 7) **Write scripts that integrate multiple applications.** For many scripters, this is the big payoff, since it allows you to essentially create

customized solutions from existing programs.

- 8) **Graduate to AppleScript Studio to create user-friendly interfaces for your scripts.** Although AppleScript Studio is beyond the scope of this book, you'll be eager to start building standalone applications that use AppleScript to integrate the powers of multiple applications.

This sequence looks long and arduous, but it can go relatively quickly, because you'll be trying out things along the way. I'll also try to anticipate head-scratching problems you may encounter, and cover them in relevant locations.



Chapter 2:
**How to Learn
AppleScript**

If You've Never Programmed Before

To a programming newcomer, the learning sequence above may appear daunting. AppleScript may not be the easiest language in the world to program, but it's pretty close. Unlike getting down and dirty into developing full fledged monolithic applications (like the productivity programs you buy in the stores), AppleScript lets you experiment by writing small snippets of programming code to accomplish big things. The scripting environment built into Mac OS X does a lot of technical work for you.

Programming at its most basic level is nothing more than writing a series of instructions for the computer to follow. We humans follow instructions all the time, even if we don't realize it. Traveling to a friend's house is a sequence of small instructions: go three blocks that way; turn left here; turn right there. Among the instructions we follow are some decisions: if the stoplight is red, then stop; if the light is green then go; if the light is yellow, then gun it. Occasionally, we must even repeat some operations multiple times: keep going around the block until a parking space opens up. Computer programs not only contain the main sequence of steps, but they *anticipate* what decisions or repetitions may be necessary to accomplish that goal (e.g., the stoplight, the parking shortage).

The initial hurdle of learning to program is becoming comfortable with the way a programming environment wants its words and numbers organized

inside these instructions. Such rules, just as in a living language such as English, are called *syntax*.

Computers, being the generally dumb electronic hulks that they are, aren't very forgiving if we, humans, don't communicate with them in exactly the language they understand. When speaking to another human, we can flub a sentence's syntax, and there's a good chance the other person will understand fully. Not so with computer programming languages. If the syntax isn't perfect (or at least something within the language's range of knowledge that it can correct), the computer has the brazenness to tell us that *we* have made a syntax error.

It's best to just chalk up the syntax errors you receive as learning experiences. Even experienced programmers get them. Each syntax error you get—and the resolution of that error by rewriting the statement—adds to your knowledge of the language.



Chapter 2:
**How to Learn
AppleScript**

If You've Done a Little Programming Before

Programming experience in a procedural language, such as BASIC or plain C—especially if it was for computers other than the Macintosh—may almost be a hindrance rather than a help to learning AppleScript. While you may have an appreciation for precision in syntax, the overall concept of how a program fits into the world is probably radically different from AppleScript. Part of this has to do with the typical tasks a script performs (automating processes or exchanging information between programs), but a large part also has to do with the nature of AppleScript's object-oriented programming.

In a typical procedural program, the programmer is responsible for everything that appears on the screen and everything that happens under the hood. When the program first runs, a lot of code is dedicated to setting up the visual environment. Perhaps there are text entry fields or clickable buttons on the screen. To know if a user clicks on a particular button, the program examines the coordinates of the click, and compares those coordinates against a list of all button coordinates on the screen. Program execution then branches to carry out instructions (probably in a subroutine) that are reserved for clicking in that space.

Object-oriented programming is almost the inverse. A button is considered an object—something

tangible. An object has properties, such as its location, color, size, label, border style, and so on. An object may also contain a script. At the same time, the system software can send a message to the object—depending on what the user does—that triggers the script. For example, if the user clicks on a button, the system software tells the button that somebody has clicked there, leaving it up to the button to decide what to do about it. That's where the script comes in. The script is part of the button, and it contains the instructions the button carries out when the user clicks on it. A separate set of instructions may be listed if the user double-clicks on the button, or holds down the Option key while clicking.

Some of the scripts you'll write may seem to be procedural in construction: they contain a simple list of instructions that are carried out in order. But when dealing with information from programs or the Finder, these instructions will be working with the object-oriented nature of programs. A range of spreadsheet cells becomes an object, as does a word processing paragraph.

In advanced uses of AppleScript, you can even create your own objects—scripts that create entities consisting of specific properties and behaviors. We'll build one in Chapter 15. The script creates a daily alarm clock object. This object continually operates in the background, comparing the Macintosh's internal system clock against the top item in a list



Chapter 2:
**How to Learn
AppleScript**

of alarms set to go off. When the times match, the object comes to life, displaying a dialog with the alarm details.

Making the transition from procedural to object-oriented concepts may be the most difficult challenge for you. When I was first introduced to object-oriented programming a number of years ago, I didn't "get it" at first. But when the concept clicked—a long, pensive walk helped—so many lightbulbs went on in my head, I thought I might be glowing in the dark. From then on, object orientation seemed to be the only sensible way to program.

If You've Programmed a Lot Before

Scripting languages can be frustrating for programmers with long experience in modern languages. While programming newcomers appreciate loose data typing (or, rather, they don't know any other way), programmers from the worlds of C/C++ and Java find the lack of strict data typing to be almost frightening. Similarly, although AppleScript exhibits numerous object-oriented tendencies, you will not see all of the concepts you've used in other languages. Here are the major differences between AppleScript and full-fledged object-oriented programming languages:

AppleScript is a scripting language. A scripting language is intended to be used to "glue together" existing objects and processes. In other words, the power lies in the object models of applications and other compiled code (such as scripting additions), while your ingenuity comes in shuffling data among the objects in productive ways. Although you can construct your own abstract objects, that capability is not the focus of the language.

AppleScript is partially object-oriented. The language, indeed, offers several object-oriented features on various levels, but you may find some advanced OO concepts missing or their implementations in AppleScript a bit clumsy. The core language and application dictionaries use the term "class" liberally, referring both to types of objects (e.g., a window class) and types of values



Chapter 2:
**How to Learn
AppleScript**

(e.g., a Boolean value class). You may also construct your own, custom objects, which readily permit the equivalent of subclassing the object (with parent-child relationships). While class objects in AppleScript commonly have properties associated with them, they typically do not have methods, as is common in other languages. Action words are called **commands**, and they are either directed implicitly to the current context (e.g., commanding BBEdit to quit) or explicitly to a previously instantiated object (e.g., commanding BBEdit to print a particular document among the several open at the time). An explicit target is signified syntactically as a parameter to the command (e.g., **print first window**).

Variable assignments are verbose. Unlike many other languages, where a simple assignment operator associates a value with a variable token, AppleScript requires one of two assignment commands (**set** or **copy**) to assign a value. The commands are not necessarily interchangeable in all situations (**copy** can create a new instance of an object, for example). Scope of a global variable is limited to the current script execution context, while, as expected, a local variable limits its scope to the current execution block. Scoping rules, however, can be complex in a variety of nesting situations.

AppleScript is loosely typed. Variables are not defined by their data types, but rather by the values they currently hold. A variable's value can change over time, if needed, to the extent that values of

different data types may occupy the same variable at different times. That's not to say, however, that data types (value classes in AppleScript-speak) aren't important in AppleScript. Quite the opposite. Many commands require that their parameter values be of a particular value class, and you will frequently change (**coerce**) value classes with the **as** operator. The range of available value classes follows:

- ▶ Boolean
- ▶ Class
- ▶ Constant
- ▶ Data
- ▶ Date
- ▶ File Specification
- ▶ Integer
- ▶ International Text
- ▶ List
- ▶ Number
- ▶ Real
- ▶ Record
- ▶ Reference
- ▶ RGB Color
- ▶ String
- ▶ Styled Clipboard Text
- ▶ Styled Text
- ▶ Text
- ▶ Unicode Text



Chapter 2:
**How to Learn
AppleScript**

► Unit Classes (for measurements)

AppleScript provides two types of value collections. An ordered collection (similar to a numerically-indexed array) is known as a **list**, while an unordered collection comprising of name-value pairs is called a **record**. In both cases, a single list or record may contain values of any class, including mixed and matched value classes. You access a list item value by its position (e.g., **item 4 of myList**). Reference a record's value by way of property syntax (e.g., **name of myRecord**).

AppleScript operators operate at a high level. You won't find any bitwise or a few other operators you're accustomed to in other languages. But in addition to the typical arithmetic, Boolean, and comparison operators, you have a range of containment operators, such as determining whether a list (array) of items contains a particular item. Most comparison and containment operators are available in more than one syntactical construction as a way to help you build more English-language statements. For example, the equality comparison operator can be referenced with the usual symbol (=) or any of the following equivalents: **equal**, **equals**, **equal to**, **is**, **is equal to**.

AppleScript provides most of the typical control structures. You'll find the usual conditional constructions (**if**, **if-else if**, etc.), as well as repeat loops of several types, including a shortcut one for iterating through lists of values. There is not,

however, a **switch-case** capability, so you'll have to use a sequence of **if-else if** constructions in its place.

AppleScript subroutines may or may not return a value. What other languages call methods, functions, or procedures are lumped together in AppleScript as user-defined **subroutines**. A subroutine returns a value only when it includes a **return** statement followed by some value. Otherwise, at the end of the subroutine's execution, control simply returns to the calling script statement. Any segregated group of statements in AppleScript is called a **handler**, and a subroutine is a handler that has been defined by the programmer as an action to be invoked by some other statement.

AppleScript subroutines cannot be overloaded. There are two types of subroutines depending on the way you wish to define parameters. One type accommodates positioned parameters, that is, a sequence of values that get assigned to parameter variables in the same order in which they are passed. The second type accommodates labeled parameters, which allows you to pass values in any order you wish. But in both cases, you must pass values at least for all parameters defined in the subroutine specification. Excess parameters are ignored, and cannot be read by statements within the subroutines. You are not permitted to create more than one subroutine with the same name within any script (the Script Editor refuses to compile the script).



Chapter 2:
**How to Learn
AppleScript**

Values are passed “by reference” and “by value.”

Any object (instance of a class) passed as a parameter to a subroutine is a reference to that object, thus allowing statements in the subroutine to read and write property values of the passed object directly. But other types of values, such as values of object properties, are passed by value, meaning that subroutines working with such values cannot derive a reference to the object from which the value comes. You’ll learn later in this book that an application’s dictionary dictates what class properties are objects in themselves or simply values.

Error trapping is by `try-catch`. AppleScript employs a fairly conventional **`try-catch`** mechanism for handling exceptions (called **errors** in AppleScript). Scripts can also explicitly throw exceptions by invoking the **`error`** command.

Memory management is not under script control.

Although you can control memory usage to some degree (e.g., emptying a variable that holds a lot of data that is no longer needed), AppleScript provides no direct access to system memory usage.

Carriage returns serve as simple statement delimiters. An AppleScript **simple statement** is one that consists of a single line. There are no semicolon or other punctuation symbols at the end of a statement line. An AppleScript **compound statement** is a multi-line statement, usually comprising of a **`tell block`** and one or more statements nested within.

AppleScript is not case-sensitive. If you’re accustomed to naming distinct variables or subroutines with different capitalization (e.g., different variables named **`foo`** and **`FOO`**), that won’t work in AppleScript, where token names are case insensitive.



Chapter 2:
**How to Learn
AppleScript**

If You've Programmed in HyperTalk Before

Although HyperCard is well past its prime (and has been officially dropped as a product by Apple), I suspect that there may be a significant number of past or present HyperCard developers who want to try their hands at AppleScript. After all, the concept of scripting little bits of code was popularized through HyperCard—a self-contained programming environment for non-propellerheads. Although it avoids the terminology of object orientation (classes, methods, etc.), HyperCard is very object-oriented. You won't have any trouble with AppleScript's object orientation.

What may drive you crazy, however, is that HyperTalk and AppleScript have enough similarities and differences to make the learning process confusing at first. For example, you can get and set properties of objects in both environments. But in HyperCard when you get a property, the information goes into a special variable called **it**; in AppleScript the information goes into a special variable called **result**. AppleScript also has an **it** word in its vocabulary, but the meaning is different from HyperTalk's **it**.

Probably the biggest difference you'll notice right off the bat is that until you start writing scripts inside user interfaces (e.g., with AppleScript Studio), AppleScript scripts are not dependent on system messages, as HyperTalk scripts are. While

a HyperTalk script (say, a stack script) may contain handlers for any of the possible system messages that reach it through the message hierarchy (e.g., **openStack**, **mouseUp**), most AppleScript scripts you write have a main sequence of statements that simply run from top to bottom when they're told to run. The initiating force is a click of a Run button in the script editor, or, if the script has been saved as a standalone application, opening it from the Finder (plus other ways we'll see later).

You'll also welcome AppleScript's syntax forgiveness—sometimes doing even better than HyperCard. For example, the greater than (>) operator has five synonyms:

comes after

greater than

is greater than

is not less than or equal [to]

isn't less than or equal [to]

Some other constructions, such as **if-then** and **repeats**, however, are slightly more restrictive in AppleScript. Still, the differences are easily surmountable.



Chapter 2:
**How to Learn
AppleScript**

Enough Talk: Let's Script!

If I haven't frightened you away by now, we're ready to get underway. Since you're running Mac OS X, you already have the minimum necessary tools to get started. In the next chapter we'll be using Script Editor and the script recording features of a popular Mac text editor program, BBEdit, to record some scripts.



Chapter 3 Your First AppleScript Script

In this chapter we'll start using the Script Editor, the primary user gateway to AppleScript. The Script Editor is where you will write scripts, even if your scripting travels take you to other applications. We'll make one of those journeys—to a modern, scriptable and recordable application, BBEdit—in this chapter. In the end, you'll see how to let AppleScript record your actions in a program, and convert them to AppleScript in the Script Editor.

Scriptable Applications

By far the most common method of AppleScript support in applications is *scriptability*. This means that by way of an external editor, such as Apple's Script Editor, you can write scripts that automate processes or capture data from documents.

Each application's level of scriptability is entirely up to the program's designers. For some programs, the designers give scripters access to almost every conceivable element and characteristic of a document. Others may have their own scripting language built in (which they likely share with Windows versions of the products). Rather than reinventing the scripting wheel for the program, the designers let scripters run the program's internal scripts. That means, of course, that a scripter must also be sufficiently knowledgeable of the program's own scripting language to do those scripts first—and then trigger them from an AppleScript script. The majority of scriptable applications are in the middle, offering some level of scriptability support for elements of a document.



Chapter 3:
**Your First
AppleScript
Script**

Recordable Applications

While the BBEdit application lets you record a script of your actions, not all AppleScript-capable programs support this feature. A program that supports script recording is said to be *recordable*.

The AppleScript recording mechanism is controlled from the Script Editor. When you switch on script recording, actions you take inside recordable programs are translated into AppleScript statements in real time.

Just because an application is scriptable doesn't mean that it is recordable. Moreover, applications generally provide no readily external clue indicating whether it is recordable. Only when you try to record a script from the Script Editor—and nothing appears in the Script Editor window as a result of actions in a program—will you know that the program is not recordable.

Recording's Learning Value

Experimenting in a recordable application is a valuable learning experience. For one thing, you see immediately how the program “thinks” in AppleScript terms as you make menu selections, move data around, and so on. Even more important is that recordable applications record in the precise syntax they prefer from scripts. You'll see syntax details in recorded scripts that don't necessarily show up while viewing a program's AppleScript dictionary (Chapter 6). It's these pesky details—which can vary

from program to program, even when they appear to do the same thing—that can make learning an application's scriptability time-consuming.

Recording's Limitations

Before you get all excited and think that recordability will write all your scripts for you, I have some sobering news. Aside from the fact that precious few programs are recordable, it's not possible to record powerful AppleScript operations, such as if-then decision trees or repeat loops.

Recording goes in a kind of brute force straight line. For example, you may record a sequence of steps that involves opening a document file. But what if that document file isn't available when the script runs some time in the future? Even if you walk a recording through the same series of steps 20 times, the recorder duly lists those steps 20 times, rather than put them efficiently inside a repeat loop—as you would do if writing the script from scratch.

Still, you can combine a program's recordability with manual scripting to help out. In the missing file situation, above, let the recording track the steps when the file is there. Later, manually add error checking to handle the case when the file isn't available to script. For the repetitive action scenario, use the recorder to snag one instance of the sequence, and then manually build a repeat loop around that sequence.



Chapter 3:
**Your First
AppleScript
Script**

Attachable Applications

While on the subject of AppleScript support, I'll mention here the third type: *attachability*. Attachability means that an object—a spreadsheet cell, a button on a database form, a folder—can contain a script. Such a script might send a query to a remote computer based on information in a field or cell, copy the query results, and paste them someplace else in the document.

The Mac OS X Finder application (yes, it is treated as a separate scriptable application) is attachable. For example, you can attach a script to a folder so that an action, such as adding a file to the folder, triggers some scripted action, thus obviating the need to stop and explicitly run a script each time you drag a file to that folder.

Script Editor

It's time to get acquainted with the Script Editor, where you'll spend a lot of your AppleScripting life. The Script Editor is an application that works very closely with the AppleScript system software.

Opening the editor, you see its primary editing window (Figure 3.1). It contains two fields. One lets you enter a description of what the script does. If you ultimately save the script as a double-clickable application, then the contents of this description field appear in the splash screen when the script opens (more about this later in the chapter). The upper field is where you enter the script.

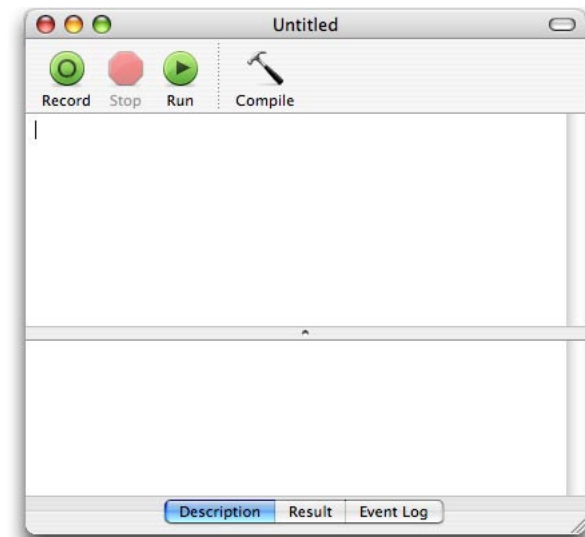


Figure 3.1. AppleScript Script Editor window.



Chapter 3:
**Your First
AppleScript
Script**

One of the first tasks you want to perform is establishing the size of the window each time a new script window appears. Drag the lower right corner of the window so that the window resizes to proportions that suit you. Then choose **Save As Default** from the **Window** menu. You can adjust this default setting anytime.

The Record Button

Two of the editor's four buttons are dedicated to script recording tasks: Record and Stop. When you click the Record button, Script Editor tells the system that recording is on. In turn, the system alerts all running applications that support script recording to send copies of its actions to the system, which then passes them on to Script Editor. Note that other than the altered state of the Record and Stop buttons in Script Editor, there is no system-wide visual clue that recording is switched on.



Figure 3.2. Record and Stop buttons.

With recording turned on, you can watch the script filling in Script Editor while you perform the real actions in the programs (provided you have enough screen real estate to view it all). Don't be shocked, however, if absolutely every action isn't immediately set to script. Some applications queue up recorded events until they sense by another action that what you have just performed is finished, and ready for recording. One or more lines of script may then flow to Script Editor.

Only applications designed to be recordable will know how to send copies of their events to the system and Script Editor. If, while recording a script, you switch to a non-recordable application, not one line of new script appears in Script Editor.

What To Record

Just like recording live action with a camcorder, AppleScript recording captures virtually everything you do in a recordable application—including mistakes. Even if an Undo were recorded, it would mean that the script still contained the error. The good news is that because the recording is translated into human readable form, you can edit out the mistakes.

More important, however, is figuring out exactly the state of your applications and documents at the beginning and end of a script. These states may be different when you're recording a script than when you want to run it later.



Chapter 3: **Your First AppleScript Script**

The script you'll be recording later in this chapter is a good example. The purpose of the script will be to assemble portions of a document into one master copy (each portion having been prepared and revised by different people). Ultimately, we just want to run the script on each day's submissions to assemble the latest version of the whole document.

In the course of recording the script, BBEdit is already open, and a blank document window greets us, ready to accept the pieces. But that may not be the state of the text editor program when we run the script next week. There may already be one or more other open windows, which we don't want to disturb. The proper thing to do would be to create a new window before doing anything else. In the recording process, this seems redundant, since a blank window is already waiting for us. Still, it's a case of anticipating the environment in which the script may be used in the future.

Stopping Recording

When you're finished with the process you want to record, switch back to Script Editor, and click the Stop button. One or more lines may complete the script, and it's ready to run.

You'll see that the script begins with a **tell** statement, which indicates where the rest of the commands in the recorded script are sent. An **end** statement finishes the script. If you then turn on recording a second time, a second **tell** statement is

added to the bottom of the script—there is no magic that knows to blend the commands into the previous script.

Running the Script

Clicking the Run button (or typing ⌘-R) in Script Editor plays back what you've recorded. While the script runs, you see the program's windows open, and other actions you had done manually. There is rarely a problem running unmodified recorded scripts, but if something doesn't work correctly, script execution stops, the Script Editor comes to the foreground, and an error message appears. Unless you've had some experience debugging scripts, the error message probably won't be very helpful in telling you what's wrong.

Saving the Script

Since the idea of a script is to automate processes, you will want to save the script so you can use it later. There are a number of save options available as indicated by the File menu and Save dialog box choices illustrated in Figure 3.3. Table 3.1 shows the range of options available for each script file format choice, as well as the Finder icon associated with each type.



Chapter 3: Your First AppleScript Script

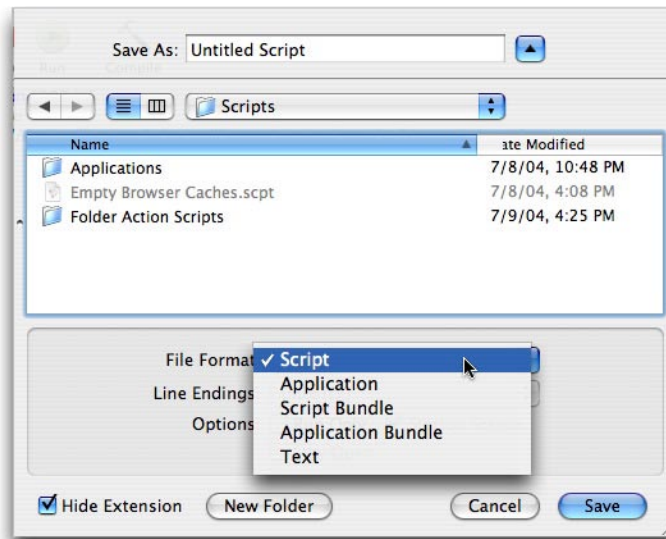
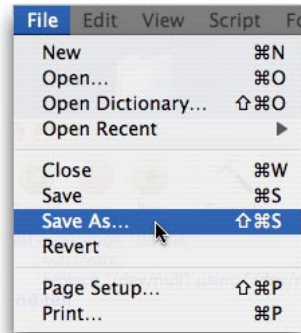


Figure 3.3. Script Editor File menu (top) and dialog format options (bottom).

Save Format	Icon	Options			
		Run Only	Stay Open	Startup Screen	Line Endings
Script		×			
Application		×	×	×	
Text					×

Table 3.1. Save script choices.

Each format offers one or more options in the Save dialog. Here's what the choices mean:

- **Run Only.** When checked, the script cannot be re-opened in Script Editor to inspect or modify the script source code.
- **Stay Open.** When checked, after the script application runs, it stays open as a process, continuing to appear as an icon in the Dock and in the $\mathbb{A}$ -Tab application rotation.
- **Startup Screen.** When checked, displays an opening splash window (see Figure 3.4). The window displays the contents you entered into the Description pane of Script Editor for that particular script (or a default advisory if you enter no description). The user sees a Quit button and a Run button. Clicking this Run button is the same as clicking the Run button in Script Editor. After the script completes executing its statements, the AppleScript application automatically quits (unless it was saved as a Stay Open application).



Chapter 3: **Your First AppleScript Script**



Figure 3.4. A typical script application splash screen.

- **Line Endings.** For the text only format, you can choose from among several platform-specific ways of creating line breaks in the source code view (various combinations of the carriage return (CR) and linefeed (LF) control codes).

Both the Script and Application types also have variations named Script Bundle and Application Bundle. The bundle varieties save the file in a format that works only on Mac OS X version 10.3 (Panther) or later. Use the non-bundle varieties if your scripts also need to be able to run on earlier versions of Mac OS X or Mac OS 9.

We'll have more to say about saving scripts as applications, and what a compiled script really is, but it's valuable to understand the main differences between the file formats at this juncture.

Saving a script as Text (with the .applescript file name extension) means that you can run the script only by loading it into Script Editor and clicking the Run button. Use the text-only version only when you're

having difficulty with a script of your own design (i.e., it won't compile—explained later). The Script version (with the .scpt or .scptd extension) is the format that you can run either from Script Editor or from the Script menu (if engaged on your Mac). Files in the Script format are the ones that go into the various *Library/Scripts* folders described in Chapter 1.

In the Application format (whose .app file name extension is hidden in the Finder), the script becomes a double-clickable application. AppleScript applications occupy very little disk space, since they depend on the system software to do most of the work when they run. To run a script saved in this form, double click it in the Finder—just as you would any application. Script applications may also be placed in the *Library/Scripts* folders.



Chapter 3:
**Your First
AppleScript
Script**

Deciding What To Script

The first class ticket to an unenjoyable scripting experience is sitting down in front of Script Editor and trying to dream up something to script. Scripting is no meandering Sunday drive.

Instead, any attempt to script should be driven by a desire to accomplish a task. The better you can visualize the goal of the script, the easier it will be to record or write a script. Even if the goal changes during the scripting process, you need to know where you're going before you begin.

The application used in the following section is called BBEdit by Bare Bones Software, Inc. This commercial text editing program is very popular among programmers as well as Web content authors who like to write their own HTML source code. Unlike a word processing program, BBEdit doesn't offer text styling (bold or italic, for instance), but does some helpful color coding and other formatting assistance to speed the work of programmers. We use this program for the recording part of this chapter because the TextEdit program that comes with Mac OS X is only scriptable, and not recordable.

If you do not already own a recent version of BBEdit (version 7 or later), you can download a free demonstrator version to complete the following tasks. The demo version is a fully functional application, but with a 30-day time limit on it. To download the demo version, visit <http://www.barebones.com/products/bbedit/demo.shtml>.

Recording Your First Script

Since you may not know exactly what you'd like to script at this point, I'll make that part of the scripting process easy for you. In the folder *Handbook Scripts/Chapter 03* (from the companion files) is a set of BBEdit text files. These files represent portions of a legal document (in this case, a mythological software license). In this scenario from a big law office, each segment of the license is written by a different junior associate. Since the final license and its constituent parts will be going through many revisions, we need a script that assembles the current versions of parts into a complete license for review by the partners and client.

Script Overview

The manual process for this consolidation consists of the following steps:

- 1) Open BBEdit if it isn't open.
- 2) Create a new blank window for our consolidation, and turn on soft word wrapping.
- 3) Open the file containing the first portion of the license.
- 4) Select all of the text.
- 5) Choose **Copy** from the **Edit** menu.
- 6) Switch back to our consolidation document window.
- 7) Choose **Paste** from the **Edit** menu.
- 8) Repeat steps 3 through 7 for each of the remaining components.
- 9) Save the consolidation with the file name



Chapter 3:
**Your First
AppleScript
Script**

“Software License.”

When you multiply the number of steps for each component by the five components in this document, you can see that this is a good solution to script. By saving the script that does this as a script application, all you do is double-click the app each time you want to assemble the latest version.

Setting the Stage

Follow along as we go through the recording process that takes us up to the point of actually working with the components.

- 10) Open Script Editor if it isn't already open. If you have a large monitor, you may want to drag the editor window to the right or bottom of the screen to watch what happens during recording.
- 11) If the Description button isn't highlighted, click it to show the Description pane. Then enter a description of what this script does. Styles vary about the language of these descriptions, but I try to envision what a user would see in this script application's startup screen. For this script, you might use: “Assembles the latest version of the software license from component parts.” Don't be afraid to be verbose in these descriptions if they will help users fully understand what clicking the Run button accomplishes.
- 12) Click the Record button. After a few seconds, the Record button goes dim, and the Stop button activates.

- 13) Switch to the Finder and launch BBEdit. If BBEdit is already open, make it the active application.
- 14) Choose New from BBEdit's File menu. Even though you may be looking at a new, untitled window before doing this, you have to anticipate that the user of this script may already be using the application with another document window already open. There is no penalty for having two untitled windows open in an application. This new window is where we'll assemble the components into a fresh copy of the whole license.
- 15) In the tool bar at the top of the new document window is a row of menu buttons. Click the third one from the left (the first one, with the pencil, may be dimmed) to reveal the Text Options menu. If the first menu item, Soft Wrap Text, does not have a bullet to its left, then select the item to turn on this feature for the current document. Turning on this feature lets the incoming text word-wrap so you can easily see all of the text.

Bringing in the Components

In this segment, we'll go through the repetitive drudgery of copying and pasting each component in sequence.

- 16) Choose Open from the File menu, and open the component named *0. Introduction*.
- 17) Choose Select All from the Edit menu.
- 18) Choose Copy from the Edit menu.
- 19) Close the *0. Introduction* window. This is



Chapter 3: **Your First AppleScript Script**

important so we don't leave a trail of window clutter after the script executes.

- 20) Choose **Paste** from the **Edit** menu. Doing one of these manually isn't so bad. Avoid clicking in the document window, which may reposition the text insertion pointer someplace other than at the end of the text. We want the pointer to always be at the end of text so that succeeding components are pasted to the end of the document.
- 21) Repeat steps 16 through 20 for each of the remaining components: *1. License*; *2. Restrictions*; *3. Warranty*; and *4. General*. This is mind-numbing drudgery that you'll only have to perform once. Do these repetitions slowly, because it's easy for your mind to wander while doing repetitive stuff like this.

Saving the New Document

We're nearing the end. All that's left is to save the document and stop recording.

- 22) Choose **Save As** from BBEdit's **File** menu.
- 23) Enter the name *Software License*, and click the **Save** button (or press **Return**).
- 24) Switch back to **Script Editor**.
- 25) Click the **Stop** button.

At this point, your recorded script should look like the following listing:

```
tell application "BBEdit"
    activate
    make new text window
```

```
    set properties of text window 1 to {soft
wrap text:true}
    open {file "Macintosh HD:Chapter 03 Script
Recording:0. Introduction"} with LF
translation
    select text 1 of text window 1
    copy selection
    close text window 1
    paste
    open {file "Macintosh HD:Chapter 03-Script
Recording:1. License"} with LF translation
    select text 1 of text window 1
    copy selection
    close text window 1
    paste
    open {file "Macintosh HD:Chapter 03-Script
Recording:2. Restrictions"} with LF
translation
    select text 1 of text window 1
    copy selection
    close text window 1
    paste
    open {file "Macintosh HD:Chapter 03-Script
Recording:3. Warranty"} with LF translation
    select text 1 of text window 1
    copy selection
    close text window 1
    paste
    open {file "Macintosh HD:Chapter 03-Script
Recording:4. General"} with LF translation
    select text 1 of text window 1
    copy selection
    close text window 1
    paste
    save text window 1 to file "Macintosh HD:
Chapter 03-Script Recording:SW License"
end tell
```

The file path names in your copy will be different



Chapter 3:
**Your First
AppleScript
Script**

(reflecting your hard disk name and the location of the sample files on your drive), but the basic flow should be the same.

Saving the Script

Before running the script for the first time, be prudent and save the script. While it's not as big an issue for recorded scripts, you never know what may happen when you run a script the first time. The vast majority of the time, things will work fine or some safe error will occur. But there is always the slim chance that some instability could crash Script Editor, and you could lose the script. Saving the script is an insurance policy that will let you get back to this state should the worst happen.

I'll help you decide how to save this script. We'll turn it into an application, since we will probably want to give it to another Mac user who may not appreciate the scary look of Script Editor or know about managing Scripts folders. Also, we'll leave it in its editable form.

- 1) Choose **Save As** from Script Editor's **File** menu.
- 2) In the pop-up menu below the file name, choose **Application**.
- 3) Enter the file name **SW License Consolidator**.
Especially in the case of script applications you intend to give to others, the script's name should help the user know what the app does from the Finder, and pick out one script from perhaps dozens.

Checking the Script

We won't go through the contents of the script at this point (we'll come back to it after you've seen more about AppleScript syntax). But it's still a good idea to give the script a run-through before turning it over to someone else to use. First delete the one Software License file that you generated while recording the script. Then click Script Editor's Run button. You'll see those dozens of steps reduced to a single click.

If you should receive an error message that indicates something is wrong with the syntax, select and delete the contents of this script and re-record it. The steps in this example should work as described. Make sure during recording that you perform no steps or errant clicks other than the ones described above.

The Next Step

You've just recorded a pretty powerful script for automating a dull process. From here we go into some essential syntax basics that will let you start writing scripts from scratch, rather than just recording.



Chapter 4 Writing AppleScript Scripts—An Overview

To gain an appreciation for writing scripts, we'll spend some more time in Script Editor. The goal is not to learn specific commands just yet. Rather, you'll gain a familiarity with scripts and the environment in which they run.

Script Editor—The Bottom Pane

If you're still in Script Editor from the last chapter, close all windows, and choose **New** from the **File** menu. You'll see a blank, untitled script editor window divided into a top and bottom pane. By dragging the divider bar between the two panes you can adjust the relative sizes. You can hide either pane if you wish.

The top pane is where script code goes, but the bottom pane can play one of three roles. Notice the row of buttons at the bottom of the window (see Figure 4.1). These buttons let you choose how you want to utilize the bottom pane of the script's window.

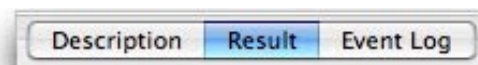


Figure 4.1. Buttons that control the bottom pane behavior.



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

Description

As shown in Chapter 3, if you save a script as a double-clickable application, you can also set it to display a startup screen. The Description field is where you insert the text you'd like your script's users to see before running the script (and giving them an option to cancel the script before doing anything). You may style text inside this field with the help of Script Editor's **Font** and **Format** menus. Select portions of text you wish to style, and then choose your font, style, color, and alignment from the menus (and some secondary dialog boxes).

Result

The vast majority of script lines produce a result of some kind. For example, if you enter a script line that adds two numbers, and then click the Run button, Script Editor displays the result of that script execution in the Result pane (Figure 4.2). Only the result of the last executing line of a multiple-line script appears in the Result window. Nine times out of ten, you won't even bother viewing these results. But at this crucial learning stage, you can use the Result window as a tool to understand how scripts work as well as to experiment with new terms you learn.

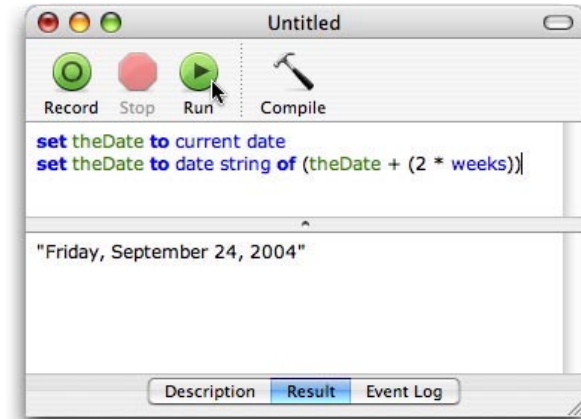


Figure 4.2. Result pane after running a simple script.

Event Log

Useful for debugging purposes, the Event Log displays a sequence of commands from your scripts that interact with applications and some indication of the result of each command. Unlike the Results display of only the last result of a sequence of script statements, the Event Log keeps a running record of the steps running in your script, but records only those steps that reach out to applications for commands or other resources. Thus, in Figure 4.3, only the first statement (which retrieves the system clock time) is recorded, but not the value manipulation of the second statement.



Chapter 4: Writing AppleScript—An Overview

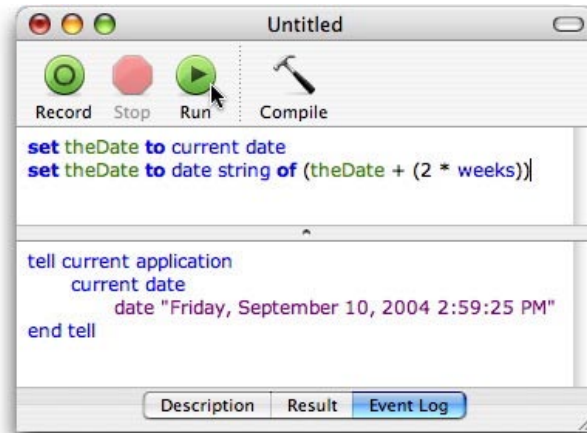


Figure 4.3. Event Log pane after running a simple script.

Setting Your Default View

Script Editor lets you assign the default setting for the lower pane as well as your desired window size for each new script window you create. Size the window, position the divider bar, and click on the desired lower pane type to make everything “just right.” Then choose **Save as Default** from the **Window** menu.

Additional Script Editor Windows

While we’re on the subject of Script Editor windows, let me introduce some additional windows that will come in handy once you start doing some hard-core scripting. Use the **Window** menu to display your choice of the **Result Log History**, **Event Log History**, and **Library**. If you’d like to have any of these windows open up automatically when you launch Script Editor, position and size them where you like, and choose **Save as Default** from the **Window** menu.

Result Log History

While the **Result** pane of the regular Script Editor window shows only the final and most recently executed script result, the **Result Log History** window records a succession of **Result** pane postings, allowing you to go back to previous results. The most recent entry inserts itself at the top of the list. If you select an older result from the left column and click the **Show Script** button, the script that formed the displayed result value appears in either a new script window or (if the original script is still open) the associated script window comes to the front. When you quit Script Editor, the history for the current session is not saved.

Event Log History

The **Event Log History** keeps a similar kind of record, but in this case, a record of what would appear in the **Event Log** pane of a script window. This history, too, is not saved when you quit Script Editor. Scripters



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

hard at work on complex scripts would likely keep both the Result Log History and Event Log History windows open at the same time. Script Editor's preferences settings let you control how many entries the Result Log or Event Log History window should hold.

Library

You'll learn more about dictionaries later in this chapter, but the Library window offers a shortcut way for you to open script dictionaries of applications that you use frequently (Figure 4.4). Apple loads its own Mac OS X scriptable applications in there by default. You can delete the ones you don't use often and add third-party applications as you like (via the + and – buttons or simply dragging an application's icon to the Library window).



Figure 4.4. The Script Editor Library window.



Chapter 4:
**Writing
AppleScript—An
Overview**

Script Editor Advanced Features

Two additional features built into Script Editor may appeal to experienced scripters. The first is the Navigation Bar. Turn it on by choosing **Show Navigation Bar** from Script Editor's **View** menu. The Navigation Bar appears between the buttons and script editing area in two segments. The first segment lets you choose the OSA scripting language for the script. For most users, only AppleScript will be available. But the second segment offers a list of property, global, and function handler definitions in the current script. If you have a large script with lots of these things in it, the Navigation Bar makes it easier to find and scroll to the relevant section of a long script.

The second feature is called Script Assistant. Turn this feature on by visiting Script Editor's Preferences for Editing, and engage the checkbox at the bottom of the Preferences window. Then quit and restart Script Editor (this step is necessary). When you start typing in Script Editor, it will try to help you by providing a list of possible words or phrases that may complete the script statement you started. When the Assistant has something to offer, you'll see an ellipsis (...) at the text pointer. Press the F5 key to view a list of possible completions that Script Editor knows about, and use the Up or Down arrow keys to scroll to the item you wish to choose. Press the Esc key or continue typing what you wanted to hide the popup list. Sometimes this kind of assistance can be distracting, so judge for yourself whether this one is helpful to you.

Compiling (Checking Syntax)

I haven't yet mentioned Script Editor's Compile button, which sits by itself to the right of the Run button. In earlier versions of Script Editor, this button was labeled Check Syntax, a label which accurately describes what the button does for script authors. The syntax of a recorded script is supposed to be accurate, so you didn't use that button in recording a script in Chapter 3. There is no penalty for compiling a recorded script, however.

You can type utter gibberish into the script, and the Script Editor won't mind—until you click the Compile button. At that point, Script Editor summons its AppleScript powers to double-check what you've typed against its own dictionary of AppleScript terms, plus some other places, as we'll see later. It also makes sure you've followed the syntax rules, such as putting commands before the things they're commanding.

What Compiling Does

For those who are experienced in programming environments, you'll understand that the Compile button triggers the AppleScript compiler to produce (in memory) a compiled version of the script. This is the same compiled version that gets saved to a compiled script and script application. Compiling the script does not run the script, although clicking the Run button on an uncompiled script compiles it first.

For those without programming experience, the brief



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

explanation is that compilation converts the human readable script you type into a more computer-readable version, which the Mac can execute much faster than having to translate the script each time you click the Run button. As you'll see later, this compilation step also makes sure that any commands you summon from other programs are, in fact, available to the script. Thus, syntax checking performs double duty.

The Outcome

If syntax checking uncovers a problem, a syntax error alert appears (Figure 4.5). The contents of these error messages can be all over the board, and we'll get into them later. It takes a bit of AppleScript experience to understand the real meaning of some of these messages.

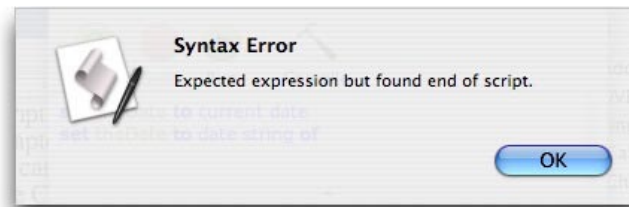


Figure 4.5. A typical syntax error alert.

You'll know if syntax checking was successful by two clues. First, you don't see any error messages. Second, the font of the script changes. As shown in Figure 4.6, as you type any new characters into a

script (including one that has already successfully compiled), the characters appear in a distinctive font, size, and color (the default is Courier 12 purple). After successful compilation (see Figure 4.7), the font changes to Verdana 12 (with some words possibly boldfaced and italic). You can change these font settings by clicking the Formatting button in Script Editor's application preferences (see Figure 4.8), and adjusting individual settings by double-clicking items to reveal Font and Color palettes.

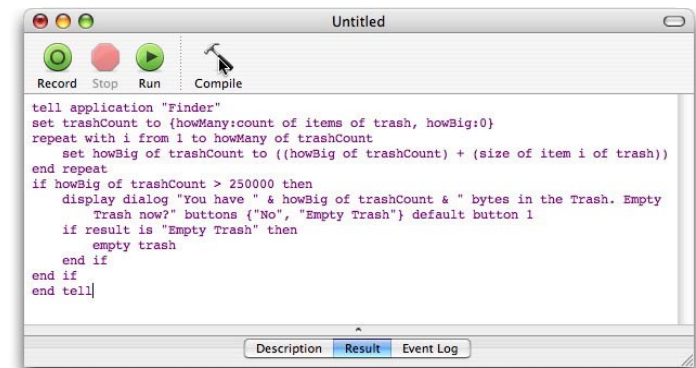


Figure 4.6. A script before compilation.



Chapter 4: Writing AppleScript Scripts—An Overview



Figure 4.7. A script after compilation.

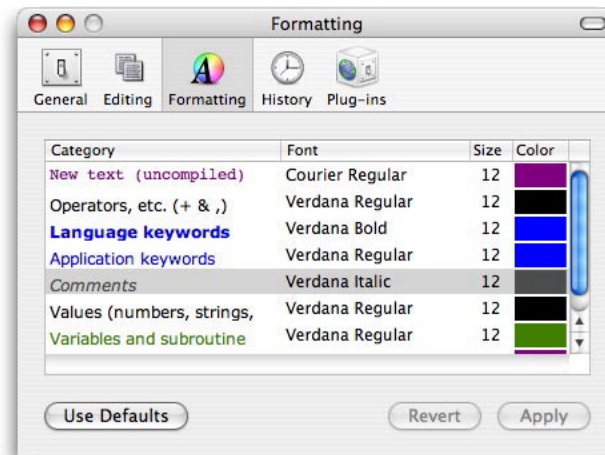


Figure 4.8. Script Editor dialog for adjusting font characteristics of scripts.

Let's try a bogus script to see what happens when compilation fails:

- 1) In a new script window, type the command **do my laundry**. This may be a valid command for some future Macintosh or iAppliance, but not the ones we're using today.
- 2) Click Compile (a handy shortcut for this button is the Enter—not Return—key). The first two words highlight, and a Syntax Error window appears (Figure 4.9). The error, we're told, is that the word "my" cannot go after the identifier "do". We know, of course, that there is far more wrong with this command than what it's telling us, but what AppleScript tells us is as much as it knows from the information we've given it.

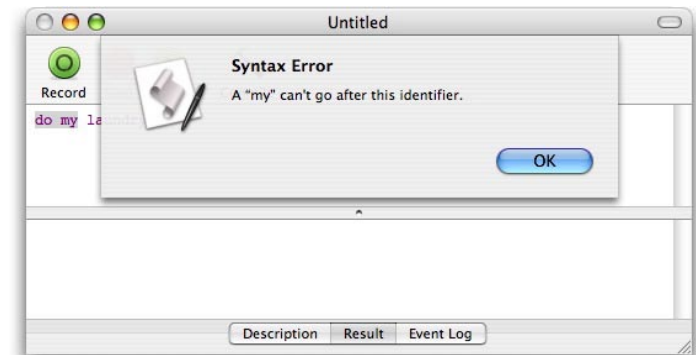


Figure 4.9. Words responsible for the syntax error highlight in the script.

- 3) Click the OK button. The script is still in Courier font, meaning that the script has not compiled.



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

Now, try a valid command:

- 4) Select all the text in the script, and press the Delete key to remove it all.
- 5) Type the command **beep**.
- 6) Click Compile. This time there is no error window, and the font of the **beep** command turns to Verdana. Making even a one-character change to the script will require recompilation.

Click the Run button to try your **beep** command. Certainly not the most impressive script in the world, but you've just *written* and run your first AppleScript script.

Smart Compiler

The compiler/syntax checker also tries to clean up messy scripts if it can, replacing a source script with a prettier version if necessary. For example, the **beep** command, when followed by a space and a number, plays the beep as many times as the number indicates. But if you type multiple spaces between the command and number, the compiler returns the proper version, with a single space. Try it:

- 1) If it's not already there, place the text insertion pointer at the end of the **beep** command.
- 2) Type six spaces and a **3**. The numeral appears in Courier font, because that part of the script hasn't be compiled yet (the spaces are in Courier, too, but you can't see that).
- 3) Click Compile. After successful compilation, the command displays with a single space between it

and its number (Figure 4.10).

- 4) Click Run to hear the beep three times.

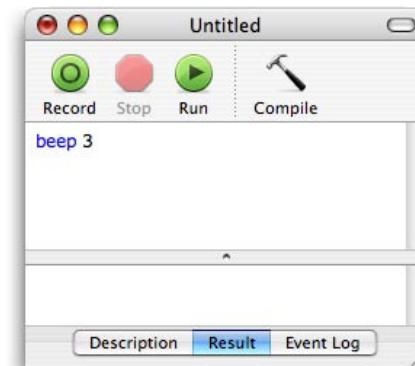
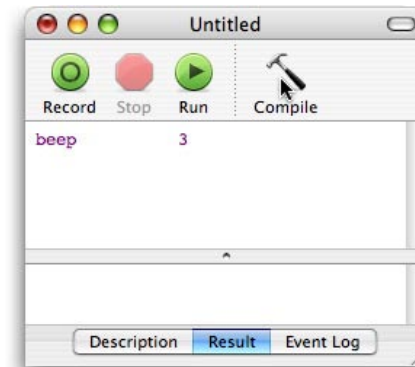


Figure 4.10. The sloppy command before (top) and after (bottom) compilation.

During compilation, AppleScript dissects each line, and figures out how each word fits into the grand scheme of things. Thus, as you see in the Formatting preferences window (Figure 4.8), you can instruct



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

Script Editor to highlight certain categories of words. In other programming environments, this feature is called “pretty printing,” but its goal is not aesthetic: it is to help someone reading the script—including you—more quickly understand how the script works. The default formatting is straightforward, with only one type of word in an active line of script highlighted in boldface. Gradually, you may evolve a different style of your own. There are no rules you must follow in this regard.

The formatting information from the Formatting preferences window (Figure 4.8) is not saved with a script file. Rather, these settings govern how your copy of Script Editor displays scripts from all sources. You can, however, make your formatting “stick” in your copy of Script Editor by choosing **Save as Default** in the **Window** menu.

Making a Statement

Every line of an AppleScript script is known as a **statement**. A script may consist of any number of statements, including a single line, such as the one-line **beep** statement you wrote earlier.

Another type of statement, called the **compound statement**, begins with only one of a few special AppleScript keywords (you’ll learn about these later), and must end with an **end** statement. The compiler performs special indenting of statements nested inside the outer edges of a compound statement. The script you recorded in the last chapter produced a compound statement, starting with the **tell** command. Notice that it had a balancing **end** statement—a requirement for all compound statements.



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

Commandments

At the minimum, a statement must contain a **command**, a word that acts like a verb to indicate some action that is to take place. Here is an abbreviated version of the script you recorded in the previous chapter (it brings in only a single component of the software license and uses a disk and folder naming system that probably differs from yours):

```
tell application "BBEdit"
    activate
    make new text window
    set properties of text window 1 to {soft
wrap text:true}
    open {file "Macintosh HD:AS Handbook
Scripts:Chapter 03:0. Introduction"} with LF
translation
    select text 1 of text window 1
    copy selection
    close text window 1
    paste
    save text window 1 to file "Macintosh HD:
Library:Scripts:Software License"
end tell
```

In the indented list of statements, there are two one-word statements (**activate** and **paste**), each one a command that the BBEdit knows how to execute.

More common, however, are statements that include additional words after the command. Collectively, items coming after a command are called **parameters**. In the statement, **make new text window**, **make** is the command, and **new text window** is its parameter.

Telling an Object What To Do

A command generally performs an action on an **object**. The term object refers to the object-oriented world I spoke about in Chapter 2. What the object *is* depends entirely on the objects that the application's designers have defined for their product.

For example, in the script statement above that starts with the **make** command, the object it's supposed to make is a text window (the **new** part of the command is optional—you'll learn later how to determine that). Some applications might require an additional parameter, such as **at beginning**, to provide instructions as to the sequence the new text window should appear if multiple windows are already open.

A couple of lines later, the **open** command also works on an object: a file with a valid pathname. And so it goes.



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

Common Actions

While it doesn't show up so much in a recorded script, a significant percentage of actions you'll be writing in scripts have to do with obtaining and inserting information. The AppleScript commands for these actions—**get** and **set**—will be the first word of most of the statements you'll write, especially for scripts that exchange information between documents.

For the moment, let's think about getting and setting in the context of one of those large warehouse type stores. In an organized warehouse, each location has some kind of location identifier, like a bin number or row and shelf number. Getting something with AppleScript is like picking out an item from a bin; setting is like putting the item someplace else—in another bin or in your shopping basket.

If this warehouse were a scriptable application, the bin would be an *object*. An item you pick from a bin would be an *element* of that bin. And the bin, itself, has a number of *properties*, such as location, size, minimum reorder point, and so on. The element, too, is an object, with its own properties, such as price, color, size (completely independent of the size property of the bin), and perhaps others.

We can summarize the purchase of an item at one of these stores in AppleScript terms:

- 1) Standing in front of the bin, we *get* the price property for one item.
- 2) If the price fits within a predefined budget, we *get*

the color property of the item on the top of the pile.

- 3) That color isn't what we want, so we *get* the color property of the second item in the pile.
- 4) It's the one we want, so we *get* the second box object in the pile (in the store, we actually remove the item from the bin, while in AppleScript, we only get a copy of the original, so we don't disturb the original item by getting it).
- 5) Finally, we *set* the contents of our shopping basket to what was already in there plus the new item object.

In the process of working with real applications and document data, your scripts will get and set elements and properties all over the place. In word processing programs, you'll encounter elements such as characters, words, and paragraphs, with properties for font characteristics, character length, and the like. Spreadsheets elements include cells, columns, rows, and ranges of cells, featuring properties such as widths, formulas, borders, and format. These objects, elements, and properties shouldn't be new to you if you've used these programs, but you probably didn't think about such items quite in those terms before. For these applications to be scriptable, their documents *must* be defined by these bits and pieces.



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

Where the Wording Comes From

I must emphasize again that the definitions of an application's commands, objects, elements, and properties are entirely up to the program's designers, and are not built into AppleScript. There is no guaranteed uniformity across similar applications (e.g., all databases) that the elements and properties (or the specific syntax for describing them) will be the same.

Application Dictionaries

Script Editor provides a tool for viewing the AppleScript terms that an application knows. Choose Open Dictionary from the File menu (Figure 4.11). The only kinds of applications that appear in the resulting open file dialog box are those that identify themselves to the operating system as being scriptable.

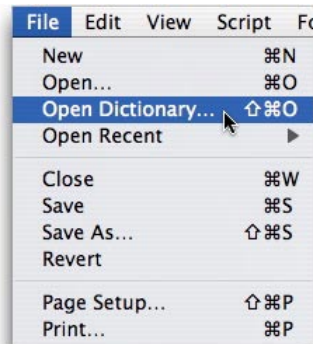


Figure 4.11. The Open Dictionary command in Script Editor's File menu.

Figure 4.12 shows the dictionary window for Apple's own "bare bones" text editor, TextEdit. When all groups in the left column are expanded, you see a list of all the AppleScript objects (classes) and commands that this application understands. When you click on a word in the column, the details for that word appear in the right. I won't go into the specifics of the words here, but it's important to understand the basic structure of a dictionary.

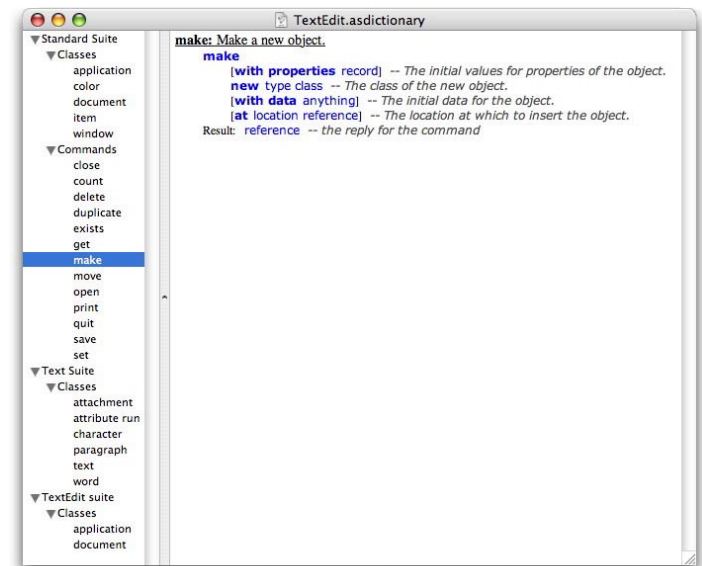


Figure 4.12. TextEdit's dictionary

How Suite It Is

All of Apple's scripting technology relies on a behind-the-scenes mechanism called Apple events.



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

AppleScript shields scripters from the complexity of working with Apple events directly. Instead, we can work in a language that often reads like English phrases and sentences to issue commands to objects and move data around.

Developers of scriptable applications follow Apple's developer guidelines to include a dictionary in their applications. Except for a relatively small set of very basic entries that virtually all applications have in common, the depth, organization, and helpfulness of a dictionary varies from one developer to the next.

Since the earliest days of AppleScript, dictionary authors were encouraged to group their entries into related bunches, thus making it easier for a scripter to locate a particular term and determine its capabilities. Each primary group in an AppleScript dictionary is called a **suite**. In Figure 4.12 you see that TextEdit's dictionary has three suites: Standard, Text, and TextEdit. Each suite contains listings of the classes (objects) and/or commands associated with that suite. Some suites list only classes, others list only commands, while others list both classes and commands—all depending on how the program's designers decided to expose its inner workings to scripters.

Every application contains the Standard Suite (previously known in the AppleScript world as the Core or Required Suite), but only a handful of the classes and commands shown in TextEdit's dictionary are absolutely required. By and large, however, you

will find the same classes and commands in the Standard Suite listings across all modern scriptable applications. Applications that work with textual data also typically implement Apple's recommend Text Suite (as listed in Figure 14.12).

Beyond those two suites, however, application developers have the freedom to assemble dictionary suites as they see fit. Sometimes there is a single application-specific dictionary. For example, Safari's dictionary lists a total of three suites: Standard, Text, and Safari. But BBEdit's dictionary list 13 suites, including ones named BBEdit, HTML Scripting, Sort Lines Scripting, and so on.

Not So Suite

Beneath each suite name are all classes and/or commands that the program's designer decided belong to that suite. One of the things that makes learning an application's AppleScript implementation difficult at times is that there is no specific roadmap in the dictionary that links commands with classes. Some commands work with only a limited number of objects listed for that suite. For example, if you're new to scripting TextEdit, you may think that you can create a new document by the statement

```
make new window
```

because the **make** command and **window** class are listed under the Standard suite (and the definition of the **make** command just says it requires class type as a parameter). What you can't tell directly from



Chapter 4:
**Writing
AppleScript
Scripts—An
Overview**

the definition is that the **make** command works effectively only with document objects. Nothing in the application's resources (from which all information shown in the dictionary window comes) can tell us specifically which command works with which objects. Initially, it becomes a trial and error experience or requires a study of whatever external documentation might be supplied by the application's developer.

In future chapters, we'll spend more time in the dictionary because it is a vital resource to help you know what a scriptable application can do. But as you've just seen, the dictionary doesn't tell the whole story.



Chapter 5 A Crash Course in Programming Fundamentals

As with a lot of programming environments I've seen, learning AppleScript sometimes seems to require knowing many parts before you can learn the first part. Subject A requires a knowledge of subject B; but you can't learn subject B without knowing subject A. If AppleScript is your first programming experience, this can be frustrating. It seems as though everything whirls around like a carousel without a brake. 'Round and 'round it goes—when you try to hop on, forces of nature repel you.

The goal of this chapter isn't necessarily to stop the wheel, but to slow it down enough so an inexperienced programmer can hop on safely. I'll introduce you to important terminology and concepts that will make learning AppleScript comparatively easy. If you are an experienced programmer or scripter, you can skim this chapter to make sure that your understanding of these terms is the same as the way they're used in AppleScript.

Open Script Editor and TextEdit, because we'll type some things in a Script Editor window (sometimes talking to TextEdit) to reinforce the meaning of these

concepts. Don't worry if you don't fully comprehend everything in this chapter the first time, because we'll revisit all these terms in more detail in succeeding chapters.



Chapter 5:
**A Crash Course
in Programming
Fundamentals**

Statements

In the previous chapter, you saw that a script consists of a series of statements. They may be all simple statements—one-line statements that appear aligned along the left margin of Script Editor's window (shown here divided into two lines to fit the page):

```
tell application "TextEdit" to -  
    open file "MacHD:Software License"
```

More commonly, especially when scripts perform a number of steps inside an application, the statements are grouped into a compound statement:

```
tell application "TextEdit"  
    open file "MacHD:Software License"  
    print front window  
    close front window  
end tell
```

Compound statements begin with only a limited number of AppleScript words (predominantly **tell**, **if**, **repeat**, **try**, **on**, and **to**), and always end with an **end** statement. After compilation, compound statements are formatted so that the simple statements within a compound statement are indented. In the example, above, the **tell** compound statement contains three simple statements.

Commands and Objects

Statements, as you've also seen, usually consist of at least one command, such as the **beep** command. Many commands also accept or require additional words—parameters—to help them carry out the action.

The **beep** command has an *optional* parameter: the number of times the beep sounds. If you don't specify a parameter, then the command has a default parameter of 1. Therefore,

```
beep  
  
and  
  
beep 1
```

achieve the same action. But if a command *requires* a parameter, then a parameter must follow the command in the statement. For example, the **close** command in TextEdit requires a parameter signifying what should be closed. Thus, in the statement

```
close front window
```

the parameter to the **close** command is **front window**.

The formal definition of the **close** command in TextEdit's dictionary, says the following:

```
close reference -- the object for the command
```

The word, "close", is clearly the command here. The parameter, however, is defined merely as something called a reference. In AppleScript, a *reference* is the manner in which we distinguish one class (object)



Chapter 5:
**A Crash Course
in Programming
Fundamentals**

from all the rest. Recall I said in the last chapter that commands perform actions on objects. For AppleScript to know precisely which object you mean—among perhaps thousands of possible objects “living” in your Macintosh at any instant—you signify the object by its reference. Take the statement

```
open file "Hard Disk:Chapter 03:Software License"
```

as an example. The object of our **open** affections is a file whose hypothetical Finder path name is “Hard Disk:Chapter 03:Software License.” The wording—**file "Hard Disk:Chapter 03:Software License"**—is a valid reference for a file as far as TextEdit is concerned.

Object Elements and Properties

An object’s dictionary definition includes what component parts (*elements*) make up that object (if any), and what *properties*—characteristics or attributes—help define the object. In some cases, an object’s property is another object, which, in turn, has its own properties and perhaps elements within it. For example, in TextEdit’s TextEdit Suite, the **document** is one of the main objects. One of its properties is an object called **text**. In turn, the Text Suite contains a **text** object, whose dictionary entry reveals that its elements are objects such as **character**, **word**, and **paragraph**. If we want to refer to a particular word in the document, we assemble a reference from the point of view of the document:

```
word 3 of text of document 1
```

If that word 3 happens to be the first word of the second paragraph, then we can also refer to it in the context of the paragraph

```
word 1 of paragraph 2 of text of document 1
```

because paragraphs, themselves, have words as elements. Both object references point to the same word, and either one would be valid in an AppleScript script directed to TextEdit (Figure 5.1). This element business sounds more complicated than it is. The more familiar you are with a program you’ll be scripting, the more logical become the relationships between objects and the elements that belong to them.

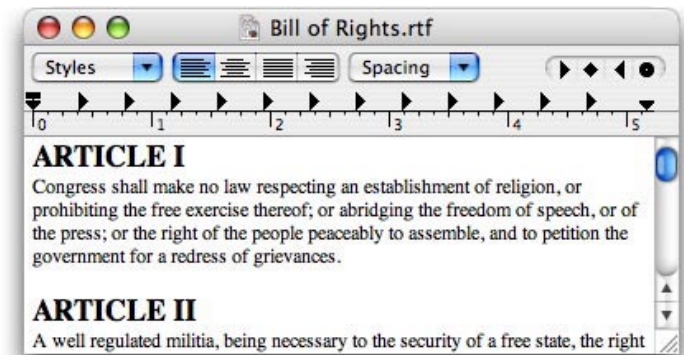


Figure 5.1. In this document, a word has position relative to the entire document and to its own paragraph. “Congress” is both word 3 of the document “Bill of Rights.rtf” and word 1 of paragraph 2 of the same document.



Chapter 5:
**A Crash Course
in Programming
Fundamentals**

Usually more important than an object's elements are the object's properties as revealed in the dictionary. Object properties define the characteristics of the object—sometimes visible, sometimes not. Table 5.1 shows the list of the properties of a TextEdit **window** object in the order in which they appear in the AppleScript dictionary:

<i>Property</i>	<i>Read-Only</i>	<i>Description</i>
zoomed	no	Is the window zoomed?
miniaturized	no	Is the window miniaturized?
name	no	Title of the window
floating	yes	Does the window float?
modal	yes	Is the window modal?
miniaturizable	yes	Is the window miniaturizable?
visible	no	Is the window visible?
closeable	yes	Does the window have a close box?
resizable	yes	Is the window resizable?
zoomable	yes	Is the window zoomable?
id	yes	Unique identifier of the window
bounds	no	Boundary rectangle for the window
titled	yes	Does the window have a title bar?
index	no	Number of the window in sequence
document	yes	Reference to the window's document

Table 5.1. *Properties of a TextEdit window object.*

You don't have to remember these properties at all, but once you start working with an application, it will be helpful to acquaint yourself with the range of properties for the objects you'll be dealing with. In this case, most of them govern the visible appearance and behavior of a window. Simple properties may be either on or off (their values set to **true** or **false**), while others consists of coordinates, numbers, or text. Many properties here are read-only, which means that scripts can only find out what their

settings are, and cannot change them.

To retrieve the contents of a property at any instant, you use the AppleScript **get** command, as in:

```
tell application "TextEdit"
    get name of window 1
end tell
```

```
tell application "TextEdit"
    get bounds of window 1
end tell
```

Changing the value of a property involves the **set** command, as in:

```
tell application "TextEdit"
    set bounds of window 1 to {20, 59, 481, 305}
end tell
```

with the items in curly braces representing screen coordinates in pixels. With at least one window open in TextEdit, try running this last script yourself with Script Editor.



Chapter 5:

A Crash Course in Programming Fundamentals

Working with Information

Virtually every statement you write in a script contains information—data. Even setting a property, like the bounds of a window, above, contains data about the coordinates. Any kind of data that can be shuffled around inside AppleScript is called a *value*. AppleScript values come in a number of different varieties, called *value classes*. One reason for this is that parameters for commands often accept only a specific kind of value.

For example, in the statement

```
beep 3
```

the **beep** command accepts an optional value to specify how many times to beep. The parameter must be an integer (a number without any decimal fraction). Trying to feed another kind of value, such as a series of letters between quotes (called a string value) would result in an error (you'll learn more about this in a moment). Table 5.2 lists examples of the most common value classes used in AppleScript:

Class	Example	Description
Boolean	true	A logical true or false
integer	233	A positive or negative whole number (no decimals allowed)
list	{1,2,3}	A series of other values—of any class—arranged in a specific order inside curly braces
real	2.5	A positive or negative number with decimal fraction
string	"Howdy"	A series of characters inside quote marks

Table 5.2 Common value classes in AppleScript.

There are additional value classes, which you'll meet in Chapter 9. By and large, values do their work automatically, as long as you help them when necessary (e.g., putting quotes around strings or curly braces around lists). Occasionally, however, it will be necessary to change one value class to another. For example, a script may fetch some real numbers, perform some math on them, and then have to insert them into another document as a string. This conversion process is called *coercion*, but we can save that discussion for later.



Chapter 5:
**A Crash Course
in Programming
Fundamentals**

Variables

In the course of any kind of programming, it is often necessary to hold values in temporary baskets while the script gets additional data or performs operations on the original values. Consider this script:

```
tell application "TextEdit"
    set oneBlock to paragraph 1 of document 1
    make new document at beginning
    set the text of document 1 to oneBlock
end tell
```

Let's focus on the three statements inside the compound statement. The first uses the **set** command to place a copy of the first paragraph of an opened document into a basket we've arbitrarily named **oneBlock**. Since the contents of that paragraph are now safely in the basket inside our script, we can forget about document 1 and move on. The script next creates a new document window and then sets the contents of that window to whatever the **oneBlock** basket holds at that instant. In AppleScript, this kind of holding basket is called a *variable*.

You assign values to variables with either the **set** or **copy** commands. The following two statements accomplish the same task:

```
set myMac to "PowerBook G4"

copy "PowerBook G4" to myMac
```

They both put the string "PowerBook G4" into the variable named **myMac**. For the most part, you can

use either syntax, depending on which one reads better to you, but as you'll learn in Chapter 6, the **set** command has additional powers that may be useful in certain situations.

An AppleScript variable can hold any type of value (string, integer, list, etc.). And unlike many programming languages, you don't have to predetermine what kind of value a particular variable will hold. In fact, you are allowed to change a variable's contents radically during script execution, holding a reference to a file in one statement and a text string in the next—if your script design so requires.

The names you assign to variables can play a critical role in your understanding how a script works. AppleScript allows single-word variable names without any coaching. You may, however, use multiple-word variable names, provided the words are contained between vertical bar characters (the shifted backslash key). For example, the following variable names are valid in AppleScript:

```
oneBlock
one_Block
|one Block|
```

My personal preference is for single-word names consisting of two or more plain language words—each word after the first being capitalized (e.g., **oneBlock**). I find this style easiest to read and also the least likely to conflict with words reserved (now or in the future) by AppleScript or scriptable



Chapter 5:
**A Crash Course
in Programming
Fundamentals**

applications. I also often use single letter variables for repeat loop counters, as you'll see in Chapter 10. The important point to remember is to assign names that describe the contents of the variable. I'll have more to say about variables in Chapter 9.

Expressions and Evaluation

We've seen that any piece of data—an integer, a string—is a value in AppleScript. Another concept that is closely related to the value is the *expression*.

We use expressions in everyday language all the time. Consider the sentence:

"I want to watch television."

The word, "television", is universally understood to be the box that replays sound and pictures carried to it from an antenna or cable. We use the word "television" as an expression to stand in for the meaning of a more formal or complete definition of that device. But it's not uncommon for us to use other expressions for that television:

"I want to watch *TV*."

"I want to watch *the tube*."

"I want to watch *the idiot box*."

All three sentences end with different words, but in our minds, we know what they mean. We automatically (often unconsciously) *evaluate* each term to our mental picture of the device.

In an AppleScript statement, all values are automatically evaluated to their true meanings. For example, an expression that is already at its truest form, say the integer 3, can be evaluated no further. But the expression $3+4$ evaluates to 7.

Variables also get involved here. When we set a variable to hold some value, the variable becomes an



Chapter 5:

A Crash Course in Programming Fundamentals

expression that evaluates to whatever the contents of the variable are. For example, after executing the statement

```
set x to 5
```

the variable **x** evaluates to its contents: 5. In the next example,

```
set y to 5 + 5
```

before **y** is assigned its value, the arithmetic expression, `5+5`, is evaluated to 10. The variable **y** holds the value 10.

Why this expression evaluation stuff is important in AppleScript is that it grants scripters great flexibility in working with values as parameters to commands. Consider the **open** command from TextEdit we've used before. We saw that it requires as a parameter a reference to a file. We had hard-wired the parameter before, as

```
open file "Hard Disk:Chapter 03:Software License"
```

But we can substitute any expression after the **open file** command that evaluates to the string that is expected for the file's path:

```
set pathName to ~  
    "Hard Disk:Chapter 03:Software License"  
open file pathName
```

Here, the **pathName** variable contained a string. Before the **open** command executed, AppleScript evaluated all expressions in the statement to their true form. Once the **pathName** variable evaluated to the string, the command read internally just like the

hard-wired version. This is particularly useful if you intend to use that value many times in a script—you need only the short variable to stand in for the long, quoted string.

The key point to remember about expressions and evaluation is that you can substitute any expression for a value, as long as the expression evaluates to the kind of value that is expected at that instant.



Chapter 5:

A Crash Course in Programming Fundamentals

Operators

It's quite common to use *operators* in expressions. In fact, in one example we used above,

```
set y to 5 + 5
```

we used the addition operator, signified by the plus symbol. An operator generally performs some kind of calculation or comparison with two values to reach a third value. Arithmetic operators are easy enough to grasp, since they let us add, subtract, multiply, or divide two values (integer or real number classes of values) to arrive at the answer.

Another kind of operator lets us join two string values into a longer string that is the combination of the two. That operator, represented by the ampersand (&) symbol, is called the *concatenation* operator. The expression

```
"John" & "Doe"
```

evaluates to "JohnDoe", because the concatenation operator is very literal when it comes to joining strings. If you want a space between concatenated strings, then you've got to put it there yourself. One way to put the space between the strings is to put the space inside one of the component strings, as in

```
"John " & "Doe"
```

Or, you can concatenate three strings, one of which is merely a space, as in

```
"John" & " " & "Doe"
```

Depending on the source of your strings, you may end up using both forms in your scripting.

You will often use multiple layers of expression evaluation with operators. In the following script sequence

```
set x to 5  
set y to 20  
set z to x + y
```

the last statement first evaluates each of the variables, **x** and **y**, before adding them. Only then does AppleScript set the value of **z** to 25.

AppleScript is also loaded with operator vocabulary to help you compare values. Not just to find out if things are the same as, less than, or greater than each other, but whether one item comes before another in a list and whether one string contains another string. You'll see what all these operators are in Chapter 11. These kinds of comparison operators return only one kind of value, a Boolean—**true** or **false**. Boolean values are used frequently as on/off indicators for object properties. But where these comparison operators and their Boolean results come into play is in constructing scripts that make decisions based on the results of comparisons. We do this in real life many times a day: if the room is dark when we enter it, then we turn on the lamp first; otherwise we go straight ahead with what we planned to do there. We'll cover this in detail in Chapter 10, when we start working with **if-then** decisions.



Chapter 5:
**A Crash Course
in Programming
Fundamentals**

Continuation Symbols

I'll introduce here a symbol (`↵`) that you will see in many scripts throughout this book. The symbol is called a *continuation symbol*. A statement may be too long to fit comfortably in the margins of this book or within the viewable area of your Script Editor window. You can break up the line into smaller chunks, provided you put a continuation symbol at the end of the broken line.

While the symbol is normally typed as Option-L, Script Editor lets us insert one by holding down the Option key while pressing the Return key. It's like a "soft" return. There are some tricks involved if you want to break up a line in the middle of a string, but we'll look into that in Chapter 9.

Comments

The last item I'll cover here is something else you'll see often in scripts throughout this book. They're called *comments*, and are pieces of a script that do not compile or execute. Comments are useful as extra plain language guides to let someone reading the script understand a statement that may not be obvious.

To instruct the compiler to ignore comments, they must be formatted in one of two ways. One way is a double hyphen preceding the comment. In this fragment

```
-- we set the x variable  
set x to 5 + 5 -- x is now 10
```

the only part that compiles and executes is `set x to 5 + 5`. Because all other items began with the double hyphen, the compiler ignores them. The double-hyphen is useful for one-line or in-line (i.e., at the end of a valid statement) commands.

Comments may also be bracketed by special symbol combinations. The comment begins with `(*` and ends with `*)`, as in

```
(* This is a long comment,  
   which can go  
   on for many lines. *)
```

Comment sections of any script appear in italics after compilation (this is the default behavior of Script Editor, although you can choose a different font or style for comments, as described in Chapter 4). Good scripts use comments liberally, because they help you remember six months from now why you did certain things in your scripts.

A Lot of Stuff

This chapter has crammed a lot of terminology and concepts into a small space. If you didn't get it all, don't worry too much, because we'll cover everything in more depth in succeeding chapters. It was important, however, to acquaint you with these concepts, because they'll help you work with the details. We begin with the details in the next chapter with basic AppleScript commands.



Chapter 6 Issuing Commands and Getting Results

In previous chapters, you've seen three of AppleScript's built-in commands—**get**, **set**, and **copy**—in action. In this chapter, we'll learn more about them, and see some more of what else AppleScript comes with. Before we get to the actual commands, however, there is some other ground to cover, namely where the commands come from, how commands return information back to a script, and how we direct commands to the proper target.

Commands Provoke Action from Something

AppleScript commands are the verbs of the language. As the first word of any simple statement, a command signals the action that is to take place as a result of the statement. You may wonder, though, that if a human language command has an intended audience, what about an AppleScript command?

AppleScript does, too. In formal terminology, the recipient of a command is considered an *object*. But which object? Consider the command

```
beep
```

in a script all by itself. When you run this script, AppleScript looks for an object to send it to. Since none is specified in this statement or elsewhere in the script, AppleScript sends the command to itself. If there is a command within AppleScript's own dictionary to match, then it carries out the command without complaint.



Chapter 6:
**Issuing
Commands
and Getting
Results**

In truth, a single **beep** command within Script Editor first goes to the “current application,” which is the very same Script Editor program (the Event Log display reveals this info). If the command is not in the current application’s dictionary, the scripting system then looks into other available commands. It turns out that the **beep** command is one of a group of convenient language extensions delivered with Mac OS X, and covered in Chapter 7. If the command were not part of the language extensions or built into the basic command set, then a syntax error would result when you attempt to compile or run the command.

When it comes time to direct a command to an application, the script must specifically do so with a **tell** statement. Here’s how a **tell** statement looks in simple form:

```
tell application "TextEdit" to -  
    make new document at beginning
```

In other words, you’re sending the command, **make new document**, to the TextEdit program. That program has the **make** command in its dictionary (it’s part of the Standard Suite built into most scriptable applications), and knows what to do with it. If your script had been the single line

```
make new document at beginning
```

then the implied target of the command is Script Editor, and you end up with a new Script Editor window instead.

Groups of Statements—The Compound Statement

It would be exceedingly cumbersome to designate the object of every command in a script. AppleScript provides a shortcut for grouping commands directed toward a single object. By creating a *compound tell* statement, all commands inside it are directed to the *default object*—the last object signified by a **tell** statement in the script. For example, look at the following script:

```
tell application "TextEdit"  
    activate  
    make new document  
end tell
```

Both the **activate** and **make new document** statements have TextEdit as their target objects.

Nested Tell Statements

As scripts get more complex and rely on multiple applications, it may be necessary to temporarily redirect the default object of some steps within a compound **tell** statement. Schematically, such a script that needs to communicate with hypothetical word processing and spreadsheet programs might look like this:

```
tell application "ScriptWord"  
    ScriptWord-compatible statement 1  
    ScriptWord-compatible statement 2  
    tell application "ScriptSheet"  
        ScriptSheet-compatible statement 1  
        ScriptSheet-compatible statement 2  
    end tell
```



Chapter 6: Issuing Commands and Getting Results

```
ScriptWord-compatible statement 3
ScriptWord-compatible statement 4
end tell
```

The important item to notice here is that ScriptSheet understands only commands in its dictionary—not those from ScriptWord’s dictionary. The **tell** statement defines the focus, no matter how deeply nested it may be.

If the script you’re writing works with two applications in a serial, rather than nested, fashion, you could also structure your script like this:

```
tell application "ScriptWord"
    ScriptWord-compatible statement 1
    ScriptWord-compatible statement 2
end tell
tell application "ScriptSheet"
    ScriptSheet-compatible statement 1
    ScriptSheet-compatible statement 2
end tell
```

Scripts continue to execute down the statements until there are no more statements to execute.

Mixing it Up

You can also combine a single statement directed at a different object within a compound **tell** statement. Here’s an example of how this might look with an intermediate statement directed to a desktop publishing program from within a script targeting TextEdit:

```
tell application "TextEdit"
    -- puts a paragraph from default object,
    -- TextEdit, into the variable oneBlock
```

```
set oneBlock to paragraph 1 of document 1
-- the next statement goes to a desktop
-- publishing application that has defined
-- a text block object where text can be
-- dropped in
tell application "Scriptable DTPer" to
set text block 3 of document 1 to oneBlock
-- back to TextEdit as default object
set paragraph 1 of document 1 to "✓" &
oneBlock
end tell
```

More Specific Targets

So far we’ve just been directing commands toward applications. But a script can be even more specific about a default target object if it helps the cause of the script. For example, if a script performs a lot of actions in a single document window, and you know there will be only one document window, then the script can set up the document window of a specific application as the default object. The following scripts grab a copy of the first paragraph of a TextEdit document and then close the document. Instead of:

```
tell application "TextEdit"
    set oneBlock to paragraph 1 of document 1
    -- directed to document 1
    close document 1 saving no
    -- and here, too
end tell
```

we can make the script more compact by saying:

```
tell document 1 of application "TextEdit"
    set oneBlock to paragraph 1
    -- directed to document 1
    close saving no -- and here, too
end tell
```



Chapter 6:
**Issuing
Commands
and Getting
Results**

Notice that by including the reference to the first document as part of the **tell** block's target, the two statements inside the **tell** block are directed at the first document open in TextEdit and don't need to repeat that reference. The point to observe for now is that you can be as granular as you like when establishing a default target.

The purpose of these exercises in default target-ry is to demonstrate that every command needs a target object as a recipient. You can specify the target in a simple statement, in a compound **tell** statement, or ignore the object entirely if the default target you want is built into the AppleScript system or is a scripting addition.

Networked Applications

It's not uncommon these days for multiple Macintoshes to be linked together on a local area network (LAN) through a TCP/IP connection. A user has the option of setting a system preference that permits, say, a network administrator to execute scripts controlled from the administrator's Mac but targeting applications running on the user's Mac. To set up a Mac to be receptive to external control, open System Preferences and the Sharing pane. Activate Remote Apple Events by checking the checkbox (if it isn't already checked). This is the only pane needed, but you will also need to know the IP address of this Mac, which you can see by temporarily turning on Personal File Sharing in the remote Mac.

To direct AppleScript commands to an application on a remote computer, you specify as part of the **application** parameter (of a **tell** statement) the URL of the Macintosh—a string consisting of **eppc://** and the machine's IP address. A volume from the other Mac does not have to be mounted on your Desktop for this feature to work.

The format for the reference to an application on another machine is:

```
application applicationName of machine -  
    "eppc://n.n.n.n"
```

Here is a networked version of our software license example from Chapter 3. In this scenario, each user has the license section on his or her machine. The job of the script is to use the BBEdit application on each user's machine to open and copy the segment, and then bring it back into the combined document:

```
tell application "BBEdit" -- our copy  
    make new text window  
    set properties of text window 1 to {soft  
wrap text:true}  
    -- now talk to Steve's copy of BBEdit  
    tell application "BBEdit" of machine  
        "eppc://192.168.1.103"  
        open {file "SteveHD:Current:0.  
Introduction"}  
        set oneDoc to text of text window 1  
        close text window 1  
    end tell  
    -- now back to our copy  
    copy oneDoc to end of text of text window 1  
    -- do the same for other parts on other Macs  
end tell
```



Chapter 6:
**Issuing
Commands
and Getting
Results**

You'll notice that we don't use the **copy** and **paste** command scheme from the recorded script. The reason is that by working through Steve's installation of BBEdit, the **copy** command would have used his clipboard—and clipboards don't transfer from machine to machine. Therefore, we place the data in a variable, which lives in our machine in our script, and makes the journey to our document.

As an author of scripts that perform actions across Macintoshes on a network, you must be aware of this and other scripting limitations. One, in particular, will affect your design. While a **tell** statement to an application on your own Mac opens the application before sending the commands, you must open a remote application through a separate operation via the remote machine's Finder application. For example:

```
tell application "Finder" of machine
    "eppc://192.168.1.104"
        open application file "Steves Mac:
        Applications:Safari"
    end tell
```

When the application object of a command is on another Mac, the application must already be open.

Where the Words Are

You probably got the point already that the native AppleScript system, by itself, comes with only a small handful of commands. The rest of the language comes from outside of AppleScript. Here are the four places commands (and objects) are made available:

- 1) **AppleScript.** All these commands are built into the AppleScript system services, and can be called anywhere in a script.
- 2) **Scripting additions.** These are system extensions of a very special type, and must reside in the *Library/ScriptingAdditions* folder (of the System or User regions of your hard disk). Scripting additions transparently extend the vocabulary of AppleScript as far as the scripter is concerned, because these commands can be called anywhere in a script. Scripting additions (sometimes called OSAXes or OSAXen, after their pre-Mac OS X file type and Mac OS X file name extension) provide enormous flexibility for AppleScript, allowing programmers in C and other languages to extend AppleScript's native abilities. You can find a number of scripting additions on electronic bulletin boards frequented by AppleScript scripters.
- 3) **Scriptable Applications.** These commands can be used only when directed to the application. You can uncover which commands are supported by an application by choosing **Open Dictionary** in Script Editor's File menu.
- 4) **User-defined commands.** These are subroutines that scripters can design and make a part of a script. You can think of a user-defined command



Chapter 6:
**Issuing
Commands
and Getting
Results**

as a scripting addition written in AppleScript. A user-defined command may exist inside the script that calls it, or may be loaded into the script when the script runs (see Chapter 14 for more about script libraries).

Who Gets What?

Each of the four kinds of commands has a different scope. The question is, then, what does AppleScript do with a command that isn't in the scope of the default target? For example, in the following script, I'll comment as to the location of the command vocabulary for each statement inside the **tell** statement (with each command shown in **boldface**):

```
tell application "TextEdit"
    activate -- TextEdit
    open file "Hard Disk:SampleFile"
        -- TextEdit
    set oneBlock to paragraph 1 of document 1
        -- AppleScript
    make new document at beginning -- TextEdit
    set text 1 of document 1 to oneBlock
        -- AppleScript
    display dialog "The job is done!"
        -- scripting addition
end tell
```

Yet in all cases, the commands first go to the default target, TextEdit. The mechanism, however, provides a pathway for these commands to follow if nothing intercepts them. Here are the rules:

- 1) If the command has AppleScript as the default target (i.e., it is not inside a **tell** statement directed at an application), then the command

follows this path in search of a target that knows what to do with the command:

- a. User-defined command
- b. AppleScript command
- c. Scripting addition command

Here is an example you should try in Script Editor:

```
choose file with prompt "Pick a file, any file:"
doMyCommand()
-- a user-defined command, the same
-- as a scripting addition
on choose file with prompt promptText
    display dialog "I trapped the choose file
        command."
end choose file

-- a user-defined command just for this script
on doMyCommand()
    display dialog "The command reached this
        user-defined command."
end doMyCommand
```

When you run this script, you get two dialog boxes that show how both commands were trapped by user-defined command handlers in the same script (even though one of them, **choose file**, is also a scripting addition).

- 2) If the command has an application as the default target (i.e., it is inside a **tell** statement), then the command follows this path:
 - a. Application command
 - b. AppleScript command
 - c. Scripting addition command



Chapter 6:
**Issuing
Commands
and Getting
Results**

If we modify the example, above, so that the two primary statements are directed to TextEdit:

```
tell application "TextEdit"
    choose file with prompt "Pick a file, any file:"
    doMyCommand()
end tell

-- a user-defined command, the same as a
-- scripting addition
on choose file with prompt promptText
    display dialog "I trapped the choose file
    command."
end choose file

-- a user-defined command just for this script
on doMyCommand()
    display dialog "The command reached this
    user-defined command."
end doMyCommand
```

then when you run the script, you get the actual choose file dialog from the scripting addition (the command never looked for a user-defined match in the script). When the running script reaches the **doMyCommand()** statement, it generates an error message that says TextEdit doesn't know this command. In other words, AppleScript has entirely different hierarchies, depending on the default object of the command (see Chapter 14 for ways to alter the message hierarchy for user-defined commands, including how to get the call to **doMyCommand()** in the above script to execute the custom handler at the bottom of the script).

More About Parameters

We saw in earlier chapters that commands sometimes have extra words appended to them that supply necessary information for the command to do its job. These words are called *parameters*. An important part of each command's description throughout this book are the details of what parameters, if any, accompany the command.

In the command definitions, you'll see two different kinds of parameters: direct and labeled parameters. Both kinds were used in the script you recorded in Chapter 3.

A *direct parameter* generally consists of the object that the command is to work with. For example, in the statement

```
select text 1 of text window 1
```

the **select** command's direct parameter is an object reference to the contents of a BBEdit document window (**text 1 of text window 1**). The ever popular **get**, **set**, and **copy** commands all use direct parameters.

In contrast is the *labeled parameter*, a parameter whose value is preceded by a plain language word describing what the value is supposed to represent. BBEdit's **open** command specification contains one direct parameter (a reference to a file) and several labeled parameters, one of which is named **LF translation**, corresponding to the way the program should handle linefeed characters



Chapter 6:
**Issuing
Commands
and Getting
Results**

within the file being opened. The value of the **LF translation** parameter is a Boolean value (**true** or **false**). Using an explicit Boolean value, the command would look like this (with *filepath* standing in for the path to the file on your hard disk):

```
open {file filepath} LF translation true
```

If you followed along with the recording of Chapter 3 inside BBEdit, however, you saw a variation in the syntax: the keyword **with** in front of a labeled parameter implies that the value assigned to the parameter is **true** (the converse would be **without**). Thus, the recorded version looks like the following:

```
open {file filepath} with LF translation
```

Here is another example of a popular command statement using one direct and one labeled parameter:

```
close document 1 saving no
```

The **close** command (as defined in AppleScript dictionaries of many programs) has both a direct parameter (a reference to an open document) and a labeled parameter (**saving**). When the designers specified the behavior of the **close** command, they realized that if a document is edited in any way, the program will ask the user about saving changes before closing the document. In real life, we have the choice of not saving those changes. We have the same choice while scripting. The command includes a labeled parameter, called **saving**, which requires either a **yes**, **no**, or **ask** value immediately

following the parameter label—**yes** to save changes; **no** to ignore changes; **ask** to have the program query the user about saving. Labeled parameters make statements much more meaningful to someone reading a script. Instead of a gibberish list of parameter data, the statement almost reads like a sentence.

A notable convenience about labeled parameters is that they may be used in any order that makes the most sense to you. For example, the formal definition of the **choose file** command (which displays an open file dialog box) indicates a statement syntax like this:

```
choose file with prompt "Select something:" ~  
of type "TEXT"
```

From a plain language syntax point of view, however, it's the file of type "TEXT" we want, not a prompt of that type. Because this command uses labeled parameters, you could just as easily reorder the statement to read:

```
choose file of type "TEXT" with prompt ~  
"Select something:"
```

Parameters of any type may be required or optional, depending on how the command was defined by its designer (and thus specified in the dictionary). If a parameter is required, then there must be a parameter value of the required class provided. In command descriptions throughout this book, you can easily see whether a command has required or optional parameters (or both). Optional parameters



Chapter 6:
**Issuing
Commands
and Getting
Results**

are surrounded by brackets, as in the following:

```
beep [ numberOfBeeps ]
```

If you don't supply a parameter to the **beep** command, it uses its default value: 1. But you can also supply any other integer, if needed.

Here's an example of a command that has both required and optional parameters (a lot of them):

```
display dialog questionString ~  
  [ default answer answerString ] ~  
  [ buttons buttonList ] ~  
  [ default button buttonNumberOrName ] ~  
  [ with icon iconNumberOrName ]
```

The first parameter, *questionString*, is a required parameter. It also doesn't have a label, so it must go where it does, immediately after the **display dialog** command. The rest of the parameters are optional (in brackets), and they are labeled. Use only the one(s) you need to specify the dialog box. We'll cover what these parameters mean in Chapter 7.

Getting Results

AppleScript commands almost always return a result of some kind, but they don't have to. A lot of the time, your script will ignore the results that come back after issuing a command, but they're there if you need them (e.g., for debugging purposes—see Chapter 13).

After a command executes, the result is stuffed into a special AppleScript variable called **result**. You don't have to initialize this variable: it's there whether you need it or not. Script Editor displays the value of the **result** variable at the end of a script's execution in its Result pane or Result Log window.

Show the Result pane in Script Editor, and type the following statements—one at a time—into the script window. Run each script to view the value that comes back as **result**:

```
3      -- this expression evaluates to itself  
  
3+4      -- evaluates to the sum  
  
"Fred"    -- evaluates to its own string  
  
get 3  
  
set genius to "Einstein"  
      -- the value assigned to genius  
  
current date  
      -- the date and time from the Mac's clock  
  
count of {"larry", "moe", "curly"}  
      -- list count: 3
```



Chapter 6:

Issuing Commands and Getting Results

You can then use the **result** variable in a script to convey the value returned by a command to a more durable variable (the value of **result** will change with the next command). For example:

```
count of {"larry", "moe", "curly"}  
set numOfStooges to result
```

You can now use the **numOfStooges** variable throughout the script, even though the contents of the **result** variable will be changing with each command.

In later chapters, you'll also see how you can use commands as parameters to other commands. This is logical, because a command returns a value—which may then be used as a direct parameter to another command in the same statement. Here's a preview of what such a construction might look like:

```
set numOfStooges to count of ~  
{"larry", "moe", "curly"}
```

The **count** command evaluates to a value (3), which is then assigned to the **numOfStooges** variable.

"Built-in" Commands

Even with a thorough study of Apple's *AppleScript Language Guide* and the numerous documents provided to Macintosh programmers, it isn't always easy to arrive at a definitive list of fundamental commands that are available to you. On the one hand is a small set of commands traditionally referred to as *AppleScript commands*. These commands are, indeed, built into the AppleScript system software and can operate without any particular application context. In practice, however, even a statement invoking one of these simple commands inside Script Editor, at least initially, sets the target to the Script Editor application. Ultimately, however, the command runs from the system.

On the other hand is a set of *application commands* that have been grouped into suites of different names through the years (such as Core and Required). These commands all require an application as their target, even if it means the Script Editor using itself as the default target. More recently the basic set of application commands have been grouped into what is known as the Standard Suite, precisely the heading under which you'll find these commands in any modern application's AppleScript dictionary.

Confusion frequently reigns, however, because you'll find some of the AppleScript commands also listed in the Standard Suite's list of application commands. The reason for this is that a command may behave slightly differently when it uses an application as



Chapter 6:
**Issuing
Commands
and Getting
Results**

its target, rather than the AppleScript system. For example, our old friend the **set** command operates as an AppleScript command when its parameter is, say, a value of the number or string class; but use **set** with a reference to an object in an application's document, and you are using the application's **set** command, and not the AppleScript command. To a newcomer, the distinction may seem frivolous or too subtle. In practice, however, unless the program's designer has added some unique parameters to the application command version of an AppleScript command, you probably won't have to think too much about the distinctions—the command works the same way, just on different kinds of stuff.

Despite the potential confusion and occasional overlap, it's helpful to have an overview of the commands that you will either rely on (as AppleScript commands) or see regularly (as Standard Suite commands) in dictionaries. Table 6.1 shows the combined lists of AppleScript and standard application commands. Commands that I consider to be traditional AppleScript commands are marked with an asterisk (*). At the same time, just because a command is listed as a standard application command doesn't mean that it is supported or operative in all applications. For example, a utility application that operates within its own little window might have no need for an **open** or **print** command.

activate*	exists	print
close	get*	quit
copy*	launch*	reopen*
count*	make	run*
delete	move	save
duplicate	open	set*

Table 6.1. Application and Traditional AppleScript Commands.*

If there is an advantage to being familiar with standard application commands, it is not that you learn one command for all programs, but that when you are scripting a program whose dictionary is new to you, you know what command to look up in the dictionary. For example, rather than wonder if a program requires a **make**, **new**, or **create** command to generate a new document or object, you should look first for a **make** command, because it is the accepted standard command for such an operation. Then use the details in the dictionary to fill in the blanks of parameters for the unique objects and document environment of that application.



Chapter 6:
**Issuing
Commands
and Getting
Results**

Traditional AppleScript Commands

The following commands are hard-wired into the AppleScript system software. Although Apple has typically listed only five AppleScript commands (**copy**, **count**, **get**, **run**, and **set**), a few more are available in all contexts. The total list is small, yet the items listed here account for a majority of the commands you will write in scripts.



activate *referenceToApplication*

Result

None.

Purpose

Makes the referenced application the active application among all open programs on your Macintosh. For crowded screens, it allows the person running the script to see the action in that application.

When To Use

Recorded scripts (i.e., in recordable applications) always record the **activate** command, because you can't help but activate the program to perform recordable actions. But activating every program in a script is not necessary. You may see an application open beneath current programs while a script runs. Unless the comfort of witnessing each action is required (to some, seeing is believing), you can leave out this step on most scripts. If your script

jumps frequently between applications, the extra time required to continually update the display can negatively impact performance.

Some application commands require that the application be the active one to work. For example, if an application lets you cut, copy, or paste with the Clipboard, the laws of Macintosh operation insist that the application be active to access the Clipboard.

Parameters

This command requires only one direct parameter in the form of a reference to an application. Such a reference begins with the word “application” or “app” plus a quoted string containing the name or pathname to a program. Most commonly, the application reference is handled by a preceding **tell** statement, as happens during script recording. In this case, the object of the **activate** command is the default object, the application referred to in the **tell** statement.

Script Editor attempts to resolve the reference to a target application on your own machine when it compiles the script. Therefore, if the application you supply is not among applications registered with the system or if it is on an unmounted volume, Script Editor displays a dialog box listing all applications it knows of and asks you to choose the one you mean (or click the Browse button to find an application not in the list). Some applications may need to be running when you compile a script targeting them.



Chapter 6:
**Issuing
Commands
and Getting
Results**

All Mac OS X applications respond to the **activate** command, including those apps that aren't otherwise scriptable. If the application is on the same machine as the script, the application will open automatically when the script executes the **activate** command directed at that application.

For applications on a different machine on a network, the reference must include the machine name address. See the discussion earlier in this chapter about references to networked applications. The most important point to remember is that an application on another machine must be open for script execution. Also, the other machine must have Remote Apple Events turned on in its Sharing preferences pane. Users of scripts that call remote applications also must gain authenticated password access to the target machine.

Examples

```
activate application "TextEdit"
activate application -
    "SteveHD:Applications:BBEdit" -
    of machine "eppc://192.168.1.122"
```

You Try It

Enter the following script statement into Script Editor, and run it:

```
activate application "TextEdit"
```

Common Errors

Forgetting the word “application” in the application reference parameter; not putting the entire pathname to an application in the parameter; Remote Apple

Events preference is turned off in a remote Mac; remote application isn't running.

Related Items (Chapters)

choose file (7); **choose application** (7); recording applications (3); default object (6); networked applications (6); application object references (8).



copy *expression to variableOrReference*
put *expression into variableOrReference*

Result

Value that was copied (any class).

Purpose

Places a duplicate of a chunk of data into an object, property, or variable.

When To Use

The **copy** command lets you put any kind of information into an object, property, or variable. An alternate syntax (the **put** command with its **into** preposition) may be used freely if that wording makes a more concrete mental image of what's taking place. Everything I say about the **copy** command in this section applies equally to the **put** command.

In many ways, the **copy** command is synonymous with the **set** command, although the latter has additional powers (described later in this chapter).



Chapter 6:
**Issuing
Commands
and Getting
Results**

Because much of what goes on in scripts that link documents or applications together is moving information around, the **copy** command can be used to fetch information from one document (placing the information temporarily into a variable), and then used again to put the information (from the variable) into another document.

Because this command is the same name as the ever-popular **Copy** item of **Edit** menus, you might be tempted to think that the **copy** command works like the Clipboard. But that's not the case at all. It simply takes a snapshot of some value and places the copy into some other container. The original is untouched. The Standard Additions offer support for use of the regular Clipboard (see Chapter 7).

Parameters

Both parameters are required. The first, *expression*, is the item that is to be copied. Any valid expression can be used here, since it is the evaluated value that is placed into the destination object or variable. The expression could also be a reference to an object, in which case AppleScript usually treats the value as a reference to the original object.

When the second parameter is written as a single word (but not an AppleScript reserved word), then AppleScript assumes that word to be the name of a variable. You do not have to declare or initialize a variable prior to using it as a destination for a **copy** command. As a result of the **copy** command, the variable contains a copy of whatever the first

parameter evaluated to. In the simplest example,

```
copy "90210" to ZIPCode
```

the string value “90210” is copied to the **ZIPCode** variable. A more complex expression might run next to insert some additional text before the existing **ZIPCode** variable's value using the AppleScript operator that joins strings together (the **&** symbol):

```
copy "Beverly Hills, CA " & ZIPCode to ZIPCode
```

After the second statement runs, the **ZIPCode** variable contains the string “Beverly Hills, CA 90210”.

Things get a little more complicated when the value you wish to copy to a variable is an object, especially an object from an application. For example, the following script copies *a reference to* the first document of the TextEdit application to a variable named **niceDoc**.

```
tell application "TextEdit"  
    copy document 1 to niceDoc  
end tell
```

When a variable contains a reference to an object, the value is far more powerful than just some value hanging in mid-air. With an object reference in hand, your script can directly influence the original object without having to go through a lengthy restatement of the original object's location in your scripting universe. The following script builds on the previous example and demonstrates how an object reference can come in handy. It begins by obtaining a reference to the **document** object inside TextEdit; next it copies the value of the **text** property of an Excel



Chapter 6:
**Issuing
Commands
and Getting
Results**

range (a single cell's text); then it assigns the new text to the **text** property of the original TextEdit document object:

```
tell application "TextEdit"
    copy document 1 to niceDoc
end tell

tell application "Microsoft Excel"
    copy text of Range "R1C1" of Sheet 1 of
        document 1 to newWords
end tell
copy newWords to text of niceDoc
```

The final statement, which replaces the TextEdit document's original text content with whatever is stored in the variable **newWords**, appears to execute without an explicit target application. In truth, however, the reference stored in **niceDoc** has sufficient information to find its way back to TextEdit (and the Event Log window shows that the final statement does, indeed, operate with TextEdit as the target application).

The above demonstration also shows a middle ground between assigning a simple expression and assigning an object reference to a variable via the **copy** command. Notice that the statement in the Excel block copies a property (**text**) of an object to the variable **newWords**. When you copy a property of an object to a variable, only the *value* of the property comes along for the ride. The source of the original value is not conveyed. Therefore, if you were to change the value of **newWords** after its initial assignment, there would be no corresponding change

to the content of the cell in the Excel spreadsheet.

Occasionally, you will encounter application object references that don't seem to behave as their dictionary entries suggest. One such example is the paragraph object in TextEdit. A paragraph is a clearly listed object class in the TextEdit dictionary, complete with a set of properties. But consider the following script:

```
tell application "TextEdit"
    copy paragraph 1 of document 1 to niceGraph
end tell
```

TextEdit doesn't copy the paragraph object to the **niceGraph** variable, but rather the text of the designated paragraph. Yet, if you simply reference a paragraph, as in:

```
tell application "TextEdit"
    paragraph 1 of document 1
end tell
```

the result contains a reference to that paragraph, complete with a paragraph object's properties. You can't always learn these variations from dictionary listings, but you can explore the differences by experimenting with simple statements inside Script Editor. You'll learn more about these techniques in Chapter 16.

In all of the code snippets above, the destination of most **copy** commands was a variable. But the destination can also be a reference to an object (if you're copying an object) or a property of an object. Thus, one example above copied the text from an



Chapter 6: Issuing Commands and Getting Results

Excel spreadsheet into a variable in one statement, and then later copied the variable value into a TextEdit document. That's one way—and a popular way at that—you'll be using the **copy** command to move data around via AppleScript.

Examples

```
copy "Fred" to oneName -- copying a value
```

```
copy cell "R1C1" to oneCell  
-- copying an object reference
```

```
copy bounds of window 1 to windowCoords  
-- copying an object's property value
```

You Try It

Locate the the *Handbook Scripts/Chapter 06* folder in the companion files for this book. Use TextEdit to open the files named “Sample 6.1.txt” and “Sample 6.2.txt”. Resize and position the windows so you can see them both. Then open a new Script Editor window and position it so you can see all three windows, as shown in Figure 6.1. You'll be entering and running a series of scripts that work directly with the two text documents.

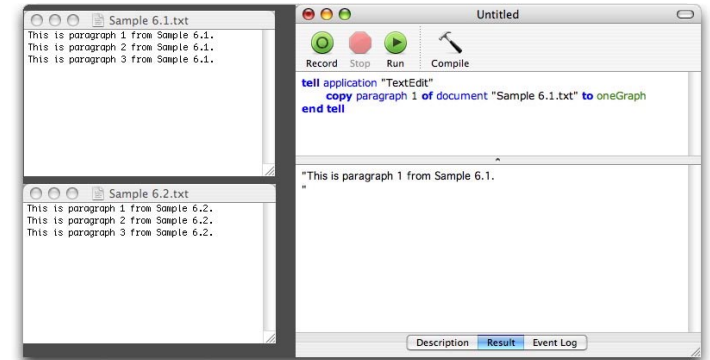


Figure 6.1. Window layout for experiments.

The first example copies the text of a paragraph of one document to a variable named **oneGraph**:

```
tell application "TextEdit"  
    copy paragraph 1 of document "Sample  
        6.1.txt" to oneGraph  
end tell
```

Notice that we're using an alternate way of referring to the document than shown previously: instead of using a numeric index among all open documents (**document 1**), we're referencing the document by its file name. Using the name of an existing object lets us forget about the numerical sequence of windows; we reference a specific window with confidence, which comes in especially handy if the application doesn't always treat document 1 as the foremost document. As shown in the Result pane of Script Editor, the **result** variable after running this one statement contains the value of the data copied to **oneGraph**—the contents of the first paragraph of the document.



Chapter 6:
**Issuing
Commands
and Getting
Results**

The next script replaces paragraph 3 of Sample 6.2 with the contents of **oneGraph**.

```
tell application "TextEdit"
    copy paragraph 1 of document "Sample
6.1.txt" to oneGraph
    copy oneGraph to paragraph 3 of document
"Sample 6.2.txt"
end tell
```

In case you're wondering if you could copy the paragraph from one document directly to a destination of the other document (without the intervening variable), the answer is yes. Choose Revert to Saved from the File menu for the second document and try this shortcut:

```
tell application "TextEdit"
    copy paragraph 1 of document
"Sample 6.1.txt" to paragraph 3 of
document "Sample 6.2.txt"
end tell
```

In our final example, we'll use the **copy** command to fetch two pieces of data from your iTunes library, and once more place the data and some hard-wired text into a TextEdit document. The script uses the string combination operator (&) to assemble a long string of text that gets placed into the TextEdit document.

```
tell application "TextEdit"
    activate
    make new document at beginning
    tell application "iTunes"
        copy album of track 1 of playlist
"Library" to oneAlbum
        copy date added of track 1 of
```

```
playlist "Library" to oneDate
    end tell
    copy "Album:" & oneAlbum & return &
"Added:" & date string of oneDate to text of
document 1
end tell
```

If you have lots of tracks in your iTunes library, try substituting some other numbers for “track 1” in both statements that communicate with iTunes.

Common Errors

Incorrect object reference syntax; an object named in the second parameter cannot accept the type of value contained in or referred to by the first parameter.

Related Items (Chapters)

Get (6); **set** (6); object references (8); variables (9).



```
count [ of ] directParameter -
[ each className ]
```

Result

An integer representing the number of elements in an object. Applications may enhance this command so that it returns multiple values in the form of a list of integers.

Purpose

Provides a count of elements in lists, records, or application objects.



Chapter 6:
**Issuing
Commands
and Getting
Results**

When To Use

Obvious applications for this command are when a script needs to know how many objects there are, such as the number of characters in a word or the number of words in a document. But a script often needs to know how many elements there are in an object so it can set up a repeat loop that performs some action on only as many items as there are elements.

Because the value returned by the **count** command often goes into a variable, it is common to employ **count** statements as parameters of other commands, such as **copy** and **set**, as in:

```
set wordCount to (count words in document 1)
```

To help AppleScript figure out your meaning in such a statement, place the parentheses around the nested statement.

Parameters

The *directParameter* must be one of the following:

- The actual data to be counted
- A variable that evaluates to the data to be counted
- A reference to an application object whose data is to be counted.

Types (a) and (b) can be used within AppleScript by itself, and are generally lists, records, or strings; type (c) must be used inside a **tell** statement directed to an application.

Here is an example of the command working with a list of various pieces of data:

```
count {"Red Sox", 3, "Yankees", 2}  
-- result: 4
```

But we can add a parameter to more narrowly define classes of items we want to count. For example, if we want to know how many teams are in the list, we actually want to know how many strings there are. Therefore we can add the **each className** parameter, as in

```
count {"Red Sox", 3, "Yankees", 2} each string  
-- result: 2
```

Here is an interesting alternative to the last example. Rather than specify “each string,” we use a reference form, **strings in {"Red Sox", 3, "Yankees", 2}**, as the *directParameter*. This reference evaluates to another list, containing just the strings: **{"Red Sox", "Yankees"}**—try it yourself in Script Editor to see the result. This evaluated reference is what gets counted, as follows:

```
count strings in {"Red Sox", 3, "Yankees", 2}  
-- evaluates to:  
count {"Red Sox", "Yankees"}  
-- result: 2
```

Incidentally, if the *directParameter* is a simple string (i.e., not in a list), then the default behavior of the **count** command is to count the characters (including spaces), as in:

```
count "William Shakespeare"  
-- result: 19
```



Chapter 6:
**Issuing
Commands
and Getting
Results**

TextEdit enhances the **count** command by letting us count things such as characters in a word, words in a paragraph, paragraphs in a document, and so on. Other programs may allow you to count cells in a spreadsheet or graphic objects in a document. The following script demonstrates how you might use the **count** command to set the first character of every paragraph of a document to a red color and a large font size. Open the supplied TextEdit file “Sample 6.3.rtf” to see this script in action.

```
tell application "TextEdit"
    -- create shortcut reference
    copy document "Sample 6.3.rtf" to myDoc
    -- perform following command just once
    set graphCount to (count of paragraphs of
text of myDoc)
    -- step through each paragraph
    repeat with i from 1 to graphCount
        set the size of character 1 of
paragraph i of text of myDoc to 22.0
        set the color of character 1 of
paragraph i of text of myDoc to {65535, 0, 0}
    end repeat
end tell
```

We used the **count** command in an in-line statement (inside the parentheses), because we needed to hang onto the value that the **count** command returned. That value became the maximum number for a repeat loop. Assigning the returned value to the **graphCount** variable was a choice of programming style, letting the names of variables clearly show what’s going on in the script.

There is a lesson here in how to examine components

of a statement in case you get errors you don’t understand. Take the in-line command, **count of paragraphs of text of myDoc**. The reference to **paragraphs of text of myDoc** evaluates to a list of paragraph within the document’s text; the **count** command then counts the number of items in the list. You can dissect a reference like this, and experiment in Script Editor to see exactly how a reference evaluates, so you know what the **count** command will be counting. See more about this as a debugging technique in Chapter 13.

Examples

```
count of {1, 3, 5} -- result: 3
```

```
count {1, 1.5, 2} each real -- result: 1
```

```
count words of "Now is the time" -- result: 4
```

You Try It

Enter and run each of the following commands in Script Editor one at a time. Keep the Result pane visible so you can see the result of the count.

```
count of {"Eeny", "Meeny", "Miney", "Moe"}
```

```
set charCount to count -
"Antidisestablishmentarianism"
```

```
count integers in -
{1, 2, 3, "four", "five", "six"}
```

Open the companion document “Sample 6.3.rtf” in TextEdit. Then enter the following script to see how the reference works:



Chapter 6: Issuing Commands and Getting Results

```
tell application "TextEdit"
    get words of paragraph 1 of document 1
end tell
```

The result (`{"This", "is", "paragraph", "1", "of", "Sample", "6.3"}`) shows a list of all the words in the first paragraph. Now, replace the **get** command with the **count** command:

```
tell application "TextEdit"
    get count words of paragraph 1 of document 1
end tell
```

The result (7) is the number of words in the list.

Here's one more example, this time counting number of tracks in your iTunes library:

```
tell application "iTunes"
    set numOfTracks to (count of tracks of
        playlist "Library")
end tell
```

Because there is only one executing statement in this script, the last value assignment (to the variable **numOfTracks**) appears in the Result pane after the script runs. Wherever an application's dictionary indicates that there can be multiples of a particular object class, you can use scripts to count the items.

Common Errors

Incorrect reference form for the *directParameter*; trying to count an application's objects without specifying the application (e.g. in a **tell** statement); using an incorrect containment hierarchy reference form (e.g., trying to count iTunes tracks without observing that tracks are contained within a specific

library playlist).

Related Items (Chapters)

List value class (9); record value class (9).



[**get**] *expressionOrReference*

Result

The value of the expression or reference.

Purpose

Assign a value to the **result** variable.

When To Use

Because this command does little more than copy a value to the **result** variable—which changes value after the very next statement in a script—the **get** command is often used for retrieving throwaway information. By this I mean information whose lifespan doesn't need to extend beyond the next statement. This can come in handy immediately preceding an **if-then** or **repeat** construction, in which the **result** variable is examined for some comparison or repeat value.

```
tell application "TextEdit"
    get font of paragraph 1 of document 1
    if result is not "Times" then
        display dialog "The first paragraph
            isn't in Times."
    end if
end tell
```



Chapter 6:
**Issuing
Commands
and Getting
Results**

In practice, it's more efficient to avoid **get** statements if you can. For example, the above script can be shortened to read:

```
tell application "TextEdit"
  if font of paragraph 1 of document 1 is
    not "Times" then
      display dialog "The first paragraph
        isn't in Times."
    end if
end tell
```

It is also a waste of script to get a value, and then copy the result to a variable: Just copy the value to the variable directly using the **copy** or **set** commands. But sometimes, a reference or expression may not evaluate for use in one side of an **if-then** statement. If that's the case, then use the two-step method with the **get** command.

The **get** command—especially the version that doesn't use the “get” word for the command—does come in handy, however, for debugging purposes. As you begin to write a script, you can test it up to a certain point, and use a **get** statement as the last line of the script to check the value of an expression or reference. See Chapter 13 for debugging guides.

Parameters

Perhaps the most striking part about this command is that the command, itself, is optional. We've actually seen this action in previous discussions. If you type any value (a number or quoted string) into a script, and run it, the result contains that value. Therefore, these two statements are identical:

```
"Jane"
get "Jane"
```

Both statements place the string, “Jane”, into the **result** variable (and display the value in the Result pane). The same goes for expressions that evaluate to other values:

```
3 + 4 -- result: 7
"John" & " Doe" -- result: "John Doe"
paragraph 1 of document 1
  -- result: the actual contents of
  -- paragraph as a string
```

For your own use, the **get** word is usually superfluous. But if you intend to use one of these statements in a script that others will read, it's best to use the **get** word to make the script easier to read and follow.

The main parameter of this command is an expression or a reference to an object (including an object's properties). It is always the value of that expression or reference that goes into result. Therefore, in the statement:

```
get 3+4
```

the result contains the evaluated value, 7, not the characters of the arithmetic problem. Similarly, the statement:

```
get words 1 thru 5 of paragraph 1 of -
  document 1
```

as defined by TextEdit, places the actual words as a list into **result**, not the wording of the reference (see Figure 6.2).



Chapter 6:
**Issuing
Commands
and Getting
Results**

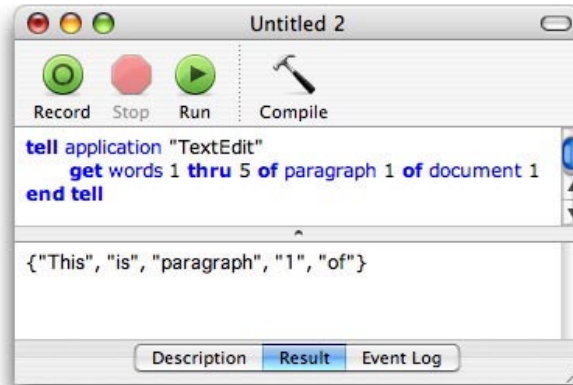


Figure 6.2. Result values appear in black in the Result pane

Consistent with an earlier discussion of the **copy** command, when you use the **get** command with an object reference, the value placed into the **result** variable is a live reference to the object. Therefore, although it's not necessarily good form, it is possible to read or write a property of that object by referencing the **result** (but this works only one time, because **result** will acquire a different value upon executing the next statement). You can usually spot values that are object references in the Result pane because you'll see a long-winded reference phrase displayed in purple font (see Figure 6.3).

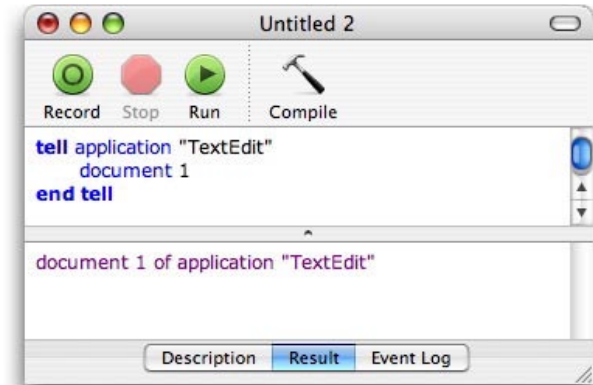


Figure 6.3. Result object references appear in purple in the Result pane

Examples

```
10 -- result: 10

10 + 10 -- result: 100

get 10 + x
-- result: nothing, because it's
-- an invalid expression

get name of window 1
-- result: "untitled 2"

words 1 thru 3 of document "Preamble"
-- result: {"We", "the", "people"}
```

You Try It

Enter several one-line arithmetic problems into Script Editor, and run each one as a script. Look for the answers in the Result pane.



Chapter 6:
**Issuing
Commands
and Getting
Results**

Next, create a compound **tell** statement directed at TextEdit. Open any document or enter a couple lines of text yourself. Try each of the following one-line **get** statements and examine the results. Some of these get contents of the document, while others get properties of objects:

```
get text of document 1

get paragraph 1 of document 1

get words of paragraph 1 of document 1
    -- words in a list

bounds of window 1

font of paragraph 1 of document 1

size of word 1 of document 1 -- font size

count of characters of word 1 of document 1
```

Common Errors

Incomplete application object references; trying to force an evaluation of mixed value classes (e.g., integer + string); confusion over an application's dictionary, mistakenly trying to get a property that doesn't exist for a particular object.

Related Items

Copy (6); **set** (6); object references (8); **if-then** constructions (10); **repeat** loops (10); debugging scripts (13).



launch [*variableOrReference*]

Result

None.

When To Use

The **launch** command is a kinder, gentler version of the **run** command. The **run** command not only launches an application or script object, it also sends a run Apple event to the launched object. It is this run Apple event, for instance, that makes TextEdit go through its initialization paces to produce a new untitled window. It is also the event that tells a script object to do its work (if there is a matching run handler in the script object to obey).

Should you try launching an already-open application, nothing happens. You can also use this command to launch non-scriptable applications.

Parameters

The **launch** command's parameter is the same as the **run** command's parameter, but is limited to applications only. No parameter is used if the command is issued inside a **tell** block aimed at the application to be launched.

Examples

```
launch application "iTunes"
```

You Try It

Close TextEdit if it's running. Then enter and run the following script:



Chapter 6:
**Issuing
Commands
and Getting
Results**

```
launch application "TextEdit"
```

Leave TextEdit running, and position Script Editor windows such that you can see part of the Text Editor's window. Then enter the following command, and run it twice:

```
launch application "TextEdit"
```

Notice how the **launch** command does not cause the program's initialization process to create a new window, as the **run** command does.

Common Errors

Omitting “application” from parameter; trying to use an alias class value as a parameter, when a string for the pathname is required.

Related Items (Chapters)

Run (6); **choose application** (6); value classes (9); script objects (15).



```
run [variableOrReference]
```

Result

For an application reference, no result; for a script object reference, the value (if any) returned by the script object.

When To Use

It's not often that you need to use the **run** command to start up an application because any statement directed at a scriptable application (via a **tell**

statement) automatically launches the application anyway. If the application is already running, this command does not bring it to the front of other open applications.

Issuing the command to an application that is already open, however, may be useful if the program performs some initializations in response to the command. For example, TextEdit always opens a new untitled window whenever it runs—either from a Finder launch or as a recipient of the **run** command. This can give you a known state to start a series of statements. But not all programs react this way. Many applications bypass their initializations if you use the **launch** command—test your favorite applications for their behavior.

You can also use the **run** command to invoke a script object. See Chapter 15 for more details. A scripting addition with a similar sounding command, **run script**, is described in Chapter 7.



Chapter 6:
**Issuing
Commands
and Getting
Results**

A companion command to **run**, called **reopen**, is supported by most applications (including non-scriptable apps), but the precise behavior upon receiving this command depends on how the application's programmer implemented its response to the internal Apple event associated with this command. Typically, applications respond to the **reopen** command the same way they respond to the **run** command. This means, for example, that TextEdit opens a blank window if no other document window is open. The **reopen** command will also start up an application that is not running—just as the **run** command does. If you confirm by experimentation that one of your target applications responds the same way to both commands, you can choose the command that makes the most sense in describing the operation within the context of your script.

Parameters

The parameter for the **run** command may be a variable or reference that evaluate to the name of the application or a script object name. If you do not supply a complete pathname for the application, AppleScript relies on the system's internal list of available applications. If multiple copies exist across multiple mounted volumes, the default behavior is to run the copy on the startup disk. You may also provide a complete pathname to guarantee that a particular copy of an application will run. Be sure to include the word "application" as part of an application reference, as in:

```
run application "TextEdit" -- for application
```

The parameter is not needed if you include the **run** command inside a **tell** block aimed at the application you wish to run:

```
tell application "Microsoft Excel"  
    run  
end tell
```

You may also place the job of selecting an application to run into the hands of the user by applying the **choose application** scripting addition command that produces a list of applications known to the system. This command returns a valid reference to an application, complete with the "application" label:

```
run (choose application)
```

Examples

```
run application ~  
    "BackupHD:Applications:SuperDuperEdit"
```

```
run application myFile -- variable holding  
    -- application name as a string
```

You Try It

Quit TextEdit if it's running. Then enter and run the following script in Script Editor:

```
run application "TextEdit"
```

Leave TextEdit running, and position Script Editor windows such that you can see part of the Text Editor's window. Then enter the following command in Script Editor, and run it twice:

```
run application "TextEdit"
```



Chapter 6:
**Issuing
Commands
and Getting
Results**

Notice how the **run** command provokes the program's initialization process to create a new window each time. If you try the **launch** command, no new additional window is created.

Common Errors

Omitting “application” from parameter; trying to use an alias class value as a parameter, when a string for the pathname is required.

Related Items (Chapters)

Launch (6); **choose application** (6); **run script** (7); value classes (9); script objects (15).



set *variableRefOrPattern to expression*

Result

The value assigned to the variable or reference.

When To Use

The **set** command is among the most used commands in AppleScript scripts. Its job is to assign a value to one or more variables or to an application object (including properties of objects). You may freely mix **copy** and **set** statements in the same script, because in common operations they perform the same basic value assignment tasks.

The command also provides a shortcut way to assign values to multiple variables. In essence, you define a pattern for variables that “receive” their values from an expression that evaluates to multiple

values in the same pattern. For example, the **color** property of various document-related objects in TextEdit is an AppleScript list of three numeric values corresponding to the red, green, and blue components of the designated color (e.g., {**48204**, **13792**, **65535**}). You can use a variable pattern with the **set** command to assign all three values to individual variables in one statement. Simply set up the pattern as a series of variable names inside square brackets:

```
tell application "TextEdit"
    set [r, g, b] to color of paragraph 3 of
        document 1
end tell
```

After the **set** command runs, you can read the values individually in subsequent statements via the **r**, **g**, or **b** variables.



Chapter 6:

Issuing Commands and Getting Results

AppleScript provides an alternate syntax for the **set** command: the **returning** command. To understand the relation between the two commands, examine the syntax definitions for both:

```
set variableRefOrPattern to expression  
expression returning ~  
    variableRefOrPattern
```

Therefore, the following two statements are completely synonymous:

```
set myName to "Danny"  
"Danny" returning myName
```

After either statement, the **myName** variable may be used to access the value stored in it. If you prefer the English syntax of the **returning** command over the **set** command, use it.

While the **set** command does everything that the **copy** command does (albeit with parameters in a different syntactic order), it does have an extra, non-obvious power when the item being set is a list, record, or script object: data sharing.

A demonstration of data sharing reveals more about it than a long explanation. Consider this series of statements:

```
set stoogeList to {"Larry", "Moe", "Curly"}  
set comedyGroup to stoogeList  
set item 3 of stoogeList to "Shemp"
```

We first assign a list of strings to a variable named **stoogeList**. Then we assign that variable to a second variable, **comedyGroup**. Finally, we

change the third value in the original variable, replacing Curly with Shemp. What's the value of **comedyGroup**?

Without data sharing, the value of **comedyGroup** would remain what it was when it was assigned. But because the **set** command automatically performs data sharing with variables that evaluate to lists, records, and script objects, the value is updated in **comedyGroup**, as well. Therefore, if you run these four script lines:

```
set stoogeList to {"Larry", "Moe", "Curly"}  
set comedyGroup to stoogeList  
set item 3 of stoogeList to "Shemp"  
comedyGroup -- examine value in result window
```

you'll see that **comedyGroup** is updated to be {"Larry", "Moe", "Shemp"} —the same as **stoogeList**. As long as the initial **set** statement for the second variable doesn't try to modify the value of the second value (e.g., concatenating an additional item in the process), data sharing will survive any subsequent modification to the second variable:

```
set stoogeList to {"Larry", "Moe", "Curly"}  
set comedyGroup to stoogeList  
set item 1 of comedyGroup to "Larry Fine"  
set item 2 of comedyGroup to "Moe Howard"  
set item 3 of stoogeList to "Shemp Howard"  
comedyGroup  
-- result: {"Larry Fine", "Moe Howard",  
            "Shemp Howard"}
```

Moreover, data sharing is bi-directional. Any change you make to one variable appears in the other.



Chapter 6:
**Issuing
Commands
and Getting
Results**

After the above script, the **stoogeList** variable evaluates to the same as **comedyGroup**.

Data sharing is more memory-efficient, especially for large data collections. Instead of creating multiple copies of the same data for use in various parts of a script, data sharing lets one copy be shared among multiple variables. Data sharing can extend anywhere throughout a single script (and all activity must be on the same Mac). For example:

```
set stoogeList to {"Larry", "Moe", "Curly"}
set comedyGroup to stoogeList
shempify(comedyGroup)
stoogeList
-- result: {"Larry", "Moe", "Shemp"}
on shempify(comedyGroup)
    set item 3 of comedyGroup to "Shemp"
end shempify
```

In this version of our earlier script, we parcel out the job of replacing Curly with Shemp to a subroutine. We create a shared copy (**comedyGroup**), which we pass as a parameter to the subroutine. The subroutine merely modifies one item of the list. But in so doing, the other shared list (**stoogeList**) is updated without having to return the “shempified” value.

Parameters

Both parameters of the **set** command are required. The second, *expression*, is the item that is to be evaluated and assigned to the variable, pattern, or object named by the first parameter. Any valid expression can be used for the second parameter,

and it is the evaluated value that is placed into the variable or object. The expression could also be a reference to an object, in which case the “live” reference is assigned to the variable or object.

When the first parameter is written as a single word (i.e., not an AppleScript reserved word), then AppleScript assumes that word to be the name of a variable. The variable does not need to be declared or initialized before being assigned a value with the **set** command. As a result of the **set** command, the variable contains whatever value the second parameter evaluated to. In the simplest example,

```
set ZIPCode to 90210
```

the **ZIPCode** variable is assigned the value, 90210. But the second parameter could also be a reference to a property of an object, as in

```
tell application "iTunes"
    set oneAlbum to album of track 100 of
        playlist "Library"
end tell
```

or to a reference to an object:

```
tell application "iTunes"
    set trackObj to track 100 of playlist
        "Library"
end tell
```

The first parameter may also be a reference to an object, provided the expression being assigned to it is of the same class. With the following script, we replace document 2’s first paragraph with the first paragraph from document 1:



Chapter 6: Issuing Commands and Getting Results

```
tell application "TextEdit"
    set niceGraph to paragraph 1 of document 1
    -- variable niceGraph
    set paragraph 1 of document 2 to niceGraph
    -- replaces paragraph
end tell
```

Examples

```
set oneName to "Fred"

set firstFormula to formula of cell "R1C1"

set windowCoords to bounds of window 1
    -- copying an object's property

set locked of file "myFile" to true
```

You Try It

On the companion disk, in the *Chapter 06* folder, are two TextEdit files, named *Sample 6.1.txt* and *Sample 6.2.txt*. Open both files. Position them and a Script Editor window as shown in Figure 6.1. Then enter and run each script separately below:

```
tell application "TextEdit"
    set oneGraph to paragraph 1 of document
    "Sample 6.1.txt"
end tell
```

As shown in the Result pane of Script Editor, the **result** variable after running this one statement contains the value of the data copied to **oneGraph**—the contents of the first paragraph of the document.

The next script replaces paragraph 3 of Sample 6.2 with the contents of **oneGraph**.

```
tell application "TextEdit"
    set oneGraph to paragraph 1 of document
    "Sample 6.1.txt"
    set paragraph 3 of document "Sample
    6.2.txt" to oneGraph
end tell
```

In our final example, we'll use the **set** command and a pattern to put values into three variables at once. To start, open the file "Sample 6.1.txt" in TextEdit. Then change the font color of the document following these steps:

- 1) Make sure the window "Sample 6.1.txt" is the front window.
- 2) From the **Format** menu, choose **Font>Show Colors** to display the color chooser palette.
- 3) Click anywhere in the color wheel and vertical slider to choose a color for the text.

Now enter and run the following script in Script Editor. It fetches the **color** property of the document and assigns each of the color values to different variables. The final statement displays an alert dialog that reveals the color values you chose:

```
tell application "TextEdit"
    set [redVal, grnVal, bluVal] to color of
    document "Sample 6.1.txt"
end tell
set msg to "The red value is:" & redVal &
return & "The green value is:" & grnVal &
return & "The blue value is:" & bluVal
display dialog msg
```

The text for the dialog box is assembled using hard-wired text and values supplied by the individual



Chapter 6:

Issuing Commands and Getting Results

variables. Each color's intensity is measured by a number in the range between 0 and 65,535.

Common Errors

Incorrect object reference syntax; an object named in the first parameter cannot accept the type of value contained in or referred to by the second parameter.

Related Items (Chapters)

Copy (6); object references (8); variables (9).

Onward to Scripting Additions

In the next chapter, we examine commands that are external to AppleScript, but are supplied by Apple as scripting additions. We'll also look more deeply at how application dictionaries describe an application's commands.



Chapter 7 Scripting Addition Commands and Dictionaries

AppleScript's internal vocabulary is intentionally small. It is meant to supply only the basic structure and wording primarily for shuffling data from point A to point B, leaving the more powerful work to scripting support inside applications. But often we scripters need more help with file, data, and interface issues on a universal scale—not just when we're inside applications that provide those powers.

To fill the gap, AppleScript provides a mechanism whereby the commands available to all scripts can be extended to include some of these helpful extensions. These extensions are called *scripting additions*. You use these commands in AppleScript statements just like you do the built-in commands. As described in the section *Who Gets What?* in Chapter 6, AppleScript scans through a sequence of locations for instructions on how to process a command. One of those locations is the *Library/ScriptingAdditions* folder (in the System or User directories), where scripting additions files reside.

Apple supplies a number of useful scripting additions with Mac OS X in a group called Standard Additions.

Some of these additions date way back to the time before the Finder was scriptable, yet they continue to be supported for the sake of compatibility—even though using the Finder's scriptable powers is the more desirable approach today.

To help you get an overview of what scripting additions you'll have at your scripting fingertips, I've divided Apple's standard scripting additions into categories by type of operation, shown in Table 7.1. The Standard Additions dictionary has a slightly different structure, but the command coverage is the same. We'll discuss these commands within the groupings to help you learn them initially, and then to make them and related commands easy to find later as a reference.



Chapter 7:

Scripting Addition Commands and Dictionaries

System Commands

beep
choose application
choose color
choose file
choose file name
choose folder
choose URL
delay
do shell script
get volume settings
set volume
system attribute

Numeric Commands

random number
round

File Commands

open for access
close access
read
write
get eof
set eof

Clipboard Commands

clipboard info
set the clipboard to
the clipboard

Finder Commands

info for
list disks
list folder
mount volume
path to

Script Commands

load script
run script
scripting components
store script

Debug Commands

log
start log
stop log

Folder Action Commands

adding folder items to
closing folder window for
moving folder window for
opening folder
removing folder items from

String Commands

ASCII character
ASCII number
offset
summarize

User Interface Commands

choose from list
display dialog
say

Date/Time Commands

current date
time to GMT

Internet Commands

handle CGI request
open location

Table 7.1. Apple's Standard Scripting Additions



Chapter 7:
**Scripting
 Addition
 Commands and
 Dictionaries**

System Commands



beep [*numberOfBeeps*]

Result

None.

When to Use

Beep sounds are intended to alert the user when some user interaction is required, often before displaying a dialog box that signals an error or other problem. The actual sound that plays is the alert sound chosen in the Macintosh's Sound preferences pane. I find multiple beeps that serve as alert sounds to be very annoying.

Scripters can also use beeps in the debugging process to know whether certain parts of a script (e.g., a branch of an **if-then** construction) execute as expected. By placing **beep** commands with varying number of beeps at various places within a script, you can get an aural clue about where inside a script the current execution is taking place. Be sure to remove debugging **beep** statements before releasing the script to other users.

Parameters

If you omit the parameter for this command, the beep sound occurs once per command. Any parameter you supply, however, must evaluate to an integer value.

Examples

```
beep
```

```
beep 4
```

You Try It

Enter and run the following script in Script Editor. It displays a dialog that asks the user to provide an answer. If the user enters an incorrect answer, the script beeps once before displaying a smart aleck alert.

```
display dialog "How much is 2 + 2" default
  answer ""
if {"4", "four"} contains text returned of
  result then
  display dialog "Awright!"
else
  beep -- alert that something is wrong
  display dialog "Since when?"
end if
```

Common Errors

Sound is turned off; variable value used as parameter does not evaluate to an integer.

Related Items (Chapters)

Display dialog (7)



choose application ~

```
[ with title windowTitle ] ~
```

```
[ with prompt promptString ] ~
```

```
[ as appOrAliasClass ] ~
```

```
[ multiple selections allowed Boolean ]
```



Chapter 7: Scripting Addition Commands and Dictionaries

Result

Selecting an application from the Choose Application dialog and clicking the OK button returns a reference to the application (in the form **application** *applicationName*) selected by the user. If multiple selections are allowed, the result is in the form of an AppleScript list, even if the user selects only one application. Clicking the Cancel button returns error number -128 and error message “User canceled” in a **try-onerror** construction (see Chapter 12).

When to Use

The Choose Application dialog box (Figure 7.1) displays a list of all applications on all mounted volumes (including volumes shared from other Macs).

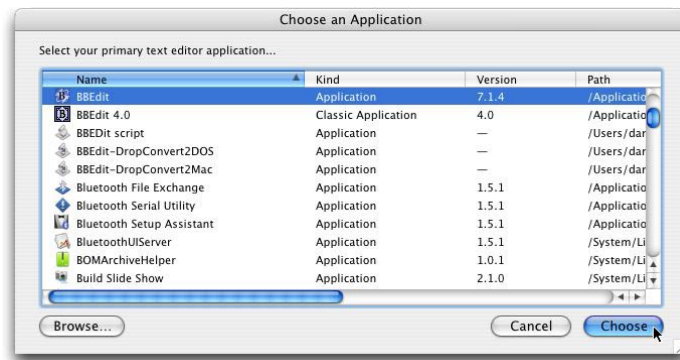


Figure 7.1. The Choose Application dialog box.

Although the command by default returns a

reference to the application, your script can obtain the pathname (as shown in the righthand column of the dialog) by requesting the result in the form of an AppleScript alias (see Chapter 8):

```
set chosenApp to choose application as alias
```

Your choice will depend on the type of data your subsequent script commands that use the application selected by the user.

As with most commands that display a dialog window, if the user clicks the Cancel button, the command throws an error (with an error message text of “User Canceled”), and script execution halts. If your script needs to recover gracefully from the user clicking Cancel, then wrap the **choose application** command statement inside a **try-on error** construction (Chapter 12).

Parameters

All four labeled parameters are optional. The **with title** parameter requires a quoted string value and controls the text that appears in the titlebar of the dialog window (the default is “Choose Application”). To provide more detailed instructions to the user, you can supply a string to the **with prompt** parameter. The text for this parameter appears in the dialog window just above the list of applications (the default is “Select an application:”).

Use the **as** parameter to control the class type of the returned value. The default is an application reference (parameter value of **application**). The



Chapter 7: Scripting Addition Commands and Dictionaries

other accepted parameter value is **alias** to obtain the pathname to the selected application file.

If you want to allow users to select more than one application (by Command- or Shift-clicking on application names), specify the **true** Boolean value for the **multiple selections allowed** parameter. When this parameter is set to **true** (the default is **false**), the command returns an AppleScript list, whose items you must access through list item reference syntax (see Chapter 9). Even a single selected application is returned as a one-item list.

Examples

```
choose application
```

```
set appList to (choose application with prompt  
    "Select one or more applications:" multiple  
    selections allowed true)
```

```
set targetApp to (choose application with  
    title "Application Chooser" with prompt "Pick  
    a program:" as alias)
```

You Try It

With the Result pane showing in the Script Editor window, enter and run the following command. Select any application from the list presented to you, and observe the class of the value returned from the command.

```
choose application with title "Apps" ~  
    with prompt "Select an application:"
```

Next, enter and run the following command. Notice

that the returned value is an alias class and that the values are in the AppleScript list format. Run the script a couple of times, selecting just one item and then Command-clicking to select multiple applications.

```
choose application as alias ~  
    multiple selections allowed true
```

Common Errors

Failing to specify the correct returned value class (via the **as** parameter) needed by a subsequent command parameter.

Related Items (Chapters):

Choose file (7).



```
choose color [ default color ~  
    RGBColorSpec ]
```

Result

A color specification, consisting of a three-item list, such as **{26143, 63756, 21232}**. The three numbers correspond to the intensity of red, green, and blue components (respectively) of the selected color. Values range between 0 and 65535, inclusive.

When To Use

Whenever a script needs to ask the user to select a color for a subsequent operation, use the **choose color** command. The command displays a color picker window, shown in Figure 7.2.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**



Figure 7.2. The Mac OS X color picker

As with most commands that display a dialog window, if the user clicks the Cancel button, the command throws an error (with an error message text of “User Canceled”), and script execution halts. If your script needs to recover gracefully from the user clicking Cancel, then wrap the **choose color** command statement inside a **try-on error** construction (Chapter 12).

Parameters

The single optional parameter of the **choose color** command is labeled **default color**, and its value is in the same three-item list format as the value returned by the command when a user clicks OK. If you omit the parameter, the default value

applied to the command is {0, 0, 0}, which is the same as complete black. This may confuse users who don’t frequently work with the color picker, because they may be unaware how to increase the brightness slider to see more colors. Unless of you have a reason to supply a specific color as the default, you should supply the maximum values for all three items in the list.

Examples

```
choose color
```

```
choose color default color ↵  
{65535, 65535, 65535}
```

You Try It

Open a new document window in TextEdit, and type a few words into the document. Then enter and run the following script in Script Editor, observing the TextEdit document window to see the results when you select a color.

```
tell application "TextEdit"  
  activate  
  set color of text of document 1 to choose  
  color default color {65535, 65535, 65535}  
end tell
```

In this case, we’re using the result of the **choose color** scripting addition command to act as the value being assigned to the **color** property of the text of document 1 of TextEdit. If there is no need to preserve the selected color for an additional operation later in the script, your script is more compact when command results are used directly to



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

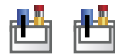
fill other values (including parameter values of other commands).

Common Errors

Presenting a user with a black color wheel, confusing them as to the purpose of the dialog box.

Related Items (Chapters)

None.



```
choose file [ with prompt ~
    promptString ] ~
    [ of type fileTypeList ] ~
    [ default location fileOrFolderAlias ] ~
    [ invisibles Boolean ] ~
    [ multiple selections allowed Boolean ]
```

Result

Clicking the Choose button (see Figure 7.3) returns a reference to the file chosen by user in the form **alias** *pathname* (or a list of pathnames if multiple selections are allowed). Clicking the Cancel button, throws an error of number -128 and error message of “User canceled”.

When to Use

The **choose file** command provides a standard open file dialog box for users to select a file from any mounted volume on their Desktop. This is a common user interface device, so users of your script should know how to use it to locate a file. To ask users to

select a folder, see the **choose folder** command, below

The command does not open any files, but does return the full pathname of the selected file in a quoted string, preceded by the word **alias**, making the value an alias class (Chapter 8), as in:

```
alias "HD:Documents:Presentations:Overhead
Template"
```

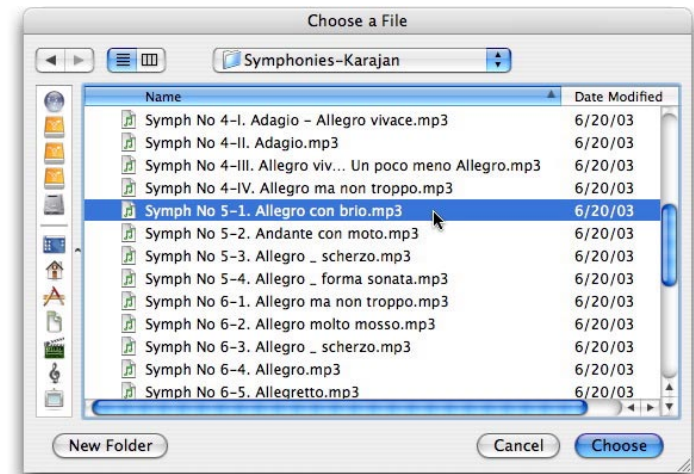


Figure 7.3. The *choose file* dialog box.

But if you want to perform any operations just on the name of the file, the returned value must be coerced to a string value. You can combine the **choose file** command and value class coercion in a single statement, such as:

```
set myFile to (choose file with prompt "Select
a file:") as string
```



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

The result from this command would look like this:

```
"HD:Documents:Presentations:Overhead Template"
```

The alias value class, however, offers an avenue to using a powerful scripting addition command called **info for** (described in detail later in this chapter). The properties of this object include data such as the file name (as a string, without the pathname), file name extension, file type, and many more. For example, if you want to extract only the file's name from the item chosen by the user, here's the quick syntax:

```
set oneFileName to name of ¬  
    (info for (choose file))
```

If the user selects a Finder alias to a file, **choose file** returns a reference to the alias file, not the original file. The **info for** command, however, can't help you track down the original file. To obtain a reference to the original file, use the scriptable Finder and a property of its **alias file** class (described in the Finder's AppleScript dictionary):

```
tell application "Finder"  
    set origFile to original item of (choose  
        file)  
end tell
```

Because this command provides an interface element that the user controls, your script should anticipate all actions the user could make in a situation like this: selecting the desired file; selecting the wrong file; clicking the Cancel button. See Chapter 13 for more about error trapping in scripts.

Parameters

All parameters are optional for **choose file**. Because they are labeled parameters, you can apply them in any order that you like.

The **with prompt** parameter makes room for one line of prompt text in the file dialog just below the titlebar. The length of the string that will show up depends on the width of various characters in the font and the width of the dialog as adjusted by the user. A long string in a narrow window will wordwrap any runover text, but the second line won't be visible. The value assigned to the parameter must evaluate to a string class variable.

You may restrict the selectable files to one or more specific file types via the **of type** parameter. A file type is a four-character identifier that helps the Finder differentiate between different kinds of files. Because the **choose file** command without any restrictions displays every file (including invisible files), things can get confusing for users. You can instruct the command to filter the views so that folders and only files of the desired type(s) are clickable. Table 7.2 demonstrates popular file types.



Chapter 7: Scripting Addition Commands and Dictionaries

File Type	Description
TEXT	Plain text file
APPL	Any application
PDF	Acrobat file (the fourth character of the type is a space)
M4A	MPEG4 audio file (fourth character space)
JPEG	JPEG image file
W8BN	Microsoft Word X document
XLS8	Microsoft Excel X spreadsheet
8BPS	Photoshop 7 file

Table 7.2. A sampling of file types

Notice that each file type is four characters (no more, no less). Also bear in mind that file types are case sensitive, so observe file type naming very carefully. If, while designing a script, you're unsure about the file type of a document, use this one line script and Script Editor's Result pane to help you:

```
file type of (info for (choose file))
```

The **of type** parameter for the **choose file** command requires a string class value for a single type or a list class value, for one or more types. For example **"PDF "**, **{"PDF "}**, or **{"PDF ", "JPEG"}**. By placing multiple file types in the list, your script can allow more than one type of file to be active at a time.

You can pre-set the folder whose contents appear in the file dialog by assigning to the **default location** parameter an alias reference to a folder or file (within the desired folder). If the script application you're writing will be installed on other computers, then writing the script with a pathname based on your own hard disk's name and folder

organization is not likely to succeed elsewhere. But with the help of another standard scripting addition (the **path to** command), you can set the default to one of several well-known directories on any Macintosh. For example, the following statement prompts the user to locate a file starting at the Documents directory within the user's domain:

```
set oneFile to choose file default location -  
    (path to documents folder from user domain)
```

The downside to specifying this parameter is that subsequent displays of the file dialog will continue to start from the specified directory, even if the user had navigated away from it to find a file earlier. When you don't specify this parameter, a subsequent display of the file dialog remembers the last directory selected by the user.

By default, the **choose file** command lists even normally invisible files in the file dialog. If you set the **invisibles** parameter to **false**, only those files that you normally see in Finder views are shown in the lists of files.

You may also allow users to select multiple files if you set the **multiple selections allowed** parameter to **true** (the default is **false**). The value returned from **choose file** when this parameter is set to **true** arrives as a list value class, even if a single item is selected. Therefore, your script will need to reference returned values as items within the list.



Chapter 7:

Scripting Addition Commands and Dictionaries

Examples

```
choose file
```

```
set myFile to (choose file with prompt "Pick a  
file, any file:")
```

```
choose file with prompt "Select an Excel  
spreadsheet:" of type {"XLS8"}
```

```
choose file with prompt "Select one or more  
documents:" default location (path to  
desktop) invisibles false with multiple  
selections allowed
```

You Try It

Enter and run each of the script statements below, selecting a file in the dialog, and watching Script Editor's Result pane to see the values returned by the command:

```
choose file
```

```
choose file with prompt "Select a file of your  
choice:" default location (path to frontmost  
application)
```

```
choose file of type {"PDF "} invisibles false
```

```
choose file with prompt "Select some iTunes  
files:" of type {"M4A ", "M4P "} with multiple  
selections allowed
```

```
get (choose file of type {"TEXT"}) as string
```

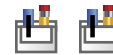
Common Errors

Expecting just the file's name or path name being returned as a string; not observing proper

capitalization of file types; not observing a blank space in a file type name; forgetting to place file types in a list.

Related Items (Chapters)

Choose file name (7); **choose application** (7); **choose folder** (7); alias and text value classes (9); **info for** (7); **path to** (7).



```
choose file name [ with prompt ~  
promptString ] ~  
[ default name fileName ] ~  
[ default location fileOrFolderAlias ]
```

Result

A file reference to the newly named file or to the existing file bearing the same name as the one entered by the user. No file is created by the successful execution of this command.

When To Use

Although perhaps not intuitively named to contrast with the **choose file** command, the **choose file name** command displays a Save file dialog box to the user (Figure 7-4). This dialog, identical to the kind presented by applications when the user chooses Save As from the File menu, prompts the user to navigate to a desired folder location and enter the name of a file to be created in that directory. If the name supplied by the user is already used by an existing file, the user receives a warning that advises



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

of the impending collision and allows the user to choose to replace the existing file.

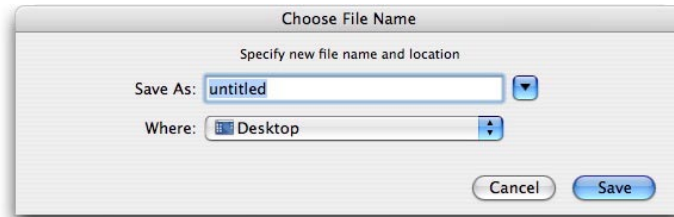


Figure 7.4. The choose file name dialog box

This command does not open, write, or overwrite any files. It is merely a way to let the user supply the path and name for a file that your script will write to the disk in subsequent statements. That's why the command is named what it is, rather than something like "save file"—no file saving occurs as a result of this command. Use the command in advance of your script writing to a potentially new file with the permission of the user.

That the command returns a value of the file class (rather than the alias class) permits the value to point to a file that does not yet exist on the disk (an alias reference must evaluate to an existing file). At the same time, if the user enters the name of an existing file and agrees to replace the old file, your script can use the returned file reference and other scripting additions (such as **info for**) to examine properties of the file about to be overwritten (perhaps to see

how recently the older file was last modified).

Parameters

All three parameters to the **choose file name** command are optional. The **with prompt** parameter makes room for one line of prompt text in the file dialog just below the titlebar. The length of the string that will show up depends on the width of various characters in the font and the width of the dialog as adjusted by the user. A long string in a narrow window will wordwrap any runover text, but the second line won't be visible. The value assigned to the parameter must evaluate to a string class variable.

If you don't specify a value for the **default name** parameter, Mac OS X supplies a default value of "untitled." But you can supply any string value you want to the parameter to pre-set the new file name.

See the discussion of the **default location** parameter associated with the **choose file** command. Its purpose and usage are the same for the **choose file name** command, allowing your script to pre-set an initial directory for the file.

Examples

```
choose file name with prompt "Save file as..."
```

```
choose file name default location (path to  
frontmost application) default name date  
string of (current date)
```

You Try It

Enter and run each of the script statements below in



Chapter 7:

Scripting Addition Commands and Dictionaries

Script Editor. Either enter a new file name or use the default value. Observe the returned value in Script Editor's Result pane:

```
choose file name
```

```
choose file name with prompt "Enter a name for  
the new file:" default location (path to  
desktop)
```

```
choose file name default name "Diary-" & date  
string of (current date)
```

For the next example, when presented with the Save dialog, expand the view to see existing directory entries. Files will be disabled, but you can click on one to pick up that file's name in the file name entry field. When prompted to replace the file, click the Replace button. Notice that the **info for** command returns information about the existing file:

```
info for (choose file name)
```

Common Errors

Forgetting that the returned value is a file class rather than alias class for use in subsequent statements.

Related Items (Chapters)

Choose file (7); alias and text value classes (9);
info for (7); **path to** (7).



```
choose folder [ with prompt ~  
promptString ] ~  
[ default location fileOrFolderAlias ] ~
```

```
[ invisibles Boolean ] ~  
[ multiple selections allowed Boolean ]
```

Result

An alias reference to the selected folder.

When to Use

This scripting addition is a sister command to the **choose file** command. Issuing the command creates a file dialog window that looks just like the **choose file** command's dialog (Figure 7.3) except that only folders are clickable. Use this command when you need a user to select a folder, such as for specifying a destination folder where files are to be written by subsequent portions of the current script or script application. Or your script may prompt a user to select a folder containing files to be processed by the rest of the script.

Like other file dialogs, this one's Cancel button throws a -128 error, halting script execution (see Chapter 13). Therefore you should place statements using this command inside a **try-on error** construction if you want to gracefully bow out of a script when the user cancels the dialog.

The returned value of the choose folder command is an alias reference form, as in:

```
alias "Macintosh HD:Users:fred:Documents:"
```

Notice that the path to a folder ends in a colon. This alias form is the one preferred by various other scripting addition commands that require a folder reference as a parameter. If necessary, you can



Chapter 7:
**Scripting
 Addition
 Commands and
 Dictionaries**

coerce the reference to text, append a file name, and create a file reference for some other file-based command:

```
set myPath to (choose folder with prompt ¬
    "Choose your Scripts folder:") as text
set myPath to myPath & "Test Script Data"
set myFile to (open for access file myPath ¬
    with write permission)
(* do file writing here *)
close access myFile
```

Parameters

The first of four optional parameters lets you insert a prompt to help the user locate the kind of folder your script requires. If no **with prompt** parameter is included with the command, the Choose Folder command omits a prompt entirely from the dialog. Otherwise the prompt string appears on one line just below the window's titlebar.

The values and behavior of the remaining three parameters—**default location**, **invisibles**, and **multiple selections allowed**—are identical to the same labeled parameters for the **choose file** command, described earlier in this chapter.

Examples

```
choose folder

set folderPath to (choose folder with prompt
    "Where should we store the backup file?")

size of (info for (choose folder default location
    (path to library folder from user domain)))
```

You Try It

Enter and run each of the script statements in the Examples section, above, watching the intermediate values in the Result pane.

Common Errors

Using “alias” or “file” in commands with results from this command, when “alias” is already part of the resulting data.

Related Items (Chapters)

Choose file (7).



choose URL [showing *servicesList*]
 [editable URL *Boolean*]

Result

When the Connect button is clicked, the URL selected by the user as a string. A click of the Cancel button throws a “User canceled.” error.

When To Use

The **choose URL** command presents a Connect to Server dialog window to the user—the same dialog that the user sees when choosing **Connect to Server** from the Finder's Go menu (Figure 7.5). As with most AppleScript standard additions that display system dialog boxes, the associated command simply returns the chosen information without acting on the information. Thus, despite the appearance of the Connect button in the script-generated dialog, the **choose URL** command does not open any



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

connection to the selected server. Instead, the script needs to be written to apply the returned URL string value to some additional command (perhaps within an application's suite) that acts on the URL.

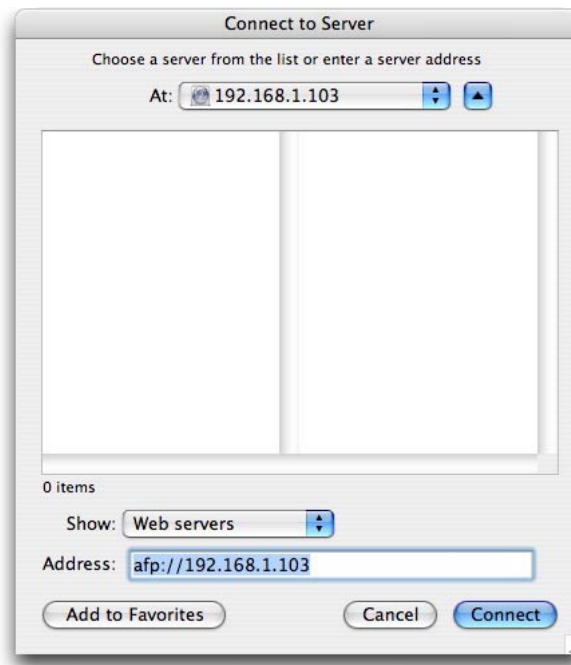


Figure 7.5. The choose URL dialog box.

Parameters

One of the two optional parameters to **choose URL**, **showing**, lets you limit the types of services from which the user may choose. The parameter value must be a list class, and the list must contain one or more of the following (non-string) constant

values:

Web servers	News servers
FTP Servers	Directory services
Telnet hosts	Media servers
File servers	Remote applications

For example, to limit the display to Web and FTP servers, the command would be as follows:

```
choose URL showing {Web servers, FTP Servers}
```

When you specify more than one service, the Show select list appears in the dialog window, allowing the user to choose a service type from the items passed as parameters. If you omit the parameter, the Show select list contains the names of all service types. The system controls the order of items in the displayed list, regardless of the order in which you supply items as parameter values.

The second optional parameter, **editable URL**, controls whether the dialog displays an editable text entry field, into which a user may type a URL. The default value is **true**. Therefore, use this parameter to hide the entry field and limit choices to those items that appear in the dialog's list box.

Examples

```
choose URL
```

```
choose URL showing -  
    {Web servers, News servers} -  
    without editable URL
```



Chapter 7:

Scripting Addition Commands and Dictionaries

You Try It

Enter and run the following script statement in Script Editor to see the dialog box appear:

```
choose URL
```

If an item appears in the dialog window's list, select it and click the Choose button; otherwise, select Web Servers from the Show list and enter any Web URL, such as `http://apple.com`, and click the Choose button. See the resulting string in the Result pane.

Next, if you're not already connected to the Internet (perhaps you use telephone dial-up access), connect now. Then enter and run the following statement, which takes the results from the choose URL command and applies it to the open URL command. Your default browser should load the URL you enter:

```
open URL (choose URL showing {Web servers})
```

Common Errors

Using quotes around the **showing** parameter's list values; not anticipating the user clicking the Cancel button.

Related Items (Chapters)

Open URL (6); list value class (9).



delay *seconds*

Result

No result.

When To Use

Insert a **delay** command whenever you wish to introduce an artificial pause between two other script statements. While the delay is active, no other processing occurs in the current script. Other applications are not affected. You can manually cancel the delay by clicking the Stop button in Script Editor or typing Command-Period.

Parameters

The one required parameter is a number of seconds before the next statement executes. Values are not limited to integers.

Example

```
delay 3
```

You Try It

Enter the following script into Script Editor and run it a few times, selecting different delay values from the initial dialog list.

```
set secs to (choose from list {"1", "2", "5",  
    "10"})  
if secs is not false then  
    delay (secs as number)  
    display dialog "The wait is over."  
end if
```

Although the example explicitly coerces the string value returned from **choose from list** to a number class value, the **delay** command is smart enough to automatically coerce string representations of numbers to the proper parameter value. It is better practice, however, to force the



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

coercion as shown.

Common Errors

Confusing this command with similar commands of other languages that use millisecond parameter values, instead of this command's seconds units.

Related Items (Chapters)

None.



```
do shell script    commandString ~  
    [ as returnValueClass ] ~  
    [ administrator privileges Boolean ] ~  
    [ password passwordString ] ~  
    [ altering line endings Boolean ]
```

Result

Text results or output from the shell script command.

When To Use

One of the powers of Mac OS X that is typically hidden from the view of most casual users is the ability to control numerous aspects of the system and files through Unix commands and sequences of commands saved as little programs called **shell scripts**. The “shell” is how programmers refer to the command-line interface through which they communicate with the system and its services. Mac OS X users with more familiarity with Unix commonly run Unix commands and invoke shell scripts via the Terminal application that is delivered

with Mac OS X. Similarly, a system process known as **cron** can invoke commands and run shell scripts at specific times. Mac OS X comes equipped with some cron jobs that perform regular maintenance on the system—tasks that are important to the integrity of the system, but have no particular user interface component. Learning how to write Unix shell scripts is a separate task you may wish to undertake, and for which numerous publications provide instruction.

The **do shell script** scripting addition command allows AppleScript to invoke Unix commands directly, rather than having to go through an application such as Terminal. Upon invoking a shell script, processing is handed off to the system until a result (if any) is returned. You can freely intermix AppleScript statements (and user interface elements you create through scripting addition dialog boxes or AppleScript Studio interface constructors) with the calls to shell scripts, as shown in one of the examples below.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Communication between AppleScript and shell scripts is a two-way street. Just as an AppleScript can invoke a shell script through the **do shell script** command, so can a shell script invoke a saved AppleScript script. The shell command is called **osascript**, and the parameters depend on what you intend to control. If the command is a direct AppleScript command (e.g., to a scripting addition), use the **-e** parameter as well as a quoted string containing the AppleScript command. For example, the following shell command invokes the **say** scripting addition to verbalize some text:

```
osascript -e 'say "You are the best!"'
```

To run a saved AppleScript script from the shell, supply the pathname (unquoted) to the script file:

```
osascript -  
Library/Scripts/collateData.scpt
```

Be aware, however, that AppleScript scripts invoked from a shell script via **osascript** cannot contain statements that involve user interaction (such as displaying dialog boxes of any kind). You'll still see things occur while the script is running (such as activating applications), but don't use statements that ask users to locate files or display results in dialog boxes.

Parameters

One required parameter is a quoted string containing the text of the command to be invoked. This could be a built-in Unix command or the name of an

executable shell script. You can combine multiple commands into one parameter by separating the individual commands with semicolons. For example, the following statement changes the operating directory (via the Unix **cd** command) to the Applications directory and returns a list of the contents of the directory (the Unix **ls** command), complete with owner and permissions information:

```
do shell script "cd Applications; ls -l"
```

You have a measure of influence over the value class of the returned data via the **as** labeled parameter. The default value class is Unicode text, but don't expect miracles of coercion to more far-flung types. For example, just because a command returns a nicely formatted, return-delimited string list of items doesn't mean that you can request the data in the form of an AppleScript list value class. Instead, use other AppleScript powers to convert such a list to a list class. The following script sequence grabs a list of application file names and converts the string list into a list:

```
set stringList to -  
    (do shell script "cd Applications; ls")  
set text item delimiters to return  
set itemList to text items of stringList
```

If the shell command or script you need to run requires administrator privileges, you can control whether the script can run with such privileges and even include the administrator password via the optional **administrator privileges** and **password** parameters. The value of the former is



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

a Boolean value, while the password value is plain text. If you set **administrator privileges** to **true** and omit the **password** parameter, the script displays a password entry dialog box. For the next five minutes, the system will not prompt for a password for the same command. Be sure to protect script applications containing the administrator password so that others may not view or edit the script. Or, safer still, don't include this information in any script.

Unix and Macintosh systems treat end-of-line characters differently, and such differences can affect scripts that rely on multi-line responses from a shell command. By default, the **do shell script** command converts Unix-style linefeed characters to Mac-style return characters. But if you wish to preserve the Unix-style linefeed characters in a returned value from the command, set the **altering line endings** parameter to **false**.

Examples

```
do shell script "uptime"
```

```
do shell script "sudo apachectl restart" with  
administrator privileges password "notmydog"
```

You Try It

Enter and run the following script in Script Editor, viewing the returned value in the Result pane:

```
do shell script "cal 3 2005"
```

The **cal** Unix shell script is delivered with Mac OS X and returns a textual monthly calendar for the

month and year supplied as parameters.

Here's another blast from the past. It creates an ASCII-style sideways banner of whatever text you enter into the dialog box:

```
display dialog "Enter a name for the banner:"  
default answer "Abraham Lincoln"  
if text returned of result is not "" then  
do shell script "banner " & text returned  
of result  
end if
```

To see the results more clearly, copy the contents of the Result pane, and paste it into a text editor with a mono-spaced font (Courier or Monaco). Of course, these days, you won't likely have fan-fold paper to print out the banner like we used to.

Common Errors

Invoking a shell script that keeps running a process without returning a value.

Related Items (Chapters)

Value classes (9).



get volume settings

Result

A **volume settings** record value whose properties are: **output volume**, **input volume**, **alert volume**, and **output muted**.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

When to Use

The **get volume settings** command lets your script read four sound-related system settings. Three settings are related to output volume, that is, the sound the user hears through the current speaker setup for the Macintosh. All three settings may be controlled manually via the Sound pane in System Preferences. Values for volume (**output volume** and **alert volume**) are integer values in the range of 0 to 100. Output volume controls the overall sound volume level for all kinds of sounds, whereas the alert volume controls what percent of the output volume is applied to system and application alerts. If the **output volume** is zero or the Mute checkbox is checked, then the **output muted** property is **true**. No sound will be heard if muting is on. Presumably the **input volume** property reports the input volume setting, but despite having a sound source connected to my Macintosh (and the input level bars responding to sounds), the value for my Macintosh always returns the **missing value** constant.

Parameters

None.

Examples

```
set currVol to output volume of (get volume
    settings)

if output muted of (get volume settings) is
    false then say "Good Morning!"
```

You Try It

Open the Sound pane in System Preferences next to a new Script Editor window. Enter the following statement into Script Editor and view the returned value in the Result pane:

```
get volume settings
```

Adjust output volume settings in the Sound pane, and run the script again to see how changes are instantly reflected in the command's returned value.

Common Errors

None.

Related Items (Chapters)

Set volume (7).



```
set volume integer0thru7 -
    [ output volume integer0thru100 ] -
    [ input volume integer0thru100 ] -
    [ alert volume integer0thru100 ] -
    [ output muted Boolean ]
```

Result

None.

When to Use

Whenever your script needs to adjust the volume level while working with various applications, use this command. Changes made to settings via the AppleScript command are immediately reflected in the Sound pane of System Preferences.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Parameters

The **set volume** scripting addition command was originally introduced with a single parameter whose value was an integer from 0 through 7. These values correspond to the possible positions of the volume slider control in the Sound control panel of Mac OS versions 9.x and earlier. While this parameter is still supported in the more modern versions of the command, the usage of that unlabeled parameter is deprecated. As long as you are scripting exclusively for Mac OS X, use the other four optional labeled parameters.

The four optional parameter labels are the same as the property names returned by the **get volume settings** command: **output volume**, **input volume**, **alert volume**, and **output muted**. The first three require integer values in the range from 0 to 100. The **output muted** parameter is a Boolean value.

Examples

```
set volume output volume 50 alert volume 80
```

```
set volume with output muted
```

You Try It

Open the Sound pane in System Preferences next to a new Script Editor window. Enter the following statements into Script Editor and view the changes in the Sound pane:

```
set volume output volume 0
```

```
set volume output volume 30
```

```
set volume output volume 50 with output muted
```

```
set volume without output muted
```

```
set volume alert volume 66
```

Common Errors

Misunderstanding the relationship between output and alert volume settings.

Related Items (Chapters)

Get volume settings (7).



system attribute -

```
[ environmentVarName ] [ has integer ]
```

Result

If no environment variable is supplied as a parameter, the command returns an AppleScript list of environment variable names that can be used, individually, as parameters to the command; otherwise the command returns the value of the environment variable passed as a parameter, usually either a string or number class value.

When to Use

Use the **system attribute** command if your script needs to know the values currently assigned to one of several environment variables, such as the login name of the current user or the path to the user's Home directory. Most scripters won't have



Chapter 7:

Scripting Addition Commands and Dictionaries

a strong need for this command, but those writing automation scripts for system administration may find the returned values useful.

Parameters

To find out which environment variables are currently available, invoke the command with no parameters. The returned value is a list of variable names as strings. The last time I ran the command this way, the list consisted of the following strings:

```
"SECURITYSESSIONID"      "HOME"  
"SHELL"                  "USER"  
"PATH"  
"__CF_USER_TEXT_ENCODING"
```

These values represent parameter values you can pass to the **system attribute** command to read the value for that variable. Environment variable names are case-sensitive.

Examples

```
system attribute  
    -- returns all environment variables  
system attribute "USER"
```

You Try It

Enter and run the command name by itself in Script Editor. Read the Result pane list to see the environment variables returned on your system. Then invoke the command with each parameter value in turn to examine the value returned for that variable.

Common Errors

Failing to observe the correct case of the parameter values.

Related Items (Chapters)

None.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Finder Commands



info for *fileOrFolderReference*

Result

A record containing fields describing numerous characteristics of the Finder item. The record is, itself, a class defined in the standard scripting additions dictionary. The class is called **file information**. The precise collection of properties in the file information class depends on whether the item in question is a folder, application, or data file. Table 7.3 shows what properties are returned in the **info for** command's result for each type of item.

Property	Folder	Application	Data File
alias	•	•	•
bundle identifier		•	
busy status			•
creation date	•	•	•
default application	•		•
displayed name	•	•	•
extension hidden	•	•	•
file creator	•	•	•
file type	•	•	•
folder window	•		
folder	•	•	•
icon position	•	•	•
kind	•	•	
locked			•
long version		•	•
modification date	•	•	•
name extension	•	•	•
name	•	•	•
short version		•	•
size	•	•	•
visible	•	•	•
package folder	•	•	•

Table 7.3. *Info For results by item type.*

When to Use

There is a strong likelihood that a number of Finder related tasks you need to perform involves one or more properties of a file or folder. These properties are generally the kinds of things you see depicted in an item's Get Info dialog box. This command lets a script retrieve that information for any file or folder. Moreover, the command returns the four-character file creator and file type strings that don't appear in the Finder.

The **info for** command only allows reading of



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

these properties. Some of these properties, however, may be changed by scripting the Finder.

Because the data comes back in a record value format, each of the items is labeled, making it comparatively easy to extract the data in readable scripts. For example, the following two-statement sequence prompts the user to select a file and then obtains the size of the file:

```
set fileToExamine to choose file  
get size of (info for fileToExamine )
```

A script can directly extract the labeled item from the **result** of the **info for** command. The key is knowing the exact names of the labels—and then knowing the class for a particular field's data for further manipulation. Table 7.4 (see next page) shows all of the property names, classes, and examples of the data returned for these properties.



Chapter 7:

Scripting Addition Commands and Dictionaries

<i>Property</i>	<i>Value Class</i>	<i>Example</i>
alias	Boolean	false
bundle identifier	Unicode text	"com.apple.AddressBook"
busy status	Boolean	true
creation date	date	date "Thursday, January 8, 2004 11:04:02 AM"
default application	alias	alias "HD:Applications:Office X:Microsoft Word"
displayed name	Unicode text	"Address Book"
extension hidden	Boolean	true
file creator	text	"MSWD"
file type	text	"W8BN"
folder window	rectangle	{196, 137, 623, 912}
folder	Boolean	false
icon position	point	{132, 68}
kind	Unicode text	"Microsoft Word document"
locked	Boolean	false
long version	text	"10.1.1 (2425), © 1983-2001 Microsoft Corp...."
modification date	date	date "Friday, January 9, 2004 02:29:49 PM"
name extension	Unicode text	".app"
name	Unicode text	"Address Book.app"
short version	text	"10.1.1"
size	integer	6.640831E+6
visible	Boolean	true
package folder	Boolean	true

***Table 7.4.** File information properties, classes, and examples.*

An item's **name** property is the name for that item as the Finder knows it. In the case of Mac OS X applications, this name includes the .app extension that is hidden when viewing applications in lists and Finder windows. Thus, the **displayed name** property reveals how the file appears in various Finder views. The meanings of most properties are self-explanatory, or easily deduced by inspecting values returned for various items on your own



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

system (some shortcuts on how to do this are shown below). Additional properties include:

visible: A Boolean value indicating whether the file or folder is visible in a Finder window. Many times you don't want to perform operations on hidden files or folders, so checking for this property can tell your script whether the item is worth messing with.

folder window: The global screen coordinates of the window of a folder. This property is available only for folders, and applies even to a closed folder, which knows the position and size to which it would open if double-clicked by the user.

default application: An alias reference to the application that would open with the document if the document were opened from the Finder. Document files that are not linked to applications may present problems for your script, because the property does not come back in the result of the **info for** command. Instead you receive an error that the script can't get that property (we'd expect the property to come back empty, but such is not the case). Therefore, if your script must extract this information about a document file, it is best to do so in a **try-on error** construction (Chapter 12) to avoid breaking scripts on plain document files.

name extension: Just that portion of the item's name following (and not including) the last period. For example, the **name extension** property of a file named sample.pdf would be **"pdf"**.

alias: A Boolean value that reveals whether the item is a Finder alias to another item.

icon position: The coordinate point of the upper left corner of the icon within the Finder window that contains the icon for the file or folder. The value is in the form of two-item list (e.g., **{41, 55}**) whose first value is the horizontal coordinate. The icon referred to here is the one that shows when the Finder's View menu is set to "by Icon."

Syntax for extracting a property from a record is flexible. For example, below we assign the record of a file's info to a variable. Succeeding statements show two methods of extracting the **modification date** property from that variable:

```
set fileData to info for alias ~  
    "HD:Documents:Letters Template"  
get modification date of fileData  
-- or --  
get fileData's modification date
```

Your choice depends on which version seems more readable to you.

A more important question you'll have to ask yourself once you feel comfortable using AppleScript and the scriptable Finder is whether you should use the **info for** command or the Finder, itself, to retrieve this information. The Finder's corresponding access to these bits about Finder "things" is an object class called **item**. The **item** class encompasses all kinds of files, folders, applications, and other entities that the Finder represents. An alias reference



Chapter 7: Scripting Addition Commands and Dictionaries

to a file, for example, is sufficient to access that entity as a Finder item (inside a **tell** block aimed at the Finder), and therefore any of its 28 or more properties (special types of items have additional properties over the basic 28). For example, in lieu of the scripting addition syntax for retrieving the creation date of a file:

```
get creation date of (info for alias ~
    "HD:Documents:Letters Template")
```

the Finder version would be:

```
tell application "Finder"
    get creation date of ~
        alias "HD:Documents:Letters Template"
end tell
```

Some properties, particularly **icon position**, are reported more accurately and reliably by the Finder. That may offer suitable incentive to prefer the Finder over the **info for** scripting addition command. But if all of this is new to you, the scripting addition is an easy way to introduce yourself to the kinds of information available for Finder items.

Parameters

This command requires an alias or file class reference to a folder or file as its sole parameter, as in

```
info for alias "HD:Applications:MacProject Pro"
    -- for file

info for alias "HD:Applications:"
    -- for folder
```

Other AppleScript commands, particularly **choose file** and **choose folder**, provide results that

feed the **info for** command parameters directly. In fact, the combination of the two commands is helpful to scripters who, in the course of developing a script, are in need of a file type for a document. This one-line script can get the info you need for just such a task:

```
get file type of (info for (choose file))
```

Examples

```
info for "Hard Disk:System Folder:System"
```

```
creation date of ~
    (info for "Hard Disk:Documents:")
```

```
get modification date of ~
    (info for (path to preferences))
```

You Try It

Enter and run each of the following scripts, watching for returned values in the Result pane:

```
info for (choose file)
```

```
info for (choose file of type "PDF ")
```

```
info for (path to system folder)
```

```
size of (info for (choose file))
```

```
display dialog (creation date of ~
    (info for (choose file))) as string
```

```
icon position of ~
    (info for (path to system folder))
```

Common Errors

Not taking into account the class of a particular

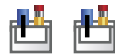


Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

record field's value before performing further manipulation with the result; forgetting the alias reference in the parameter; trying to get a file-only property of a folder.

Related Items (Chapters)

Choose file (7); **choose folder** (7) **path to** (7); record value class (9); date value class (9); Boolean value class (9).



list disks

Result

An AppleScript list of all volumes mounted on your Desktop.

When to Use

This command offers a convenient way to check whether a particular volume is mounted on the script runner's Macintosh. If your script expects files or applications to be available on the local Mac, the script can use this command to check before it makes any calls to the desired volume, and elegantly handle user messages or other actions.

This command treats all volumes—hard disks, CDs, DVDs, volumes shared across a network—the same, and does not distinguish one type from another in the list of volumes it returns:

```
list disks (* result: {"Hard Disk", "Backup  
HD", "U2-Vertigo"} *)
```

There will always be at least one item in the list: the name of the startup disk.

Parameters

None.

Examples

```
list disks
```

```
count of (list disks)
```

```
if (list disks) does not contain "Big Daddy"  
then  
    mountServer()  
    -- subroutine that opens a server alias  
end if
```

You Try It

Enter and run the following scripts:

```
list disks -- returns list in Result pane  
  
display dialog "You have " & (count of (list  
disks)) & " volume(s) mounted."
```

Common Errors

Forgetting that returned values are in a list.

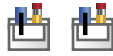
Related Items (Chapters)

If - then (10); list class values (9).



Chapter 7:

Scripting Addition Commands and Dictionaries



```
list folder folderReference -  
[ invisibles Boolean ]
```

Result

An alphabetical AppleScript list of file and folder names (but not pathnames) in the designated folder.

When to Use

With this scripting addition you can retrieve a list of all items in a folder. From here, you can see if the list contains a particular item your script may be interested in. This can be especially valuable for folders that hold transient information, such as Trash or Temporary Items. Bear in mind that the Desktop and Trash are treated as folders (as, indeed, they are within the Finder), so you can use the **list folder** command to see if any files or folders are in those locations. Your script can determine if these folders are empty or full, as in:

```
list folder (path to trash)
```

If the Trash folder is empty, the result would be a list containing no items:

```
{ }
```

Or if there were two items in the Trash, the result would look like this:

```
{"Documents", "SuperDweeb"}
```

Items in a list do not distinguish between file and folder items. They're all listed as equal citizens.

Parameters

The required parameter, *folderReference*, is an alias or file class value, but the notation must indicate a folder—ending with a colon, as in:

```
list folder alias "MacHD:Users:jane:Desktop:"
```

If you omit the trailing colon after a valid folder name, the colon is inserted for you at compile time. The command also accepts a folder path as a string, but I recommend using an alias or file class value.

For listing special system folders, nothing beats nesting the **path to** command as a parameter to the **list folder** command. The **path to** command returns the required alias class value, as in:

```
list folder (path to music folder)
```

If your script requires a list of pathnames to, say, document files in a folder, use Finder scripting to extract all such files within a particular folder and coerce each item's reference to an alias value. For example:

```
tell application "Finder"  
    set fileList to document files of  
    (choose folder)  
    get item 1 of fileList as alias  
end tell
```

The optional second parameter allows you to filter out invisible items, such as the ubiquitous `.DS_Store` and other normally hidden items. When you enter the command and set the **invisibles** property to



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

false, the compiler turns the phrase around a bit, but the meaning is the same. If you enter

```
list folder alias "Macintosh HD:" -  
    invisibles false
```

the compiler displays

```
list folder alias "Macintosh HD:" -  
    without invisibles
```

The revised version reads better, and you can enter it that way if you prefer.

Examples

```
list folder "MacHD:Users:tarzan:Desktop" -  
    without invisibles  
  
list folder (path to preferences folder)  
  
count of (list folder (path to extensions))
```

You Try It

Enter and run each of the script statements below, watching the returned values in the result window:

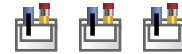
```
list folder (path to system folder)  
  
list folder (path to startup disk)  
  
list folder (path to startup disk) -  
    without invisibles  
  
count of (list folder (path to -  
    utilities folder))
```

Common Errors

Forgetting to supply an alias or file class value as a parameter; trying to list the contents of a file.

Related Items (Chapters)

Path to (7); alias and file class values (9); list class values (9).



```
mount volume nameOrURL -  
    [ as user name userName ] -  
    [ in AppleTalk zone zoneName ] -  
    [ with password password ]
```

Result

A file class reference to the “folder” container that the mounted volume represents.

When To Use

It’s not uncommon in automating processes across networks for the script to need to mount a remote volume (an external server represented as a disk volume) onto the Desktop. The **mount volume** command is the scripted equivalent of the **Connect to Server** item of the Finder’s **Go** menu. The command has exhibited various bugs in some versions, so be sure to test the operation on the machine(s) on which scripts using this command will be deployed. Operation has been relatively reliable with Mac OS X 10.3.2 and later. The version of AppleScript that came with that version (and later versions) added the ability to mount a wider range of servers, expanding from the original AppleTalk server range to Windows (smb) and FTP servers. As is frequently the case working with servers on a network, you may have to wrestle with permissions



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

and passwords to complete a script performing more sophisticated tasks. For instance, access to a mounted FTP server may be limited to being read-only. Operating within the permissions imposed on your connection to the server, you may use Finder scripting commands on the contents of the mounted volume, just as if it were a local volume.

Network problems are also a possibility. Networks and servers occasionally become unavailable. Any such problems (or the user failing to enter a correct username/password pairing) will generate a variety of network-related error messages. Therefore, always build your **mount volume** command inside a **try-on error** block to handle errors gracefully. Mounting an already-mounted volume does not result in any error.

Although there is no “unmount” command, you can use the Finder’s **eject** command to remove the mounted volume from your Desktop.

Parameters

Successfully mounting a remote volume requires knowing the URL of the server, the name of the server volume you wish to mount, the user name, and password. These are the same pieces of information you need to perform the same action via the **Connect to Server** menu item, although when you follow that process, you’re more or less prompted for this information as it is needed. When you script it, you need the information up front to pass as parameters to the **mount volume** command.

You can use one of two formats for passing this information to the command. The more verbose way is to use the command’s explicit parameters. First, the required parameter is the URL of the server and volume. Assuming for now that you’re mounting an AppleTalk server, the format is as follows:

```
mount volume -  
    "afp://serverNameOrIPAddress/volumeName"
```

Using a simplified example in which the remote server is another Macintosh on the same router as the Mac running the script, when Personal File Sharing is enabled in the remote Mac, the Sharing preference pane notes the URL of the newly available server (perhaps something like `afp://192.168.1.104/`). If you to supply this to the **Connect to Server** dialog plus the necessary user name and password, you would be presented with another dialog presenting a list of available volumes on that server. The desired volume name should be part of **mount volume**’s URL. For example:

```
mount volume -  
    "afp://192.168.1.104/TomsLaptopHD"
```

It’s vital to note that because this parameter is a URL, no spaces or odd punctuation symbols are allowed. Instead, use URL-escape equivalents for such characters. A space is represented by **%20**. Thus, if the name of the volume above were actually “Toms Laptop HD”, the command would have to be represented like this:



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

```
mount volume -  
    "afp://192.168.1.104/Toms%20Laptop%20HD"
```

If you supply no additional information to the command, the system presents a username/password entry dialog box. To bypass this dialog, you can build the user name and passwords into the command by supplying those strings as labeled parameters. If your user name for a particular server were “smokey” and your password were “notMyDog”, the complete command would be as follows:

```
mount volume -  
    "afp://192.168.1.104/Toms%20Laptop%20HD" -  
    as user name "smokey" -  
    with password "notMyDog"
```

Include user names and passwords only in uneditable versions of scripts so that others cannot gain access to these valuable pieces of information.

An alternate format allows you to include all of the information as a single parameter to the mount volume command. The format is as follows:

```
mount volume "username:password@afp://  
    serverNameOrIPAddress/volumeName"
```

Using the values from the earlier example, the single URL version would be:

```
mount volume "smokey:notMyDog@afp://192.168.1.  
    104/Toms%20Laptop%20HD"
```

When working with an FTP server, you may experience problems passing a user name and password by way of the parameters. Instead, you will be presented with a dialog requesting permission

to use an entry from your keychain associated with the URL. If you have a user account with the server and allow the keychain data to be used, you will be granted access to the desired server (although perhaps with read-only permission) from the Desktop; if you deny use of keychain data, the server may produce instead a public directory that is normally available for unregistered guest access.

Note that the AppleScript dictionary for this command indicates a required parameter called **on server**. This parameter does not work and is not needed when you follow the instructions above. Similarly, the **in AppleTalk zone** parameter is backward compatible with older Apple networks.

Examples

```
mount volume "afp://192.168.1.101/shared"
```

```
set remoteVol to mount volume -  
    "afp://192.168.1.101/shared" -  
    as user name "janedoe" -  
    with password "notMyDog"
```

You Try It

If you have access to a shared AppleTalk server on your network, follow the instructions described earlier for turning on Personal File Sharing, locating the URL, and entering necessary parameters to the mount volume command to bring the volume to your Desktop. The following script begins by mounting a remote volume, preserving a reference to the volume in a variable. Next, it presents the user



Chapter 7:
**Scripting
 Addition
 Commands and
 Dictionaries**

with a list of files and folders from the top level of the mounted volume. Upon selecting an item, the user sees some information retrieved about the file. Finally, the remote volume is unmounted with the help of the Finder.

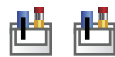
```
try
  set remoteVol to mount volume
  "afp://192.168.1.104/Toms%20Laptop%20HD" as
  user name "smokey" with password "notMyDog"
  set remoteItem to choose from list (list
  folder remoteVol)
  set creation to creation date of (info for
  alias ((remoteVol as string) & remoteItem))
  display dialog "The remote file " &
  remoteItem & " was created on " & creation
  tell application "Finder"
    eject remoteVol
  end tell
end try
```

Common Errors

Using an incorrect address for the remote server;
 failing to anticipate network and user errors.

Related Items (Chapters)

None.



path to *applicationOrFolder* ~
 [from domain] ~
 [as class] ~
 [folder creation Boolean]

Result

The path name to the designated application or folder as an alias reference.

When to Use

This command is intended to help your script locate the fully qualified pathname to specific system-related folders and applications on the startup disk of your Mac. Typically, a script would use the value returned by this command to assemble a complete pathname for a file in the system-related folder, and then open or save the file there.

Values returned from this command are in the alias class, such as:

```
alias "Macintosh HD:Users:hubertb:Movies:"
```

If you wish to build onto the returned value to create a path to a particular item inside the original target, you can retrieve the value as a string, and then append the rest of the known path to the string. The following script gets the path to the user's home directory as a string. Then it concatenates the rest of a path to the StickiesDatabase file inside the home library.

```
get path to home folder as string
set stickiesDB to result & ~
  "Library:StickiesDatabase"
set stickiesDBSize to size of ~
  (info for alias stickiesDB) as integer
```

The last line of this script re-coerces the string path value back to an alias, as required by the **info for** command. Notice, too, that the value returned by the



Chapter 7:

Scripting Addition Commands and Dictionaries

path to command ends in a colon when the item it's working on is a folder.

Parameters

This command requires one of a specifically worded list of items as a parameter. Several of the items are leftovers from the pre-Mac OS X days, and work only when the command is directed to a Mac Classic domain (explained below). Table 7.5 lists all parameter constants that compile when writing scripts under Mac OS 10.3.2 or later. Items indicating Mac OS 9.2 must be aimed at the Classic domain, while some items (with no bullets) compile successfully but point to no particular default item installed with either system version. Constant words in square brackets are optional.

Table 7.5. *Path to Command Parameter (see right column)*

Constant	MacOS 9.2	MacOS 10.3
apple menu [items folder]	•	
application support [folder]	•	•
applications folder	•	•
control panels [folder]	•	
control strip modules [folder]	•	
current application	•	•
current user folder		•
desktop [folder]		•
desktop pictures folder	•	•
documents folder		•
editors [folder]		
extensions [folder]	•	
favorites folder	•	•
Folder Action scripts	•	•
fonts [folder]	•	•
frontmost application	•	•
help [folder]	•	•
home folder		•
internet plugins folder	•	
keychain folder	•	•
library folder		•
me	•	•
modem scripts [folder]	•	•
movies folder		•
music folder		•
pictures folder		•
preferences [folder]	•	•
printer descriptions [folder]	•	•
printer drivers [folder]		
printmonitor [folder]	•	
public folder		•
scripting additions [folder]	•	•
scripts folder	•	•
shared documents [folder]		•
shared libraries [folder]	•	•
shutdown items folder	•	
sites folder		•
speakable items	•	
startup disk	•	•
startup items [folder]		•
stationery [folder]	•	
system folder	•	•
system preferences	•	•
temporary items [folder]		•
trash [folder]		•
users folder		•
utilities folder	•	•
voices [folder]	•	•



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Constant values **current application** and **me** point to the path to the application context for the script that executes the **path to** command. When you run a script in Script Editor, the command returns a path to the Script Editor application. For a script saved as an application, the context is the script application file itself. For a script saved as just a script, the application context is a normally hidden Mac program called System Events.app. And if the script is invoked from another application, the current context is that application.

You may also substitute a reference to an application as the parameter to the command. For example, to obtain the path to the iTunes application on your hard drive, the command is:

```
path to application "iTunes"
```

The **path to** command also accepts a long list of four-character codes that correspond to not only the places listed in Table 7.5, but a host of other special system folders in Mac OS X. You can find the entire list online at http://developer.apple.com/documentation/Carbon/Reference/Folder_Manager/folder_manager_ref/constant_6.html.

Using the optional **as** parameter for class coercion is faster than letting AppleScript perform the coercion. Therefore, while these two statements yield the same result,

```
path to preferences as text
```

```
(path to preferences) as text
```

the first version runs marginally faster.

Another optional parameter, **from**, lets you direct the request for a path to a specific operational domain. Permitted values are: **system domain**, **local domain**, **network domain**, **user domain**, and **Classic domain**. For example, to obtain the path to the Mac OS 9 applications directory on a machine, the command is:

```
path to applications from Classic domain
```

Aside from the obviously different paths returned for Classic and non-Classic domains, some paths reflect differently when directed to different domains within Mac OS X. For example, the default path for the library folder is the Library folder directly inside the startup disk (e.g., **alias "Macintosh HD:Library:"**). But by directing the call to the system domain, the returned path becomes **alias "Macintosh HD:System:Library:"**, and directing the call to the user domain yields **alias "Macintosh HD:Users:username:Library:"**. Each path constant has its own default domain.

If the folder named in the **path to** command does not exist in the target domain, the default behavior of the command is to create the folder in that domain. This can lead to a lot of litter if you're not careful. But you can prevent the folder creation by adding the



Chapter 7:

Scripting Addition Commands and Dictionaries

without folder creation parameter.

Examples

```
path to desktop folder
```

```
path to startup disk
```

```
set extensionsPath to (path to extensions -  
from Classic domain as string)
```

```
set printersPath to (path to library folder -  
from user domain as string) & "Printers:"
```

You Try It

Enter and run each of the following scripts in Script Editor, watching the returned values in the Result pane:

```
path to scripts folder
```

```
path to Folder Action scripts as string
```

```
path to frontmost application
```

```
tell application "TextEdit"  
  activate  
  path to frontmost application  
end tell
```

```
path to library folder from local domain
```

```
path to library folder from user domain
```

Common Errors

Not using one of the predefined parameters;
targeting the wrong domain; forgetting that unless

coerced by the **as** parameter, the result is an alias value class.

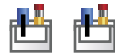
Related Items (Chapters)

Info for (7); **list folder** (7); alias and string value classes (9).



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

String Commands



ASCII character *integer0to255*

Result

A single-character string of the character corresponding to the integer value in the ASCII table.

When to Use

This command is a common programming capability that allows you to convert an ASCII number to the actual character it represents. The value returned will be the same in all non-symbol fonts for characters 32 to 127. For integer values above 127, the actual rendered returned may differ from font to font. The so-called control codes (zero through 31) return empty strings except for typographically significant codes: 9 (Tab), 10 (Line Feed), 12 (Form Feed), and 13 (Carriage Return).

While AppleScript often ignores the case of a character for things like file names, uppercase and lowercase characters have different numerical values in an ASCII table. Changing characters of a string from one case to another is a common application of this command (and its counterpart, **ASCII number**). Appendix B shows an ASCII table for values zero through 127.

Parameters

The only valid ASCII table values range from 0 through 255 (for a total of 256 values), and the ASCII

character command accepts integers within this range only. A parameter is required. You can supply only one parameter at a time.

Examples

ASCII character 65

```
-- result: "A"
```

ASCII character 97

```
-- result: "a"
```

ASCII character 13

```
(* result:      "  
   "  a return character *)
```

You Try It

Enter and run the following script, which changes any string you enter to all uppercase:

```
tell application "TextEdit"  
    display dialog "Enter text to be made into  
uppercase:" default answer "sample"  
    set mainString to text returned of result  
    activate  
    make new document at beginning  
    set text of document 1 to mainString  
    -- check each character of the string  
    repeat with i from 1 to count of mainString  
        set testChar to (ASCII number item i  
of mainString)  
        -- see if character is in  
        -- lowercase ASCII range  
        if testChar ≥ 97 and testChar ≤ 122  
then  
            -- adjust corresponding  
            -- character in document  
            set character i of text 1 of  
document 1 to ASCII character (testChar - 32)
```



Chapter 7: Scripting Addition Commands and Dictionaries

```
end if
delay 1
-- slow down so we can see it work
end repeat
-- replace variable with adjusted
-- copy from document
set mainString to text of document 1
close document 1 saving no
end tell
mainString -- in the result pane
```

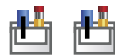
Among the lessons in this script, perhaps the most important is that we're using an application's commands and objects to help out with a chore. We chose an available application (TextEdit) that provides text object services that make substituting one character for another much easier. When we're finished with the task, we close whatever windows we opened, and still have the data in a local variable.

Common Errors

Confusing this command with **ASCII number**; value of the parameter is not an integer or is out of range.

Related Items (Chapters)

ASCII number (7).



ASCII number *characterAsString*

Result

An integer value between 0 and 255.

When to Use

This command performs the inverse action of the **ASCII character** command. This version converts a character to its equivalent numeric value in the ASCII table (see Appendix B). As demonstrated in the “You Try It” section of the **ASCII character** command description, it is common to use the two ASCII commands together in a script: one converts characters to numbers for calculations, while the other puts the numbers back into characters for reinsertion into a string.

Parameters

The required parameter must be a single character in quotes or a variable containing a single string character. If you need to obtain ASCII values for multiple characters (e.g., all characters in a word), your script will need to perform this command on each character.

Examples

```
ASCII number "G"
-- result: 71
```

```
ASCII number (item 1 of {"a","b","c"})
-- result: 97
```

You Try It

First enter and run the examples, above. Then consult the “You Try It” section for the **ASCII character** command, which includes a hands-on example of both commands.



Chapter 7:
**Scripting
 Addition
 Commands and
 Dictionaries**

Common Errors

Confusing this command with **ASCII character**; value of the parameter is not a string; the parameter character is outside the 255 ASCII character set.

Related Items (Chapters)

ASCII character (7).



offset of *containedString* in *containerString*

Result

An integer that signifies the character number of *containerString* at which *containedString* begins.

When to Use

Many operations involving strings require knowing precisely where a substring starts within a larger string. For example, you can build a subroutine that performs a search-and-replace within a chunk of text. On the companion files (*Handbook Scripts/Chapter 07 folder*) is a script called *Offset Script (Search&Replace)*, which works with a pre-formatted document (*Offset Document.rtf*) to demonstrate the **offset** command. The script grabs the contents of the document, and then sends it off to the generic search-and-replace subroutine:

```
-- We use TextEdit and a pre-written document
-- as the source of text for the search
-- and replace subroutine.
tell application "TextEdit"
    activate
```

```
choose file "Locate the file "Offset
Document.rtf": "
    open result
    set theDocument to text of document 1
    -- now we call the searchReplace
    -- subroutine, passing three parameters:
    -- 1) the document text
    -- 2) the text to search for
    -- 3) the text with which to replace
    -- the found text
    searchReplace of me into theDocument at
    "..." given replaceString:"SuperSubScript Plus"
    -- since the subroutine returns
    -- value in result, we place result
    -- back into document
    set text of document 1 to result
end tell

-- universal search and replace subroutine
-- operates strictly in AppleScript, not
-- an application
on searchReplace into mainString at searchString
    given replaceString:replaceString
    -- keep doing this until all instances
    -- of the searchString are converted
        repeat while mainString contains
        searchString
            -- we use offset command here to
            -- derive the position within the
            -- document where the search string
            -- first appears
            set foundOffset to offset of
        searchString in mainString
            -- begin assembling remade string by
            -- getting all text up to the search
            -- location, minus the first
            -- character of the search string
            set stringStart to text 1 thru
        (foundOffset - 1) of mainString
```



Chapter 7:

Scripting Addition Commands and Dictionaries

```

-- get the end part of the
-- remake string
set stringEnd to text (foundOffset +
(count of searchString)) thru -1 of mainString
-- remake mainString to start,
-- replace string and end string
set mainString to stringStart &
replaceString & stringEnd
end repeat
return mainString
-- ship it back to the caller
end searchReplace

```

At the very core of this script is the **offset** command, which, every time through the repeat loop, returns the number of the character in the string at which the search string begins. That value then becomes a fixed point to be used to extract text before and after the search string, and assemble a new string around the replacement string.

It's best to use the **offset** command in AppleScript-only environments, rather than while executing commands within an application's **tell** block. The reason is that the word *offset* is a common enough name that it could be used as a property within text-based applications. To avoid a potential conflict with the application's dictionary, pass the data to a subroutine, which works strictly within the confines of AppleScript.

Parameters

Both parameters for this command are required, and both must evaluate to strings. **offset** command parameters tend to be literal strings (in quotes) or

variables containing strings.

There is no penalty for the *containedString* not being in the *containerString*. All that happens is that the command returns 0 (zero) as the result. Consequently, the repeat loop in the subroutine above could have been controlled this way:

```

repeat while offset of searchString in ~
mainString > 0

```

The loop would end when the search string no longer exists in the main string.

In the early days of AppleScript, the **offset** command was case-sensitive. But it is currently not so. Thus, the following two statements return the same result, 12:

```
offset of "World" in "We are the world"
```

```
offset of "world" in "We are the world"
```

Examples

```
offset of "We" in "We the people"
-- result: 1

```

```
offset of "people" in "We the people"
-- result: 8 (spaces count)

```

```
offset of "America" in "We the people"
-- result: 0

```

```
offset of "row" in "row, row, row your boat"
-- result: 1 (first instance in a string)

```

You Try It

Enter and run each of the following scripts in Script



Chapter 7:

Scripting Addition Commands and Dictionaries

Editor with the Result pane visible so you can see the values returned by the offset command:

```
offset of "a" in "abc"

offset of "b" in "abc"

offset of "desire" in ~
    "A Streetcar Named Desire"

offset of "Desire" in ~
    "A Streetcar Named Desire"

set temp to ~
    "We compile no script before it's time."
offset of "script" in temp
```

Common Errors

Forgetting one of the prepositions in the parameters; specifying a parameter that does not evaluate to a string; conflict with **offset** class or command in an application.

Related Items (Chapters)

String values (9); repeat loops (12).



```
summarize textOrTextFileRef ~
    [ in sentenceCount ]
```

Result

Sherlock-generated summary of the text passed to the command.

When To Use

Originally introduced with Mac OS 8.5, the **summarize** command provides script access to the Sherlock text indexing mechanism built into the Mac OS. Applied to an HTML source code file, the summary will likely include HTML tags. I'll leave it to your experimentation with your own data to evaluate how effective the mechanism is. Sometimes it amazes; other times it disappoints.

Parameters

One required parameter is either a batch of text or a reference (file or alias class) to a text-only file from any mounted volume. Some file types, even when they contain only text, can cause an error when invoking the command. Therefore, it's best when using the summarize command with files to wrap its call inside a **try-on error** construction.

By default, the command returns a one-sentence summary. But you can increase that amount by specifying an integer to the optional **in** parameter.

Examples

```
summarize "HD:Documents:Meeting Minutes"
```

```
summarize textBlock
    -- variable containing string data
```

You Try It

You can find a text file containing the U.S. Constitution's Bill of Rights in the companion file folder for Chapter 7's scripts. From Script Editor, run each of the following statements, choosing to open



Chapter 7:

Scripting Addition Commands and Dictionaries

the file `billOfRights.txt` from the companion files:

```
summarize (choose file)

summarize (choose file) in 2

summarize (choose file) in 3
```

Common Errors

Attempting to read a non-text file.

Related Items (Chapters)

None.

Numeric Commands



```
random number [ topNumber ] ~
[ from bottomNumber to topNumber ] ~
[ with seed number ]
```

Result

For integer parameters, a random integer; for real number parameters, a random real number.

When to Use

Random numbers come into play quite often in entertainment applications (e.g., the roll of the dice) and in education applications, in which the random presentation of quiz items prevents a lesson from turning into rote exercises time after time.

Parameters

How your script specifies parameters to this command has a great deal of impact on the results returned by the command. If you specify no parameters, then the command returns a real number value between zero and one (inclusive), displayed with precision to 12 digits to the right of the decimal, as in:

```
random number
-- result: 0.513712886507
```

Supplying a single value as an unlabeled parameter instructs the command to return a value between zero and that number (inclusive)—in the same value class as the parameter. If the parameter is an



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

integer, then the command returns an integer; if the parameter is a real number, then the result is a real:

```
random number 3
-- result: 2

random number 3.0
-- result: 1.455524846616
```

In lieu of the unlabeled parameter use the pair of **from** and **to** parameters to set different low and high limits to the range of random numbers. Both the **from** *bottomNumber* **to** *topNumber* parameters must be provided together to work properly, even though their dictionary definitions makes them look like independent parameters.

Finally, random numbers are a misnomer in most personal computers, but the **random number** command reportedly uses the 60Hz ticks counter of the system clock to establish a seed value for generating pseudo-random numbers. If you apply the same number to the **seed** parameter in two different sessions, you can force the command to generate the same pseudo-random sequence of numbers. Begin the process by establishing a seed value in a separate call to **random number**; then invoke the command (without the **seed** parameter) as needed to obtain the pseudo-random numbers based on that seed value. To start the sequence over again, invoke the command once again with the same seed value; then invoke the command without the **seed** parameter to obtain the exact same sequence.

Examples

```
random number 10
```

```
set randChar to ASCII character -
(random number from 65 to 90)
```

You Try It

Enter and run the script below, which appears to roll two dice, and supplies their values in a dialog box:

```
repeat
  set die1 to random number from 1 to 6
  set die2 to random number from 1 to 6
  display dialog "The dice values are: " &
    die1 & " " and " " & die2 & "." buttons
    {"End", "Roll Again"} default button 2
    if button returned of result is "End"
  then exit repeat
end repeat
```

We set up this script in a repeat loop, so you can continue to roll the dice by pressing the Return or Enter keys. Only if you click the End button does the script exit the repeat and end. To enter the curly quote symbols into the script, type Option-[(left quote) and Option-Shift-[(right quote).

To demonstrate how the **seed** parameter works, the following script works with TextEdit and an RTF version of the Bill of Rights, found in the *Handbook Scripts/Chapter 07* folder of the companion files. The file in that format allows changing the color of individual words, which is what the script does in a random fashion. Each time you run the script without the **seed** statement, a different selection of 20 words is turned red. You can choose Revert



Chapter 7: Scripting Addition Commands and Dictionaries

to Saved from TextEdit's File menu to restore the document to its original state between tests.

```
tell application "TextEdit"
    activate
    open (choose file with prompt "Locate
        \"billOfRights.rtf\":")
    set wordCount to (count words of text of
        document 1)
    --random number wordCount with seed 100
    repeat 20 times
        set color of word (random number
            (count words of text of document 1)) of
            document 1 to {65535, 0, 0}
    end repeat
end tell
```

If you remove the comment symbols from the statement that assigns a seed value, you'll find that the same series of 20 words is changed to red each time you run the script. Set the seed value outside of the calls to **random number** from which you wish to derive pseudo-random numbers. Repeated calls to **random number** with the **seed** parameter result in the same number being returned time after time.

Common Errors

Forgetting the word "number" in the command.

Related Items (Chapters)

Integer value class (9); real value class (9).



```
round realNumber [ rounding up | down -
    | toward zero | to nearest | -
    as taught in school ]
```

Result

An integer representing the rounded value.

When to Use

The **round** command performs double duty. Its obvious function is to round a real number to an integer in whatever direction you specify in the optional **rounding** parameter. But since so many properties and parameters of other commands require integer values, you can use the **round** command to prepare a real number to be sent along as a parameter.

Parameters

One required parameter is any real number in the range from -1.797693e+308 to +1.797693e+308. Any real number larger than or equal to 10,000.0 and any number less than or equal to 0.0001 it is converted to scientific notation at compile time:

```
round 123456.7
-- becomes
round 1.234567E+5 -- after compilation
```

If you specify no additional parameters to the **round** command, the real number is rounded to the nearest integer, with .5 being rounded up. But you can alter the way rounding is performed on any value by adding the **rounding** parameter.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

This optional parameter requires one of five hard-wired parameters: **up**, **down**, **toward zero**, **to nearest**, and **as taught in school**. The default is **to nearest**. For example, you can force a value that would normally round up to round down:

```
round 35.74 rounding down    -- result: 35
```

Remember, too, that you can nest this command in a statement as an argument to another command that requires an integer value.

Examples

```
round 3      -- result: 3
round 6.5    -- result: 7
round 0.23 rounding up    -- result: 1
```

You Try It

Enter and run each of the following script lines in Script Editor, watching for values (when appropriate) in the Result pane:

```
round 25.4
round 25.5
round 0.9 rounding down
beep (round 2.3)
```

Common Errors

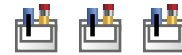
Forgetting to specify the correct rounding method.

Related Items (Chapters)

Integer value class (9); real value class (9).

Script Commands

One group of commands deals exclusively with script objects. Three commands help scripts use other script objects in their execution without having to duplicate script lines everywhere.



load script *aliasReference*

Result

A script object, which displays in the Script Editor Result pane as **<<script>>**. The resulting object should be copied to a variable or property and treated like an application object (i.e., used as a target for commands).

When to Use

It is often helpful to incorporate a pre-written library of AppleScript routines into one of your scripts. First of all, it fosters the idea of reusable code, because multiple short scripts you write can share the same library of sophisticated code with essentially no memory overhead penalty. Second, it lets your scripts take advantage of successful work others have done to write these libraries, which you may have obtained from a commercial publisher or through an AppleScript-related Web site.

To help demonstrate this AppleScript feature, you can find in the companion files (in the *Handbook Scripts/Chapter 07* folder) a script library called *daysBetweenDatesLib.scpt*. This library provides a



Chapter 7: Scripting Addition Commands and Dictionaries

single handler named **daysBetweenDates**, which takes two dates as labeled parameters (**from** and **to**). The following script loads the script library as an object and calls the routine:

```
set date1 to date (text returned of (display
  dialog "Enter a starting date in the form
    "mm/dd/yyyy":" default answer "1/1/2005"))
set date2 to date (text returned of (display
  dialog "Enter an ending date in the form "mm/
    dd/yyyy":" default answer "1/1/2006"))
set targetLib to (choose file with prompt
  "Locate "daysBetweenDatesLib.scpt":")
set dateLib to load script targetLib

tell dateLib
  -- like working with a scriptable app
  set difference to daysBetweenDates from
    date1 to date2
end tell
```

The **daysBetweenDates** command is defined in the library, and we call it just like a command in an application—with the library addressed as a target of a **tell** block, just as you do for an application. None of this can happen, however, unless we load the script as an object through the **load script** command.

While script libraries typically do not have scripts that execute automatically (i.e., as they would with **run script**), there's nothing that prevents you from having these kinds of executable lines. They won't execute, however, unless you send a **run** command to the script object. If your only goal is to run executable lines of a script, use the **run script** command (below) instead of **load script**.

Parameters

The required parameter is an alias reference to the script library you want to load, as in:

```
load script file "HD:Library:Scripts:StringLib"
```

The only kind of scripts you can load this way, however, are those that have been saved as applications or compiled scripts. In other words, they must be pre-compiled before being loaded into another script. Scripts saved as text won't work.

For more about creating and using script libraries, and making calls to them, see Chapter 14. See Chapter 9 for more about file and alias references.

Example

```
load script file ~
  (choose file with prompt "Locate a library")
```

You Try It

Enter and run the script above that loads the date library (**daysBetweenDates.scpt**). Run the script a second time, but alter the name of the handler call (perhaps just drop the trailing "s") to see the same error message you'd get by asking an application to perform a command not in its dictionary.

Common Errors

Forgetting to assign the result of the command to a variable for a later **tell** block; incorrectly including the word "application" in a **tell** statement directed at a script object.



Chapter 7:

Scripting Addition Commands and Dictionaries

Related Items (Chapters)

Run script (7); file and alias references (9); script libraries (14).



```
run script variableOrReference ~  
    [ with parameters parameterList ] ~  
    [ in scriptingComponent ]
```

Result

Whatever value (if any) the script or script object returns.

When to Use

The **run script** command is more flexible than you may think at first glance. Used one way, the command gets into the realm of script objects (Chapter 15). In other guises, it lets you mix scripts written in different OSA scripting languages that may be installed on a Macintosh.

An AppleScript script object is a software object that you can define, much like a subroutine saved as a separate script file. Script objects have properties and behaviors as you see fit. You can pass parameters to them, and they can return values. As separate files, their value is in their reusability by any number of other scripts, provided they are designed to be called as generic operations.

When you need the services of such an object, you use the **run** command to make that script go. If the script returns any data, it goes into the **result**

variable, just like any other command.

For run-and-stay-opened scripts, use the **launch** command (Chapter 6) before the **run** command.

Script files, of course, may be written in any OSA scripting language. When they are run with the **run script** command, they are run as compiled scripts, so their language of origin is of no concern to your AppleScript script.

You may want to use the facilities of another scripting language in a simple one-line statement within your AppleScript script. Instead of having to write a separate script file for that one line, you can tell AppleScript to compile and run that statement (passed as a string parameter to **run script**) using the other language.

The **run script** command should not be confused with the **do script** command found in scriptable applications that have their own scripting languages. **Run script** is strictly for OSA-compatible scripts and script objects.

Parameters

When the script object you wish to run is saved as a separate script file, the required parameter for the **run script** command must be an alias or file reference to the script file. For example:

```
run script alias ~  
    "HD:Library:Scripts:dateRecord.spt"
```

The surefire way to provide a valid pathname to a script file is to let the user choose the file via the



Chapter 7:

Scripting Addition Commands and Dictionaries

choose file command, limiting the available items to those whose file type is that of a script:

```
set fileToOpen to choose file with prompt ~  
    "Choose a script to run:" of type {"osas"}  
run script fileToOpen
```

The **choose file** command returns the chosen file as a class of alias.

Forcing the user to open the desired script object every time would be an unwelcome user interface “feature.” Fortunately, you can have your script store the path name as a property, so the user is faced with the file dialog only the first time:

```
property fileToOpen : ""  
if fileToOpen is "" then  
    set fileToOpen to choose file with prompt  
        "Choose a program to run:" of type {"osas"}  
end if  
run script fileToOpen
```

You may pass parameters to the external script being run with the **run script** command. Pass parameters via the **with parameters** parameter (gets confusing, doesn't it?). Actual parameter values are items of a list, indicating that the script object you create should have positional, rather than labeled, parameters. To see this form in action, create a script object containing a **run** handler and positional parameters. Here is an example of a handler that adds any number of days and hours to the current date and time, based on the values passed as parameters:

```
on run {deltaDays, deltaHours}  
    set theDate to current date  
    if deltaDays ≠ "" then  
        set theDate to theDate + (deltaDays  
* days)  
    end if  
    if deltaHours ≠ "" then  
        set theDate to theDate +  
(deltaHours * hours)  
    end if  
    return theDate  
end run
```

If you enter this script into the Script Editor and try to run it, nothing happens (or you may get an error), because the **run** command sent by the Run button in Script Editor does not send any parameters. Instead, save this script as a compiled script with the name “DeltaTimes.sct.”

In another Script Editor window, you can now run the script and pass parameters to it (this one requests the date and time for 2 days, 6.5 hours from whenever you run the script):

```
run script (choose file ~  
    with prompt "Select DeltaTime.sct") ~  
    with parameters {2, 6.5}
```

You can also pass parameters when the script has been loaded with the **load script** command:

```
copy (load script (choose file with prompt ~  
    "Select DeltaTime.sct")) to deltaTime  
run script deltaTime with parameters {2, 6.5}
```

So far, we've been discussing running external scripts written in AppleScript (all Mac OS X computers



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

come equipped with the AppleScript component already installed). If your Mac is equipped with another OSA scripting component, you can instruct that component to execute either a literal script statement or a file by specifying the **in** parameter and supplying as the parameter's value the name of the OSA scripting component. You can use the **scripting components** command to uncover which OSA scripting components are deployed in your machine.

Additional scripting components are supplied by third-party developers. One such component, written by Mark Aldritt, provides scripting opportunities in the JavaScript language (<http://www.latenightsw.com/freeware/JavaScriptOSA/index.html>). A very basic example of how to invoke JavaScript from within an AppleScript script is to include semicolon delimited JavaScript statements as the parameter to the **run script** command. For example, the two JavaScript statements shown in the next example reports back the value of the last item in a three-element JavaScript array (similar to an AppleScript list):

```
run script "var a = [10, 100, 1000]; a[2]" -  
in "JavaScript"
```

Similarly, you could create a text file consisting of a sequence of JavaScript statements, and then run the script statements in that file via the **run script** command:

```
run script (choose file with prompt -  
"Where is test.js?") in "JavaScript"
```

The value of the last statement executing in the external script file is returned to the result of the **run script** command.

Example

```
set newFolders to (run script alias -  
"HD:Library:Scripts:Find New Folders")
```

You Try It

In the companion files' *Handbook Scripts/Chapter 07* folder are two scripts that demonstrate script objects. Use Script Editor to open the one named Run Script Script. The first time you run this script, it asks you to find a script named dateRecord. Open it when asked. Texts of both scripts are listed here:

Run Script Script

```
property scriptObjectPath : ""  
  
if scriptObjectPath is "" then  
    set scriptObjectPath to choose file with  
    prompt "Find dateRecord script" of type  
    {"osas"}  
end if  
  
-- now run external script object  
-- and capture result  
copy (run script scriptObjectPath) to  
todaysDate  
display dialog "Today is " & dayOfWeek of  
todaysDate & "." buttons {"OK"}
```



Chapter 7:

Scripting Addition Commands and Dictionaries

dateRecord

```
on run
    set todaysDate to (current date) as text
    set theDay to word 1 of todaysDate
    set theMonth to word 2 of todaysDate
    set theDate to word 3 of todaysDate
    set theYear to word 4 of todaysDate
    return {year:theYear, month:theMonth,
        date:theDate, dayOfWeek:theDay}
end run
```

Run the first script. It runs the second script, which places elements of today's date (as supplied by the built-in **current date** scripting addition command) into a record, and returns those values as a result of running that object. The last line of the first script extracts one field of that returned record (the day of the week), and displays some knowledge in a dialog box.

Common Errors

Omitting “script” from the command; forgetting to use an alias class value as a parameter.

Related Items (Chapters)

Launch (6); **choose file** (6); **run** (6);
scripting components (7); value classes (8);
script objects (15).



scripting components

Result

An AppleScript list of all OSA-compatible scripting components installed in the Mac's system.

When to Use

With this command, your script can make sure that a scripting component other than AppleScript is installed in the system before branching to a script that requires that component. For example, if you have the JavaScript OSA component installed in your Mac, a call to the command reveals all components:

```
scripting components
-- result: {"JavaScript", "AppleScript"}
```

Most typically, you would use this command in an **if-then** construction, as in

```
if scripting components contains "JavaScript"
then
    (* run JavaScript-based script *)
end if
```

If you call this command from AppleScript, you can be assured that the AppleScript component will always be listed.

Parameters

None.

Examples

```
scripting components
```



Chapter 7:

Scripting Addition Commands and Dictionaries

You Try It

There isn't much to play with here except entering the command and checking the result.

Common Errors

Forgetting that even a single component result comes back as a list value.

Related Items (Chapters)

Run script (7).



```
store script scriptObject ~  
    in aliasOrFileReference ~  
    [ replacing ask | yes | no ]
```

Result

None.

When to Use

The **store script** command is specialized in that it works only with values that represent compiled script objects—precisely the kind of objects that you get into your scripts via the **load script** command. In other words, you can't store a script unless you've previously loaded an existing script. Why would you want to do that? For a number of reasons.

The need for saving a copy of a script comes from a script object's ability to store data in the form of properties (discussed fully in Chapter 15). It's very possible that a script application will contain updated

information (such as: how many times the script has been run; the date and time of the last database transaction; the latest delinquent customer list) that needs to be propagated to other Macintoshes on the network. A script can automate that process by running the **store script** command after a loaded script has been run and its properties modified.

Parameters

As the first parameter, the *scriptObject* must be a variable that holds a reference to a script object. The **load script** command returns everything that the **store script** command needs.

As for the destination of the saved file, you can supply a path and name to the saved file via the optional **in** parameter. If you omit the parameter, the user sees a Save dialog through which the file may be positioned and named.

You can also use the **store script** command in a script to save itself—valuable for updating property values of a script object not saved as an application (Chapter 15). The parameter for *scriptObject* is the self-referencing **me** variable, as in

```
store script me in file ~  
    "HD:Library:Scripts:MyObject"
```

To compile and save the script the first time, use the file reference so the compiler doesn't choke on an alias to a non-existent file.

Another optional parameter lets you control the



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

user interface for asking about overwriting an existing copy of the file. Hard-wired choices for the **replacing** parameter are **ask**, **yes**, or **no**. Without the parameter, the default is **replacing ask**. In some senses the default is the best choice for knowledgeable users, since the resulting dialog box (Figure 7.6) gives you a chance to locate and name the file differently. But if you're trying to do this updating behind the scenes, the **replacing yes** parameter is a better choice: the name and location of the saved copy will be under script control, and your script will know where to find the object the next time it's needed.



Figure 7.6. *The replacing ask alert.*

Given the number of user cancellations that can occur during a user-controlled save action, it is a good idea to incorporate the **store script** inside a **try** statement with trapping for errors, as in:

```
try
    store script theLib in alias targetLib
    replacing ask
on error errMsg number errNum
    if errNum is -128 then -- user canceled
        display dialog "Network update was
```

```
not made!"
    end if
end try
```

If the user cancels, you can then handle what needs doing to maintain the integrity of the system you've scripted.

Examples

```
store script transactionLog in alias ~
    "Server:Shared Objects:Transaction Log" ~
    replacing yes
```

You Try It

It will be pretty boring, since you can't see much of what's going on behind the scenes, but enter and run the following script in Script Editor. It loads a library (any previously saved script file) of your choice, and saves it back to itself (unless you save a copy elsewhere). Since the script below doesn't do anything to a library in the process (and it may or may not have any properties to change anyway), the contents of the library you choose won't be harmed (unless there is a disk error, so make sure you have a backup before running the script).

```
set targetLib to (choose file with prompt
    "Select a library:") as string
load script file targetLib
set theLib to result
try
    store script theLib in alias targetLib
    replacing ask
on error errMsg
    display dialog errMsg buttons {"OK"}
end try
```




Chapter 7:

Scripting Addition Commands and Dictionaries

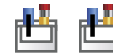
Common Errors

Specifying an object of the wrong class for the *scriptObject* parameter; forgetting the alias word in the *aliasReference* parameter.

Related Items (Chapters)

Load script (7); file and alias references (9).

User-Interface Commands



```
choose from list itemList ~  
  [ with prompt promptString ] ~  
  [ default items itemList ] ~  
  [ OK button name string ] ~  
  [ cancel button name string ] ~  
  [ multiple selections allowed Boolean ] ~  
  [ empty selection allowed Boolean ]
```

Result

A list class value of the full text of zero or more items selected if the user clicks the OK button, or Boolean **false** if the user clicks the Cancel button.

When to Use

The **choose from list** command presents a dialog window containing a list of plain text items supplied as a parameter to the command (see Figure 7.7). The height of the dialog window is determined by the available screen space in the user's video display (allowing for "breathing space" above and below the window). If the list of items is greater than the number that can appear within the window size, the list turns into a scrollable list.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

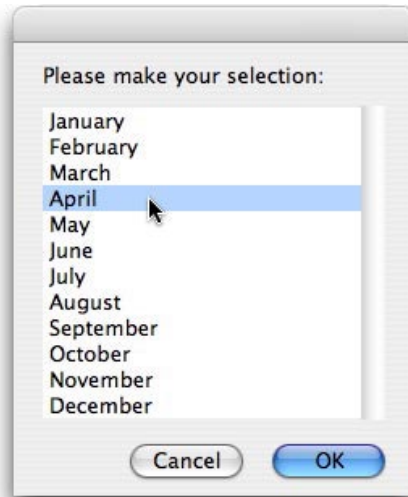


Figure 7.7. *The choose from list dialog box.*

Because the value returned can be one of two value classes (Boolean or list), you should process the resulting value carefully, in anticipation of the user clicking on either dialog button. For example, if you wish to perform further processing only if the user makes a selection, ensure that the result of the command is not **false**:

```
set colorChoice to choose from list {"red",  
  "yellow", "green", "purple", "blue"} with  
  prompt "Select a Base Color:"  
if colorChoice is not false then  
  -- process the user choice here  
end if
```

Also remember that when the user clicks the OK button, the value returned is a list value class. Therefore, if the user selects only one item (the

default behavior of this dialog), extract the selected text by getting item 1 of the result. Some commands accept one-item lists as the value class of the single item. For instance, the canonically correct syntax to display the text of a selected item from the previous example would be:

```
display dialog item 1 of colorChoice
```

But the **display dialog** command accepts a list containing a single string as if it were a string value:

```
display dialog colorChoice
```

For optimum compatibility, however, use list item references even for a single item.

Parameters

The first parameter is required. It consists of a list of string values that you wish to display in the list dialog. Each string appears as its own line item in the dialog. You cannot insert images or icons in these locations, or assign any value to an item other than its displayable text.

A host of optional parameters give you substantial control over a variety of user interface elements in the dialog box. As with most other dialog-related scripting addition commands, the **with prompt** parameter here lets you insert a custom prompt message to replace the default “Please make your selection:”. Supply any string value you like, including fairly long instructions. Unlike most other dialog prompts, however, long text is displayed as multiple lines.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Your script can pre-select one or more items in the list presented to the user. For each pre-selected item, supply the same string value in the list value assigned to the **default items** parameter as the one in the required parameter for the command. Be aware, however, that items are treated in a case-insensitive manner. For example, if you have both “green” and “Green” in the list presented to the user, any default item with upper or lower case letters for g-r-e-e-n will cause the first item matching those letters to be pre-selected by default.

To supply your own labels for the two buttons at the bottom of the dialog, provide string values for either or both **OK button name** and **cancel button name** parameters. The default values for these two buttons are, as their parameter names imply, “OK” and “Cancel.”

If you wish to allow or encourage users to select more than one item from the list (by Command- or Shift-clicking on items), set the **multiple selections allowed** parameter to **true**. Or use the alternate syntax, **with multiple selections allowed**.

And finally, the default behavior of the dialog is to disable the OK button (or whatever you call it) until the user selects an item. But if your script needs to accommodate the selection of no items (rather than just a Cancel), set the **empty selection allowed** parameter to **true**. With this parameter turned on, if the user fails to click on an item and

clicks the OK button, the command returns an empty list (**{}**).

Examples

```
choose from list -  
  {"Larry", "Moe", "Curly", "Shemp"}
```

```
choose from list -  
  {"0-17", "18-45", "46-65", "65+"} -  
  with prompt "Select your age group:"
```

```
choose from list -  
  {"Mac OS", "Windows", "Linux", "Unix", "Other"} -  
  default items {"Mac OS"} with prompt -  
  "What is your favorite operating system?"
```

You Try It

Enter and run each of the examples above in Script Editor with the Result pane open. Monitor the results for your choices, as well as what happens when you click the Cancel button.

Common Errors

Incorrect formatting of lists with misplaced quotes around each string item, or commas in the wrong places; not anticipating the user clicking the Cancel button.

Related Items (Chapters)

List value class (9).



Chapter 7:

Scripting Addition Commands and Dictionaries



```
display dialog string ↵
    [ default answer string ] ↵
    [ buttons buttonList ] ↵
    [ default button integer | string ] ↵
    [ with icon integer | string ] ↵
    [ giving up after seconds ]
```

Result

A record (of class **dialog reply**) consisting of one to three labeled fields depending upon parameters passed with the command. The possible fields and their values are:

```
button returned:buttonNameString
text returned:string
gave up:Boolean
```

When to Use

The **display dialog** scripting addition command is an important user interface device when your scripts run without the aid of an interface builder (such as AppleScript Studio). In lieu of a full debugging environment, it also helps scripters read intermediate values of variables while a script is under construction.

With one command, you control whether the dialog is a read-only dialog or is the kind that provides a field for the user to enter some information via the keyboard. Even a read-only dialog can be a user-input interface device, because the command lets a script display up to three buttons, each of which can contain a different label—your script can then process a

different branch or subroutine for each clicked button.

When the dialog appears as a read-only type of alert, the returned value is a record with a single property, as in

```
{button returned:"OK"}
```

The value of the **button returned** field is the text your script had assigned to the clicked button. Only one button can be returned for each invocation of the dialog.

For a user-input dialog, the returned record has two properties, as in

```
{text returned:"David", button returned:"OK"}
```

You can then use the powers of record class values to extract the information that comes back. Here's a series to demonstrate an example for a read-only dialog, where subsequent processing branches according to the value of the button clicked:

```
display dialog "Make an ice cream flavor
choice:" buttons {"None", "Vanilla",
"Chocolate"} default button "Chocolate"
set favFlav to button returned of result
--<<returned value
if favFlav is "Vanilla" then
    display dialog "Potentially boring."
    buttons {"OK"} default button 1
else if favFlav is "Chocolate" then
    display dialog "Now you're clicking!"
    buttons {"OK"} default button 1
else
    display dialog "Sorry you don't like
these." buttons {"OK"} default button 1
end if
```



Chapter 7:

Scripting Addition Commands and Dictionaries

Here's another version, this time for a dialog that asks the user for some input:

```
display dialog "Enter a number between 1 and  
10:" default answer ""  
set userValue to (text returned of result) as  
real  
if (userValue < 1) or (userValue > 10) then  
    display dialog "That value is out of  
    range." buttons {"OK"} default button 1  
else  
    display dialog "Thanks for playing."  
    buttons {"OK"} default button 1  
end if
```

The answer supplied by the user (the **text returned** value of the result) comes back as a string, which we coerce to a real number. We do this because the **if-then** construction tests whether the number is within the specified range.

The button labeled Cancel in a dialog has a special power. When a user clicks it, script execution stops, but not before an error message ("User canceled.") and number (-128) are sent back to the script. If your script needs to perform some other action as a result of a user cancellation, build the **display dialog** command within a **try** statement, and trap for the error (Chapter 12).

Exercise care when building dialogs into your applications. Too much of a good thing gets in the way of the user. Good practice also dictates that dialogs allow the user to make a choice about proceeding with an action. Provide a Cancel button to let the user back out at the last minute. If you

need more complex dialogs for your scripts, consider applying an interface builder to this application.

Parameters

A key to success with this command is understanding its many parameters. Only one, the hard-wired string that appears in either style of dialog, is required. It can be any string, but exercise care that the amount of text doesn't overrun the screen space (height) in which the full text and disposition buttons can be displayed. A dialog that is too tall for the display may obscure some of the text and action buttons.

Little known is that the string can contain return characters to allow the script to control to some degree the formatting of text within the dialog (beyond the automatic word wrapping). There are a few different techniques, but one of the most readable styles builds strings with the **return** constant placed where necessary, as in:

```
display dialog ~  
    "From one line..." & return & "to two."
```

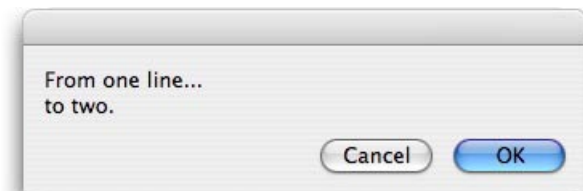


Figure 7.8. Multiple line text in a dialog.

As an alternative, you can use the `\r` special



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

character, which accomplishes the same thing, but at compile time often divides the string in the script into two lines, making the script look funny and more difficult to read:

```
display dialog "From one line...\rto two."  
-- compiles to...  
display dialog "From one line  
to two."
```

If you want the dialog to contain a field for the user to enter some text, then the command must contain a **default answer** parameter. Many dialogs supply default answers for users, making it a simple task of clicking the OK button or pressing the Return key to accept what's there. That default answer string is an argument to the **default answer** parameter. But if you want the field to be empty when the dialog opens, then that parameter value must be an empty string, as in:

```
display dialog -  
    "Enter your age:" default answer ""
```

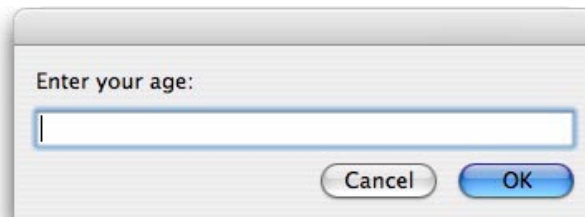


Figure 7.9. Dialog box with a field for user entry.

You have many choices about how to handle the

buttons that appear in these dialogs. If you do nothing about it in the parameters, the dialog displays a Cancel and an OK button, with the OK button being highlighted as the default button (the one that takes effect when the user presses Return or Enter). By providing one, two, or three string values in a list class value for the **buttons**, however, you can change the contents and order of all the buttons.

Buttons start filling in from the right (i.e., one button appears at the right edge of the dialog), and their order in the dialog from left to right is the same as the order of items in the list. Here are some examples of the **buttons** parameter settings and the corresponding button appearances:

```
display dialog "Are you OK?" buttons {"Yes"}
```



Figure 7.10. One dialog button.

```
display dialog -  
    "Are you OK?" buttons {"No", "Yes"}
```



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**



Figure 7.11. Two dialog buttons.

```
display dialog "Pick a car company:" buttons  
{"General Motors", "Chrysler", "Ford"}
```

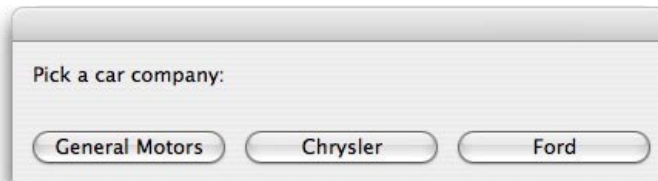


Figure 7.12. Three dialog buttons.

The size of the buttons is governed by the amount of text in the longest button. But there is a limit (defined more by space of the proportional font than by character count), so you should try your long button's names and make sure they don't look too goofy or aren't longer than the longest button can possibly be.

Note that the **buttons** parameter says nothing about which button, if any, should be highlighted as the default button: that's what the **default button** parameter is for. The argument for this parameter can be either a number representing the number of the button (with the leftmost button

being number 1) or a string representing the text of a particular button in any location, as in:

```
display dialog "Pick a car company:" buttons  
{"General Motors", "Chrysler", "Ford"}  
default button 3  
-- or  
display dialog "Pick a car company:" buttons  
{"General Motors", "Chrysler", "Ford"}  
default button "Ford"
```

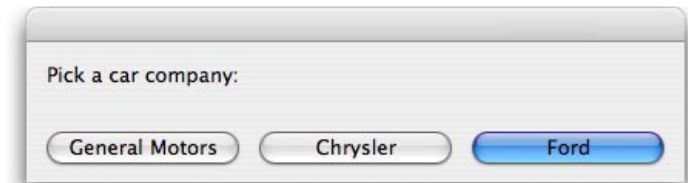


Figure 7.13. Three dialog buttons with default.

The **with icon** parameter allows your script to display one of several possible icons in the dialog. These are useful primarily for read-only, alert kinds of dialogs, since the icon can often convey subtle meanings about the message or choice to be made.

Although you can occasionally display a limited range of icons associated with an application, the most reliable icons are the three built-in icons available to scripts running in all contexts. These three icons have meanings according to the Macintosh User Interface Guidelines, as summarized in Table 7.6.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Icon	Name	Number	Meaning
	Stop	0	It's not recommended to go any further down the original path. (Offer ways out for the user.)
	Note	1	Information for your benefit, but nothing serious.
	Caution	2	Something may be overwritten or lost if you proceed. Confirm that you want to continue.

Table 7.6. *Built-in icon values and meanings.*

To display one of the icons, supply either the number or constant value to the **with icon** parameter, as in:

```
display dialog "Do you want to delete this  
file?" with icon caution
```

You can mix an icon with an answer field in a dialog box, but the icon takes up width that may be better devoted to the user field. Also, this combination is highly unusual in Macintosh usage.

If you happen to know (or know how to use resource inspection tools to uncover) the number of an application's 'cicn' icon resource, you can use that icon in an AppleScript dialog window when the application is the target of a **tell** block. Use the resource ID number as the value for the **with icon** parameter. Not many applications have icons of this type anymore, but some established products may still have some buried within the code. For instance, the following script directed to BBEdit 7 might bring you a smile:

```
tell application "BBEdit"  
    display dialog "Have a nice day."  
    with icon 5001  
end tell
```

The last optional parameter for **display dialog**, **giving up after**, lets your script close a dialog box after a specified number of seconds (the integer value assigned to the parameter). This prevents a dialog box from completely halting a script that may run without a user present to interact with dialog boxes. When this parameter is passed to the command, the returned value bears the **gave up** field, whose value is **true** if the time expired and the dialog box closed automatically. When the **gave up** field is **true**, the **button returned** field is an empty string (and, if specified, the **text returned** field contains the supplied **default answer** string).



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Examples

```
display dialog "Howdy, pardner!"

display dialog "What is your name?" default
  answer ""

display dialog "What color would you like?"
  buttons {"Red","Green","Blue"} default button
  1 with icon 1

set theAnswer to text returned of (display
  dialog "What is the meaning of life?" default
  answer "Dunno")
```

You Try It

Enter and run the following scripts in Script Editor. When prompted, enter text into a field, and/or click a button, watching for the records displayed in the Result pane:

```
display dialog "How ya doin'?"

display dialog "What city do you live in?"
  default answer ""

display dialog "Pick a card..." buttons
  {"Jack","Queen","King"}

display dialog "Place your bets..." buttons
  {"Win","Place","Show"} default button 1

display dialog "An error has occurred. Please
  try again." with icon stop buttons {"OK"}
  default button "OK"
```

Common Errors

Leaving out a word in one of the optional parameter labels; forgetting to put button names in a list;

forgetting to specify a default button.

Related Items (Chapters)

List value classes (9); error trapping (13).



```
say text [ displaying text ] ~
  [ using voiceName ] ~
  [ waiting until completion Boolean ] ~
  [ saving to aliasOrFileRef ]
```

Result

Nothing.

When To Use

The Macintosh has a long history of embedding speech synthesis technology within its system. Its speech capabilities have improved through the years, and you have a chance to explore customized application possibilities by way of AppleScript's standard additions **say** command.

In addition to the obvious application of speech synthesis for a vision-impaired Mac user, you can use speech to “read” aloud any text that you might otherwise miss on a screen or would prefer to have read to you while you attend to other matters. Verbal alerts can also be less intrusive as debugging cues than the **display dialog** command, which halts all execution in its tracks and demands your attention. Instead, a verbal signal of processing can let you know what's happening in a script you're trying to debug.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

As a practical example, the following script works with the Microsoft Entourage email program to download mail from the server and read aloud the key items from all unread messages in the Inbox folder. Entourage doesn't like the **say** command within its own **tell** block, so the command is broken out in a separate handler, invoked as needed from the main block that extracts pieces of each message, and assembles one long string to be vocalized.

```
tell application "Microsoft Entourage"
    connect to POP account "Internet Connection"
    repeat with oneMsg in messages of folder
        "Inbox"
            if read status of oneMsg is
                untouched then
                    my speakit(((time sent of
                        oneMsg) as string) & ". From " & display name
                        of sender of oneMsg & ". Subject " & subject
                        of oneMsg & ". Message " & content of oneMsg)
                    end if
                end repeat
            end tell

    on speakit(txt)
        say txt
    end speakit
```

You could even extend this application with a recent parameter addition to the command to save the vocalized sound to a file, which could then be brought into your iTunes library (and then synched up with your iPod for verbal email on-the-go).

Parameters

One required unlabeled parameter is the text that you wish the Mac to speak. Any text or string value works here, and the speech synthesis mechanism of the Mac OS does its best to turn that text into understandable speech.

You can influence the speech engine by way of what are called **intonation characters**—special control codes and phonetic spelling of words to improve intelligibility. Finding information about these codes can be a challenge, but one archived document at Apple's developer Web site contains some helpful information (http://developer.apple.com/documentation/Hardware/Developer_Notes/Macintosh_CPUs-68K_Desktop/Centris_660AV_Quadra_840AV.pdf). Of course, such codes won't help if the source of the text comes from an external source (such as an email message or Web site page), but if you build some hard-wired speech into a string being vocalized, you may wish to improve the quality of those known bits.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Details about working with the intonation characters lie outside the scope of this book, but I provide one example that will help you interpret and apply the information provided in the PDF file noted above. The following statement begins with plain text and then switches the input mode to phonetic (Speech Manager directives and their values are placed inside double square brackets). The phonetic characters put the emphasis (the 1 character) on the first phoneme, which is also lengthened (>) slightly. After the second phoneme, the input mode switches back to the default text mode for some more plain language interpretation. Next comes a 200 millisecond silence, followed by a switch into a mode that treats the ensuing string as literal characters, spelled out one by one. After the last spelled character, the mode switches back to the default normal mode, where the final period influences the downward pitch of the sentence. And here's the statement:

```
say "I love my [[inpt phon]]1>AY=pAAd  
[[inpt text]], spelled [[slnc 200]]  
[[char LTRL]]ipod[[char NORM]]."
```

If you have Speech Recognition turned on (via the Speech preference pane), a graphic symbol of a microphone appears on the screen. The text being spoken appears in a temporary box below the microphone icon while the sound is playing. However, if your spoken text includes some intonation characters or funky spellings to improve intelligibility, you'll want the human-readable text to appear in the display. You can display any text

you desire while the **say** command is operating by specifying a text value to the optional **displaying** parameter.

Another setting in the Speech preference pane (in the Default Voice tab) is a choice of the synthesized voice to be used for speech. You can select any one of the voices for a **say** command without changing the default setting. Note that Speech Recognition must be turned off for this to work. Specify one the names from the list as a string value to the **using** parameter, as in:

```
say "How low can I go?" using "Ralph"
```

With Speech Recognition turned on, you can order the **say** command not to wait until the speech completes before allowing subsequent processing to occur. The default behavior is for the command to halt processing while the speech is sounding. But if you set the **waiting until completion** parameter to **false**, script processing continues while the speech is audible.

One final parameter, **saving to**, forces the **say** command to save the audio of the spoken output to a file whose path you supply as a parameter. To facilitate the file being played through programs such as iTunes or the QuickTime Player, save the file with the .aiff extension. When you save the audio to a file, the sound does not play when you invoke the **say** command.



Chapter 7:

Scripting Addition Commands and Dictionaries

Examples

```
say "Good evening ladies and gentlemen."
```

```
say "I am not a number." using "Trinoids"
```

```
say txt saving to (path to desktop -  
as string) & "mail.aiff"
```

You Try It

The **say** command is easy enough to try out with not only the examples shown above, but with your own imagination. Have a blast!

Common Errors

The required parameter is not a string or text value class; applying the **say** command inside an application's **tell** block and the application doesn't like it; specifying a voice name not installed on the current computer; supplying an invalid path for the file being saved.

Related Items (Chapters)

Display dialog (7); value classes (9).

File Commands

Although other sections of this chapter discuss commands alphabetically, this section organizes the discussion functionally. First come commands for opening and closing files; then come commands for reading from and writing to such files; then come two related commands that assist file reading and writing.



```
open for access -  
fileOrAliasReference -  
[ write permission Boolean ]
```

Result

File reference number (integer) generated by the system. You can use this number—which will likely be different each time you issue the command—for further commands that act on the file, including reading, writing, and closing.

When to Use

To access or modify the data from any file directly from a script, you must first open the file with the **open for access** command. This is also the command you use to create a new file by specifying a path and file name to a file that does not yet exist. This scripting addition creates files of creator 'txtt' and file type 'TEXT'.

That doesn't mean, however, that only text files can be opened with this command. Existing binary



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

files may also be opened, but you had better know what you're doing with the data in such files before mucking around with them. In fact, when it comes to opening and writing to documents created by applications, you should rely on AppleScript support within the applications to work with file data. Use these scripting addition file commands for simpler file reading and writing tasks (such as appending to a log file for your script's activity) or preserving AppleScript data structures (e.g., lists or records) in their native form for use in subsequent sessions.

It is very important to balance each **open for access** command with a **close access** command (below). Keep the file open only as long as needed for reading from or writing to the file. If you receive an error that the file can't be opened because it already is, you'll have to write and execute a one-line AppleScript statement using the **close access** command whose parameter is the path to the existing file. Once the file is closed, it can be opened again by script without complaint.

Parameters

The required parameter is the pathname of the file to be opened. While you may use either a file or alias reference, you will be able to compile a script for an as-yet uncreated file only with a file reference. Use alias references only for pre-existing files. If the file is to be created with the command, the name you use for this parameter is the one assigned to the new file (see the **choose file name** command, above, for

user interface help in this regard).

Unless specified with the optional **write permission** parameter (a Boolean value), a file opened with the **open for access** command is set to be read-only. Before you can write to the file, it must be opened with the **write permission** parameter set to **true**. You cannot change the write status of a file once it is open: close it and reopen it with the desired write permission. If you enter the **write permission** parameter with the **true** or **false** Boolean value, the compiler converts the statement to the **with** or **without** version, as in:

```
open for access file "Test File" ~  
    write permission true
```

becomes

```
open for access file "Test File" ~  
    with write permission
```

Examples

```
open for access file ~  
    "Macintosh HD:Users:jane:Desktop:Memo"
```

```
set fileRef to ~  
    (open for access (choose file name))
```

```
open for access alias "HD:Names" ~  
    with write permission
```

You Try It

Enter and run the script below, watching the process in the Event Log pane of Script Editor:



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

```
set theFile to open for access file "Test File"  
with write permission  
write "Howdy" to theFile  
    -- using file reference number  
close access theFile
```

Note the integer value returned by the **open for access** command. The number increments each time you run the statement, but its use is limited to related file commands. Do not regard it as a property worth saving for later.

This script creates a file named “Test File” in the startup disk root level. Open the file with TextEdit or any other editor to see the results. Delete the file when you are finished.

Common Errors

Using an alias reference for a new file; not including the entire pathname for the file; forgetting to set the **write permission** parameter to **true**; opening a file that was not closed the last time the script ran.

Related Items (Chapters)

Close access (7); **read** (7); **write** (7); **choose file name** (7).



```
close access fileReference
```

Result

None.

When to Use

Any time a script opens a file for access, the script should also close the access once all file reading and writing has been completed. If you don't close the file, you will not be able to reopen it with the **open for access** command. While developing scripts that use the file scripting addition commands, script execution will halt before the file is successfully closed, causing the next **open for access** command to fail with an error. A handy tool is a separate Script Editor window with the following one-line script:

```
close access (choose file)
```

Whenever you receive an error that a file is already open, run this script and select the target file. You will then be able to run your development script again.

Parameters

An acceptable *fileReference* parameter may be a pathname (in either file or alias reference style) or, more easily, the file reference number that comes back as the result of the **open for access** command. Because file pathnames get clumsy when used multiple times within a script (and they are prone to typographical errors when entering them), the file reference number mechanism makes the job of working with files (including using the other file scripting additions) much easier. As demonstrated in the **open for access** command discussion, capture the reference number in a variable upon



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

opening a file, and then use that variable for the file reference in all commands directed to that file. Because the number will likely be different each time you open the file, don't rely on any hard number for the file reference: let the variable do the work.

Examples

```
close access myFile
-- myFile variable is a reference number

close access alias "MacHD:Memo to Linda"
```

You Try It

See the “You Try It” section in the **open for access** command, above, for an exercise of the **close access** command.

Common Errors

Forgetting the word “access” in the command;
typographical error in a file or alias pathname.

Related Items (Chapters)

Open for access (7); **read** (7); **write** (7).



```
read fileReference -
[ from startInteger ] -
[ for numberOfBytes | to endInteger | -
until includedDelimiter | -
before excludedDelimiter ] -
[ as dataType | class ] -
[using delimiter[s] delimiterList ]
```

Result

Data from the file referred to in the *fileReference* parameter. The amount of data, position of data from within the file, and form of the data are determined by various optional parameters.

When to Use

This is the command that retrieves data from a file previously opened via the **open for access** command. Files that may be read are not limited to text files. Binary data files may also be read, including files containing AppleScript data, such as lists and records, in their native format.

While Apple's *Scripting Additions Guide* goes to great lengths to demonstrate using the Read and Write commands to replicate a flat-file database, you are far better off using an application, such as FileMaker, to work with substantial data collections. For simple data collections that don't need a lot of maintenance, consider creating a list of AppleScript records and writing/reading the data in that form using the scripting addition file commands. Then, when accessing the data within a script, use AppleScript list and record access syntax to zero in on a particular data point (e.g., the value of a property of a particular record item within the list).

The bottom line, therefore, is to use these file commands for quick access to small collections of data that apply to the script (whether the data be rigidly structured or not). Leave the job of sophisticated database management to scriptable



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

applications better suited to the task.

Parameters

The range of parameters to the **read** command may seem daunting, but they're not that complex when you examine the job of the command. For the most part, you'll use only a couple of parameters at a time.

A helpful concept to understand about reading a file is that of the *file mark*. When a file opens for access, the Macintosh automatically places this mark immediately before the first character of the file.

The mark indicates where the next **read** command will begin grabbing a copy of data from the file (reading does not disturb the contents of a file in any way). As you'll see in the next few paragraphs, the **read** command can be instructed to read a specific sequence of characters. After one **read** command, the file mark moves ahead in the file, past the characters grabbed by the first **read** command. The next **read** command then starts where the marker is—somewhere further within the file. This file mark advancement continues until the mark reaches the end of the file, at which point the **read** command returns no further data.

The file mark moves only in one direction: from the beginning toward the end of a file. Therefore, if you read a segment near the end of a file and then need another segment from the beginning, you cannot move the file mark back to the beginning of the file during this access. You may, however, close access to the file and reopen it to re-set the marker to the

beginning of the file.

The only required parameter to the **read** command is a reference to the file to be read. A file reference number obtained from the **open for access** command is the best bet, but you may also read a file without opening it for access by simply providing a valid file reference to an existing file. If you specify no more parameters, the command reads the entire file, and returns its contents as the result, ready for assignment to a variable, as in

```
set myData to (read myFile)
myData
-- result: [entire contents of the file]
```

In file mark terms, the marker reaches the end of the file after a single **read** command (with no other parameters). Any subsequent reads return End of File errors.

To begin reading the file from a character other than the first character, use the **from** parameter. Specify the number of bytes from the beginning of the file (the first byte is byte number 1) where the file mark should be placed prior to reading. For text files, one byte equals one ASCII text character. For example, to read a file beginning with the 20th character,

```
set myData to (read myFile from 20)
myData
-- result: [file contents from
-- character 20 to end]
```

If the need is to read just the last x characters of a file, you may use a negative number, relative to the



Chapter 7:

Scripting Addition Commands and Dictionaries

very end of the file. For instance, to obtain the last 50 characters of a file,

```
set myData to (read myFile from -50)
myData
-- result: [last 50 bytes of the file]
```

Be aware, however, that if the file ends with a carriage return (as most database-style files do), the final carriage return counts as the very last character, and must be taken into account when figuring the number of bytes to read.

Without further parameters specifying an end to the read segment, the file mark is at the end of the file after this command. No subsequent reads are possible without closing and re-opening the file.

Whether the file is read from the very beginning or from some other starting point (with the **from** parameter), you may specify one of four types of limits on any **read** command. They are:

```
for numberOfBytes
to endInteger
until includedDelimiter
before excludedDelimiter
```

The argument to the **for** parameter is a positive integer representing the number of bytes to be read from the file mark. Like all four of these choices, the **for** parameter may be used with or without the **from** parameter. Two examples are:

```
set myData to (read myFile for 100)
set myData to (read myFile from 201 for 100)
```

Here, after the first **read** command, the file mark

is located immediately before byte 101 of the file. If the next **read** command were identical, it would retrieve bytes 101 through 200. But in this example, we start the next read at byte 201 for another 100 bytes. After this second command, the file mark is located before byte 301.

Another way to express the end point of a **read** command is with the **to** parameter and an integer representing the byte of the last character to be read. If the integer is positive, then the count is from the beginning of the file; for a negative argument, the count is from the end of the file. See the Examples section, below, for samples of this parameter.

While we said earlier that the file marker moves only from front to back, a combination of the **from** and **to** parameters, coupled with negative number produce some interesting results. If the last seven characters of a file were “Kennedy,” notice the results from the following command:

```
set theData to (read myFile from -1 to -7)
theData
-- result: "ydenneK"
```

Despite the backward counting motion, the file mark remains at the further point along the file (in this case at the end of the file) after the command, even though the reading started from that location.

The **before** and **until** parameters both require arguments consisting of a single character. The **read** command starts from the file mark and grabs a copy of all bytes from that point to the first instance



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

of that character (or end of the file, whichever comes first). The distinction between the two parameters is how the **read** command treats this delimiting character. In the case of the **before** parameter, the delimiter is not returned as part of the data; for the **until** parameter, the delimiter is part of the returned data. Importantly, specifying the **before** parameter causes the file mark to jump *past* the delimiting character, so that the next **read** command starts *after* the delimiter.

All of this makes sense particularly if you are reading a file with delimiters between chunks of data, and you want to retrieve just the data. In fact, you may use constants (such as **return**, **space**, and **tab**) as delimiter arguments for these parameters, as in:

```
set theData to (read myFile before tab)
```

The final optional parameters for the **read** command (**as** and **using delimiter[s]**) let you determine the form of the data from a file for further manipulation by your script. Acceptable value classes are: list, record, integer, text, real, short, Boolean, and data. There are limitations to many of these, primarily revolving around the suitability of the data for coercion into a particular value class. For example, a text file containing words can't be coerced into an integer class; similarly, only data previously written as an AppleScript record can be read as a record.

Perhaps the most common coercion will be converting a comma- or tab-delimited file into an

AppleScript list. For illustration purposes, we'll use a tab-delimited file containing four "fields" for each U.S. state (full state name, the year the state entered the Union, postal abbreviation, and state capital city—and one top row with column headings). It is located in the companion files for Chapter 7 titled `USStates.tab` (we'll also assume that the following examples are surrounded by the requisite **open for access** and **close access** commands, and that the file reference number is in a variable called **myFile**). The following statement relies on the default value class (text) in the data read from the file (the first row of data up to the return character):

```
set theData to (read myFile until return)
(* result: "State      Entered Union
Abbreviation    Capital
               " *)
```

To retrieve data from this file such that each row (record) is an item of an AppleScript list, we can direct the **read** command to recognize the return character as a delimiter between list items:

```
set theData to (read myFile as text using
delimiter return)
(* result: {"State      Entered Union
Abbreviation    Capital",
"Alabama 1819 AL    Montgomery", ...
"Wyoming 1890 WY    Cheyenne"} *)
```

The tab characters between items in each line's "record" are preserved in this arrangement, and this may be the way we want it. But it is also possible to pass two delimiters (in the form of a list) as arguments to the **using delimiters** parameter,



Chapter 7:
**Scripting
 Addition
 Commands and
 Dictionaries**

such that the **read** command recognizes each of the named delimiter characters as item delimiters for the AppleScript list that comes back:

```
set theData to (read myFile as text using
  delimiters {return, tab})
(* result:{"State", "Entered Union",
  "Abbreviation", "Capital", "Alabama",
  "1819", "AL", "Montgomery", "Alaska",
  ... "Cheyenne"} *)
```

Examples

```
read myFile

read myFile from 25

read myFile for 20

read myFile to 20

read myFile from 25 for 50

read myFile from 25 to 74

read myFile before tab

read myFile until return

read myFile from -20
```

You Try It

Because we don't know the location of various files on your hard disk, we'll have to leave it to you to locate the USStates database file (from the Chapter 7 folder) on your disk. Activate the Event Log pane of Script Editor, enter the following script into the top pane, and then run the script, watching the returned

values in the Event Log pane:

```
set myFile to (open for access (choose file))
read myFile for 11
read myFile from 20 to 32
read myFile from 33 for 9
read myFile until return
read myFile before return
read myFile until return as text ~
  using delimiters {tab, return}
read myFile from -10
close access myFile
```

Common Errors

Inserting the word “file” between the read command and file reference number; reaching end of file; trying to read more bytes than there are in the file.

Related Items (Chapters)

Open for access (7); **close access** (7);
write (7).



```
write expression to fileReference ~
  [ for numberOfBytes ] ~
  [ starting at integer ] ~
  [ as class ]
```

Result

None.

When to Use

Use the **write** command to modify a file opened by the **open for access** command. Before you may write to the file, the **open for access**



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

command must include the parameter that allows write permission. Also see the discussion in the **read** command about the kinds of files and value types you may work with from AppleScript directly, and when scriptable applications are better vehicles for reading and writing data.

The **write** command operates by default in overwrite mode. At the same time, multiple writes to a file within the same open-close block cause the writing to continue on from the next previous point (see the discussion about the file mark, below). If you wish to modify data within a file previously written by AppleScript, you should first read all the data into a variable, massage the data within the variable, empty the file (via the **set eof** command described below), and commit the entire freshly revised data block to the file in one **write** command.

Parameters

The **write** command requires at the minimum parameters containing the data to be written to the file and a reference to the file. The data can be any data that is appropriate for the file. That may include a tab-delimited text record that you've assembled in your script or from other documents, plain text, binary data extracted from other sources, or any AppleScript value class, such as a list or record (including a list of records).

Writing AppleScript value classes brings up an important point about the **read** command when reading such data. Consider the following script,

which writes an AppleScript list to a file:

```
set myFile to (open for access file "Test File"
    with write permission)
set eof myFile to 0
    -- empty anything previously in the file
write {30, 40, 50, "forty"} to myFile as list
close access myFile
```

If you were to open the file in a text editor, this is what you would see:

```
list    long    -long    (long    2TEXT    forty
```

This is how a text editor displays the binary data that AppleScript uses to write a list to a file. If you then perform a plain **read** on the file, your script would grab exactly that text—something your script would have no use for. But that's why the **read** command contains a parameter that lets you retrieve AppleScript data in a particular value class. Therefore, to retrieve this list from a file in its original list format, the script would be:

```
set myFile to (open for access file "Test
    File")
set myData to (read myFile as list)
close access myFile
myData
    -- result: {30, 40, 50, "forty"}
```




Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

If you are using the scripting addition file commands to preserve AppleScript records, and if the script needs to add a record to the file each time the script runs, you should manage the data as a list of records. In doing so, you need to pay attention to value classes when you append the new record to the data. The following example creates a record in a variable named **oneRecord**. The list of previously saved records is read from the file, the new record is appended to it, and the combined result is written back to an emptied file. Pay special attention to the curly braces and list coercions occurring within this script.

```
set oneRecord to {theDate:current date,
choice:dlogResponse}
set myFile to open for access file
"Test File" with write permission
if ((get eof myFile) is 0) then
    -- file is empty so use only the
    -- new record list item
    set fileData to {oneRecord}
else
    -- read file and append
    -- new record to data
    set fileData to (read myFile as
list) & {oneRecord}
end if
-- clear file
set eof myFile to 0
-- write all data to file
write fileData to myFile as list
close access myFile
```

Back to parameters, the file reference may be any file or alias reference, or most efficiently, a file reference

number returned by the **open for access** command, as demonstrated above.

When you write to a file, you must pay attention to the file mark, just as you do for reading. Upon opening the file, the marker is at the very beginning of the file. If the file contains data, and you write data to the file, the **write** command overwrites existing data for only the length of the data you're writing. Notice that it overwrites, rather than inserts, data. But that also means that if the file contains 20 bytes, and you write 10 bytes, bytes 11 through 20 of the original data remain in the file.

If your goal is to completely overwrite a file, however, you can do so with the help of the **set eof** command (detailed below). This command essentially lets you determine where the end of the file is ("eof" stands for "end of file"). By setting the eof to zero, you tell AppleScript to delete everything in the file, cleaning the slate for your next **write** command.

One optional parameter lets you determine where in a file you wish the **write** command to place your data. The **starting at** parameter requires an integer as an argument. The integer represents the character (counting from the beginning) where the data should be written (or negative numbers from the last character). To append data to an existing file, there is a shortcut for this parameter: substitute the **eof** constant for the integer. The following statement adds a new line of data to a return-delimited file of



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

U.S. presidents:

```
write return & "George W. Bush" ~  
to presFile starting at eof
```

You have a number of ways to delete data from a file, depending on where in the file the data to be deleted is located. In general, however, the task requires reading the file's data into one or more variables, performing text manipulations upon those variables, and then rewriting the entire file again (after setting eof to zero). Hence my recommendation to use scriptable applications that provide easier support for inserting and deleting data in their documents—rather than AppleScript's file access capabilities.

One last parameter, the **for** parameter, lets you write only a portion of a chunk of data to the file. For example, if you have a body of text to write, but you only want to write the first 100 characters of that text to the file, the statement would be something like this:

```
write bodyText to myFile for 100
```

You may also combine the **for** and **starting at** parameters to perform tasks such as overwriting a fixed number of bytes within a file.

Examples

```
write date string of (current date) to myFile  
  
write myText to myFile starting at 25  
  
write myText to myFile starting at -25 for 10
```

```
write myText to myFile starting at eof
```

You Try It

First, set up one Script Editor window to read the file you'll be writing so you can see intermediate results in the Result pane. Enter the following script in one Script Editor window:

```
set myFile to open for access file "Test File"  
set theData to read myFile  
close access myFile  
theData
```

Next, enter and run the following script:

```
set myFile to (open for access file ~  
"Test File" with write permission)  
set eof myFile to 0  
write "My first file writing!" to myFile  
close access myFile
```

Replace the middle two lines with each of the following statements, which build on the data entered with the previous script. Execute each script, and then run the file reading script in the other window to see the results.

```
write "my" to myFile  
  
write ", NOT!" to myFile starting at -1  
  
write " Cool." to myFile starting at eof  
  
write " was a blast. A real trip." ~  
to myFile starting at 22 for 14
```

Common Errors

Inserting the word “file” after the **write** command when referencing a numeric file reference; the file is



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

already open from a previous **open for access** command when an error interrupted the script (see the **close access** command for a solution); accidentally overwriting existing data; failing to clear a file with **set eof** before writing entirely new data.

Related Items (Chapters)

Open for access (7); **close access** (7);
read (7); **choose file name** (7); **set eof** (7).



get eof *fileReference*

Result

Integer representing the number of bytes of data in the file (i.e., the offset of the end of the file).

When to Use

You may obtain the length of an open file with this command. As with other commands that access a file, the file must already be open for the **get eof** command to work. As described in the **read** command discussion, it isn't necessary to specify the end of file if you plan to read the file from the current position of the file mark to the end of the file: the plain **read** command automatically reads to the end of the file.

Parameters

The only parameter is a reference to the file, either by name, alias, or the reference number returned by the

open for access command.

Examples

```
copy get eof myFile to fileLength
```

```
write myData to myFile for (get eof myFile)
```

You Try It

Enter and run the script below several times, choosing different data files from your hard disk each time. Watch the returned values in the Result pane:

```
set myFile to (open for access (choose file))  
set fileLength to get eof myFile  
close access myFile  
fileLength
```

Common Errors

Failing to open the file for access prior to the command; failing to include the *fileReference* parameter; forgetting that “get” is part of the command name, and omitting it.

Related Items (Chapters)

Open for access (7); **close access** (7);
read (7); **write** (7); **set eof** (7).



set eof *fileReference to integer*

Result

None.

When to Use

This command lets you control where you wish



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

the end of an open file to be. That means you may truncate a file to any length, or, by setting the value to zero, empty the file of all data (though this does not delete the file). As demonstrated in the **write** command discussion, above, this command is most often used to empty a file prior to rewriting its entire contents from a script. And, as with any file command that alters the content of a file, the file must be open with write permission enabled for the **set eof** command to work.

Parameters

Two parameters—a reference to a file (file, alias, or file reference number) and an integer—are required. The integer represents the offset value where you want the file to end. All bytes beyond that offset value will be erased from the file.

You cannot specify a negative number to set the eof as a negative offset of the original eof. Instead, use the **get eof** command, as in:

```
set eof myFile to (get eof myFile - 10)
```

Examples

```
set eof myFile to 512
```

```
set eof myFile to 0
```

You Try It

Begin by creating the file reading script from the “You Try It” section of the **write** command, earlier in this chapter. You’ll use this to inspect the sample file after several commands below.

Next, enter and run the script below (in a separate Script Editor window) to establish a starting point for further experimentation:

```
set myFile to (open for access file ~  
    "Test File" with write permission)  
set eof myFile to 0  
write "Now is the time for all good men." ~  
    to myFile  
close access myFile
```

Replace the middle two lines with each of the following statements, which build on the data entered with the previous script. Execute each script, and then run the file reading script to see intermediate results.

```
set eof myFile to (get eof myFile - 1)  
  
set eof myFile to 15  
  
write "." to myFile starting at eof  
set eof myFile to 0
```

Common Errors

Failing to open the file for access and write permission prior to the command; failing to include the *fileReference* parameter.

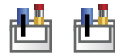
Related Items (Chapters)

Open for access (7); **close access** (7);
read (7); **write** (7); **get eof** (7).



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Debug Commands



```
log string  
start log  
stop log
```

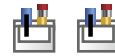
Result

None.

When to Use

Although these commands started life as scripting additions, they are now built into AppleScript. See Chapter 13, Debugging Scripts, about using these commands in conjunction with the Script Editor's Event Log pane.

Date/Time Commands



```
current date
```

Result

A value of class date.

When to Use

This scripting addition gives your scripts the ability to retrieve the current local date and time from your Mac's internal clock. A date class value is more powerful than it may look at first glance. See the discussion about date class values and date arithmetic in Chapter 9.

The returned value looks like this:

```
date "Friday, October 29, 2004 4:23:26 PM"
```

As you learn in Chapter 9, the properties of a date class value let you read and write individual component parts of a date object's date and time, as well as extract string values for display in documents and dialog windows. It is possible to use the data returned from this command for things like date comparisons, as we do in the alarm agent script described in Chapter 15.

Parameters

None.



Chapter 7:

Scripting Addition Commands and Dictionaries

Examples

```
set timeStamp to current date

if (current date) is greater than or equal to
    date "1/1/2006" then
    playHappyNewYear -- calling a subroutine
end if
```

You Try It

Enter and run the following script line, watching the returned value in the Result pane:

```
current date
```

Common Errors

Forgetting that the value is a date class, rather than a string; an improperly set system clock.

Related Items (Chapters)

Date class values (9).



time to GMT

Result

A positive or negative integer representing the number of seconds difference between the local time zone set in the Time Zone tab in your Date & Time preferences pane and Greenwich Mean Time (also known as Coordinated Universal Time, or UTC).

When to Use

For this command to be of any value to a script, the Macintosh on which the script runs must have

its local time zone set correctly. The convenience of having the Mac synchronize its internal clock with a source on the Internet assures greater accuracy. Moreover, recent versions of the Mac OS automatically take care of time differences during the year (such as Daylight Saving Time in North America). Thus, the value returned by time to GMT will vary if you change your local clocks throughout the year.

You may use the date and time objects in AppleScript to help with the math in calculating the number of hours or minutes between your Mac's local time and GMT. For example,

```
(time to GMT)/hours
-- result: -8.0
```

meaning that a Macintosh in the Pacific time zone is eight hours earlier than GMT in non-Daylight Saving Time months. During the summer season, the result would be -7.0 . For time zones east of GMT, return values are positive integers. See Chapter 9's details about the date value class for more about working with date and time arithmetic in AppleScript.

Parameters

None.

Examples:

```
time to GMT
```

```
(time to GMT)/hours
```



Chapter 7:

Scripting Addition Commands and Dictionaries

You Try It

There isn't much to play with here, but by entering the command and checking the result, you will see if your Mac's Date & Time preferences have been set correctly.

Common Errors

Incorrectly set system time zone.

Related Items (Chapters)

Date and Time arithmetic (9).

Clipboard Commands



clipboard info

Result

An AppleScript list of one or more lists, each of which contains a value class type and an integer representing the number of bytes of data occupied by data of that particular value class.

When To Use

When you select some text or graphic in an application program and choose **Copy** from the **Edit** menu, you probably aren't aware that far more information about the copied data occupies the Clipboard than what you see on the screen. For example, select and copy any four characters from text you type into a newly opened TextEdit window. Then run the **clipboard info** command in Script Editor to see the results:

```
clipboard info
(* result: {{styled Clipboard text, 22},
{string, 4}, {uniform styles, 144},
{Unicode text, 8}, {«class RTF », 238}} *)
```

The value returned from the command is a list of lists. Each nested list contains information about a particular data type stored in the Clipboard. The obvious value, string, contains the four bytes (one byte per ASCII character) you copied. Unicode characters are double-byte characters (to accommodate the thousands of characters in



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

the Unicode character set), and thus occupy eight bytes of memory. Another class type is called styled Clipboard text, which contains both the characters and information about how they were styled by the application when the data entered the Clipboard. Although you can view this data, it isn't particularly human-friendly. But when the data meets another application that knows how to work with data of that value, that other application can choose to paste the styled text instead of just the simple characters. These are the kinds of decisions that applications developers make while construction their programs.

Notice that one of the items shown above lists its value class as `«class RTF »`. The display of a class name in this format means that the value class is not defined in AppleScript, any scripting additions, or in the current application. In this case, the value appears to be representing a Rich Text Format (RTF) value. The 238 bytes that this data occupies contains everything that an RTF-aware application would need to insert the styled Clipboard data into the document (e.g., with a Paste command). Typically, the appearance of the chevron (double angle bracket) characters indicates that the item isn't something that you normally script directly (or at least the developer didn't anticipate that you'd be scripting directly). You may, however, still capture data of such value classes in variables and pass the data to another document or application. If you inspect the Clipboard information after copying data from, say, Microsoft Word, you'll see more data classes (all with

four-character identifiers) than you ever imagined.

As to how best to use the information returned by the **clipboard info** command, consider a script whose job is to copy text from one application's document into the document of another application. If the original copying operation is performed by a scripted text selection and equivalent of the Edit menu's Copy command, the Clipboard may have style and other data coming along for the ride. In the destination document, however, the script does not want to inherit the styled text (as might happen with a scripted equivalent of the Paste command). Instead the script inspects the Clipboard's contents to find out if there is any plain string or Unicode data in there. If so, then the script takes only that data (with the help of the **the clipboard** command, described later):

```
tell application "TextEdit"
    set textToPaste to ""
    repeat with oneItem in (clipboard info)
        if (item 1 of oneItem as
string) is "string" then
            set textToPaste to the
clipboard as string
            exit repeat
        end if
    end repeat
    -- insert textToPaste somewhere
end tell
```

Parameters

One optional parameter, **for**, lets you filter the response to return information about only one of



Chapter 7:

Scripting Addition Commands and Dictionaries

the value classes. For example, instead of receiving all five data types in an earlier example, you could retrieve information about only the styled Clipboard text class:

```
clipboard info for styled Clipboard text
-- result: {{styled Clipboard text, 22}}
```

Examples

```
set allInfo to clipboard info
set stringLen to item 2 of item 1 of ~
    (clipboard info for string)
```

You Try It

Manually select and copy all different types of data in documents from a variety of applications on your Mac. After each copy, inspect the contents of the Clipboard by running the **clipboard info** command in Script Editor.

Common Errors

Forgetting that the result is a list of lists.

Related Items (Chapters)

Set the clipboard to (7); **the clipboard** (7); value classes (9).



```
set the clipboard to ~
    variableOrReference
```

Result

None.

When To Use

If an application provides AppleScript commands for Clipboard manipulation, it is generally a good idea to use those features if possible. Such commands typically mimic the action of the **Copy** or **Cut** choices of the **Edit** menu. But they also usually require that the user or script select document data on which the operation is to take place. In some cases, a selection of content may be impractical, or your script assembles a special collection of data that you wish to preserve in the Clipboard for transfer to another document. The **set the clipboard to** command lets you script the insertion of any data into the system Clipboard.

Bear in mind that capturing a document's data to the Clipboard generally requires that the application be the active application at the time (just as when you manually work with the Clipboard). You should also execute the **set the clipboard to** command within a **tell** block directed at the application. This helps keep the context straight as your script dashes around your applications and document.

Parameters

The required parameter is the data you wish to place into the Clipboard.

Examples

```
set the clipboard to myText
-- variable containing string
```



Chapter 7:

Scripting Addition Commands and Dictionaries

You Try It

In this simple example, you'll use the Script Editor application as the context to place a string into the system Clipboard:

```
set the clipboard to "Clippy"
```

You can now go to any application document and paste that text into the document.

Common Errors

Trying to capture a document's content without it being the active application and document.

Related Items (Chapters)

clipboard info (7); the clipboard (7).



the clipboard [as class]

Result

Contents of the system Clipboard.

When To Use

If an application provides AppleScript commands for retrieving Clipboard data, it is generally a good idea to use those features if possible. Such commands typically mimic the action of the **Paste** choice of the **Edit** menu. This lets the application decide which data type to extract from a potentially complex collection of data elements in the Clipboard. But if you need to extract a particular data type from the Clipboard (and the application's scripting dictionary

doesn't provide this service as a parameter to the **paste** command), then use **the clipboard** command to grab just the data you want.

When dealing with complex Clipboard data, especially a set that includes uncommon data types, operate your scripts within the context of the application. Execute all commands within a **tell** block directed to the application, and **activate** the application.

Parameters

One optional parameter, **as**, lets you extract a particular class of data from the Clipboard. Once you retrieve that data, however, the application must be able to do something with it if you intend to insert the data somewhere in a document. Not all documents can handle all Clipboard data types (at least not independently of other data in the Clipboard).

If you wish to capture one of the non-English value classes (the ones inside the chevron characters), you can use the non-English terminology as a parameter, such as:

```
tell application "Microsoft Word"
  activate
  set HTMLData to the clipboard as «class
    HTML»
end tell
```

Entering the chevron characters into Script Editor can sometimes be a problem. Theoretically, you should be able to type them via the Option-\ and



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Shift-Option-\ keyboard sequences. If the compiler fails to recognize those characters, copy some sample data from your application, open a separate Script Editor window, and run the **clipboard info** command. Now you can select and copy the chevron sequence from the Result pane and paste it into the script under construction.

Examples

```
the clipboard
```

```
the clipboard as styled Clipboard text
```

You Try It

Playing with the **the clipboard** command in Script Editor is a little tricky because the application tries to display renderable content from the Clipboard in the Result pane, even when data of other types is included in the Clipboard. Technically, the result of the **the clipboard** command is a list of all pieces of data, but you cannot access items of the result in Script Editor. For instance, inspecting the class of the result of the command via

```
class of (the clipboard)
```

yields the string class if there is data of that type in the Clipboard. Yet, you can also extract a different value class with the help of the **as** parameter. Use the **clipboard info** command to determine what data types are contained in the Clipboard at any particular instance. Then use the **the clipboard** command to inspect the data returned for that class (inside a **tell** block directed at an application that

knows how to handle the data type).

With that in mind, open a new document in TextEdit, and enter text of any kind. Select segments and adjust various font and color styles. Select the text and choose Copy from the Edit menu.

Then switch to Script Editor and try each of the following script blocks, observing the results in the Result pane:

```
tell application "TextEdit"
  activate
  set clipData to the clipboard as string
end tell
```

```
tell application "TextEdit"
  activate
  set clipData to the clipboard as styled
  Clipboard text
end tell
```

In the second example, the data returned is in a form consisting of raw binary data. This value class is usually valuable only to the inner workings of an application.

Common Errors

Not taking into account the multiple data types returned by the command.

Related Items (Chapters)

Clipboard info (7); **set the clipboard to** (7); value classes (9).



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Folder Action Commands

Although the standard additions contain a group of commands called “Folder Action Commands,” the five commands in this group do not operate like the other commands discussed in this chapter. You don’t write scripts that invoke these commands. Instead, the system invokes these script commands on folders that have the Folder Actions feature enabled. The trigger that causes a command to be invoked is one of the five actions inflicted on a folder. For a folder to respond to the command, it must have a folder action script attached to it, and the script must include a handler that responds to the command. In other words, the system invokes the commands; you write scripts that respond to the commands.

Let’s create a simplified example of a script that displays a dialog whenever a file or folder is added to an existing folder. The dialog box reports the total number of items inside the folder. The scripting addition command that is invoked is **adding folder items to**. Figure 7.14 shows the AppleScript dictionary definition for this command.



adding folder items to: Called after new items have been added to a folder, but only when the folder’s window is open
adding folder items to alias -- Folder receiving the new items
after receiving a list of alias -- a list of the items the folder received

Figure 7.14. AppleScript dictionary entry for *adding folder items to*.

The command has two required parameters, both of which are filled in by the system when the command is invoked. The first, unlabeled parameter is a reference to the folder into which the item is being added; the second parameter (**after receiving**) is a list of one or more item alias references to the items being added. These parameter values allow you to write a generic folder action script that can be attached to any number of folders because the details are acquired at the moment the system invokes the command. (The dictionary caution about the folder needing to be open does not apply to recent Mac OS X versions.)

Here, then, is the folder action script that performs the desired actions when triggered by the **adding folder items to** command from the system:

```
on adding folder items to targetFolder after
receiving newItem
    tell application "Finder"
        set folderName to name of
targetFolder
        set itemCount to (count of items of
folder targetFolder)
    end tell
    set howMany to count of newItem
    if howMany is greater than 1 then
        set msg to (howMany as
string) & " new items have"
    else
        set msg to "One new item has"
    end if
    set msg to msg & " been added to folder \"
set msg to msg & folderName
set msg to msg & "\". It now contains "
```



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

```
set msg to msg & itemCount & " items."
display dialog msg buttons {"OK"}
end adding folder items to
```

There is nothing too extraordinary occurring here. We use the Finder to get some information about the folder (its name and number of items), but the rest is simply assembling an informative string to display in a dialog box.

Save a folder action script in one of the Folder Action Scripts folders located in the *Library/Scripts* folder either of the system or user. Then enable folder scripting for the desired folder by Ctrl-clicking on the folder's icon, and choosing **Enable Folder Actions** from the context menu. Next, choose **Attach a Folder Action** from the same menu, and locate the script file you created. That's all there is to it.

For more details on managing folder actions, including how to use the Folder Actions Setup application, visit <http://www.apple.com/applescript/folderactions/>

The complete list of folder actions commands are as follows:

adding folder items to -

aliasReference after receiving -
aliasReferenceList

closing folder window for -

aliasReference

moving folder window for -

aliasReference from *rectangle*

opening folder *aliasReference*

removing folder items from -

aliasReference after losing -
aliasReferenceList

All of these commands pass as their first parameter a reference to the folder on which the action occurs. Several powerful scripts come pre-installed, including some that perform graphic format conversion. You are encouraged to inspect and learn from those scripts.



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Internet Commands



A suite of Internet-related commands contains two quite different commands. The first command, **open location**, is one that your scripts would invoke to cause your Mac to load the designated destination in the program associated with the type of URL. The syntax for the command is as follows:

```
open location URLText ~  
[ error reporting Boolean ]
```

The required parameter is a string consisting of the complete URL string of the page to open in the browser. To reach a Web page, you must include the `http://` or `https://` protocol as part of the URL. If you supply an `ftp://` protocol, the action is instead redirected to the Finder, where the remote computer is mounted as a volume on the Desktop. Using a `file://` protocol and path to a local file causes the file to open in the application associated with the file. Upon invoking the command with a Web address, the default Web browser activates, and the page loads. The **open location** command returns no value. And, despite the indication in the AppleScript dictionary, the optional **error reporting** parameter neither forces nor bypasses typical browser errors, such as DNS or 404 errors that can occur during the page retrieval process.

The other Internet command, **handle CGI request**, is another one of those commands that is invoked by another process, rather than by your own scripts. Building a handler for this command

allows you to use AppleScript to write server scripts running on a Macintosh server (including Apache in Mac OS X). Your script may then process the data received by the server from Web page forms. Writing server scripts is outside the scope of this book. But you can learn a great deal more and see a working example by visiting <http://www.apple.com/applescript/guidebook/sbvt/pgs/sbvt.12.htm>.



Chapter 7: Scripting Addition Commands and Dictionaries

Understanding Application Dictionaries



This section could be better be titled “Trying to Understand Application Dictionaries.” A dictionary supplies a great deal of information about an application’s scriptability, but until you get to know a program’s scriptable characteristics, you may feel as though the dictionary is leaving out important bits. Sometimes the information you’re looking for is in another place within the dictionary; sometimes the information really is missing because there isn’t a logical place for the information to be placed within the dictionary.

When you use Script Editor to open a dictionary, the window has two panes (Figure 7.15). On the left is a list of Apple Event suites, classes (objects), and commands. Click on any item(s) in the left, and a fuller explanation appears on the right. These explanations come from the application—embedded by the programmers who made the application scriptable.

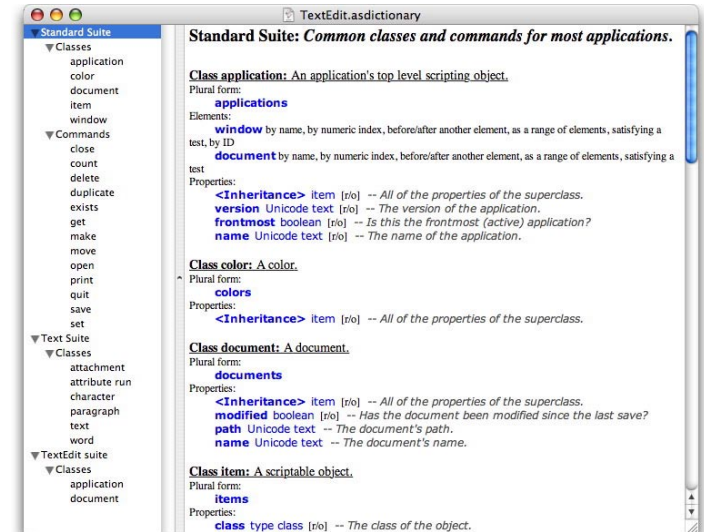


Figure 7.15. Dictionary window in Script Editor.

If you look at the outline of information supplied in a typical application, here’s what it comes down to (optional items in square brackets):

Suite name	
Class (object) name	
Description	
	[Plural form name]
	[Elements]
	[Properties]
Command	
Purpose	
Syntax	
	[Result]



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

Let's look at each of these items, and see what's great about the definitions, and what's usually missing.

Suites

An application's AppleScript dictionary is typically divided into multiple suites. Historically, suites were grouped according to types of Apple events (commands). These days, suites are more of a conceptual convenience for both the application developer and scripter. An application that supports hundreds of commands needs a way to organize and group objects and commands in a way that make it easier for scripters to find what they need for scripting tasks.

A dictionary typically features a Standard suite (once known as the Core suite) that shares much with the basic AppleScript commands described in Chapter 6, plus other commands that virtually every scriptable application has (e.g., a command to open an existing file or create a new, empty document). Applications that work with textual data also commonly include a Text suite, whose objects and commands provide basic scripting powers for dealing with the text of a document, and, more specifically, paragraphs, words, and characters. Beyond that, the application will usually have one or more suites for definitions tailored to the application and the types of data in its documents.

While almost all applications support the Standard Suite, an application may support some or all of the

standard commands, and often enhance some of the standard ones to work better with specific data or interface elements of the application. It's always a good idea to check through each of the Standard Suite objects and commands for a new application, and see if there is something you haven't seen before. Deviations from the standard won't necessarily be noted in any of the comments.

Classes

As described in Chapter 4, it is convenient to think of classes as objects. An object can represent something tangible (e.g., a document window, an iTunes playlist, a spreadsheet cell) or a bit more abstract (e.g., a dialog window reply, printer settings, the results of a global search command). All AppleScript dictionary entries refer to objects as classes.

As important as it is to know what kinds of classes an application permits you to script, dictionary class definitions can simply boggle the mind with both superfluous and insufficient information at the same time. Figure 7.16 shows the object definition for a paragraph in TextEdit. This object is defined in the Text Suite (and is defined the same way in virtually all built-in Apple application dictionaries).



Chapter 7: Scripting Addition Commands and Dictionaries

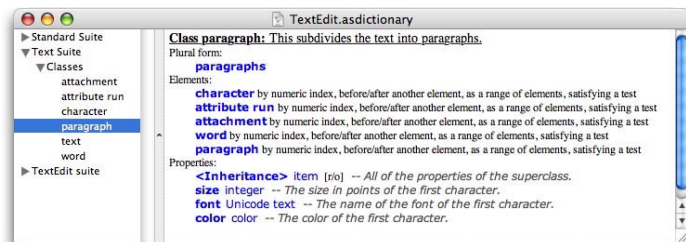


Figure 7.16. Paragraph object definition in TextEdit's dictionary.

Notice the list of elements (items that a paragraph class object can supposedly contain). The element listed as “attachment” turns out to be an object you don’t access from a script in this program. In fact, attempting to uncover if any such elements exist within a paragraph yields a script error. Conversely, it turns out that the **paragraph** object has some additional properties that it inherits from the **item** class (one of the Standard Suite classes), as indicated by the **<Inheritance>** notation in the Properties section. Fortunately, this application’s dictionary isn’t so big that you’d have a hard time finding the **item** class, but in a larger dictionary, you might have to go on a brief hunting expedition to follow an inheritance chain through several objects to get the rest of the properties for the paragraph class.

Years ago, early scripters of the first scriptable generation of Microsoft Excel ran into difficulty with its AppleScript dictionary. For example, nowhere in the dictionary did it explain exactly how to compose a reference to a specific cell in a spreadsheet. If you were used to the A1 type of cell addresses in Excel macros, you’d be frustrated as all get-out in AppleScript. Only by some trial and error did early scripters figure out that cell references were in the “R1C1” format. To compound matters, if your script retrieved a formula for a cell, it would retrieve the formula in whichever format it was written in, including the A1 format. But then Excel would not accept that formula in another cell, because the data was in the A1 format, rather than the R1C1 format. Fortunately, modern versions of Excel are far more forgiving, although the dictionaries can still leave new scripters in the dark by omitting descriptions of what roles numerous classes play within a document.

Commands

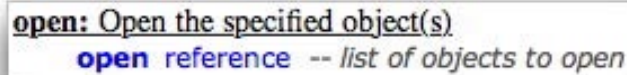
Commands are vital to scripting an application. A command definition usually consists of the syntax form and a note about the result that the command returns after execution. How well the syntax is described can make the difference between success and failure your first few times out with a command. Ideally, the syntax description should provide an explicit listing of the classes of values required for each parameter.

Figure 7.17 illustrates an unfortunate, yet all too



Chapter 7:
**Scripting
Addition
Commands and
Dictionaries**

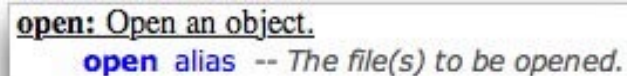
common, example that tells a scripter absolutely nothing about an application's **open** command. I'm not making this one up. It comes from the Microsoft Excel AppleScript dictionary.



```
open: Open the specified object(s).
open reference -- list of objects to open
```

Figure 7.17. An unhelpful dictionary entry.

Figure 7.18 shows a better (albeit not entirely perfect) way to define the same command. This comes from Apple's own Script Editor dictionary. At least this tells you that the parameter must point to a file in the form of an alias reference. Both dictionary entries shown here intend to induce you to perform the same action with the same syntax. It's just that without some experience and perhaps a bit of trial and error, you may have trouble interpreting the first entry to do the right thing.



```
open: Open an object.
open alias -- The file(s) to be opened.
```

Figure 7.18. A slightly more helpful dictionary entry.

Command definitions also rarely describe which objects they work with. When you look at the list

of commands and classes for a given suite, you may be led to believe that the commands work with all objects in the suite. Most of the time this is impractical, but nothing in the command definition helps you narrow down the objects that the command can affect.

Also, be aware that the comments supplied are not tied to the way a command has been written or modified (at the last minute) to behave. I have seen numerous examples of incorrect information about the result produced by a command or the default behavior of a command when a parameter is omitted.

Scripting by Dictionary Alone

The bottom line is that until you get a lot of scripting under your belt, learning an application's scriptability only by its built-in dictionary is fraught with peril. If an application you intend to use frequently offers documentation for scripters, it will be well worth obtaining it if it's not part of the program's regular documentation. It's not uncommon to find details either buried in the Apple Help documentation that comes with a program or published online at a support or developer section of the program publisher's Web site.

Just the same, be prepared for lots of trial and error as you gain comfort with the way each program expects its commands and object references. I'll have much more to say about how to approach scripting a new application in the final chapter of the book.



Chapter 7:

**Scripting
Addition
Commands and
Dictionaries**

Next Stop

Learning the AppleScript built-in commands and scripting additions is a big part of AppleScript. But an equally important piece of the puzzle is how to work with objects. Learning how to refer to objects in scripts can be one of the most challenging parts of AppleScript, but once you've got a good feel for that in the next chapter, you're well on your way to successful scripting.



Chapter 8 Describing Objects—References and Properties

I've said it before, and I'll say it again: commands perform actions on objects. In the previous two chapters we looked at lots of actions. This chapter focuses on the objects.

Objects In Real Life...

Most of the commands we issue in real life have well-defined objects as part of the command statements. For example, if you ask Joe to open the window, a couple of objects are involved in the process. First, Joe is an object, and the fact that he was the only person named Joe to be within earshot at the moment means that he is the sole target of your request. Your request then triggers Joe to do something with another object: the window.

While this may be a simple request in real life, you don't realize the hidden parts of the command you issued. For instance, by asking Joe to perform a certain task, you first assume that there is only one Joe who could possibly hear your request. Your second assumption is that Joe is capable of doing that task: that he knows what a window is and how to open one. And as for which window you mean, you may not have noticed the glance, nod, or other gesture that indicated to Joe which window you had in mind.



Chapter 8:
**Describing
Objects—
References and
Properties**

...Translated to Computer

Computers tend to ignore subtlety. As a consequence, they require extreme precision in how we label things. If our window opening request of Joe were part of a computer script, the computer wouldn't be able to see our gestures or nods to help it know what objects we're talking about. In fact, it may not even know precisely which Joe we mean from the dozens it may know.

We've already seen how this affects **tell** statements. Consider this statement fragment:

```
tell application "Microsoft Excel"...
```

We help the script narrow down who it is we're going to tell something to by specifying the application's name. If necessary, we could be even more specific about the object being addressed. For example, we could supply a pathname to a backup copy of the application on a different volume. The parameter to the **tell** command essentially contains a kind of roadmap for the command.

The same goes for other objects. If a computer script were in charge of the real-world window-opening example above, the script would need a roadmap to the precise "Joe" and precise window. The window we want a certain Joe to open is one window of a certain wall, of a certain room, of a certain building, and so on, until there is an ample description to make it clear exactly which window we mean.

For objects we'll be working with in AppleScript,

we need these same kinds of object roadmaps. The length of the directions on the roadmap depends on how small the object is within the scope of the application and its realm of objects. Watch the progression of depth the following statements encounter as the objects become smaller components of the outermost object, the application:

```
-- application level
tell application "TextEdit"
    quit
end tell

tell application "TextEdit" -- document level
    get name of document 1
end tell

tell application "TextEdit" -- paragraph level
    get paragraph 3 of document 1
end tell

tell application "TextEdit" -- word level
    get word 2 of paragraph 3 of document 1
end tell

-- character level
tell application "TextEdit"
    get character 1 of word 2 of paragraph 3
    of document 1
end tell
```

As each object becomes more deeply nested within others, the description of the object grows. These sometimes long object descriptions are necessary, since each part of the description is an integral part of the object description.



Chapter 8:
**Describing
Objects—
References and
Properties**

References

These verbal roadmaps to objects are called object **references**. Knowing how to write proper object references is one of the most important aspects of mastering AppleScript. The difficulty lies not so much in knowing the various kinds of references there are, but in the fact that each application can—and does—treat references differently, even among similar applications.

If you dissect any object reference, you will discover that it has three pieces:

1. Type of object—what kind it is
2. Form—a distinguishing characteristic about a particular object
3. Container—the object that holds the specific object you're referring to

The first word of a reference identifies the **class** of the object of your command's affections. Class names—terms such as **word**, **paragraph**, **cell**, **row**, **record**—clearly state what kind of object is to receive the action of the command.

The **form** that distinguishes one object in a class from another object of the same class is some identifier, such as a number or name.

A **container** is any object that can hold other, smaller objects. For example, a document contains one or more paragraphs. When you view the AppleScript dictionary of an application's object, the container-ish aspects of the object are represented by a list of

what kinds of other objects—**elements**—the object is allowed to contain. For instance, a document object may list as its elements paragraphs, words, and characters. The container of a particular object is simply part of the roadmap to the object, and must be part of the reference to the object.

When an object is deeply nested, the reference must include the series of containers that point to the object. An extreme example with the TextEdit character object is:

```
character 1 of word 2 of paragraph 3 of -  
document 1 of application "TextEdit"
```

Notice that the series of containers are listed in this reference from the smallest to the largest. In other words, we're referring to the first character of the second word of the third paragraph of document 1 of a specific application. The sequence demonstrates the logical steps AppleScript follows to reach the object.

As you'll see later in this chapter, the extreme case above can be shortened if your script can accommodate a different frame of reference. Consider the three-paragraph text entry in a TextEdit document shown in Figure 8.1.



Chapter 8:
**Describing
Objects—
References and
Properties**

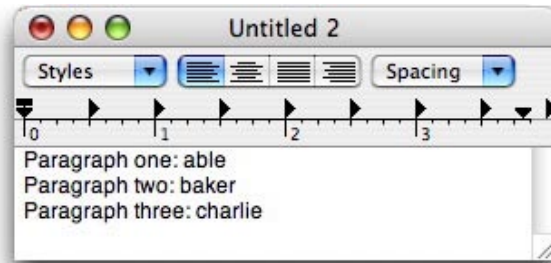


Figure 8.1. A three-paragraph document.

Here are the various ways we can refer to the first letter of the word “baker”:

```
character 1 of word 3 of paragraph 2 of -  
document 1
```

```
character 1 of word 6 of document 1
```

```
character 36 of document 1
```

The **index** (i.e., numeric) values that signify character, word, and paragraph counts are all intertwined in the sequence of objects. When the frame of reference is the document and its character elements, the script can get straight to the character by its count among all the character elements of the document.

Container Syntax

So far, all of our examples use the **of** preposition between elements. AppleScript accepts the **in** preposition as an alternate. The following references get you to the same TextEdit object:

```
word 3 of paragraph 2 of document 1
```

```
word 3 in paragraph 2 in document 1
```

```
word 3 of paragraph 2 in document 1
```

AppleScript also recognizes the possessive form of containers. For example, the reference

```
paragraph 3 of document 1
```

can also be written as

```
document 1's paragraph 3
```

The possessive form is more often used in concert with object properties and the ordinal form of numbers, as in:

```
first window's bounds
```

For references that feature a series of nested containers, the possessive form can become quite cumbersome. Use this form only when it helps the readability of the script.



Chapter 8:
**Describing
Objects—
References and
Properties**

Default Object References

When sending a command to an application's object, the application must also be part of the reference.

Therefore, to send a command to the third word of document 1 inside TextEdit, the formal reference is:

```
word 3 of document 1 of application "TextEdit"
```

A reference that includes the application name, like this one, is called a **complete reference**.

When a reference doesn't have enough information to fully specify the object, the script looks to the default object to fill out the reference. If no default object is explicitly named, then the default object is the application hosting the script—Script Editor, if that's where you run the script. But if you save such a script as an application and launch that script, the context becomes that very script application, which won't know anything about other applications unless statements are directed to those apps.

Conveniently, we can shift part of a complete reference to a **tell** statement. In fact, we've been doing a lot of that in the book thus far. When a script says

```
tell application "TextEdit"  
  get word 3 of document 1  
end tell
```

TextEdit is the default target object for all commands inside the **tell** block. AppleScript completes the **partial reference** in the **get** statement with the application name, as required. We can also go further along this line by defining a narrower default object,

if it suits the needs of the script. Therefore, we can also say

```
tell document 1 of application "TextEdit"  
  get word 3  
end tell
```

to accomplish the same thing. In this case the partial reference is filled out with default object information about document and application identifiers. We can use this behavior to our advantage when a number of statements apply to a specific object belonging to the application. It makes for shorter, more readable script lines. Thus,

```
tell application "TextEdit"  
  set size of word 1 of paragraph 1 of  
  document 1 to 20  
  set size of word 1 of paragraph 2 of  
  document 1 to 20  
end tell
```

becomes:

```
tell document 1 of application "TextEdit"  
  set size of word 1 of paragraph 1 to 20  
  set size of word 1 of paragraph 2 to 20  
end tell
```

You have the power to make any container the default object, as long as each object in the reference is a legitimate element of the next larger container (as defined in the dictionary).

Creative uses of **tell** statements can provide interesting flexibility in how you structure a script. Here is how nesting **tell** statements can be used for efficiency:



Chapter 8:
**Describing
Objects—
References and
Properties**

```
tell application "TextEdit"
  tell document 1
    tell paragraph 1
      -- commands for objects
      -- in paragraph 1
    end tell
    tell paragraph 2
      -- commands for objects
      -- in paragraph 2
    end tell
  end tell
end tell
tell document 2
  tell paragraph 1
    -- commands for objects
    -- in paragraph 1
  end tell
  tell paragraph 2
    -- commands for objects
    -- in paragraph 2
  end tell
end tell
end tell
```

Object Properties

Now that we've been introduced to the concept of referring to an object, how does a script get or set information about that object? In most applications, it all relies on an object's **properties**. Properties of an object consist of a well-defined series of specifications that help shape the appearance and characteristics of an object.

Real-life objects have properties as well. If we were to define a standard pencil object, the kinds of properties we might give it are: length; diameter; outer color; lead number; point sharpness; lead color; has eraser; eraser height; has toothmarks. Every standard pencil has those properties, although the values (or settings) of those properties likely differ from one pencil to another.

Scriptable object properties are often more complex than that, but your scripts will quite commonly inspect the value of an object's property. Here are some examples from various scriptable programs demonstrating how to read the property value of an object:

```
paragraphs 1 thru 5 of document 1 -- TextEdit
played count of track 2084 of ¬
  playlist "Library" -- iTunes
formula of cell "R35C12" -- Excel
value of email 1 of person 24 -- Address Book
```

Each of these property references point to data stored in a document—very likely the kinds of things your scripts will be shuffling about. And, as described back in Chapter 5, these phrases are AppleScript



Chapter 8:
**Describing
Objects—
References and
Properties**

expressions that evaluate to values you can place into variables or pass as parameter values to commands.

Because the dictionary for an object may not be clear about how references evaluate, you may also have to use trial and error to discover what the mere reference to an object returns. In some cases, it may be the data you're looking for; in other cases the returned value is a detailed record of the object's properties, from which you must still extract meaningful data as a property of that record.

Object Reference Syntax

AppleScript has defined a number of object reference forms, or rules for the structure of object references. For the rest of this chapter, we'll examine each of these forms in detail. The form types are:

Property	Index
Relative	Name
ID	Middle Element
Range	Arbitrary Element
Filter	Every Element

Before you get too involved with all these forms, I must give you one important caution that should make learning these easier:

Not every application and not every object supports every form.

In the next pages, you'll see some very interesting, if not exciting, ways to refer to objects or groups of objects. Be aware that while AppleScript provides a framework that allows these references to work, the level of support for any form is solely the responsibility of the AppleScript implementation in a program. Sometimes information about what forms an object supports is provided in printed documentation about a program's AppleScript support. Unfortunately, many AppleScript dictionaries mislead scripters about support for the more esoteric forms (if they say anything at all). Trial and error will be the course of action in many cases.

In the syntax descriptions to follow, recall that a reference includes a class, an identifier, and a



Chapter 8: Describing Objects— References and Properties

container. To help make the property syntax more readily understandable, I will include the container part inside angle brackets (e.g., `<of object>`). This should help you place these references in better context as you learn them or look them up for reference.

Property References



Syntax

```
[ the ] propertyLabel <of objectOrRecord>
```

How to Use

The property reference form is perhaps the simplest to use. It is also typically the most reliable, because AppleScript dictionaries tend to do a decent job of listing properties and the types of values the properties return or require when being set. All properties evaluate to values, but some properties are read-only.

As an illustration, examine Figure 8.2, which shows the properties for a **word** object in TextEdit. Let's look more closely at property definitions and how they translate to property reference forms.

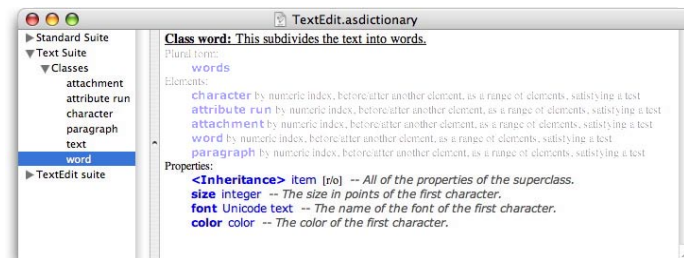


Figure 8.2. TextEdit's properties for a word object.

The first property is a pointer to the **item** class (listed in the left column under the classes of the Standard Suite—not in view here), whose properties a **word** object also has. This way of pointing to inherited properties keeps more complex dictionaries from exploding in physical size, but makes it more cumbersome to trace down the available properties for a particular class.

Getting down to properties specific to the **word** object, the value of the **size** property is an integer, representing the font size assigned to the entire word. Assuming the default object of a statement is the TextEdit application, the reference to the **size** property of the first word of an open document would be:

```
size of word 1 of document 1
```

The name of the property, **size** (as listed in the dictionary), is the **propertyLabel** part of a property reference form.

You will often encounter property listings showing the read-only designator ([r/o]). Such properties



Chapter 8:
**Describing
 Objects—
 References and
 Properties**

cannot be modified by script. In the case of inherited properties, the inheritance pointer property is always read-only, but you'll have to hunt down whether particular inherited properties are, themselves, read-only or settable.

You also use the property reference form to get or set values inside an AppleScript record (covered in Chapter 9). For example, a record from an employee object might look like this:

```
{name: "Morris Sloan", hourly rate: 9.35, -
  department: "Manufacturing"}
```

You can think of each field within this record as a property, with the field value corresponding to a property value. Syntax for referring to the data within a record is the same as referring to a property of an object, as in:

```
set oneRecord to {name: "Morris Sloan", hourly
  rate: 9.35, department: "Manufacturing"}
hourly rate of oneRecord
-- result: 9.35
```

If you like, you can precede any or all property reference forms with the word “the”. This can add to the readability and friendliness of a script for newcomers. These two scripts produce exactly the same results:

```
set font of document 1 to "Times"
set the font of document 1 to "Times"
```

Here are some examples from various scriptable applications plus an AppleScript record object:

```
bounds of window 1 -- TextEdit
```

```
the formula of cell "R3C15" -- Microsoft Excel
artist of track 20 of -
  playlist "new arrivals" -- iTunes
cellValue of cell "City" -- FileMakerPro
the ZIP of -
  {city: "Chicago", state: "IL", ZIP: 60611}
```

The more you work with scriptable applications other than TextEdit, the more importance you will give to properties. While TextEdit allows your script to access the contents of a text object directly (e.g., word 3 of paragraph 1), many other application objects don't offer such shortcuts. For example, the route to capturing the text of a line within a BBEdit document has its share of twists and turns:

```
contents of line 21 of text 1 of window 2
```

If you fail to explicitly ask for the contents of a line (i.e., the **contents** property of a **line** object), your script receives a reference to a **line** object, which, as explained in the dictionary, has 10 properties.

You Try It

Enter and run each of the following scripts, watching the returned values in the Result pane:

```
tell application "TextEdit"
  name of window 1
end tell
```

```
tell application "TextEdit"
  bounds of window 1
end tell
```

```
tell application "TextEdit"
  activate -- so we can watch
  -- grab bounds property value for a window
```



Chapter 8: Describing Objects— References and Properties

```

set windowRect to the bounds of window 1
-- add 50 to each value
repeat with i from 1 to 4
    get item i of windowRect
    set item i of windowRect to
(result + 50)
end repeat
-- set the property for the window
set the bounds of window 1 to windowRect
end tell

```

```

set teamRecord to { team: "Mudhens", wins: 40,
losses: 32 }
set theWins to wins of teamRecord
set theLosses to losses of teamRecord
set gameTotal to theWins + theLosses
display dialog "This team has won " & theWins
& " out of " & gameTotal & " games."

```

Enter and run the following script to see what happens in an application (Apple's Address Book) that requires a property reference form to access its data:

```

tell application "Address Book"
    set testValue to email of person 2
end tell
(* result: {email 1 of person id "FF066B00-
7E74-11D7-8000-0003938335A0:ABPerson" of
application "Address Book"} *)

```

The result here is a reference to an **email** object as known to the Address Book application (the ID will be different in your database). Notice, too, that the result comes back as a list, and that the reference begins with a numbered reference (discussed in the next section). This means that the **email** property might return multiple references if there are multiple email entries for that person and that you'll need to

reference a numbered item (e.g., **email 1**) to get to the actual data.

In a lot of applications (including Address Book), you can use Script Editor to inspect the properties and their current values by reading AppleScript's **properties** value of an object. Such property lists are in the form of AppleScript records, as shown here (sample data from my Address book appears in the result):

```

tell application "Address Book"
    properties of (email 1 of person 2)
end tell
(* result: {id:"E1B97F8E-2B7B-11D9-A96B-
0003938335A0", class:email, label:"work",
value:"scriptBoy@dannyg.com"} *)

```

To get human-usable data from the **email** object, we must access its **value** property:

```

tell application "Address Book"
    set testValue to value of email 1 of
person 2
end tell
-- result: "scriptBoy@dannyg.com"

```

Common Mistakes

Forgetting the second word of a two-word property or field name; forgetting to complete the reference when addressing a property of a deeply nested object (e.g., contents of a line of the text of a document).

Related Items (Chapters)

Record value class (9); **get** (6); **set** (6); **copy** (6); indexed object references (8).



Chapter 8:

Describing Objects—References and Properties

Indexed References



Syntax

```
[ the ] className [ index ] index ~
    <of objectOrItem>
[ the ] ( first | second | third | fourth | ~
    fifth | sixth | seventh | eighth | ~
    ninth | tenth ) className ~
    <of objectOrItem>
[ the ] index( st | nd | rd | th ) ~
    className <of objectOrItem>
[ the ] ( last | front | back ) className ~
    <of objectOrItem>
```

How to Use

An index is nothing more than an integer that represents the count of elements from the beginning or end of a containing object. In AppleScript, numeric index counting starts with 1 (some other languages start with zero). For example, in an Excel worksheet, your script may have to refer to the first cell of the third row. The index reference form would be:

```
cell 1 of row 3 of sheet 1
```

In truth, there are three indexed references in that phrase. The first index is the **1** indicating which cell of row 3; the second index is the **3** indicating which row among all rows in the table; the final index is the **1** indicating which worksheet object we're talking to. The basic syntax for an indexed reference starts with the name the class of object followed by the index count (the word **index** is optional and rarely used, since it adds nothing to readability). A leading article, **the**, is also optional in case you want your

statements to sound more like English.

In an effort to offer alternatives that read more like a natural sentence, AppleScript's designers provide other versions of the index reference form. Therefore, instead of saying

```
cell 1
```

your script can refer to that object as

```
first cell
```

AppleScript knows about the special ordinal numbers as words—"first" through "tenth"—and accepts them without a blink as long as the ordinal word is followed by a space and the class name of the object being counted. It's even smart enough to know that any integer immediately followed by the ordinal endings ("st", "nd", "rd", or "th") is a valid index (again followed by a space and class name of object to be counted). The following index references are valid:

```
first word
1st word
word 1
the 234th word
92th word
```

That last one ("ninety-twoth word") may seem odd, but AppleScript accepts it as one of the ordinal endings, regardless of the numeral preceding it.

Nothing limits you from mixing the regular and ordinal indexed references in the same statement. Therefore, both of the following statements are acceptable:



Chapter 8:
**Describing
 Objects—
 References and
 Properties**

```
third paragraph of document 1
paragraph 3 of the first document
```

So far, the examples have assumed that index counts start from the beginning of objects in a series. What if the operation you need to perform is at or near the end of the series of objects? While you can usually find out how many objects are in its container (e.g., **count of words in paragraph 1**), and use that value as an index, some types of data (such as lists) let you use a shorthand notation. They consider the value **-1** to represent the last object in a series. Here are some examples:

```
item -1 of {10,20,30}
-- result: 30

item -2 of {10,20,30}
-- result: 20

tell application "TextEdit"
  get word -1 of document 1
  -- result: (last word of the document)
end tell
```

The formal definition of this reference form also indicates that the words “last”, “front”, and “back” are accepted as if they were ordinals (i.e., followed by a class name). In truth, not all kinds of data support this syntax. Aside from definitely working with items of lists, these variants work with layered objects, such as windows:

```
front window -- topmost window (i.e., window 1)
back window -- most hidden window
```

To find out if a certain object can be referenced by

a numeric index, poke through other object class definitions in search of one that lists the desired object as one of its elements (i.e., an object that is a container of the object in question). For example, if you were unsure about using an index reference for a character in TextEdit, look at definitions of its possible containers (word, paragraph, document). Where you see the character element listed (Figure 8.3), the comment indicates whether you can refer to a character within that container by a numeric index. For TextEdit, all three containers say they accept the character element referred by a numeric index.

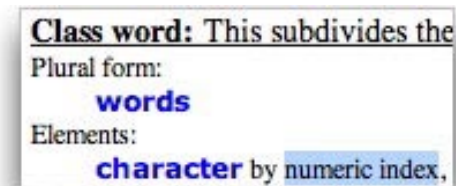


Figure 8.3. The dictionary indicates a character element can accept indexed references.

You Try It

Open the *Bill of Rights.rtf* document (TextEdit document) in the *Handbook Scripts/Chapter 08* folder of the companion files. Position this document and the Script Editor windows so you can see as much of the document as your screen allows. Then enter and run the following script in Script Editor, watching for returned values in the Result pane:



Chapter 8: Describing Objects— References and Properties

```
tell document 1 of application "TextEdit"
    get paragraph 1
end tell
```

Substitute the **get paragraph 1** line, above, with each of the following lines and run the script:

```
get paragraph 2

get word 1 of paragraph 2

get front word of paragraph 2
    -- same as word 1

get last character of first word of -
    paragraph 2 -- "s"

get 11th character
    -- of entire document 1, "C"

get 11th character of paragraph 5 -- "u"

get character 20 of paragraph 1
    -- error, because there aren't 20 characters
    -- there the index is therefore invalid
get paragraph 3
    -- result: just a carriage return in a string
```

Enter and run each of the next lines by themselves, since AppleScript knows how to handle items:

```
item 3 of {"Chico", "Harpo", "Groucho", -
    "Zeppo", "Gummo"}

first item of {"Chico", "Harpo", "Groucho", -
    "Zeppo", "Gummo"}

back item of {"Chico", "Harpo", "Groucho", -
    "Zeppo", "Gummo"}
```

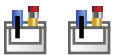
```
last item of {"Chico", "Harpo", "Groucho", -
    "Zeppo", "Gummo"}
```

Common Mistakes

Incomplete references when referring to a deeply nested object; invalid index number (for more objects than are available).

Related Items (Chapters)

Partial and complete references (8); list class values (9).



Relative References

Syntax

```
[ className ] ( before | [in] front of ) -
    baseReference
[ className ] ( after | [in] back of | -
    behind ) baseReference
```

How to Use

A relative reference is an object whose location is defined by its position in relation to another object. For example, in the (vintage TV show title) phrase:

Who Do You Trust

the word “Who” is before (or in front of) the word “Do”; the word “Do” is after (or in back of or behind) the word “Who.” This can be useful for a script that is moving objects around. Be careful with this reference, however: it is not universally supported in applications (in fact, it doesn’t work on AppleScript items). TextEdit (and probably most other text processing and layered object kinds of applications,



Chapter 8:
**Describing
Objects—
References and
Properties**

such as draw-type graphics programs) supports this reference form fully, so all examples in this section will be from that application.

The key to working with a relative reference is knowing how to specify the *baseReference* component. Typically, the *baseReference* is a more absolute reference, such as an indexed or named reference. Think of the *baseReference* as the anchor, with the relative reference being an object on either side of that base object.

Going back to our “Who Do You Trust” phrase, we could use a relative reference to refer to the first word, as in:

```
word before second word
```

Such wording is kind of clumsy because when dealing with identical types of objects (words and words), other reference forms are usually simpler to write and easier to read. One strength of the relative reference form is that if the application is willing (unfortunately, TextEdit is not one of those), your script could mix and match objects, as in:

```
word before paragraph 3  
paragraph before word 30
```

Some rules apply here, however. The objects are usually related by their containment definitions. For example, in:

```
word before paragraph 3
```

a **word** is typically an element of a **paragraph** object, so there’s no problem for the application to

pick out what you mean. But for:

```
paragraph before word 30
```

things get trickier, especially if word **30** is in the middle of a paragraph. AppleScript interprets this request to mean “paragraph before the paragraph containing word 30”. In other words, if the base object isn’t an element of the relative object, AppleScript tries to make the base object into the same class as the relative object.

Common sense also applies here. It may be fine to say:

```
window behind window 1
```

But you cannot mix grossly unlike objects as in:

```
paragraph behind window 1 --<<No can do!
```

Prepositions shown above for the two syntax descriptions need just one point of explanation. When the reference is to something **in front of** a base object, it means *immediately* in front of. If the items were indexed, and the base object were indexed as value 3, the object in front of it would have an index of 2. For an object **after** the base object, we’re talking immediately following (the imaginary index value would increase by one).

Notice that the syntax description lists the leading *className* component as optional. It is optional only in some circumstances. For example, you may encounter a text editing program that interprets the omission of the *className* component as telling the program that you mean the **insertion**



Chapter 8:
**Describing
Objects—
References and
Properties**

point object. An **insertion point** object (in those applications that define them) is typically a location relative to another object (it may hard to think of a location as an object, but strictly speaking, it is). When you click the cursor at the beginning of a paragraph and the flashing text insertion pointer flashes there, the insertion point object is defined by its location:

```
beginning of document 1
```

Press the right arrow key to move the pointer, and the insertion point object is defined by its new location:

```
after character 1 of document 1
```

Therefore, when a script says,

```
copy "Fred" to after word 1 of document 1
```

the relative reference (**after word 1 of document 1**) understands you to mean the insertion point immediately following word 1 of document 1. Insertion points don't know about spaces (in fact, an insertion point can go between a space and a character), so you have to supply spaces with your text, as in

```
copy ", I must emphasize, " to before word 12
```

Working with this somewhat invisible **insertion point** object takes some getting used to, but it becomes essential for inserting text into existing text where that operation is supported.

Earlier I said that the relative reference form also works well with layered objects. You can experiment

with that concept with document windows, which are treated as layers that can go before, in front of, after, in back of, or behind other windows. For instance, in TextEdit you can reference a window in front of or behind another:

```
get name of window behind window "Demo"
```

As a final note (and I don't know why you'd want to do this), the *className* component of a relative reference must be a simple, valid class name for the default application of the script. You cannot use another kind of reference (as in **second word after first word of paragraph 2 of document 1**) for this component. You should probably be using an indexed reference for this kind of object location.

An AppleScript dictionary will usually indicate when an object supports relative referencing. Figure 8-4 demonstrates a typical advisory from TextEdit. What the dictionary doesn't tell you, however, is how flexible relative referencing is for the object or within the application. For instance, TextEdit doesn't let you reference a character relative to other classes of element, such as words or paragraphs.



Chapter 8: Describing Objects— References and Properties

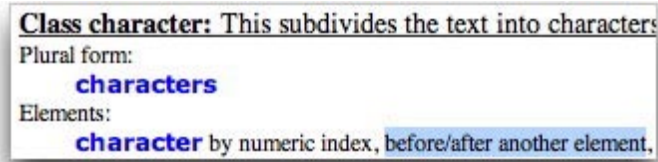


Figure 8.4. The dictionary indicates a character element can accept relative references.

You Try It

Open the *Bill of Rights.rtf* document with TextEdit. Enter and run the following script:

```
tell document 1 of application "TextEdit"
    word after first word of paragraph 2
end tell
```

Substitute the **word after first word of paragraph 2** line, above, with each of the following lines and run the script:

```
word before last word of paragraph 2

word before word -5 of paragraph 2
-- 6th word from end

word after paragraph 1 -- error
-- TextEdit doesn't support
-- mixed relative references
```

Close *Bill of Rights.rtf*. Now open the three documents for Chapter 8 named *Layer 1*, *Layer 2*, and *Layer 3* in that order. Enter and run the following script to see the results.

```
tell application "TextEdit"
    name of front document
end tell
```

Then substitute the middle statement with each of the following lines and run the script:

```
name of back document

name of document in front of document "Layer 1"

name of document before document "Layer 2"

name of document behind document "Layer 2"
```

Common Mistakes

Incomplete reference for the *baseReference* object parameter; relative reference form not supported by the application or object.

Related Items (Chapters)

Indexed reference form (8); partial and complete references (8).

Name References



Syntax

```
className [ named ] nameString ~
    <of ObjectOrItem>
```

How to Use

If an object has a **name** property in its definition, then your scripts can use the name reference form to identify that object. Two required components for the name reference form are the class name of object, followed by the complete name (in quotes or a variable that evaluates to a string) assigned



Chapter 8:
**Describing
Objects—
References and
Properties**

to that object. Sometimes the name is assigned by an application (e.g., the “untitled” name of a new window), while in other cases, your scripts can assign names (via the **set** command). Optionally, you can insert the word “named” between the two components if you believe it helps the readability of your scripts.

Name references are most often used in concert with other references to get or set specific information about that object. If the reference is simply to the named object (and the program accepts it as a valid reference), you may get more information than you bargained for. For example, in Microsoft Excel, we can assign a name to a cell, because a cell object in that program has a **name** property (also settable in the program via the **Insert>Name>Define** menu choice). Let’s say you’ve assigned the name “Product” to the first cell, which contains the value 456. If your script says:

```
get cell "Product"
```

Excel returns a reference to the cell range within the worksheet. What you really want is the data from the cell, retrievable through the **value** property of that cell. Therefore, the reference to the value consists of a property reference form combined with a name reference form:

```
get value of cell "Product"
```

Name reference forms are commonly required for parameters to commands (AppleScript’s and an application’s commands). Look at these commands:

```
open file "HD:Documents:Bill of Rights.rtf"

close window "Jane Doe"

activate application "HD:Apps:Microsoft Excel"

list folder alias "HD:Trash:"
```

Each one contains a command followed by a required name reference form. The *className* component sets up the type of data the command needs (**file**, **window**, **application**, and **alias**, in the examples above). The command chokes without the *className* parameter part of the reference.

Whenever an application you’ll be scripting can have many objects available for a command, it is vital that you use name reference forms if at all possible. The importance of a name reference over an indexed reference will become readily apparent the instant a user or your script disturbs the order of objects (e.g., window, documents), and your script expects a particular object to be in a particular order. Assigning unique names to objects and then using those names in references goes a long way towards avoiding unexpected execution errors.

Of course, there may be times when more than one object in the default object (e.g., multiple cells in a spreadsheet document) has the same name. If that happens, the name reference will apply to only one of those objects. Which one depends on how the application handles it. Chances are, the program will pick the one whose indexed reference would be the



Chapter 8:
**Describing
 Objects—
 References and
 Properties**

lowest number. Alternatively, the application may return a value that is a list of all objects of the same class and name and you refer to an individual object as an item of that list.

You Try It

Open the three Layer documents (*Handbook Scripts/Chapter 08* folder) in TextEdit. Enter and run the following script:

```
tell application "TextEdit"
    text of document "Layer 1"
end tell
```

Substitute the **text of document "Layer 1"** line, above, with each of the following two statement groups and run the scripts:

```
text of document "Layer 3"

set winName to name of window 2
bounds of window winName
```

The second example grabs the **name** property value for the window because the official window name may contain additional path information that will differ with each user's computer. But the example equally demonstrates that a string value variable may also be used to supply the name for a name reference.

Common Mistakes

Incomplete reference to a named object; forgetting the property name when accessing values of a named reference object.

Related Items (Chapters)

Indexed reference form (8); partial and complete references (8); properties (8); string class values (9).

ID References



Syntax

```
className id IDvalue <of Object>
```

How to Use

A few applications assign unique ID (identifier) values to each object they create. This value is like a serial number stamped in steel on the object. An object's ID number (usually an integer) stays with that object for its entire life. Applications that do IDs correctly won't let the same ID be assigned to any other object in the file—even if an object is deleted, no new object of the same class will ever receive the same ID while the application remains open.

An ID guarantees a uniqueness to a property that cannot be altered by the user changing the object's order (index) or name. Therefore, an ID reference is guaranteed to work while the object is still "alive" in the file.

Two required components of an ID reference are the class name of the object containing the **ID** property and the **ID** property value (usually an integer). Between them is the hard-wired word "id" (in any combination of uppercase and lowercase), which is pronounced as if the letters were two initials (not like the Freudian term). Here are some examples of what the syntax looks like in real life:



Chapter 8:
**Describing
Objects—
References and
Properties**

```
window id 9029
```

```
background field id 23
```

Be prepared, however, to not be able to access the ID of some objects whose property list includes an ID. Often such properties are present for internal use only, but end up in the dictionary, causing scripters no end of trial and error—mostly error.

You Try It

Several Apple applications bundled with Mac OS X include ID properties with which you can experiment. TextEdit, for example, creates an ID integer for each window that opens. The ID does not survive closing the window, but it allows your script to refer to a particular window with confidence, even if its stacking order changes (altering its index reference) or it is renamed by the user (altering its name reference).

iTunes also establishes a unique ID for each track it stores in its Library. The **track** object property containing the value is **database ID**. This is the value that lets multiple playlists share a single track file listed in the iTunes Library. Importantly, this value sticks with the track between iTunes sessions.

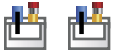
Common Mistakes

Forgetting the “id” word; incomplete reference.

Related Items (Chapters)

Partial and complete references (8); properties (8); integer class values (9).

Middle Element Reference



Syntax

```
middle className <of objectOrItem>
```

How to Use

Some objects allow this reference form, which points to the center element of a series. For example, in a three item list, the middle reference form points to the second item:

```
get middle item of -  
    {"Executive", "Legislative", "Judicial"}  
-- result: "Legislative"
```

When the number of items in the series is even, the reference points to the item that is the last item of the first half of the group (the actual formula is $((n + 1) \text{ div } 2)$, where n is the total number of objects in the series). Therefore,

```
get middle item of -  
    {"parsley", "sage", "rosemary", "thyme" }  
-- result: "sage"
```

TextEdit supports this reference for some objects, as in:

```
middle word of paragraph 1 of document 1  
middle character of word 3 of paragraph 8
```

You may see similarities between the middle and indexed reference forms. In a way, the middle reference form is a special case of the indexed form, where AppleScript does the math of figuring out which is the middle (between the first and last).

You Try It

Open the *Bill of Rights.rtf* document in TextEdit.



Chapter 8:

Describing Objects—References and Properties

Then enter and run the following script:

```
tell document 1 of application "TextEdit"
    get middle paragraph
end tell
```

The above script returns a carriage return, because the middle paragraph happens to be a blank line between other paragraphs. If you were to insert a carriage return before the first line and run the script again, you'd get one of the full paragraphs. Revert the document to its original state before proceeding with the rest of the experiments. Substitute the **get middle paragraph** line, above, with each of the following lines and run the script:

```
get middle word of paragraph 1

set size of middle word of paragraph 2 to 48
```

Close the *Bill of Rights* document, without saving changes. Next, open the three *Layer* files in order. Run the following script:

```
tell application "TextEdit"
    name of middle window
end tell
```

While previous examples were calculating horizontally, the middle reference can work “vertically” through layers of objects of the same class, such as these three windows.

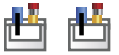
Common Mistake

Incomplete reference.

Related Items (Chapters)

Partial and complete references (8); indexed reference form (8).

Arbitrary Element References



Syntax

```
some className <of objectOrItem>
```

How to Use

Related to the indexed reference, the arbitrary element reference points to a random item in a series of objects. AppleScript handles the random number generation for things like lists, but not for an application's objects—that's entirely up to the application if its designers elect to support this reference form. This form is not appropriate for some classes of data, so don't expect it to be present in all instances (although it is available for several object classes in TextEdit).

As with any kind of indexed reference form, the arbitrary reference requires a complete reference, as in:

```
some word of paragraph 1 of document 1
some paragraph of document 1
```

You can also use this form with lists, as in,

```
some item of {20, 40, "sixty", 80, 100}
(* result: (could be any item and
changes each time) *)
```

Even though random numbers are involved here, there is no relation between this reference form and the **random** scripting addition discussed in Chapter 7.



Chapter 8:
**Describing
 Objects—
 References and
 Properties**

You Try It

Open the *Bill of Rights.rtf* document in TextEdit. Then enter and run the following script several times to see the random results:

```
tell document 1 of application "TextEdit"
    get some paragraph
end tell
```

Substitute the **get some paragraph** line, above, with each of the following lines and run each script a few times to see random results:

```
get some word of paragraph 2

set size of some word of paragraph 2 to 48
some word of paragraph 1
    -- only two words to choose from
```

Revert the *Bill of Rights.rtf* file. Then enter and run the following script, which sets up to ten random words of the document to red:

```
tell document 1 of application "TextEdit"
    activate
    repeat 10 times
        set color of some word of paragraph
        2 to "red"
    end repeat
end tell
```

Close the *Bill of Rights.rtf* document, without saving changes.

Common Mistake

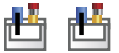
Incomplete reference.

Related Items (Chapters)

Partial and complete references (8); indexed

reference form (8).

Every Element References



Syntax

```
every className <of objectOrItem>
pluralClassName <of objectOrItem>
```

How to Use

This variation of the indexed reference form points to all items of the class named in the *className* component. It's a little tricky to work with sometimes, because programs may react differently to this reference. The best way to use it, however, is to retrieve all the values of a particular property of an entire collection of objects, as in,

```
every word of paragraph 1
value of every cell of range "Price" of sheet 1
```

What you may not expect from this reference form, however, is that such references evaluate to AppleScript lists of value. For example, if the first paragraph of a TextEdit document is

"Four score and seven years ago..."

and we

```
get every word of paragraph 1
```

the result is:

```
{"Four", "score", "and", "seven", "years", "ago"}
```

In other words, the command asks for all the words, and AppleScript returns an itemized list of them. By asking for every word—instead of the paragraph—we get just that: every word in a form that allows us



Chapter 8:
**Describing
Objects—
References and
Properties**

to perform convenient AppleScript item-by-item parsing or looping.

If you look at the last word AppleScript extracted from the paragraph, you also get a clue about how important spaces are to TextEdit's definition of a word. Any punctuation immediately preceding or following a word from the language is ignored. This definition, I emphasize, is determined by TextEdit, and may be different in other text-based scriptable applications.

Things can get pretty granular, if you like:

```
get every character of "Big Bird"
(* result: {"B", "i", "g", " ", "B",
           "i", "r", "d"} *)
```

Or if you go the other way:

```
get every paragraph of document 1
```

the returned value is a list, with each paragraph as a quoted string item in that list.

AppleScript returns an empty list (instead of an error) if your reference to a list specifies every object of a class that is not contained in the list. For example,

```
get every string in {1,2,3}
-- result: { }
```

Referring to classes of objects actually contained in the list returns the desired values:

```
get every string in {1, "two", 3}
-- result: { "two" }
```

For application objects, you will most likely get

an error when mismatching objects (e.g., **every window in paragraph 1**).

Most applications also define a plural class definition for objects your scripts tend to use all the time. Figure 8.5 shows how this looks in a dictionary entry for an object with a plural class form. Plural class forms are usually just the plural of the object (although in Apple's Address Book, the plural of the **person** class is **people**). You can substitute a plural class form for every element references. For example, the following two statements are identical:

```
get every word of paragraph 1 of document 1
```

```
get words of paragraph 1 of document 1
```

Both return an itemized list of all the words in paragraph 1.

Class word: This subdivides the text into words.
Plural form:
words

Figure 8.5. A plural class form in a dictionary.

You Try It

Open the *Bill of Rights.rtf* document in TextEdit. Then enter and run the following script:

```
tell document 1 of application "TextEdit"
  get every word of paragraph 1
end tell
```



Chapter 8:
**Describing
 Objects—
 References and
 Properties**

Substitute the **get every word of paragraph 1** line, above, with each of the following lines and run the script:

```
get every word of paragraph 2
get words of paragraph 5
get every paragraph
set size of every word of paragraph 1 to 18
set font of every word of paragraph 2 to "Times"
```

Close the *Bill of Rights* document, without saving changes.

To see how this reference works in iTunes, enter and run the following script. It assumes that you have some items in your Purchased Music playlist. If not, the substitute the name for any playlist you have created (or use **playlist "Library"** as the default).

```
tell application "iTunes"
  album of tracks of playlist "purchased music"
end tell
```

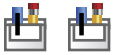
Here we get the values of every album name in the library. Notice that the property name (“album”) is singular, while the references (“tracks”) are plural—this is an important syntax point for this reference form.

Common Mistakes

Incomplete reference; forgetting to specify a property for the objects; using a plural class form when none is defined for that object; using a plural property name.

Related Items (Chapters)

Partial and complete references (8); list value classes (9).



Range References

Syntax

```
className startIndex ( thru | through ) -
  stopIndex
pluralClassName startIndex -
  ( thru | through ) stopIndex
every className from boundaryReference1 -
  to boundaryReference2
pluralClassName from boundaryReference1 -
  to boundaryReference2
```

How to Use

A range reference is a powerful extension to the every element reference. While the every element reference applies to all items of a class, you can use the range reference to target a smaller, consecutive group (or subset, for math-prone folks) of items. You specify the range by indicating the starting and ending points as indexed references.

Let’s walk through an example, using a TextEdit paragraph that reads:

When in the course of human events...

Starting with the every element reference form,

```
get every word of paragraph 1
(* result: {"When", "in", "the", "course",
  "of", "human", "events"} *)
```

But we can limit the range to be just the first three words by saying:



Chapter 8: Describing Objects— References and Properties

```
get every word from word 1 to word 3 of ¬  
    paragraph 1  
-- result: {"When", "in", "the"}
```

The indexed references (**word 1 to word 3**) are inclusive. And, like the every element reference, the data is generally conveyed in a list form.

Notice an important point about this construction: When you compile the statement, AppleScript automatically converts it to the following:

```
get words 1 thru 3 of paragraph 1
```

This is valid syntax for the range reference form. Why did it do this? Because it recognized that all the objects referred to in the original statement were of the same class and were contained by the same paragraph object. In the process of reducing the statement to the *startIndex* variation of this reference form's syntax, AppleScript has made the statement more readable. When an object supports a plural class name (as many objects do in TextEdit), the conversion automatically turns the *className* component to the *pluralClassName* during compilation—all it means is that AppleScript can accept either version from us humans, and it knows what to do with them during compilation.

Note that there may be a bug in TextEdit that doesn't let the syntax conversion operate successfully. If you add a space anywhere in the script and recompile, the script runs as expected.

Lists also can take part in this reference form, using the plural **items** as needed. For example,

```
get items 2 thru 4 of {"eMac", "iBook", ¬  
    "iMac", "Power Mac", "PowerBook"}  
-- result: {"iBook", "iMac", "Power Mac"}
```

For both the *startIndex* and *boundaryReference* components of this form, you can use the AppleScript keywords **beginning** and **end** to signify the respective locations among a series of objects. The previous statement could have also read:

```
get items beginning thru 3 of {"eMac", ¬  
    "iBook", "iMac", "Power Mac", "PowerBook"}  
get items from beginning to 3 of {"eMac", ¬  
    "iBook", "iMac", "Power Mac", "PowerBook"}  
-- result: {"eMac", "iBook", "iMac"}
```

You can also obtain a list of objects in from the end of a group of objects. For example,

```
get items end thru -3 of {"eMac", "iBook", ¬  
    "iMac", "Power Mac", "PowerBook"}  
-- result: {"iMac", "Power Mac", "PowerBook"}
```

While the way the statement reads might lead you to expect the objects to come back in reverse order (i.e., the last object listed first), such is not the case.

You use the *boundaryReference* version of this reference form when the objects involved are of different classes. To illustrate this feature, we'll place a simple paragraph of text into a variable named **myText**, and then apply AppleScript relative references on the contents of the text (outside of an application):



Chapter 8:
**Describing
Objects—
References and
Properties**

```
set myText to "To be or not to be."
```

We start by getting all of the text of the string variable via AppleScript's **text** object:

```
get text of myText  
-- result: "To be or not to be."
```

AppleScript returns the text of the entire document as one long quoted string value (not as an itemized list). But if we want just a range of that text, we can limit what comes back, as in:

```
get text from word 1 to word 3 of myText  
-- result: "To be or"
```

Here we're mixing objects: text and words. Therefore, the *boundaryReference* syntax is required. Notice that the reference for each boundary is a complete reference. AppleScript gets confused if we try a shortcut:

```
get text from word 1 to 3 of myText  
-- result: "To "
```

If the application is up to the task, you can even mix *boundaryReferences*, as long as they make sense. Thus, we can try:

```
get text from character 4 to word 3 of myText  
-- result: "be or"
```

meaning all the text from the fourth character of the string to the end of the third word of the string, inclusive.

If we want to extract the individual characters (as a list) of the first two words, the proper reference would be:

```
get characters from word 1 to word 2 of myText  
-- result: {"T", "o", " ", "b", "e"}
```

But you might be tempted to use another construction, which produces quite different results:

```
get characters of words 1 thru 2 of myText  
-- result: {}
```

What gives? If you examine this statement more closely, you'll recognize that it actually contains two references. The first is an every element reference (in plural class form) to all the characters; the second is a range reference to the first two words of the paragraph. AppleScript begins its evaluation process of the statement by getting the value of the range reference:

```
words 1 thru 2 of myText  
-- result: {"To", "be"}
```

Then it gets the characters of that list, but a list value class doesn't expose characters as elements, causing the result to be an empty list.

What this last issue demonstrates is that you need to be careful with range references that involve mixing object types. Think the reference through, and make sure you haven't combined multiple references that get evaluated differently than you expected.

You Try It

Open the *Bill of Rights.rtf* document in TextEdit. Then enter and run the following script, which operates a little differently from most other examples. We use TextEdit to grab a copy of the complete text of the document, but then use range references on



Chapter 8:

Describing Objects—References and Properties

that text from outside of the tell block, thus avoiding syntactical conflict between AppleScript's own text object and the one defined in the TextEdit dictionary.

```
tell application "TextEdit"
    set myText to text of document 1
end tell
get words 1 thru 2 of myText
```

Replace the bottom line, above, with each of the following lines and run the script:

```
get words 1 thru 10 of paragraph 2 of myText

get text from word 1 to word 10 of -
    paragraph 2 of myText

get text from end to word -5 of -
    paragraph 2 of myText
```

Common Mistakes

Incomplete reference; nesting multiple references when mixing objects; the application doesn't support range references for some object(s); forgetting to specify a property for an object range.

Related Items (Chapters)

Partial and complete references (8); every element reference forms (8); indexed reference forms (8); list value classes (9).

Filtered References



Syntax

reference whose | where *BooleanExpression*

How to Use

The filter reference form is the most powerful of all AppleScript forms. Unfortunately, it is limited to application objects in scriptable applications that support them (like TextEdit and Finder). They may not be used on AppleScript objects, such as lists or other types of values. To know whether an object in an application supports filtered references, look in the Elements listings of various classes, and look for the comment that the element can be referenced by "satisfying a test."

In the real world, when you run a liquid through a filter, you are trying to filter out impurities, ending up with only the stuff you want. That's precisely what a filter reference does with a series of objects: it lets a script extract just the information you want from the whole. Let's start with a TextEdit document with a single paragraph:

```
My sister Susie sells sea
shells by the sea shore.
```

If we ask AppleScript for all the words with the every element reference form, we get them all in a list:

```
tell document 1 of application "TextEdit"
    every word
end tell
(* result: {"My", "sister",
            "Susie", "sells", "sea", "shells",
            "by", "the", "sea", "shore"} *)
```



Chapter 8:
**Describing
Objects—
References and
Properties**

But with the filter reference, we can ask for only those words whose first letter is an “s”:

```
tell document 1 of application "TextEdit"
    every word whose first character is "s"
end tell
(* result: {"sister", "Susie", "sells",
           "sea", "shells", "sea", "shore"} *)
```

What makes this work is that TextEdit tests each **word** object in the container (the whole document, here) to see whether it is true that its first character is the letter “s”. If the test returns a value of **true** (internally), then the word is added to the list of values to be returned to the script. This **true** value is what we mean by the *BooleanExpression* component of this reference form.

Let’s go further, and see how this might be useful in our example. We can extract all the words of the document that match a particular word. A first effort at such a reference might be something like:

```
every word whose word is "sea"
```

but this doesn’t work, because the *BooleanExpression* parameter cannot evaluate correctly at the same object level as the original. To accommodate this dilemma, AppleScript has a special predefined variable: **it**. In a filter reference form, “it” points to each successive occurrence of the object within the series as AppleScript performs its tests through the group. The correct syntax, then, is:

```
tell document 1 of application "TextEdit"
    every word where it is "sea"
end tell
-- result: {"sea", "sea"}
```

It may not seem that you can do much with this information. On the contrary, you now have a list of items that match your filter criteria (like the every element reference form, the filter form returns a list of data matching the criteria). You can count items:

```
tell document 1 of application "TextEdit"
    get every word where it is "sea"
    count of result
end tell
-- result: 2
```

Your choice between the **whose** and **where** keywords in a filter reference is entirely up to your taste in how a phrase reads. The words are interchangeable, although the common terminology for this kind of reference is that an application “supports whose clauses.”

To take full advantage of this reference form, you should have a good grasp of AppleScript’s Boolean operators (Chapter 11). They provide a very full range of possibilities for constructing filter references. Here are some syntax examples to give you ideas:

```
words of document 1 whose length > 10

every paragraph where it contains "normal"

words whose style contains bold

every word where it contains "ing"
```



Chapter 8:
**Describing
Objects—
References and
Properties**

every paragraph where it starts with "Once"

You could say that these examples use the filter reference to find “things that are.” You can use other Boolean expressions, however, to filter results to show “things that are not”:

words of document 1 whose length is not 10

In theory, we should be able to use the **not** operator to invert the returned value of another Boolean expression. For example, to find all words whose style is not bold, we should be able to say:

words whose style not (contains bold)

Using the synonyms available for AppleScript’s comparison operators (whose results are Booleans), we should then be able to make that statement more readable:

words whose style doesn’t contain bold

Incidentally, a Boolean expression can be the aggregate of multiple Boolean expressions. For example, if you want to refer to every word whose first character is “s” and whose last character is “e”, you can combine the two expressions with the Boolean **and** operator. This is what the reference would look like (parentheses added for help in showing the two components:

words where (its first character is "s") and ~
(its last character is "e")

This filter catches only paragraphs that meet both criteria. If one *or* the other would do, use the **or** operator:

words where (its first character is "s") or ~
(its last character is "e")

Feel free to string as many of these smaller Boolean expressions into one larger one as needed to make your selection criteria clear.

Filter references can be tricky to work with, especially the first time you work with them in a new application. If the acceptable syntax for such references isn’t documented with the application, it could take some trial and error to make one work the way you expect. One way to help figure these things out is to test the *BooleanExpression* parameters on some known sample data first. Let’s try this out with the following reference:

paragraphs where it contains "telephone"

Since the Boolean test is whether a paragraph contains a specific word, try each of these scripts:

```
tell document 1 of application "TextEdit"  
    paragraph 1 contains "telephone"  
    -- assuming paragraph 1 really does  
end tell
```

```
tell document 1 of application "TextEdit"  
    paragraph 2 contains "telephone"  
    -- assuming paragraph 2 does not  
end tell
```

You should get “true” and “false” returned values, respectively, in the Result pane for these two scripts, meaning that the Boolean expression does work, and its place in the filter reference is secure.

Although we haven’t shown any examples with the



Chapter 8:
**Describing
Objects—
References and
Properties**

Finder here, you can begin to imagine how powerful filtered references can be when working with the contents of folders. You could, for example, let a script gather a list of all files whose modification dates are after a particular threshold, or whose sizes are greater than so many megabytes.

One final note about testing filter references, especially in the Finder. If there are lots of objects that the filter reference needs to check (e.g., lots of items in a folder), be prepared for a wait. The repeated examination of items is not script execution, per se, so command-period won't necessarily halt the process. You'll just have to wait until it's finished.

You Try It

Open the *Bill of Rights.rtf* document in TextEdit. Then enter and run the following script:

```
tell document 1 of application "TextEdit"
    paragraphs where it starts with "article"
end tell
```

Substitute the **paragraphs where it starts with "article"** line, above, with each of the following lines and run the script:

```
words where it contains "mm"

set size of paragraphs -
    where it starts with "article" to 24

copy "(blank paragraph)\n" to paragraphs -
    where it is "\n"
```

```
copy "\n" to paragraphs -
    where it = "(blank paragraph)\n"
```

Close the *Bill of Rights* document, without saving changes. Next, open the three *Layer* documents. For good measure, open a couple of new, untitled windows, and make them small enough so you can see the Layer windows, too. Enter and run the following scripts to see range references work with other kinds of application objects:

```
tell application "TextEdit"
    name of windows whose name contains "Layer"
end tell
-- result: names of Layer windows only

tell application "TextEdit"
    bounds of windows whose name contains
        "Layer"
end tell
-- result: bounds properties
-- for Layer windows only
```

Close all documents.

With the Finder, you can witness the power of whose clauses in the following script:

```
tell application "Finder"
    name of every item of desktop whose kind
        is "volume"
end tell
```

Common Mistakes

Invalid Boolean expression; improper or unsupported Boolean operator synonym; incomplete reference.



Chapter 8:
**Describing
Objects—
References and
Properties**

Related Items (Chapters)

Boolean operators (13); partial and complete references (8); every element reference forms (8); list value classes (9).

File and Alias References



Syntax

```
file pathname  
alias pathname
```

How to Use

There is considerable confusion about the differences between file and alias references. Part of the confusion stems, I believe, from the fact that “alias” has a specific meaning to most Mac users: a small file that stands in for a Finder item located elsewhere on a hard disk or network. AppleScript alias references are not related to these kinds of alias files (but they do share a helpful characteristic).

The distinctions between file and alias reference types revolve around the way file pathnames are treated during and after script compilation. To help explain the differences, let’s use the following example references:

```
file "HD:Correspondence:Memo to Mike"  
  
alias "HD:Correspondence:Memo to Mike"
```

Both references point to the same file.

When a script contains a file reference, AppleScript does not search the disk volume(s) for the file indicated by the pathname when the script compiles.

In other words, the file does not even have to exist for the script to compile (it may need to exist, of course, for the script to execute, depending on the command that uses the reference as a parameter). AppleScript leaves any error detection about the file’s existence for runtime. If the file exists where stated, then the script runs fine, looking for the file in the precise location in the reference each time. Incidentally, this behavior is what allows the **open for access** command to compile successfully with the name of an as-yet uncreated file when the parameter is a file reference instead of an alias reference.

When a script contains an alias reference, however, AppleScript looks for the file according to that path during compilation (the technical term is **resolving the alias reference**). If the file does not exist at that path, then you receive a compilation error indicating that the file does not exist. With a successful resolution of an alias, some extra magic comes into play. As part of the compilation process, AppleScript creates its own internal alias for the file. AppleScript treats this internal alias just as a user treats an iconized alias in the Finder. No matter where the original file is moved, the alias will know how to find it. Therefore, if you compile a script with an alias reference, you may then move the file into another location on the hard disk—the next time the script runs, it will know how to find the original.

You must be careful, however, about moving a file



Chapter 8:
**Describing
Objects—
References and
Properties**

bearing an alias reference in a script. Each time you compile the script (even if it is to correct a problem in a different place of the script), AppleScript must go through its internal alias creation—the original file must be in the spot indicated in the path of the alias reference at compile time. Before you start moving your aliased files around, the script should be completed and compiled (or saved as an application).

One further distinction between file and alias references is that only alias references may be stored in variables directly (for use as parameters to other commands in a script). For example,

```
set myFile to alias "HD:Tables:States"
tell application "FileMaker Pro"
    open myFile
...

```

For file references, however, the variable must contain a reference class value to the file (Chapter 9), and the reference should have some application context, as in

```
tell application "Finder"
    set myFile to a reference to file "HD:
    Tables:States"
end tell
tell application "FileMaker Pro"
    open myFile
...

```

In both cases, of course, you can save just the string part of a pathname in a variable, and turn it into the desired reference with the file command:

```
set myFile to "HD:Tables:States"
tell application "FileMaker Pro"
    open alias myFile
...

```

The upside of this clarification is that virtually every command that works with existing files compiles and runs with either a file or alias reference. Therefore, there is rarely a need to convert one reference to another. But should the need arise, here's how to coerce an alias reference to a file reference:

```
set myFile to (choose file with prompt "Pick
one") -- returns alias
set myFile to a reference to file (myFile as
string)

```

Converting any pathname string to one of these references is as simple as prepending the reference name to the string. For example, here we assemble a pathname string to a file in the System folder, and then pass the alias form to a command:

```
set pathname to ((path to desktop) as string)
& "Lab.sct"
get creation date of (info for alias pathname)

```

In the first line, we extract the text-only pathname from the alias reference returned by the **path to** command, and stick the file name on the end. Then we convert the value to an alias reference in the second line as we pass it as a parameter to the **info for** command.



Chapter 8:

Describing Objects— References and Properties

Common Mistakes

Assigning a reference to a variable already containing the reference; trying to compile an alias reference to a non-existing file.

Related Items (Chapters)

File-related scripting additions (7); scriptable Finder (18).

Next Stop

So far we've seen the commands we send to objects and the process of making sure the commands get to the desired objects. In the next chapter, we cover the third and last big topic: working with the information—values—that our scripts shuffle about.



Chapter 9 Working with Data—Values, Variables, and Expressions

A strong argument exists for saying that computing is all about information—data. Most of us use our Macintoshes for retrieving, creating, storing, and massaging information in the form of words, pictures (still or moving), sounds, and combinations thereof. Scripting’s role in this information-filled environment has evolved from simple macro processing—the electronic equivalent of clicking buttons—to directly manipulating information. A scripting language, such as AppleScript, can extract or insert information and even make decisions based on the content.

AppleScript calls any kind of information a *value*. Think about a paper application form from the real world. One of the blank spaces would be a kind of object, and what you write in the blank becomes the value of that object.

Kinds of Values

Some blanks on a form expect information entered into them to be of a certain type before the information is deemed valid. For example, a text entry field for someone’s name wouldn’t look strange if letters and numbers appeared in there (John Smith, 3rd). But the field asking for an age is usually small, denoting its expectation of the entry of numbers only (“32”, not “thirty-two”). Therefore, the value of the name field could be any combination of numbers and letters, while the value for the age field must be a number.

AppleScript calls each different type of value a *value class*. This class terminology comes from the object-oriented programming world. You can think of a class as a master copy of something: anything belonging to the same class behaves the same way. Therefore, AppleScript treats all values of the same class identically (e.g., you can use arithmetic operators with all values of the integer class).

By dividing values into classes, it helps scripters know how to work with data that flows through a



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

script. We know, for instance, that to join together two values of the string class, we use the concatenate (“&”) operator. Classes also help us know what kinds of information can work together. For example, the expression

```
3 + "Fred"
```

won’t work, because it applies an arithmetic operator (+) to a value (“Fred”) that doesn’t understand that operator. Conversely,

```
3 & "Fred"
```

doesn’t accomplish what you might expect, because the concatenate operator (“&”) doesn’t work with integer class values.

But if we turn that integer into a string value, then the operator works, creating a valid expression:

```
"3" & "Fred"  
-- result: "3Fred"
```

One more advantage to classifying values in this way is that when we look at a formal definition for a command or reference, the definition generally states what class of value is required for the various parameters and what class of value (if any) a command returns. Therefore, when we see that a property of an application’s object is a value of the Boolean class, we know that any value that satisfies the definition of a Boolean class value can be assigned to that property.

Variables and Value Classes

I introduced you to the concept of variables in Chapter 5. There you saw how a variable is a named holder of a value or series of values. A variable name is formally known in AppleScript as an *identifier*—a term you may see in error messages that refers to a variable. Most of the data we work with in scripts spends at least part of its time in variables. Even if we use the simple **get** command to extract a piece of information, the value goes into the predefined AppleScript variable named **result**:

```
get 5 + 5  
display dialog result
```

We can plug variables into slots for command and reference parameters (like we just did for **display dialog**), because AppleScript *evaluates* the variable to its contents before the contents are passed as parameters.

While a variable you define in a script can contain a value of any class (unlike some other programming languages, you don’t have to tell the variable ahead of time what class of value it will be holding), a variable evaluates to the class of the data it holds. For example, if we assign an integer to a variable, we can use that variable in any operation that works with integers:

```
copy 25 to yourAge  
yourAge + 1  
-- result: 26
```

At the same time, a variable evaluates to the same



Chapter 9:

Working with Data—Values, Variables, and Expressions

class as the value that goes into it. Thus, the following sequence does perhaps unexpected things when trying to concatenate a numeric variable value to a literal string:

```
copy 25 to yourAge
yourAge & " Fred"
-- result: {25, " Fred"}
-- list value of the components
```

While we could turn an earlier example (3 & “Fred”) into a valid expression by placing quotes around the integer—turning the integer into a string—you cannot do that directly with a variable. Putting quotes around a variable name turns the name (*not its value*) into a literal string:

```
copy 25 to yourAge
"yourAge" & " Fred"
-- result: "yourAge Fred"
```

To accomplish the desired task, you must use a facility of AppleScript that allows a script to change a value from one class to another: *coercion*.

Coercing Value Classes

AppleScript uses the dictionary definition of “coercion” (the application of force) in the way it lets us alter a value’s class. The most readable method is to use an AppleScript operator—**as**—to force the class issue. This operator is covered in Chapter 11, but it’s important to understand it in the context of the values we’re discussing here.

A script can change the class of a value by adhering to this syntax:

```
expression as className
```

where *expression* is the value you want changed, and *className* is the class you want it to become. Here are some examples of coercion with hard-wired values and their results:

```
100 as string
-- result: "100"
"Joe Blow" as list
-- result: {"Joe Blow"}
45 as real
-- result: 45.0
"1024" as integer
-- result: 1024
```

Coercion is a critically important concept to master in AppleScript, because quite often a command will receive results of one class, which your script must then feed as a parameter to another command in another class. We’ve seen this with some of the file commands and scripting additions in Chapter 7. If we use **choose folder** to get an alias class value of a folder, we need to convert the value to a string to



Chapter 9:

**Working with
Data—Values,
Variables, and
Expressions**

append a file name that we can ultimately feed to the **open for access** scripting addition command. Here's one way to do it:

```
get (choose folder with prompt ~  
    "Choose your Data folder:") as string  
set myPath to result & "Test Data"  
set myFile to (open for access file myPath ~  
    with write permission)  
...
```

In this case, the **choose folder** command returns an alias class value, which is immediately coerced to a string class value before going into the **result** variable. At this point, **result** contains a complete path to a folder, including the trailing colon. We then concatenate a file name to the path, and store the combination string in a variable named **myPath**. That string value is then passed as a parameter to the **open for access** command, which expects a reference in the form of either a file or alias reference. As you saw in Chapter 8, you can create such a reference by preceding a path string with either reference name (**file** or **alias**). Incidentally, we could have also done the coercion later:

```
get (choose folder with prompt ~  
    "Choose your Data folder:")  
set myPath to (result as string) & "Test Data"  
set myFile to (open for access file myPath ~  
    with write permission)  
...
```

In both scripts, parentheses help AppleScript more accurately perform necessary evaluations. In the

first, we want AppleScript to get the alias reference from the **choose folder** command intact—and then we coerce it. In the second script, we need AppleScript to perform the coercion on the **result** variable before sending it as a parameter to the **open for access** command.

With all this in mind, we can go back to the **yourAge** example above, and coerce the variable containing the age value from its original integer value to a string for the desired concatenation:

```
copy 25 to yourAge  
(yourAge as string) & " Fred"  
-- result: "25 Fred"
```



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Coercion Caveats

As cool as coercion sounds, you must be realistic about it. Just as you cannot turn yarn into gold (except in fairy tales), some values simply can't be coerced to specific classes. For example, a string of alphabet characters can't be made into a numeric class of any kind. Also, a real number with a decimal fraction cannot be made into an integer (although you can round the number, and then coerce it).

One other gotcha afflicts primarily the application, script, file, and alias class values, all of which use a file path name as a further parameter. Coercion among these classes doesn't work the same way as for other class coercions. That's why we have to use a construction like this:

```
get (choose folder with prompt ¬
    "Choose your Data folder:") as string
set myPath to result & "Test Data"
set myFile to (open for access file myPath ¬
    with write permission)
...
```

instead of:

```
get (choose folder with prompt ¬
    "Choose your Data folder:") as string
set myPath to result & "Test Data"
set myFile to (open for access ¬
    (myPath as file) with write permission)
...
```

If we try the latter, AppleScript advises us that it's not possible to coerce a string class to a file class. For these kinds of classes, it's best for now to coerce one type to a string, and then pass the string as an

argument after the command and the class name, as demonstrated in the prior example.



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Value Class Details

In the next several subsections, we'll examine each of AppleScript's value classes. To help you learn about them in the first place, and refer to them later for reference, I've divided the classes into three categories:

Common	Less Common	Rare
Boolean	Date	Class
Integer	File Specification	Constant
List	Record	Data
Number	Reference	International Text
Real	RGB Color	Styled Clipboard Text
String	Styled Text	Unicode Text
Text	(Unit Types)	

As you learn AppleScript for the first time, you can get pretty far just with the common value classes. You can become familiar with the rest as you run into them in the course of expanding your AppleScript experience.

Boolean Class



Computerdom owes a big debt to the nineteenth century mathematician, George Boole. He developed an arithmetic system based on the logic of true and false properties. These two states play a large role in computer languages, because they can be used to specify common real world states: on/off; yes/no; true/false; one/zero.

AppleScript uses Boolean class values of **true** and **false** (no quotes around the words, and they may be upper or lowercase) for these situations. For example, one property of an application's document may be whether its contents have changed since the last time it was saved. The property is defined as being **true** if the contents have been modified; **false** if they're the same as when the document opened. The minute a user or script changes the contents, the program internally, and automatically, changes the property from **false** to **true**. Our scripts can read that property, which comes back as the unquoted text:

```
true
```

Variables can hold Boolean values just as they can any other value. For example:

```
set modifiedState to modified of document 1
```

stores **true** or **false** into the **modifiedState** variable.

I devote the majority of Chapter 11 to making comparisons in scripts—actions that rely on Boolean



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

results from operators, such as:

```
4 > 3
-- result: true
10 = 20
-- result: false
"mouse" contains "us"
-- result: true
"Microsoft" comes before "Apple"
-- result: false
```

All these expressions evaluate to Boolean values. As a result, they are considered Boolean expressions.

Multiple Boolean expressions can be combined into a single expression that evaluates to a single **true** or **false**. The glue that holds these Boolean expressions together are the **and** and **or** operators. Consider the expression:

```
(10 < 20) and ("a" comes before "b")
-- result: true
```

Evaluation of this entire expression begins with each of the smaller expressions. In this case, *both* smaller expressions evaluate to **true**. Then the **and** operator, working on the two **true** values (**true and true**), returns the final evaluation: **true**. If one of the smaller expressions had evaluated to **false**, then the entire expression would be **false**:

```
(10 < 20) and ("z" comes before "a")
-- result: false
```

That's because the **and** operator worked on dissimilar values (**true and false**).

The **or** operator returns a **true** if either of the two sides of the expression returns a **true**:

```
(10 < 20) or ("z" comes before "a")
-- result: true
```

In other words if the left side is **true** or the right side is **true**, then the expression is **true**.

All this **and** / **or** operator stuff explains how we can combine multiple Boolean expressions into things like filtered references:

syntax:

```
reference whose | where BooleanExpression
```

example:

```
every paragraph where ~
  (it starts with "The") and ~
  (it contains "general")
```

The two smaller Boolean expressions are applied independently to each paragraph of the document. If both expressions prove to be true, then the paragraph qualifies for this filter.

A Boolean value's class name is **boolean**. A Boolean value can be coerced only to a list class (representing a single-item list).

See Chapter 11 for more about operators that affect Boolean values. Chapter 10 shows Boolean values in use for **if-then** decisions.



Chapter 9:
**Working with
 Data—Values,
 Variables, and
 Expressions**

Class Class



No, this isn't doubletalk, but this value class starts to make all this class stuff sound somewhat recursive. Anything in the AppleScript world that is considered an object belongs to a class. For example, in our Chapter 8 discussions about references, most references require a parameter called *className*. In these object references, the class name is the type of object we're referring to (e.g., **word**, **paragraph**, **cell**, **row**). In the definition of the indexed reference form

```
className index
```

the *className* parameter expects a class value—one of the object class names defined for the application's AppleScript dictionary (within the Classes grouping). Therefore, in the partial reference

```
word 6
```

“word” is the class value, and “6” is the index value.

This probably sounds more complicated than it is. By and large, you won't think much about the class value, because it becomes a virtually automatic part of object references.

But micromanagement of this value can be helpful at other times. For example, every value discussed in this chapter has a **class** property, which allows a script to determine what type of value—class—it is. A script can use this information to decide how to handle a value. Let's say a script subroutine handler receives some data as a parameter that may be a real

or integer class value. The handler can make sure it handles each type accordingly:

```
on mySubroutine(passedValue)
    if class of passedValue is real then
        (* do real number processing *)
    else if class of passedValue is integer then
        (* do integer processing *)
    end if
end mySubroutine
```

Most of the value classes covered in this chapter have a **class** property consisting of the name of the value class (**boolean**, **class**, **constant**, **data**, **date**, **file specification**, **integer**, **list**, **real**, **record**, **reference**, **string**, **styled text**, **unicode text**). Notice, too, that the class name, when referred to in a script, is not a quoted string:

```
class of "Howdy Doody"
-- result: string
-- not "string"
```

To bring this discussion full recursive circle, the **class** property of the class value is, well, **class**. The first time you encounter this stuff, it can make your head hurt, but once you see enough value class references in property descriptions in application dictionaries, it begins to make sense.



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Constant Class



Several commands and objects provide in their definitions a number of words that appear to be hard-wired as possible parameters. For example, in TextEdit, an optional parameter to the **close** command lets you specify how to handle the saving of a changed document. The syntax definition is:

```
close reference [saving yes/no/ask]
```

Arguments for the **saving** parameter can be one of the three choices provided. These three words are constant class values. Their contents are predetermined by the application and cannot be changed. They are also represented in script lines without quotes (which would turn those words into strings). Therefore, if we ask for the class of one of these values, AppleScript tells us it's a constant class:

```
class of yes
-- result: constant
```

A constant class is different from what other programming languages might call a constant. For example, while **true** and **false** are reserved words in AppleScript (and are even listed as constants in the *AppleScript Language Guide*), their value class is **boolean**.

By and large, you won't have to worry much about the class of constant values because the values are presented as options in definitions where needed. The only thing you have to remember is that when a definition specifies one of these constants, you must use one of the ones provided. Incidentally,

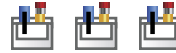
you cannot coerce a string to a constant to fill one of these parameters, although AppleScript does allow assigning a constant to a variable, which then assumes the constant class.

The class name of a constant class value is **constant**; a constant class value may be coerced to a single-item list value or string.



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Data Class



Because scripts and results (as viewed in Script Editor) deal only in text, there are other kinds of information that can't be represented or viewed in the scripting environment. But very often, our application of AppleScript depends on moving this kind of data—graphics, sounds, movies—from one document to another.

In earlier implementations of AppleScript, it was not uncommon to see values holding binary data from applications to show themselves in the Script Editor result window in a form such as the following:

```
«data PICT021F0000000000C701»
```

Nowadays, values that point to an application's document objects maintain their references for display in the Result pane. Thus, even in script statements that refer to, say, a Microsoft Excel chart object point to that object within the Excel application:

```
ChartObject "Chart 1" of Worksheet "Sheet1" ↵  
  of Workbook "Workbook1" ↵  
    of application "Microsoft Excel"
```

In this case, the object value does not reveal a class name.

A data class value can be coerced only to a single-item list value.

Date Class



Values of the date class are in many ways similar to the classes that signify things like applications, files, aliases, and scripts: the class name appears as part of the value, and you cannot coerce another class to a date class with the **as** operator. The **current date** scripting addition, for example, returns a date class value:

```
current date (* result: date "Tuesday,  
November 9, 2004 3:53:02 PM" *)
```

The leading word, “date,” is required for any subsequent string to be treated as a date value.

While the amount of information in a date value appears rather large (day, month, date, year, hour, minute, second, AM/PM designation), it's not necessary to give AppleScript every part of the information for it to convert a string to a date. Here are examples of date values AppleScript accepts (e.g., when assigning these values to a variable):

```
date "12/25/04"  
date "12/25/2004 1:01 PM"  
date "Dec 25, 2004 1 PM"  
date "1:01 PM"
```

In all cases, when some pieces are missing, AppleScript does its best to fill in the blanks. For example given any date, AppleScript calculates and returns the full day name, month name, four-digit year, and the midnight time (12:00:00 AM) for that date. Given any time by itself, AppleScript applies today's date. In fact, if you simply enter



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

any date value (like the above examples) as a script and compile that script, AppleScript automatically converts what you entered to the full value.

A number of valuable operators work on date values—primarily those that let you test for the equality of dates or whether one date precedes another. Bear in mind that the date value includes whatever time is attached to it. For example, let's say today's date is 11/9/2004. A script that performs this test:

```
date "11/9/2004" = current date
-- result: false
```

returns a **false**, because the left operand evaluates to midnight on the date, while the **current date** command will most likely return a time on the same day that is after midnight. The **current date** evaluates to a value greater than "11/9/2004."

Working with Dates and Times



To work successfully with dates and times in AppleScript, you'll need to have a good grasp of what a date object offers your scripts. Each time you create a date object, it represents a snapshot of time—it is not a "living" clock. You can create a date object with an initial date and time set to anything you want (well, at least from the year 1000 onward); or use the **current date** scripting addition to get today's date and the current time. You can modify individual properties of a date object (e.g., to get the date 10 days from today) and compare the values of two different date objects, if you like. The date object is the gateway to this power.

The Date Object

A date class value by itself is not particularly helpful for performing calculations with dates such as figuring the number of days between dates or finding the date two weeks from today. Fortunately, AppleScript treats dates as objects that contain a number of properties that help in this arena.

Here are the properties for a date object:



Chapter 9:

Working with Data—Values, Variables, and Expressions

Property	Value Class	Example
weekday	class	Wednesday
month	class	December
day	integer	8
year	integer	1993
date string	string	"Wednesday, December 8, 1993"
time string	string	"5:25:12 PM"
time	integer	36000 (seconds since midnight)

It's important to note that the values for **weekday** and **month** properties are not strings, but values of the class `class`. For example, if you wish to use the values in a comparison, then use the values as-is:

```
if weekday of (current date) is Friday -  
    then beep
```

But if you need to capture the weekday or month for use in assembling strings, you must coerce the property values to string classes:

```
set msg to "Today is " & -  
    (weekday of (current date) as string)
```

You can change a date object's value by assigning new values to the **month**, **day**, and **year** properties of a date value. AppleScript does its best to calculate the meaning of the adjustment. For example, if you assign a date value to a variable, and set the year property, AppleScript recalculates the date based on that year:

```
copy date "Sunday, June 5, 2005 12:00:00 AM" -  
    to myDate  
set year of myDate to 2006  
myDate
```



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

```
(* result: date "Monday, June 5, 2006  
12:00:00 AM" *)
```

where the day changes to Monday, which is the day of the week for the date in the following year. Moreover, if you set the **day** value to an integer greater than the number of days in the month of the date value, then AppleScript calculates the date of the fictitious day:

```
copy date "Sunday, June 5, 2005 12:00:00 AM" →  
to myDate  
set day of myDate to 35  
myDate  
(* result: date "Tuesday, July 5, 2005  
12:00:00 AM" *)
```

Even so, see below for better ways to deal with date arithmetic.

One potential disappointment among scripters is that the date string and time string properties come in only one flavor each. Without the help of third party scripting additions, you cannot have AppleScript deliver short versions of those strings (e.g., “6/5/2005” or “5:25 PM”) with reliability across Date & Time preference settings.

Date and Time Arithmetic

You may perform the following kinds of arithmetic on dates and times:

- ▶ subtracting one date from another to achieve the difference in seconds
- ▶ adding time (in seconds) to a date to achieve a new date

- ▶ subtracting time (in seconds) from a date to achieve a new date

The *second* is the key unit of time and date measurement in working with AppleScript dates and times. When you subtract one date from another, the result of the operation is returned in seconds. For example:

```
set firstDate to date "1/1/2005"  
set secondDate to date "2/1/2005"  
secondDate - firstDate  
-- result: 2678400
```

The result in this example is the number of seconds in the 31 days between the two date values.

To help make the math appear more like plain language, a helpful set of AppleScript constants define standard chunks of time:

Constant	Number of Seconds
minutes	60
hours	60 * minutes
days	24 * hours
weeks	7 * days

To let AppleScript figure out the number of days between the dates in the previous example, we’d modify the script as follows:

```
set firstDate to date "1/1/94"  
set secondDate to date "2/1/94"  
(secondDate - firstDate) / days  
-- result: 31.0
```




Chapter 9: Working with Data—Values, Variables, and Expressions

To add time to, or subtract time from, a date value, you must use either seconds or the constants listed above. For example, if your script wants to record the date for ten days from today, the script would look like this:

```
set nextAppointment to (current date) + -  
    (10 * days)
```

You aren't restricted to integer calculations here, either. To get the time 3-1/2 hours from now,

```
set nextAlarm to (current date) + -  
    (3.5 * hours)
```

The only restriction in using these constants in addition or subtraction operations is that the date value must come first in the algebraic expression for AppleScript to know how to evaluate the constants. Also, the constants are defined only as plurals, so you must use the plural form even if the amount of time is singular, as in:

```
set nextWeek to (current date) + (1 * weeks)
```

or

```
set nextWeek to (current date) + weeks
```

Despite the simplicity of the AppleScript date class, it has sufficient innate powers to handle a wide range of time and date arithmetic tasks.

Year Formats

While date references can accept a number of variants (above), you should exercise care when depicting a year with fewer than four digits. The

re-use of the last two digits every century wreaks havoc with the Macintosh's internal date tracking. To be somewhat helpful, the Mac toolbox interprets two-digit years to be years near the mid-1990s. For example:

```
date "Sep 5, 94"  
(* result: date "Monday, September 5,  
    1994 12:00:00 AM" *)
```

```
date "Sep 5, 05"  
(* result: date "Monday, September 5,  
    2005 12:00:00 AM" *)
```

But beyond a very specific range, the results may be other than what you expect. Use the following table as a guide:

Two-Digit Entry	Converted to Year
00-90	2000-2090
91-99	1991-1999

These two-digit entry conversions apply only to date class entries, not to the **year** property of a date value. For example, if you attempt to set the **year** property of a date value to a two-digit value, the year will be the actual two-digit year:

```
copy date "3/4/2005" to someDate  
(* result: date "Friday, March 4, 2005 -  
    12:00:00 AM" *)  
set year of someDate to 95  
(* result: date "Friday, March 4, 0095 -  
    12:00:00 AM" *)
```



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

To be on the safe side, use four-digit years whenever specifying dates as date value classes.

A value of the date class can be coerced to a single-item list or to a string. In the latter case, the entire date and time strings are returned as part of the string value.

File Specification Class



You will occasionally see references in AppleScript dictionaries to the file specification class value, usually in connection with a file-related command. This kind of value represents a file that does not yet necessarily exist on a volume, but will soon be opened and written to by an application. An earlier incarnation of the **choose file name** scripting addition returned a value of the file specification class.

If you find that you have a value of this class stored in a variable (put there, for example, as a value returned by an application command), you can coerce the value to a string. This will leave you with a string version of the full path name to the file.



Chapter 9:

Working with Data—Values, Variables, and Expressions

Integer Class



An integer is any positive or negative number (including zero) that does not also contain any decimal fraction. The following values are integers:

```
0
-1
1
495
23948570
```

The following values are *not* integers, because they are represented as having a fractional part:

```
0.1
1.0
-45.0
3.14159
```

The distinction between integers and other kinds of numbers (real number values, below) is primarily necessary because of the way computers and programs work. Internally, computers work with integers and real numbers (also called floating-point numbers in other environments) very differently. As a result, we who write programs or scripts are saddled with having to make the distinction in our values. While we may think 12 and 12.0 represent the same amount of something, the computer has to treat them differently for calculation purposes.

AppleScript integers may be in the range between -536870911 to +536870911. Any number outside that range is automatically coerced to a real number value class during compilation (and displayed in exponential notation). Note that all large numbers in

AppleScript (integer or real) do not allow commas or other delimiters between numerals. The value for 100,000 must be entered as:

```
100000
```

In some dictionary entries, you may see class references to “small integer.” AppleScript makes no distinction between categories of integers, as some compilers for other programming languages do. Typically, a small integer means any value in the range of 0 to 255. This works for something like the **ASCII character** scripting addition, whose parameter must be within that range. To AppleScript, an integer is an integer is an integer. I’d rather see dictionary definitions simply state the integer class for the parameter, and supply the required range in the comment following that item.

You may coerce an integer class value to a single-item list value or a real class value. For example:

```
get 12 as real
-- result: 12.0
```

If you coerce an integer value to the number class, the class property of the value continues to report **integer**.



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

International Text Class



A value of the international text class is similar to a text or string value, but it contains some additional information that specifies the language and (writing) script codes of the text value. Mac systems that include support for the designated language use these codes to render the correct characters. By including support for the international text class value, AppleScript allows for the transfer of text data between applications, guaranteeing that the extra internationalization codes survive the journey from one **tell** block to another.

If your Macintosh and applications are equipped to handle international text values, you will be able to test for the class name in your scripts, but don't expect to see the resulting text rendered in places such as the Script Editor Result pane. You may, however, coerce an international text class value to a plain string, in which case the extra international codes are stripped from the value. International text class values may also be coerced to styled text and Unicode text value classes.

List Class



AppleScript relies heavily on the list value class. It's worth the effort to spend time learning about lists early in your AppleScript learning process, because you'll use them a lot for carrying and manipulating data.

A list is defined as an *ordered collection of values*. What this means is that a single list value can carry around multiple values. We've seen that some commands expect or are ready to receive multiple pieces of information for a single parameter. For example, the **display dialog** scripting addition command lets a scripter pass with the **buttons** parameter the names of up to three buttons. The convenient method of passing the names as a group is the list value.

Each component of a list is called an *item*, and a single list may contain any number of items whose individual values may be from any class (i.e., items within a list don't have to be the same value class).

You can recognize a list by its curly braces. An empty list looks like this:

```
{ }
```

That is a valid list class value, whose number of contained items is zero.

More likely, you'll see lists of similar data:

```
{ "Howdy", "Doody" }
```

This list contains two string values. Here's a list with



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

values of three classes (string, real, and Boolean):

```
{ "piano", 88.0, true }
```

The majority of the lists you will deal with in scripts come as the returned values of commands to objects. A number of reference forms (particularly the every element reference form and those derived from it) return the contents of an object as a list. In fact, if a result has an ability to contain multiple pieces of information, the returned value will be in the list class, even if the actual data returned is one or zero items. For example, if a TextEdit document consists of this sentence,

```
Let them eat cake!
```

then here's a script that gets all the words of that document:

```
tell application "TextEdit"
  get every word of document 1
end tell
-- result: { "Let", "them", "eat", "cake!" }
```

Once the data is in this form, a script has a great deal of flexibility due to the special status a list has in AppleScript.

When all items of a list are string values, you can coerce the list to a string class value, in which case all items are concatenated together into a single string. But beware that the coercion jams all the individual item values together. For example, if the list's items are individual words from a document, coercing them to a string or text value class will cause the words to be strung together in one huge, gibberish

word:

```
set wordList to ~
  { "Let", "them", "eat", "cake!" }
wordList as string
-- result: "Letthetheatcake!"
```

But here's the magic to get a list of words to yield a sequence of space-delimited words:

```
set wordList to ~
  { "Let", "them", "eat", "cake!" }
set text item delimiters to " "
-- that's a space character
wordList as string
-- result: "Let them eat cake!"
```

See the discussion about the string class for more about the **text item delimiters** global property.

A List Object

While every value is considered an object within AppleScript, a list has more of the traditional pieces you would expect: defined elements and a complement of properties to help you work with contents of the list.

The elements of a list are called items. An item is one of the values inside a list, separated from other items by commas:

```
{ 1, 3, 5, 7, 11 } -- five elements
{ "Joe", true, 45 } -- three elements
```

To set or get the value of one item, a script refers to the item by its index value, with the leftmost item being item 1:



Chapter 9:

Working with Data—Values, Variables, and Expressions

```
item 4 of { 1, 3, 5, 7, 11 }  
-- result: 7 (an integer)  
item 2 of { "Joe", true, 45 }  
-- result: true (a Boolean)
```

You can place a list in a variable, and then manipulate the items of the list via the variable:

```
set bigCities to -  
  { "London", "Tokyo", "New York" }  
set item 3 of bigCities to "Los Angeles"  
bigCities  
-- result: { "London", "Tokyo", "Los Angeles" }
```

A script may also add items to the beginning or end of the list with the help of the concatenation (&) operator:

```
set bigCities to -  
  { "London", "Tokyo", "New York" }  
set bigCities to bigCities & "Los Angeles"  
bigCities  
(* result: { "London", "Tokyo",  
  "New York", "Los Angeles" } *)
```

Concatenation also works if the two groups you're joining are both lists. The result is one list consisting of elements from both original lists:

```
set list1 to {1, 2, 3}  
set list2 to {9, 8, 7}  
set jointList to list1 & list2  
-- result: {1, 2, 3, 9, 8, 7}
```

You can also insert an item at the front or end of an existing list by specifying where in the list the new item should go. For example:

```
set bigCities to -  
  { "London", "Tokyo", "New York" }  
set end of bigCities to "Los Angeles"  
bigCities  
(* result: { "London", "Tokyo",  
  "New York", "Los Angeles" } *)
```

and

```
set bigCities to -  
  { "London", "Tokyo", "New York" }  
set beginning of bigCities to "Los Angeles"  
bigCities  
(* result: { "Los Angeles", "London",  
  "Tokyo", "New York" } *)
```

These ways of adding single items to a list move less data around, resulting in a more efficient way to assemble a long list.



Chapter 9:

Working with Data—Values, Variables, and Expressions

Three properties of a list object are helpful in scripts.
They are:

Property	Value Class	Description
rest of	list	items in the list except for the first one
reverse	list	items in the list in reverse order
length	integer	number of items in the list

The **rest of** property can be useful when a script needs to examine or operate on the content of each element in turn. In the following example the list grows smaller by one each time through a repeat loop:

```
tell application "TextEdit"
    set wordList to words of document 1
    repeat until wordList is { }
        (* do something with each word *)
        set wordList to rest of wordList
    end repeat
end tell
```

This example shows an alternate to performing a repeat loop using the number of items in the list as a repeat counter. But you can also use the **rest of** property to construct a list when you know the original data coming back from another command has a leading item in the list that you don't want.

The **reverse** property allows a script to extract a copy of a list, but with items in reverse order:

```
reverse of {1,2,3}
-- result: {3,2,1}
```




Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Retrieving this property does not alter the order of the original list.

A list's **length** property is a read-only value of the number of items—of all value classes—in the list:

```
length of {"sunny", 65, "35%"}  
-- result: 3
```

Notice that this property ignores the contents of the items, themselves. If you want to get the length of an individual item within the list, construct the reference accordingly:

```
length of item 1 of {"sunny", 65, "35%"}  
-- result: 5
```

The **length** property duplicates the work of AppleScript's **count** command, which works on lists. Thus,

```
count of {"sunny", 65, "35%"}  
-- result: 3
```

provides the same results as requesting the **length** property. It's a good habit, however, to use the **count** command, because it can be more selective in what it counts:

```
count of integers of {"sunny", 65, "35%"}  
-- result: 1
```

Like many other objects, you can use a number of reference forms to access items within a list. Here are the ones that work:

Reference Form	Example
Property	length of { 1, 2, 3 }
Index	item 3 of { 1, 2, 3 }
Middle	middle item of { 1, 2, 3 }
Arbitrary	some item of { 1, 2, 3 }
Every Element	every item of { 1, 2, 3 }
Range	items 1 thru 2 of { 1, 2, 3 }

The last two references, since they have the capability of returning multiple items, evaluate to list values. All the rest return results in the class of the item extracted from the list (integer, string, etc.). Some of the more powerful reference forms—relative, filter, name, and ID—don't work with lists.

Nested Lists

You may encounter lists within lists. Applying the rules above, you should be able to work with them (or figure out how you got them in the first place when you didn't expect them).

If we take our "Let them eat cake!" document from above, we saw what happened when we get the words of the document:

```
tell application "TextEdit"  
    get words of document 1  
end tell  
-- result: {"Let", "them", "eat", "cake!"}
```

But look what happens when we alter the **get** command to get all the characters of all the words in the document:



Chapter 9:

Working with Data—Values, Variables, and Expressions

```
tell application "TextEdit"
    get characters of words of document 1
end tell
(* result: {"L", "e", "t"}, {"t", "h",
"e", "m"}, {"e", "a", "t"}, {"c",
"a", "k", "e", "!"} *)
```

We end up with lists within an outer list. That's because the command first acted on getting the words as a list, followed by dissecting the words into smaller elements. To work with the data, however, we have to be cognizant of the item status of this giant list. There are four items of the outer list:

```
tell application "TextEdit"
    get characters of words of document 1
    length of result
end tell
-- result: 4
```

Each of those items is a list:

```
tell application "TextEdit"
    get characters of words of document 1
    count of lists of result
end tell
-- result: 4
```

To work with the characters of a given word, we need to extract the word's list of characters, as in:

```
tell application "TextEdit"
    get characters of words of document 1
    get item 1 of result
end tell
-- result: {"L", "e", "t"}
```

Now we can treat those items as we would any list. Of course, we could also directly reach one of the characters by combining item references:

```
tell application "TextEdit"
    get characters of words of document 1
    get item 1 of item 1 of result
end tell
-- result: "L"
```

Let's apply some of this list knowledge to a real script example that neatly overlaps all open windows in TextEdit. The script, below, is available in the companion files in the *Handbook Script/Chapter 09* folder under the name Item Value Practice. I'll explain what it does after you study the code:

```
-- Item Value Practice
tell application "TextEdit"
    -- start with number of windows
    set windowCount to count of windows
    -- prepare first window's bounds
    -- property value
    if windowCount = 0 then
        display dialog "There are no open
        windows to arrange." buttons {"Phooey"}
        default button 1
        else
            -- do this only if there
            -- are open window(s)
            activate -- so we can watch
            -- for each window...
            set nextWindowBounds to {1, 39,
            481, 305}
            -- loop through all the windows
            repeat with i from 1 to windowCount
                -- set the bounds property
                set bounds of window i to
                nextWindowBounds
                -- bring window to
                -- front of pile
                move window i to front
                -- add 20 to each value
```



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

```
-- of bounds property
repeat with j from 1 to 4
    set item j of
        nextWindowBounds to
            (item j of nextWindowBounds) + 20
    end repeat
end repeat
end if
end tell
```

The first steps include getting a count of the open windows at the moment. We assign that number to a variable (**windowCount**) which serves double-duty: first as a value to check to make sure at least one window is open (if not, display a dialog alerting of the fact); second as a counting value later on to make sure we cycle through all open windows.

If there is at least one window open, then we bring TextEdit to the front and establish an initial set of values (in the variable **nextWindowBounds**) that will eventually be assigned to the first window's **bounds** property (a list of four coordinates). Now we use the **windowCount** variable as a limit to the number of times through an activity loop. In that loop we set the actual property of the designated window to the contents of the **nextWindowBounds** variable. In the next line, we bring the newly sized window to the front so it overlaps any other windows previously set. Finally, we include yet another repeat loop, this one working with each of the items in the **nextWindowBounds** property. To effect the overlap in a uniform fashion, we add 20 pixels to each of the four coordinates.

Since AppleScript doesn't let us perform math operations directly on a list item, we set each value of the list to its original value plus 20. This action sets up the coordinates for the next window if execution goes through another loop for another open window. As soon as TextEdit runs out of windows, so does the repeat loop, and execution continues uninterrupted out to the end. All open windows are now in an attractively overlapped state, with each window the same size.

The list object is your friend. It lets you devise simple data structures in which values are stored in a position-specific series.



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Number Class



If you are unsure about whether a value class should be an integer or real, you always have the catch-all number class to count on. By coercing a value to a number class, you let AppleScript determine whether the value falls into the integer or real class based on the value.

Observe what happens to the class name of a variable that receives values coerced as a number class:

```
set myNum to "34" as number
class of mynum
-- result: integer

set myNum to "34.5" as number
class of mynum
-- result: real
```

Therefore, if your script will be coercing a string value to a number, and you can't predict what kind of number it will be (e.g., a user types a value into a dialog box), you can coerce it to a number for your math, and let AppleScript worry about the details.

Real Class



In AppleScript, a real number is any positive or negative number that includes a decimal point. The number doesn't have to include any fractional part: a whole number represented with a decimal point and a zero is treated as a real value class. Here are some real numbers:

```
0.92
1.0
-45.5
0.0
```

AppleScript accepts scientific notation for real numbers (in fact, all scientific notations compile to real values). The format for scientific notation starts with a real number, followed by the letter “E”, followed by an exponent integer (positive or negative). Any hard-wired number you write into a script that has more than four digits on either side of the decimal is automatically compiled to scientific notation. Here are some examples and their decimal equivalents:

```
1.0E+4  -- 10000.0
-1.0E+4 -- -10000.0
1.0E-5  -- 0.00001
```

When performing math with all real numbers, the results also come back as real values:

```
3.0 + 0.14159
-- result: 3.14159
```

But if you perform math with a mixture of real and integer values, the class of the result varies, depending on which side of the arithmetic operator



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

the real value is, and what that value is. If the real value is on the left side of the operator, you are guaranteed that the result will be a real (AppleScript essentially coerces the integer on the right side to a real for the math):

```
10.0 + 5
-- result: 15.0
```

When the left operand is an integer, two things can happen, depending on the real operand's value:

```
5 + 10.0
-- result: 15
5 + 10.1
-- result: 15.1
```

In most cases, as you'll learn in Chapter 11's discussion about operators, AppleScript takes the clue from the left operand to try to coerce the right operand to the same value class to make the operator work. In the first example, that's true: the real operator (10.0) was coerced to an integer, and the result is that of integer arithmetic. But in the second example, the real-ness of the right operand took precedence, causing the arithmetic and result to be in real number terms. Without this reverse coercion, potentially valuable information (the decimal fraction of the operand) would have been lost. If you intended to lose that fraction, your script should first round the real, and then let the left integer operand govern the action.

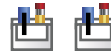
It's unlikely you'll run into this limit, but the range of values for real numbers is from the smallest (positive

or negative) values with an exponent of -324 to the largest with an exponent of +308. If you ever outrun these limits in the course of applying AppleScript to an everyday solution, conjure up a séance with Dr. Carl Sagan (of "billions and billions" fame).



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Record Class



While the formal definition of a record is a *collection of properties*, I like to think of a record as a labeled list. Look at the items in the following list value:

```
{"PowerBook", "15", "1.5GHz"}
```

The list contains three strings, each referring to specific information about a Macintosh portable model. If we turn this list into a record by assigning names to the pieces of information, we are said to be establishing properties of a record:

```
{Family:"PowerBook", LCD:"15", Speed:"1.5GHz"}
```

A property name is not in quotes and is separated from its value by a colon. Like items in a list, properties are separated from each other by a comma. Values of a property can be of any value class (including lists or other records).

It may be easy to confuse an AppleScript record value class with database records. The similarity exists when you see that a property name is like a field label in a database record. But an AppleScript record is not an entry in a database. It is a convenient way to pass along multiple values together (as a single record value), and make those values more readable and accessible because of their names.

A script gains access to data in a record via a property reference (just like accessing properties of an application object). Let's take the Mac record above. If we want to find out what the **LCD** property is, the script would be:

```
get LCD of {Family:"PowerBook", LCD:"15", ~
    Speed:"1.5GHz"}
-- result: "15"
```

This example is not particularly realistic, because it's kind of goofy getting the named property you just typed into a hard-wired record in the script. Where records come in handy is when some application object property returns a record value, which typically would go into a variable in your script. The **display dialog** scripting addition is a good example.

As described in Chapter 7, the **display dialog** command returns a record value. Properties of that record are **text returned** (i.e., typed into the editable field of a dialog), **button returned** (i.e., text of the button clicked to close the dialog), and **gave up** (i.e., a Boolean value that reports **true** if the time specified in the optional **giving up after** parameter has elapsed). Here's a script (Record Value Practice in the Chapter 09 companion files) you can try, which demonstrates how to access the data of a record class value:

```
-- request user's age
set dialogResult to display dialog "Please
    enter your age:" default answer "21" buttons
    {"None of your business", "OK"} default
    button "OK"
-- build in error detection for
-- non-integer entries
try
    -- first get button returned property
    if button returned of dialogResult is
        "OK" then
```



Chapter 9:

Working with Data—Values, Variables, and Expressions

```
-- check text returned property
-- value is an integer
set myAge to (text returned of
dialogResult) as integer
    display dialog (myAge as string) &
" is a fine age." buttons {"OK"} default
button "OK"
    else
        display dialog "Sheesh! I wasn't
going to tell anyone." buttons {"OK"} default
button "OK"
    end if
on error errMsg
    -- if coercion to integer fails
    display dialog text returned of
dialogResult & " is not a good answer."
    buttons {"OK"} default button "OK"
end try
```

The first task of this script is to place whatever results come from the initial **display dialog** command into a variable named **dialogResult**. This variable contains the record containing information about what was typed into the dialog and which button was clicked. Because we need to check whether the user entered an integer, we build the rest of the script into a **try** statement, allowing an error handler at the end to take care of erroneous entries.

The script next checks one of the properties of the **dialogResult** record value: the **button clicked**. If it is the “OK” button, then the script tries to coerce the entry (derived from the **text returned** property of **dialogResult**) to an integer and then displays a friendly message. A user’s click of the other button displays an air of

disappointment. In the error handler, we summon the **text returned** property of the dialog’s resulting value so we can display it as an example of an invalid entry.

Scripts not only extract information from a record by its label, but they can set data as well. It’s just like setting an object property:

```
set oneRecord to {model:"SuperDorf", -
    price:9.95, taxRate:0.08}
set price of oneRecord to 10.95
oneRecord
(* result: {model:"SuperDorf",
    price:10.95, taxRate:0.08} *)
```

Once a record has been defined by its property labels, you cannot insert a new property. Of course, the order of properties is inconsequential. You can, however, prepend or append new properties with the help of the concatenation (&) operator:

```
set employee to {name:"Joe", age: 32}
set department to {dept:"Manufacturing"}
set fullRecord to employee & department
fullRecord
(* result: {name:"Joe", age:32,
    dept:"Manufacturing"} *)
```

Finally, the **count** command and **length** property both work with records. Results from these actions produce the number of properties in a record:

```
count of {name:"Joe", age:32, -
    dept:"Manufacturing"}
-- result: 3
```

You may coerce a record to a list class value. When you do so, the returned value consists of a list of



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

only the record's value, without any reference to the property names.

I encourage the application of record values in your scripts. Rather than load up a script with numerous individual global properties, consider grouping related values into a record with clearly labeled property names. Property labels make scripts much more readable for others to follow, and should also make the script writing easier for you as well.

Reference Class



In Chapters 6 and 7, we saw some commands whose parameters require references to objects. The values expected there are of the reference class. These are the same kinds of references we saw in Chapter 8's exposition of reference forms—but here we're talking specifically about those reference forms that point to an entire object, not a subset. The most common reference forms to fit the bill of a reference class value are indexed, named, and property reference forms.

Some examples of reference class values are:

```
word 1 of document 1 of application "TextEdit"
```

```
item 3 of {"Yankee", "Red Sox", "Tigers"}
```

```
row "Total Sales" -  
  of worksheet "Sales Spreadsheet" -  
  of application "Microsoft Excel"
```

```
bounds of window 3 -  
  of application "Microsoft Project"
```

When used in an AppleScript statement (particularly as a parameter to a command), reference values evaluate to the value of the reference.

```
tell document 1 of application "TextEdit"  
  get paragraph 1  
end tell
```

Here the **get** command's parameter is a reference to the first paragraph (the full reference also includes the default document and application objects in the **tell** statement).



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Contents Property of Reference Classes

The reference class works very much like a programming pointer. For example, if we work with the *Bill of Rights* document in TextEdit, we can create a variable that contains a pointer to the first paragraph:

```
tell application "TextEdit"
    set oneObject to a reference to paragraph
    1 of document 1
    oneObject
end tell
-- result: paragraph 1 of document 1
-- of application "TextEdit"
```

The script can then use that value to represent whatever may be in paragraph 1 at the time it is called. In other words, if we had simply set **oneObject** to paragraph 1, the value would contain the text of paragraph 1. If the contents should change in the course of running a script, however, we'd have to re-do the assignment of data to **oneObject**. Instead, we set **oneObject** to a reference to the paragraph. Each time **oneObject** is used in a statement, it points to that spot in the document and retrieves the current content.

To get to the actual contents of the paragraph by way of that reference, we can access the **contents** property of a reference class. Therefore,

```
tell application "TextEdit"
    set oneObject to a reference to paragraph
    1 of document 1
    contents of oneObject
end tell
-- result: "ARTICLE I
"
```

While TextEdit also allows getting the text of a reference value class, other programs may not. AppleScript, on the other hand, can get the contents of virtually any reference value.



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

RGB Color Class



In Chapter 7, you saw that the `choose color` scripting addition returns a value of class `RGB Color`. The class name is somewhat artificial, but it does impose limits on the actual value.

An `RGB Color` class value is a list consisting of three integer values. Each value is in the range between 0 and 65535 (inclusive). Although unlabeled (this is a list, after all, not a record), the values correspond to the red, green, and blue components, respectively, of the color. In other words, each component can have 65536 different values (yes, that's one boatload of possible color combinations).

In truth, however, the value is simply a list. The **class** property of such a value reveals itself as **list**.

String Class



The term string goes way back in computer programming, but it doesn't seem to be a ready metaphor for something in the real world, except as a string, or series, of characters. String values in AppleScript are always placed inside standard quotation marks. I make the distinction between standard ASCII character quotes instead of the so-called smart quotes (or curly quotes) used in desktop publishing and Microsoft Word's sometimes unhelpful auto-correct default settings. Curly quotes don't work in AppleScript scripts unless they are surrounded by standard quotes. Here are some string value examples:

```
"Pride and Prejudice"  
"a"  
"Beverly Hills 90210"  
".285"
```

The instant you place quotes around any set of characters—including numbers—the set of characters becomes a string. Numbers lose whatever arithmetic abilities they may have had prior to becoming a string (although they can be coerced to numbers if needed).

Because the double-quote symbol, which must surround a string, may also need to be inside a string, AppleScript provides a mechanism that lets us designate a quote inside a string. By preceding the in-string quote with a backslash symbol (`\`), we instruct AppleScript to look at the next quote not as the end of a string but an in-string quote:



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

```
"Have you read \"Walden\" lately?"
```

If you then want a real backslash character in your string, you extend the rule to mean that you enter two backslash characters:

```
"The backslash character (\\) has special  
meaning in AppleScript."
```

Since the value of a string displayed in the Result window is also a string, these backslash-coded characters don't convert to what you'd expect until you copy them to a document. Still, the **length** property of a string, which returns the number of characters in the string, takes these special characters into account:

```
length of "\"Walden\""  
-- result: 8
```

Here, the **length** property ignored the leading backslashes, and counted the double-quotes we want. Without the double-quotes:

```
length of "Walden"  
-- result: 6
```

AppleScript has two more of these backslash special characters. One represents a return character (**\r**); the other a tab character (**\t**). For example, if you tell a TextEdit to

```
copy "Line 1\rLine 2" to text of document 1
```

the following text appears in the document:

```
Line 1  
Line 2
```

Quite often in the compilation process, a string

containing **\r** and **\t** characters gets put back into the editor window in its true form. For example, if we type:

```
tell application "TextEdit"  
    copy "Line 1\rLine 2" to text of document 1  
end tell
```

after compilation, the script reads:

```
tell application "TextEdit"  
    copy "Line 1  
    Line 2" to text of document 1  
end tell
```

This true-to-life formatting of strings can really throw off the look and readability of scripts, but you can't control the compiler's behavior using this backslash notation. As a slightly more wordy but definitely more readable alternative, you can concatenate helpful constants for three useful string values:

Constant	Value
return	"\r"
tab	"\t"
space	" "

Here's how the previous example would look using the constant value:

```
tell application "TextEdit"  
    copy "Line 1" & return & "Line 2" to text  
    of document 1  
end tell
```



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

You can use either the special characters or their constant equivalents in the string value passed as a parameter to the **display dialog** command.

String values are among the most easily coerced into different value classes, but AppleScript won't let you be indiscriminate about it. While you can coerce a string to any of the numeric classes, the original string value must consist of numbers and characters suitable for true number classes. You may also coerce a string to a Unicode text or international text class, but the coerced value does not accrue additional data that normally accompanies such values when they are created from within an application.

Continuation Strings

Mixing the continuation character (↵, typed with Option-Enter or Option-L) and strings requires some care. If you need to break a statement into two lines, and the break falls within a string, you cannot simply insert the continuation character, as in

```
"Now is ↵  
the time."
```

Instead, you must break the string into two pieces, and insert both a concatenation and continuation symbol, as in

```
"Now is " & ↵  
"the time."
```

You are also responsible for maintaining spaces between words, if you break up the string between words.

String Objects

AppleScript strings, as objects, have four elements: **character**, **paragraph**, **text**, and **word**. In concert with a number of reference forms, script statements can get and set any one or all of an element of a string. It's important to understand how AppleScript defines these elements so that you can establish your expectations for their behavior.

Character. Any single character that can appear in a string.

Paragraph. A series of characters that meets any of the following criteria:

<i>Starts At</i>	<i>Ends With</i>
beginning of string	return character
beginning of string	end of string
character after return	return character
character after return	end of string

Text. A series of characters between text item delimiters (see below). Used primarily to extract the running characters of a subset of a string (getting just the characters results in a list value being returned).

Word. A series of characters that starts at either the beginning of a string or after any character not counted as part of a word (e.g., space, en-dash symbol, em-dash symbol, leading or trailing apostrophe). Characters that count as part of a word include letters, numerals, non-breaking spaces



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

(Option-Spacebar), currency symbols, percent signs, commas between numerals, periods preceding numerals, apostrophes between characters, and plain hyphens. Here are examples of AppleScript words:

```
alpha  
semi-transparent  
45,000  
it's  
$19.95
```

Given these element classes and the **length** property of the string value class, the following reference forms are valid for strings:

Reference Form	Example
Property	length of "Now is the time"
Index	word 3 of "Now is the time"
Middle	middle character of word 3 of "Now is the time"
Arbitrary	some word of "Now is the time"
Every Element	every word of "Now is the time"
Range	words 2 thru 3 of "Now is the time"

As with any use of the Every Element and Range reference forms, the returned values for strings are lists:

```
words 2 thru 3 of "Now is the time"  
-- result: {"is", "the"}
```

If, however, your script needs the running text, you'll need to summon the **text** element and **text item delimiters** property. Note that AppleScript strings do not respond to whose clauses (filtered references).



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Text Item Delimiters

AppleScript maintains a global property, called **text item delimiters**. The value of this property is stored as a single-item list value.

Normally set to empty (a value of ""), the **text item delimiters** property defines how AppleScript picks out text items from a string (this has nothing to do with items in a list). For example, with the default property value, each text item consists of a single character:

```
text item 1 of "Now is the time."  
-- result: "N"
```

But if you set the property to a space character (**space** is another AppleScript string constant), the text item element behaves as if each word is a text item:

```
set AppleScript's text item delimiters to space  
text item 1 of "Now is the time."  
-- result: "Now"  
text items 2 thru 3 of "Now is the time."  
-- result: {"is", "the"}
```

Notice how the last example (using a range reference) returned all items in a list. Yet if you coerce the result to text while text item delimiters are still set to space, you get the original text with the proper word spacing:

```
set AppleScript's text item delimiters to space  
(text items 2 thru 3 of "Now is the time.") as string  
-- result: "is the"
```

This global property also comes in handy for other kinds of parsing. For example, folder and file path names are colon-delimited strings. To extract the file name from a full pathname, you can do the following:

```
set pathName to ~  
    "HD:Documents:Letters:Linda 6/3/05"  
set AppleScript's text item delimiters to ":"  
last text item of pathName  
-- result: "Linda 6/3/05"
```

Do not confuse text items with AppleScript's character, word, and paragraph objects. These are very independent concepts, and changing the **text item delimiters** property makes no change to how AppleScript views characters, words, and paragraphs of strings.

You must exercise care, however, when changing this property. It is the most global you can get—AppleScript global (not script global)—so any setting you make in one script remains in effect during execution of any other script. A safe way to work with this property is to save the existing setting in a script variable, and restore the property at the end of the script:

```
set oldDelim to ~  
    AppleScript's text item delimiters  
set AppleScript's text item delimiters to space  
(* do your text item stuff here *)  
set AppleScript's text item delimiters to oldDelim
```

Safer still, is to surround a script that changes the **text item delimiters** property in a **try**



Chapter 9:

Working with Data—Values, Variables, and Expressions

statement (Chapter 12), as in:

```
try
    set oldDelim to AppleScript's text item
    delimiters
    set AppleScript's text item delimiters to
    space
    (* do your text item stuff here *)
    set AppleScript's text item delimiters to
    oldDelim
on error
    set AppleScript's text item delimiters to
    oldDelim
end try
```

This way, if your script should fail with an execution error, the global property will still be restored to its original setting before the script bails out.

Styled Clipboard Text Class

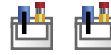


If an application supports retrieving text from a document in the form of styled text, you may be able to transfer that styled text value to the Clipboard (with the application activated). The text data copied and then retrieved from the Clipboard would be of the styled clipboard text class. You can use this facility to transfer styled text from one application or document to another by way of the system Clipboard, rather than through an AppleScript variable value.



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Styled Text Class



Normally, when a script grabs a chunk of text from a document, it takes just the characters. There is nothing in there about font, size, or style. But it will become vital at times to preserve this information, particularly when extracting formatted data from one document and inserting it into another. AppleScript provides a class identifier, called *styled text*, which is actually a variation of the string class.

To obtain a string and its style, a script must explicitly request the data as styled text:

```
tell document 1 of application "TextEdit"
    set goodGraph to paragraph 2 as styled text
    (* do something with goodGraph *)
end tell
```

Some application commands may also return values as styled text.

This value class reports itself as a string value, so your scripts might not be able to distinguish a value that is a simple string from a value containing style information. The purpose of this value is to allow scripts to convey style information along with characters between documents and between applications that support scripting of styled text. Just because a program offers its users font and style menus doesn't automatically mean strings can be operated on as styled text. Adjustments to the style settings can only be done with the value inserted into a document and using the style adjustment commands or properties associated with the string of characters (e.g., in a selection).

Text Class



The text class name is merely a synonym for the string class name. Any value that allows itself to be coerced to a string value can also be coerced to a text value. The class name of a value coerced to a text class is **string**. To a non-programmer, "text" sounds friendlier than "string."

Unicode Text Class

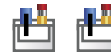


A value of the Unicode text class is similar to a text or string value, but it internally specifies each character with twice the amount of data per character as a straight string value. This allows for the vast number of characters defined for the Unicode character set to be stored in an AppleScript value. If your written language uses the Latin alphabet, the Unicode class won't play much of a role, if any, in your scripting. Some applications, including TextEdit, return strings as the Unicode text class, but the values are easily coercible into regular string values, if needed.



Chapter 9:
**Working with
Data—Values,
Variables, and
Expressions**

Unit Type Classes



AppleScript conveniently provides a wide range of value classes that are comprised of common units of measure. Your scripts can use this infrastructure to perform unit conversions, provided the source and destination units are within the same category shown in the following table.

<i>Length</i> centimeters centimetres feet inches kilometers kilometres meters metres miles yards	<i>Area</i> square feet square kilometers square kilometres square meters square metres square miles square yards	<i>Weight</i> grams kilograms ounces pounds
<i>Cubic Volume</i> cubic centimeters cubic centimetres cubic feet cubic inches cubic meters cubic metres	<i>Liquid Volume</i> gallons liters litres quarts	<i>Temperature</i> degrees Celsius degrees Fahrenheit degrees Kelvin

For example, the following script asks the user to enter a number of miles that will be converted to feet



Chapter 9:

Working with Data—Values, Variables, and Expressions

(both in the Length category):

```
set input to display dialog "Enter a number of  
miles to convert to feet:" default answer "1"  
buttons {"Cancel", "Convert"} default button  
"Convert"  
if button returned of input is "Convert" then  
    set output to ((text returned of input as  
    number) as miles)  
    set output to (output as feet) as string  
    display dialog "The answer is: " & output  
    & " feet."  
end if
```

For simple conversions, use principles of AppleScript coercion you've seen elsewhere. For instance, the following statement returns a temperature conversion:

```
(212 as degrees Fahrenheit) as degrees Celsius  
-- result: degrees Celsius 100.0
```

Next Stop

In the course of the last several chapters, we've been through the three basic building blocks of a script: commands, objects, and values. From here we can start looking at scripts with a bigger view, such as how scripts make decisions based on values. That's what we'll cover next.



Chapter 10 Going with the Flow (or Not): Control Structures

When you examine yesterday's activities, it may seem as though the day was one long series of events, which started from the instant you came-to in the morning until the instant you dropped off at night. In truth, the day doesn't exactly go like that. Upon close examination, the flow of events during the day can be represented in anything but a straight line. A trip to the grocery store—just one of the day's tasks—is a good example of how the flow of events goes this-way-and-that, round-and-round.

Shop Til You Drop

The overall flow of this shopping task takes you from entering the store until you go through the checkout aisle. No sooner do you enter the store than you are faced with your first decision that influences your route: will you need more items than you can carry? If you think there will be too many, then you pick up a basket or extricate a wheeled cart from the snaggle of carts. In other words, you've taken one of two or three paths of action between the front door and the first aisle of the store.

Next, you begin winding your way down the aisles. If you encounter a desired item on a shelf, you pause while picking the item from the shelf. If you have a basket or cart, you place the item in there. When you reach the beginning of one of the aisles, you check whether the categories of items in the aisle match anything on your shopping list. If so, then you work your way down the aisle, else you skip over to the next aisle.

You then pause at the deli counter, and tell the clerk that you'd like some sliced turkey. That instruction



Chapter 10:

Going with the Flow (or Not): Control Structures

applies only to the deli counter clerk, because no one in the produce or bakery departments could possibly help you with your turkey request.

When you later reach a produce bin, you pause. Then you go through a repetition of looking for just the best looking, best feeling peach. You repeat the process—pick up one sample, gently squeeze it, smell it—over and over until you find the one that meets your standards. That one goes with the rest of the items you’ve picked from the shelf, and you move on to the checkout line.

In just this brief foray into a store, the series of motions you made and the results you obtained were influenced by decisions, repetitions, and directing instructions to specific people who could carry out a specific task. If you understand this, then you already understand the nature of AppleScript control structures.

AppleScript Flow Control

Unlike a recorded script, scripts you write rarely flow from top to bottom without some diversion along the way. Such diversions may be needed to direct instructions to a particular application, to provide alternate paths for different results of decisions, and to provide rules to follow while repeating a task. AppleScript provides structures for each of these flow controls: the **tell** statement; the **if-then** statement; the **repeat** statement. We’ll spend some time on each.



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

Tell Statements



You've already seen, in dozens of examples thus far, how **tell** statements direct commands to a particular object and/or application. By naming the application in the **tell** statement, you instruct AppleScript to open the application's dictionary and to be ready to use any of the commands or object references available in that dictionary. To compile a script successfully, AppleScript must find all commands and object references in one of the following:

- ▶ the application or object pointed to by the most recent **tell** statement
- ▶ AppleScript's own dictionary
- ▶ any scripting additions installed on the system

Tell statements come in two varieties, the simple and the compound. Their syntaxes are summarized here:

```
tell reference to statement
```

```
tell reference  
  [ statement ] ...  
end [ tell ]
```

You use the simple type when the complete object reference and statement are written as a single statement, as in:

```
tell document 2 of application "TextEdit" -  
  to print
```

The simple **tell** statement includes a complete

reference to the object directed to carry out a command (**print**, in this case).

Compound **tell** statements are more prevalent. The syntax allows a script to establish a default object to which all nested statements are directed. The default object may be as broad as the application down to the tiniest element:

```
tell application "TextEdit"  
  set font of paragraph 2 of document 1 to  
    "Helvetica"  
  set size of paragraph 2 of document 1 to 14  
end tell
```

```
tell paragraph 2 of document 1 of application  
  "TextEdit"  
    set font to "Helvetica"  
    set size to 14  
end tell
```

In the first example, the default object is the application. To send multiple commands to the same nested object contained by the application, each statement must include the rest of the reference to the desired paragraph. In the second example, we make the second paragraph the default object, so the two short statements for setting font characteristics automatically apply to that paragraph. In both cases, the compound **tell** statement ends with the word **end** and, optionally, the word **tell** (if you don't type it, the compiler inserts it for you). This instructs AppleScript to sever its ties to the default object designated in the previous tell line.

You can nest **tell** statements inside one another, as



Chapter 10:
**Going with the
 Flow (or Not):
 Control
 Structures**

well—even if the objects they point to are in entirely different applications. Here's a skeleton of what such a construction could look like:

```
tell application "SuperWriter"
    (* statements that work in SuperWriter *)
    tell application "SuperSheet"
        (* statements that work
           in SuperSheet *)
    end tell
    (* more statements for SuperWriter *)
end tell
```

The **end tell** line nested inside applies to the nearest preceding **tell** line—the SuperSheet application in this example. You could accomplish the same results with a longer script, by writing a sequence of **tell/end tell** statements, as in:

```
tell application "SuperWriter"
    (* statements that work in SuperWriter *)
end tell
tell application "SuperSheet"
    (* statements that work in SuperSheet *)
end tell
tell application "SuperWriter"
    (* more statements for SuperWriter *)
end tell
```

You gain some advantages in compiled size and possibly execution speed by nesting **tell** statements, however. They also seem more readable when you're following the flow of a script.

It and Me Variables in Tell Statements

AppleScript's vocabulary includes two predefined variables, **it** and **me**, which have specific purposes

inside **tell** statements (and elsewhere). The **it** variable is a shortcut reference to the default object—the object pointed to in the most recent **tell** statement. By and large, the **it** variable is optional, but it may improve the readability of your script. Here's an example of this variable in use:

```
tell window 1 of application "TextEdit"
    get bounds of it
end tell    -- result: {1, 39, 485, 362}
```

The statement inside the **tell** statement could have just as easily read:

```
get bounds
```

because the default object (**window 1 of application "TextEdit"**) completes the reference. The **it** variation, however, makes more sense to someone reading the script for the first time. We also saw this variable's contribution to some variations of the filtered reference form (Chapter 8).

The **me** variable, on the other hand, points to the script containing the statement, instead of the default object of the surrounding **tell** statement. For example, if we have a script that defines a **bounds** property for itself, we may want our script inside a **tell** statement to get the property from our script, not the property of an object belonging to the default object:

```
property bounds : {0, 0, 0, 0}
tell window 1 of application "TextEdit"
    get bounds
end tell    -- result: {1, 39, 485, 362}
```



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

Here we still want the **bounds** property of the default object (**window 1**). To get the **bounds** property belonging to the script, we add two little words:

```
property bounds : {0, 0, 0, 0}
tell window 1 of application "TextEdit"
    get bounds of me
end tell
-- result: {0, 0, 0, 0} (script property)
```

AppleScript also allows the alternates:

```
get my bounds
```

and

```
tell me to get bounds
```

In both cases, the **me/my** variable breaks up the normal flow of the object reference, ignoring the default object, and pointing directly at the script itself.

(If you once programmed in HyperCard, you may remember the same global variable names for **it**, **me**, and **result**. But be aware that their meaning and usage in AppleScript is different. Don't be influenced by what HyperCard taught you about these variables.)



If-Then Constructions

The way AppleScript makes decisions during script execution is to test the truth of a statement. If the statement is true, then the script follows one path; if the statement is false, the script follows another path. Whichever path the script follows, it eventually returns to the main path. It's like taking either the high road or low road: both get to Scotland eventually.

Like **tell** statements, **if-then** statements come in two flavors: simple and compound. A simple statement takes place in a single script line in this format:

```
if BooleanExpression then statement
```

The *statement* parameter is any AppleScript statement. It executes only if the *BooleanExpression* parameter evaluates to **true**, after which execution continues with the next script line. If the expression evaluates to **false**, the *statement* parameter is ignored, and execution continues with the next script line.

Understanding Boolean expressions (and operators of all types from the next chapter) helps expand your application of **if-then** statements. Any operation that yields a **true** or **false** value qualifies as a Boolean expression for the purposes of an **if-then** statement. Here are some examples of simple **if-then** statements:

```
if x < 3 then beep x
```



Chapter 10:

Going with the Flow (or Not): Control Structures

```
if word 1 of paragraph 4 is "Article" then
    set style of paragraph 4 to bold
```

```
if modified of document 1 then save document 1
```

Sometimes, you want a script to take a sidetrip if something is *not* true. In that case, you can use the **not** operator to negate the Boolean expression (i.e., turn a **false** value into a **true** one). Here is an example:

```
if not (paragraph 1 of document 1 ~
    contains "Jimbo") then
    open file "Jimbo Report"
```

In most cases, however, AppleScript operators (Chapter 11) have anticipated this need, and provide a number of plain language synonyms for these negatives, such as

```
if paragraph 1 of document 1 ~
    doesn't contain "Jimbo" then
    open file "Jimbo Report"
```

Compound **if** statements are designed to allow multiple statements to execute within their confines. The construction also allows for additional Boolean tests to take place together. Here are syntax examples of four common compound **if** statement forms:

```
if BooleanExpression [ then ]
    [ statement ] ...
end [ if ]
```

```
if BooleanExpression [ then ]
    [ statement ] ...
else
    [ statement ] ...
end [ if ]
```

```
if BooleanExpression [ then ]
    [ statement ] ...
else if BooleanExpression [ then ]
    [ statement ] ... ...
end [ if ]
```

```
if BooleanExpression [ then ]
    [ statement ] ...
else if BooleanExpression [ then ]
    [ statement ] ... ...
else
    [ statement ] ...
end [ if ]
```

The first form is required whenever more than one statement needs to be executed as a result of the *BooleanExpression* evaluating to **true**. If you like, you may place a single statement here, instead of creating a long, simple **if-then** statement. The **then** parameter and **if** parameter after **end** are optional, but I strongly recommend their use for readability purposes (and the Script Editor compiler inserts them for you anyway).

In the second form, the script executes one of two possible groups of nested statements. The first group executes if the Boolean expression evaluates to **true**; the second only if the expression evaluates to **false**. In other words, execution can't slip through this construction untouched: One of the two branches will execute no matter what.

Form three is a more elaborate version of the previous form. Its logic is as follows: if the first Boolean expression evaluates to **true**, then the first nested group of statements executes; if the



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

first Boolean expression evaluates to **false**, then the second Boolean expression takes control. If the second expression evaluates to **true**, then the nested statements in the second group execute; if the second expression evaluates to **false**, then nothing in the **if** statement evaluates. It's possible, therefore, that under some conditions (when both Boolean expressions evaluate to **false**), no statements in the compound **if** statement execute.

The last form is one more version, which has a final trap in the form of a final **else** clause. If the first two Boolean expressions are **false**, then the statements nested under the solo **else** clause will execute.

You may also nest **if** statements when appropriate. It takes a comparatively complex set of tests and desired results to require nested **if** statements, but they may be necessary:

```
-- put current date data into
-- a list for convenience
set today to (current date)
set theMonth to month of today
set theYear to year of today
-- start checking month cases
if theMonth is February then
    -- take leap years into account
    if theYear mod 4 = 0 then
        set howMany to 29
    else
        set howMany to 28
    end if
else if {April, June, September, November}
    contains theMonth then
        set howMany to 30
```

```
else -- "all the rest have 31"
    set howMany to 31
end if
display dialog "The current month has " &
    howMany & " days in it." buttons {"OK"}
default button 1
```

In the above script, which tells you how many days are in the current month (using the U.S. system for date format), notice what happens with the February situation. Because February has two possible outcomes, depending on the year, we have to perform the second test about the year as a separate, nested **if** statement. (For demonstration purposes, the calculation operates on the year's divisibility by 4, but doesn't take into account the 400-year rule for leap years.) An alternate script could have used compound Boolean expressions in the February tests, as in:

```
...
if theMonth is February and theYear mod 4 ≠ 0
    then
        set howMany to 28
    else if theMonth is February and theYear mod 4
        = 0 then
            set howMany to 29
    ...
```

But the nested version is easier to read, because each nested compound **if** statement helps the reader focus on the conditions being tested.

This days-of-the-month script is also an example of how compound **if** statements can replicate **case** statements available in other languages. By using a



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

series of **else if** clauses, a script can define each case and the statements to be executed for each case. Here's another example:

```
display dialog "Please enter A, B, or C:"  
  default answer ""  
set answer to text returned of result  
if answer is not "" then  
  -- here come the cases  
  if answer is "A" then  
    beep 1 -- or other "A" stuff  
  else if answer is "B" then  
    beep 2 -- or other "B" stuff  
  else if answer is "C" then  
    beep 3 -- or other "C" stuff  
  else  
    display dialog "That wasn't one of  
the letters." buttons {"OK"} default button 1  
  end if  
else  
  display dialog "You didn't enter  
anything." buttons {"OK"} default button 1  
end if
```

Notice that each compound **if** statement must end with an **end if** statement. The compiler makes sure all **ifs** are balanced by **ends**, so when you are writing a complex nested **if** construction, you can use the Script Editor's compiler to perform a pass through the script and point out where things don't balance.

Repeat Statements

As demonstrated earlier in an example from real life, we sometimes need to repeat the same operation on a series of things. These operations might be to perform some kind of test on each item to find out which ones meet certain criteria; or to perform the same adjustment to every item in the series. How often or under what conditions we perform those repetitions is the realm of the **repeat** statement. AppleScript provides a range of statement forms that accommodate a wide variety of repeat conditions shown in the following table:

Type	Description
Repeat	Loops forever, or until a statement tells execution to exit the loop
Repeat Times	Loops a number of times fixed at the beginning of the loop
Repeat Until	Loops forever until a condition is met
Repeat While	Loops only so long as a condition is met
Repeat With	Loops a number of times set either by a counting variable or by the number of items under test

After seeing examples of these different types, you should begin to get a feel for the circumstances recommending each repeat variant in your scripts. It turns out that more than one variety can work for the same loop, and your choice will depend on personal taste and readability of the script.

All repeat statements begin with the word **repeat**, and must have an **end repeat** statement at the



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

end. This **end** statement tells AppleScript where to stop executing statements and start the loop over.

Repeat



The most basic of repeat constructions is the infinite (or forever) loop. Its syntax is:

```
repeat
    [ statement ] ...
end [ repeat ]
```

Any number of statements can go inside a **repeat** loop. The reason this is called an infinite loop is that the construction provides nothing on its own to indicate how to get out of it. This would be a mighty boring script segment, since it would sit there doing the same thing over and over until you broke the script (by typing Command-Period or clicking the script editor's Stop button), force quit the script, or restarted the Macintosh.

To provide a way out of the loop, you usually set up an **if** statement that tests the condition of values being manipulated inside the loop. If the condition is met, then the **exit repeat** statement forces execution out of the loop at that point.

```
set randomList to { }
-- so we can concatenate in loop
repeat
    set randomValue to random number 10
    set randomList to randomList & randomValue
    if randomValue = 5 then exit repeat
end repeat
display dialog "It took " & (count of
    randomList) & " loop(s) to reach 5."
```

This demonstration is a good candidate for the infinite repeat form, because before going into the repeat loop, the script doesn't know how many repetitions it will take. The only place with enough knowledge to find a way out of the loop is inside the loop. An **if** statement tests the value of a random number generated inside the loop each time the loop's statements execute, and escapes the repeat loop when the Boolean test (**randomValue = 5**) evaluates to **true**.

Notice something else in this script. Prior to the loop, we initialize the **randomList** variable as an empty list. Each time through the loop, the list grows by one more item: the most recently generated random number. After the repeat loop ends, the list still contains all those values, which we use to establish a count of the number of times through the loop. An equally viable method would be to initialize a variable with an integer value of zero, and then add 1 to that variable on each pass. Here's what this variant would look like:

```
set counter to 0
repeat
    set randomValue to random number 10
    set counter to counter + 1
    if randomValue = 5 then exit repeat
end repeat
display dialog "It took " & counter & "
    loop(s) to reach 5."
```

This version might run a hair's breadth faster, because it doesn't have to evaluate the count of items for the **display dialog** command parameter.



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

Repeat X Times



When a script knows how many times it must execute a repeat loop, the Repeat X Times form is a good one to use. As its name suggests, the leading **repeat** statement specifies exactly how many times execution loops through the statements nested inside. Syntax for this form is:

```
repeat numberOfTimes [ times ]  
    [ statement ] ...  
end [ repeat ]
```

The value for the *numberOfTimes* parameter must be an integer. This value can be a hard-wired integer or any expression that evaluates to an integer. Whether you include the optional “times” word is up to you, although it does aid readability. Here is an example that plays with some text in a TextEdit document, making the start of the first paragraph dance around:

```
set text item delimiters to ""  
tell document 1 of application "TextEdit"  
    activate  
    repeat 10 times  
        set paragraph 1 to space &  
paragraph 1  
    end repeat  
    repeat 10 times  
        set paragraph 1 to (characters 2  
through -1 of paragraph 1) as string  
    end repeat  
end tell
```

We use the Repeat X Times form, because the script has a specific goal in mind—adding and deleting a fixed number of spaces to the start of a paragraph.

There’s nothing preventing a script from breaking out of any repeat loop early with the **exit repeat** statement. Here’s a variation of the earlier random number script that looks for a 5 during the execution of 10 loops; if a five comes up in fewer than 10 loops, then the script breaks out of the loop:

```
set counter to 0  
repeat 10 times  
    set randomValue to random number 10  
    set counter to counter + 1  
    if randomValue = 5 then exit repeat  
end repeat  
if counter < 10 then  
    display dialog "I broke early by getting  
a five in " & counter & " loops."  
else  
    display dialog "I never got a five in 10  
loops."  
end if
```

Because execution continues after the loop, whether the loop ran its course of ten or broke out early, we include a test to handle the value of the **counter** variable, and produce a message that describes what happened.



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

Repeat Until



The Repeat Until form is related to the infinite repeat form, except the repeat statement specifies the condition that needs to be met for the loop to break (without an **exit repeat** statement). The syntax is:

```
repeat until BooleanExpression
    [ statement ] ...
end [ repeat ]
```

The best way to use this form is when your script knows that the Boolean expression is **false** when the loop starts, and it expects the condition to change as a result of one or more times through the loop. Each time the execution loops back to the top, AppleScript re-evaluates the Boolean expression to see if it is **true** this time around. If it's **true**, then execution won't go through another loop; otherwise, it goes one more round.

Since the expression is re-evaluated each time, the statements in the loop must change the value of at least one of the operands in the Boolean expression, preferably working toward a point at which the expression will evaluate to **true**. Generally, the change would occur after an indefinite number of times—otherwise you could use the Repeat X Times form. We can use a variation of the random number script to demonstrate this form:

```
set counter to 0
set randomValue to 0
repeat until randomValue = 5
    set randomValue to random number 10
    set counter to counter + 1
end repeat
display dialog "I got a five in " & counter & "
loop(s)."
```

Notice a big difference over the infinite repeat version. First of all, we initialize the **randomValue** variable to zero, since it must be defined for us to use it in the Boolean expression of the Repeat Until statement. We go into the loop knowing that the Boolean expression is **false** to begin with. After that, execution loops on and on—each time, however, the **randomValue** variable is put to the test of being 5. If it is, then no further execution occurs within the loop.

In some scripts, you'll have values coming in from documents or other sources that only occasionally need adjustment. In the companion files (*Handbook Scripts/Chapter 10* folder) are two Repeat Until files that demonstrate this form. Open both the script and the document file (TextEdit). The task of this script is to insert ("pad") spaces between the automobile brands and model names so that the result can be displayed in a monospaced font, such as Courier, in neat columns (perhaps for posting to an on-line service that doesn't have pretty fonts).



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

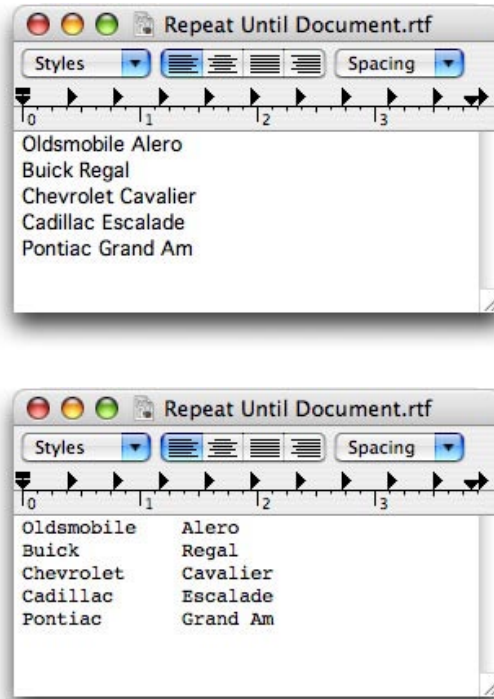


Figure 10.1. *The Repeat Until Document file before (top) and after (bottom).*

The original document appears on the top in Figure 10.1, while the results after running the script appears on the bottom. Here's the script:

```
tell application "TextEdit"
    repeat with i from 1 to count of paragraphs
        of text of document 1
            set testWord to word 2 of paragraph
            i of document 1
            set testGraph to paragraph i of
            document 1
```

```
        repeat until (checkOffset(testWord,
        testGraph) of me) = 15
            set word 2 of paragraph i of
            document 1 to space & word 2 of paragraph i of
            document 1
            set testGraph to paragraph i
            of document 1
            end repeat
        end repeat
        set font of every paragraph of document 1
        to "Courier"
    end tell

on checkOffset(testWord, testGraph)
    return offset of testWord in testGraph
end checkOffset
```

The script is divided into two parts, because we need AppleScript to do its **offset** thing with strings in variables—something TextEdit isn't equipped to do (this is actually a subroutine, a technique we'll detail in Chapter 14).

The key element for our demonstration is the **repeat until** construction. Its goal is to keep examining the location of the model name in a paragraph of the document, and keep inserting spaces until the second word begins at character position 15. The value for the Boolean expression is the value returned by the **checkOffset()** subroutine, which is nothing more than the integer result of the **offset** command. Importantly, if the Boolean test returns **true** before going through the loop (as the first paragraph in this sample does), then nothing in the loop executes for those values: everything is okay the way it is and doesn't need any adjustment.



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

In a way, you could say that our choice of the **repeat until** construction and the equals (=) operator for our Boolean test was the pessimistic outlook: we assumed that the majority of the tests would fail, and would need the space padding to fix things up. It's not a big deal, since we could have changed the operator and used the more optimistic sounding **repeat while** construction.

Repeat While



In contrast to the Repeat Until form, the Repeat While form assumes that everything's A-OK in the Boolean test, and the loop should keep going until something upsets the balance. The syntax is:

```
repeat while BooleanExpression
    [ statement ] ...
end [ repeat ]
```

As long as the Boolean expression evaluates to **true**, the loop continues to go round and round.

Often the choice between the Repeat With and Repeat Until forms is the choice of operator in the Boolean expression. We'll make a one-character change in the **repeat until** version of the random number script shown earlier, and make it a **repeat while** version:

```
set counter to 0
set randomValue to 0
repeat while randomValue ≠ 5
    set randomValue to random number 10
    set counter to counter + 1
end repeat
```

```
display dialog "I got a five in " & counter & "
loop(s)."
```

The difference is all in the point of view. In the **repeat until** version, we say to keep looping until you get a 5; in the **repeat while** version, we say to keep looping so long as you don't get a 5. Neither version is better than the other except that positive operators are often more readily understandable by script readers than negative ones.

Repeat With-From-To



AppleScript defines two versions of the **repeat with** statement, but their operating methods are so different, I prefer to treat them as two separate repeat varieties. The first form I'll cover is one I call Repeat With-From-To. Here's its syntax:

```
repeat with counterVariable from startValue to
stopValue [ by stepValue ]
    [ statement ] ...
end [ repeat ]
```

The purpose of this repeat form is to establish a range of values that will also be used as a counting variable each time through the loop. Let's take an example from the world of the post office. A postal worker has to stuff a group of post office boxes with the latest advertising circular. Of the five hundred boxes at this particular post office, she has a handful of circulars addressed to box numbers 150 through 200. If she were to analyze her actions in AppleScript **repeat** statement form, they would look like this:



Chapter 10:

Going with the Flow (or Not): Control Structures

```
repeat with boxNumber from 150 to 200
    stuff circular addressed to P.O.Box
    boxNumber into box boxNumber
end repeat
```

The repeat loop initializes itself by first establishing a range of numbers and assigning a variable name (**boxNumber**) that will take on numbers from that range. Because of the repeating nature of this construction, the **boxNumber** value increments by one each time through the loop. Inside the **repeat** statement is another statement that uses the value of that variable to help it carry out a job. In our postal example, we use the variable twice—to identify an address and the physical box into which the circular goes. The first time through the loop, the statement evaluates to:

```
stuff circular addressed into P.O.Box 150 -
to box 150
```

The second time through the loop, the values increase by one:

```
stuff circular addressed into P.O.Box 151 -
to box 151
```

and so on, until the one valued at 200 is performed. Then the repeat loop automatically exits, and script execution continues right after the **end repeat** statement (if there is more to execute).

The time to use this repeat form is when your script needs to repeat an operation based on the value of a loop counting variable. Here's an example using TextEdit to number paragraphs in a document:

```
tell application "TextEdit"
    repeat with i from 1 to (count of
        paragraphs of text of document 1)
        set paragraph i of document 1 to (i
            as string) & ". " & paragraph i of document 1
        end repeat
    end tell
```

Notice a couple of important points. First, the *stopValue* parameter here is determined by an expression whose value depends entirely on the conditions at the moment—both the *startValue* and *stopValue* parameters can be any expression that evaluates to an integer. Second, we use the letter “i” as the *counterVariable* parameter. You can use any letter or word that would qualify as an AppleScript variable for this job (Chapter 9). Some scripters use letters, such as **i** (for index, and proceeding to **j**, **k**, and so on for nested repeats), while in some instances it may make the script more readable if you use a more descriptive word that identifies what the variable represents (e.g., **graphCount**).

It's also important to note that the variable you use as the *counterVariable* parameter survives past the **end repeat** statement. Therefore, if your repeat loop contains a test that may exit the loop early, you can later examine the value of *counterVariable* to find out the highest value that the counter variable reached.

If your script needs to increment the *counterVariable* by something other than 1 (e.g., the postal worker is doing just the even numbered boxes), then use the



Chapter 10:

Going with the Flow (or Not): Control Structures

optional **by** *stepValue* parameter. For example, to number every fifth paragraph of a document, you'd modify our earlier examples to read this way:

```
tell application "TextEdit"
    repeat with i from 1 to (count of
        paragraphs of text of document 1) by 5
        set paragraph i of document 1 to (i
            as string) & ". " & paragraph i of document 1
        end repeat
    end tell
```

Only paragraphs 1, 6, 11, 16, etc. would be numbered, because the value of the counting variable jumps by 5 each time the loop executed. The value of the variable **i** at the end of looping is only as high as the variable reached during the last executed loop.

You can also instruct this style of counting repeat loop to count backwards. Make the *startValue* larger than the *stopValue*, and include a negative *stepValue* parameter:

```
tell application "TextEdit"
    repeat with i from (count of paragraphs
        of text of document 1) to 1 by -1
        set paragraph i of document 1 to (i
            as string) & ". " & paragraph i of document 1
        end repeat
    end tell
```

This script numbers each paragraph starting with the last one, and works its way down to the first.

Repeat In List



An even more powerful repeat construction is one I call Repeat In List. This one is ideal when a repeat loop needs to work with items in an AppleScript list value class. The syntax is:

```
repeat with loopVariable in listReference
    [ statement ] ...
end [ repeat ]
```

Unlike the *counterVariable* parameter in the Repeat With-From-To form, the *loopVariable* value in the Repeat In List form is not an integer, but rather a value representing an object from the list. Each time through the loop, *loopVariable* becomes a reference to the next object from the list or record. Some examples are definitely in order.

As a preliminary example, we'll create a simple list of integers, and then use the powers of this construction to add them together:

```
set itemTotal to 0
repeat with oneItem in {10, 20, 30, 40, 50}
    set itemTotal to itemTotal + oneItem
end repeat
itemTotal
-- result: 150
```

The script first initializes a variable **itemTotal** to zero, so we can add other integers to it later. At the opening of the repeat loop, we assign references to the successive items of the list to the variable **oneItem**. Inside the loop, when we perform the arithmetic, we can use the **oneItem** value for the arithmetic directly. A longer way to write this script



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

(with the Repeat With-From-To form) would be:

```
set itemTotal to 0
set integerList to {10, 20, 30, 40, 50}
repeat with i from 1 to count of integerList
    set itemTotal to itemTotal + (item i of
        integerList)
end repeat
itemTotal
-- result: 150
```

Perhaps the biggest difference is that in the Repeat In List form, the list becomes part of the loop construction, whereas in the other method, the parameters concern themselves only with the count and position of items in a predefined list. We end up having to refer to the **integerList** value more often. The Repeat In List form is compact, with the *loopVariable* value essentially evaluating to “item 1 of listReference”, “item 2 of listReference”, “item 3 of listReference”—incrementing by one each time through the list.

As a result, when the items in a list are objects containing properties, your script can access those properties inside a repeat loop with a compact reference:

```
tell document 1 of application "TextEdit"
    set letterTotal to 0
    repeat with oneWord in words of paragraph 2
        set letterTotal to letterTotal +
            (count of characters of oneWord)
    end repeat
end tell
letterTotal
```

When you consider that a lot of data, like the every element reference shown above (**words of paragraph 2**), comes back as a list, you see how valuable this repeat construction can be in AppleScript. You can place references to groups of objects directly in the repeat loop header, and let the nested statements work with the data directly.

Pay particular attention to the fact that the variable value that seems to represent an item of the list each time through the loop does not truly evaluate to the value of the item, but rather to a reference to the value. Therefore, if the list consists of three strings, it would be in vain to test whether the loop variable “is” a fixed string—because the script will be comparing a reference class object against a string class object, which will always return false. For such comparisons, it is necessary to coerce the value prior to the test:

```
repeat with oneItem in {"Rob", "Buddy", "Sally"}
    if oneItem as string = "Sally" then
        -- coercion required
        display dialog "We just reached
            Sally!"
    end if
end repeat
```

As an added note, contrary to what the *AppleScript Language Guide* says about this repeat structure, it does not coerce records to lists for use as the *listReference* parameter. Your script can manually coerce a record to list if necessary (including coercing the list in the repeat statement if you like),



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

but the coercion is not automatic. Thus, if the data source in the above example is a record, the coercion could be accomplished as follows:

```
repeat with oneItem in {boss:"Rob",  
    writer1:"Buddy", writer2:"Sally"} as list  
    ...  
end repeat
```

Timeout Flow Control



Each time your script sends a command to an application, AppleScript waits two minutes for the application to send back a reply (this is part of the Apple event mechanism, which is fortunately hidden from our view). A downside to this is that until the acknowledgment comes back from the program, script execution pauses during this time.

Under normal circumstances, this is no problem, since the application responds within a reasonable time. But there are some things that can bog down the process. For example, if your command is talking to an application running on another Mac, there could be extraordinarily heavy network traffic; or the other machine could be performing a processing intensive task with the very program you're trying to control with your script; or there could be a bug in the program (perish the thought).

Unless directed otherwise, AppleScript waits two minutes for an acknowledgment. If nothing comes back (during which time you've been twiddling your thumbs), AppleScript finally reports a timeout error.

There may be circumstances in which you find two minutes aren't enough to guard against a timeout for a particular process that a script is performing. A script object running in the background (Chapter 15) might not care (nor would you) that it's having to wait five minutes for the target application to respond. To lengthen (or shorten) the time a command or series of commands waits for the reply,



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

you can surround those statements with the **with timeout** statement. The syntax is:

```
with timeout [ of ] numberOfSeconds seconds [s]
  [ statement ] ...
end [ timeout ]
```

If you're genuinely concerned about a process timing out too soon and messing up your script, then you should not only extend the timeout time, but also nest the timeout statement inside a **try** statement so you can more gracefully trap the timeout error:

```
try
  tell application "MagicBase" of machine
    "eppc://192.168.1.103"
    -- increase to 5 minutes
    with timeout of 300 seconds
      (* time-consuming statements *)
    end timeout
  end tell
on error errMsg number errNum
  if errNum is -1712 then
    -- timeout error number
    (* timeout error handling
      statements *)
  end if
end try
```

It's vital to be on the lookout for timeout bottlenecks, because when a command times out, it not only sends back an error message, but any expected result data won't be there for you. For more information about the **try** statement and error handling, see Chapter 12.

A script can blow past the timeout issue entirely by requesting that statements be executed without

waiting at all. The method involves the **ignoring** statement, which I've reserved for the next chapter (most applications of this statement involve operators, the subject of the next chapter). This **ignoring** tactic should be used only in rare circumstances, because your script won't get any feedback about whether the command was ever received by the target application. Nor do any results come back while this **ignoring** switch is on. Caveat Scriptor!



Chapter 10:
**Going with the
Flow (or Not):
Control
Structures**

Other Control Statements

Some other AppleScript constructions affect the flow of a script. Two of them—**considering** and **ignoring**—have more influence over the use of operators, so I'll hold that discussion for Chapter 11. Another subject very close to the matter is that of subroutines. I've reserved an entire chapter (Chapter 14) for this powerful aspect of AppleScript.

The remaining control statement to keep in mind in case your applications operate in the area is the **with transaction** statement. This statement can apply to (primarily) database retrieval applications that support the concept of transactions. From the AppleScript side, placing a series of statements inside a **with transaction** statement instructs the database application to handle the statements as a kind of single action.

Syntax for this statement is:

```
with transaction [ sessionID ]  
    [ statement ] ...  
end transaction
```

In a typical database application, your script will first open a session, which returns a session ID (to distinguish one session from others that may be open at the same time). That session ID can then be passed as a parameter to the **with transaction** statement to make sure any nested statements go to the proper transaction. Here's the skeleton of such a series in a fictional application:

```
tell application "Mega Base"  
    -- create session and get session ID  
    set mySession to make session with data {...}  
    -- specify which sessionID  
    with transaction mySession  
        (* storage/retrieval statements *)  
    end transaction  
end tell
```

Details about sessions and commands that go inside transactions will accompany applications that offer these facilities. AppleScript's role here is primarily as a facilitator, to make sure instructions to such an application and transaction object are packaged correctly.

Next Stop

From here we tackle the last large group of words in the AppleScript vocabulary: the operators.



Chapter 11 AppleScript Operators

AppleScript is rich in *operators*: words and expressions that perform operations on one or two values to arrive at another value. The values that operators work on are called *operands*. An expression may contain one operand and an operator, or two operands separated by an operator. These formal descriptions sound more complicated than actually working with operators.

Four Operator Types

In an effort to help you readily learn the variety and scope of AppleScript's operators, I've divided them into four categories, some of which have untraditional names—but names that I believe identify their purpose in the language (my apologies to classically-trained computer scientists). The four types are:



Chapter 11:
**AppleScript
Operators**

Type	Description
connubial operators	Join two operands together to produce a single value that is a result of a math or other operation on the two.
comparison operators	Compare the values of two operands, deriving a result of <code>true</code> or <code>false</code> (used extensively as parameters to <code>if-then</code> statements).
containment operators	Determine whether one operand is a part of, or is in a particular position relative to, a second operand.
Boolean operators	Perform Boolean arithmetic on one or two Boolean operands.

Much of what operators are all about harkens back to our discussion in Chapter 5 about expression evaluation. By definition, an operator is part of an expression that evaluates to some other value that the script reads. For example, the expression

`2 + 3`

shows two integer operands joined by the addition (+) operator. This expression evaluates to **5**. It's the operator that provides the instructions for AppleScript to follow in its never-ending pursuit of expression evaluation.

As you will see, it's not uncommon for two operands that look very different in a script to be compared for their equality. AppleScript cares not how the operands read in the script but how their evaluated values compare. Two very dissimilar looking values can, in fact, be of identical value. For example, the expression



Chapter 11:
**AppleScript
Operators**

`a = 325`

would evaluate to **true** if the variable named **a** had a value of 325 assigned to it in an earlier statement in a script.

To help you grasp the range of operators and their requirements quickly, the following section lists the operators of each type in a table that shows the most vital information: operator syntax, the class of operand values the operator accepts, and the class of the resulting value. Most of these specifications come down to common sense anyway, especially when you look at the plain language wording for the operators.

Connubial Operators

Syntax	Name	Operands	Results
<code>+</code>	Plus	Integer, Real	Integer, Real
<code>-</code>	Minus	Integer, Real	Integer, Real
<code>*</code>	Multiply	Integer, Real	Integer, Real
<code>/</code> or <code>÷</code> (Option- <code>/</code>)	Divide	Integer, Real	Integer, Real
<code>div</code>	Integral Division	Integer, Real	Integer
<code>mod</code>	Modulo	Integer, Real	Integer, Real
<code>^</code>	Exponent	Integer, Real	Real
<code>&</code>	Concatenation	All (See below)	List, Record, String
<code>as</code>	Coercion	(See below)	(See below)
<code>[a] ref[erence] [to]</code>	A Reference To	Reference	Reference



Chapter 11:
**AppleScript
Operators**

Comparison Operators

Syntax	Name	Operands	Results
= is equal[s] [is] equal to	Equal	All	Boolean
≠ (Option-=) is not isn't isn't equal [to] is not equal [to] does not equal doesn't equal	Not equal	All	Boolean
> [is] greater than comes after is not less than or equal [to] isn't less than or equal [to]	Greater than	Date, Integer, Real, String	Boolean
< [is] less than comes before is not greater than or equal [to] isn't greater than or equal [to]	Less than	Date, Integer, Real, String	Boolean
>= ≥ (Option->) [is] greater than or equal [to] is not less than isn't less than does not come before doesn't come before	Greater than or equal to	Date, Integer, Real, String	Boolean
<= ≤ (Option-<) [is] not less than or equal [to] is not greater than isn't greater than does not come after doesn't come after	Less than or equal to	Date, Integer, Real, String	Boolean



Chapter 11:
**AppleScript
Operators**

Containment Operators

Syntax	Name	Operands	Results
contain[s]	Contains	List, Record, String	Boolean
does not contain doesn't contain	Does not contain	List, Record, String	Boolean
is contained by	Is contained by	List, Record, String	Boolean
is not contained by isn't contained by	Is not contained by	List, Record, String	Boolean
start[s] with begin[s] with	Starts with	List, String	Boolean
end[s] with	Ends with	List, String	Boolean

Boolean Operators

Syntax	Name	Operands	Results
and	And	Boolean	Boolean
or	Or	Boolean	Boolean
not	Not	One Boolean	Boolean

Operators work with a variety of value classes as operands. The specific treatment may vary from class to class, so it's worthwhile discussing how the operators work with each class of values.



Chapter 11:
**AppleScript
Operators**

Integers, Reals, and Operators

Many operators accept both integer and real numbers as operands (although some return only a real number as a result). In fact, operators such as math operators allow you to mix and match integers and reals in either operand. The value of the result depends largely upon the class on the left side. Here are examples that show all the possibilities:

```
10 + 10
-- result: 20 (integer on both
-- sides yields integer)

10 + 10.0
-- result: 20 (left integer governs,
-- since result is a whole number)

10 + 10.1
-- result: 20.1 (right real governs,
-- since result must be a real)

10.0 + 10
-- result: 20.0 (left real governs)

10.0 + 10.0
-- result: 20.0 (real on both sides
-- yields real)
```

For comparison operators, the two operands can be any combination of integer and real values. Their classes are irrelevant, because the operator determines whether the test wins or fails based on the math. Here are some examples:

```
10 = 10
-- result: true
```

```
10 = 10.0
-- result: true

9.9 = 10.0
-- result: false

10 > 10
-- result: false

10 ≥ 10
-- result: true
```

Remember that an operand is any expression that evaluates to the desired value class. Therefore:

```
item 2 of {10, 20, 30} = 20
-- result: true

count of {10, 20, 30} = 3
-- result: true

49.9 + 50.1 = 100
-- result: true
```

That includes information from an application's objects, such as properties and data. As long as you can form the operands into expressions that evaluate to a real or integer class value, you can place them on both sides of a comparison operator.



Chapter 11:
**AppleScript
Operators**

Strings and Operators

Many operators accept string values as one or both operands. In the connubial group, most commonly used is the concatenation operator (&). This operator joins the left operand string to the right operand string with no intervening spaces. As a result, if you are joining two words or phrases together, it is important that you take care of any space that may need to go between words. These examples should give you ideas:

```
"Howdy" & "Doody"  
-- result: "HowdyDoody"
```

```
"Howdy" & " Doody"  
-- result: "Howdy Doody"
```

```
"Howdy" & space & "Doody"  
-- result: "Howdy Doody"
```

Use the concatenation operator with any string value, as in:

```
get ((path to scripting additions folder) -  
    as string) & "StringLib.sct"  
(* result: "Macintosh HD:System:Library:  
ScriptingAdditions:StringLib.sct" *)
```

If the right operand of a concatenation operator is an integer or real value, AppleScript tries to coerce the value into a string to carry out the concatenation. For example:

```
"Vancouver Olympics " & 2010  
-- result: "Vancouver Olympics 2010"
```

The “weight” of a left operand demonstrates itself if you reverse the string and number sides of the

operator:

```
2010 & " in Vancouver"  
-- result: {2010, " in Vancouver"}
```

As with most operators that try to coerce right side values, it is the left side that runs the show.

Comparison operators treat strings like a series of lowercase ASCII character values. For two strings to be equal, absolutely every character—punctuation, spaces, tabs, returns, foreign symbols—must be the same in both. Case of letters, however, is irrelevant, because all characters are essentially converted to lowercase for the comparison (unless directed otherwise by **considering** or **ignoring** statements, described below).

This reduction of strings to ASCII values helps explain how strings can be checked to determine which comes before or after another. Comparisons are performed character by character from left to right (in Roman languages, at least). The instant a character of the left string is different than the corresponding character in the right string, the operator determines precisely what condition is being checked for, and returns the **true** or **false** as needed. If one string is shorter than the other (and matches corresponding characters in the second string), the shorter string is said to come before (or be less than) the longer string. Here are some examples of string comparisons:



Chapter 11: AppleScript Operators

```
"Beantown" comes before "Boston"
-- result: true (some would rather use
-- the "is less than" synonym)

"MaNcHeStEr" = "Manchester"
-- result: true (no case sensitivity)

"123" = "one, two, three"
-- result: false

"123" is greater than "one, two, three"
-- result: false (numerals sort
-- before letters in ASCII)

"aye" is "eye"
-- result: false (spelling, not sound, rules)

"Here and now" < "Here and now."
-- result: true (note trailing period
-- in second operand)
```

String treatment with containment operators is very straightforward. The conditions must be met like they sound for the result to be **true**. Similar left-to-right, character-by-character analysis takes place to determine whether one string starts or ends with another or whether one string contains another.

Lists and Operators

The test for equivalency of two lists depends on the evaluated values of the contents of both lists. For example, while

```
{10, 20, "It's me"} = {10, 20, "It's me"}
-- result: true
```

is obviously true, so is the following:

```
{10.0, 100÷5, "It's" & " me"} = {10, 20, "It's
me"} -- result: true
```

Each item in the left operand evaluates to the same value as the corresponding item in the right operand. Such comparisons are performed on an item-by-item basis, so the item orders can't be mixed if you expect a **true** result. Therefore,

```
{10, 20, "It's me"} = {20, 10, "It's me"}
-- result: false
```

returns **false**, because even though both operands have the same items in them, the items are in different order. The comparison fails the instant it sees that 10 does not equal 20—the first item of both operands.

Containment operators examine corresponding items in lists the same way. For example:

```
{"Ulysses", "by", "James", "Joyce"} starts
with "Ulysses" -- result: true
```

```
{"Ulysses", "by", "James", "Joyce"} ends with
"Joyce" -- result: true
```

```
{"Ulysses", "by", "James", "Joyce"} contains
"by" -- result: true
```



Chapter 11: AppleScript Operators

```
{"Ulysses", "by", "James", "Joyce"} contains
  "by James"  -- result: false
```

```
{"Ulysses", "by", "James", "Joyce"} contains
  "Joy"  -- result: false
```

```
{"Ulysses", "by", "James", "Joyce"} contains
  {"by", "James"}  -- result: true
```

```
"James" is contained by {"Ulysses", "by",
  "James", "Joyce"}  -- result: true
```

A personal note about this last construction: While AppleScript accepts the **is contained by** operator, some writers (add me to the list) may have a difficult time when reading scripts with this blatantly passive grammar. Spare me the pain by placing the order of operands in your scripts such that you can use the active (**contains**) version.

Use the concatenation operator to join two lists into a single list. This allows you to prepend or append one list to another fairly simply:

```
{"Two", "lists" } & {"become", "one."}
  -- result: {"Two", "lists", "become", "one."}
```

When the left operand of this operator is a list, you can append items of other classes to that list, because AppleScript turns the right operand into an item of the list (but still bearing its same class as the original):

```
{"She", "loves", "you"} & "yeah yeah yeah"
  (* result: {"She", "loves", "you",
  "yeah yeah yeah" } *)
```

```
{"She", "loves", "you"} & "yeah" & "yeah" &
```

```
"yeah"
  (* result: {"She", "loves", "you",
  "yeah", "yeah", "yeah" } *)

{"As", "easy", "as"} & 3.14159
  -- result: {"As", "easy", "as", 3.14159}
```

As an alternate to the concatenation operator (&), you may also use the **beginning** and **end** specifiers to prepend and append to a list, respectively. For example,

```
set theMacs to ~
  {"iMac", "PowerBook", "PowerMac"}
copy "iBook" to beginning of theMacs
theMacs
  (* result: {"iBook", "iMac", "PowerBook",
  "PowerMac" } *)
```

and

```
set theMacs to ~
  {"iMac", "PowerBook", "PowerMac"}
copy "iBook" to beginning of theMacs
copy "eMac" to end of theMacs
theMacs
  (* result: {"iBook", "iMac", "PowerBook",
  "PowerMac", "eMac" } *)
```

The same would work for copying another list to the beginning or end of an existing list. While this mechanism would be useful for working with strings in variables, AppleScript does not make that kind of concatenation possible.

If you need to insert an item into a list, the procedure is more complex, because you have to extract the component parts before and after the insertion point, and then use the concatenation operator:



Chapter 11:
**AppleScript
Operators**

```
set oneList to {"This", "is", "so", "easy"}  
set oneList to items 1 thru 2 of oneList & -  
    "not" & items 3 thru 4 of oneList  
-- result: {"This", "is", "not", "so", "easy"}
```

A lot of AppleScript is tied to lists. The more you know about them, the better your scripts will be.

Records and Operators

Equivalency of records is a bit freer than of lists, because records, by their very nature, are not ordered collections of data, but rather collections of labeled data. The distinction plays a role in determining whether two records are equal.

All that AppleScript cares is that both records on either side of the equivalency operator have the same number of labels, the same label names, and data that evaluates the same. The order of the labels and data is not a factor. Therefore:

```
{name: "Joe", age: 30, weight: 165} = {age:  
    30, name: "Joe", weight: 165}  
-- result: true
```

because the labels and values are the same.

For the values, evaluation is again the key:

```
{name:"Joe", age:30, weight:165} = {age:90 ÷  
    3, name:{"J", "o", "e"} as string, weight:100  
    + 65}  
-- result: true
```

As long as the labels are there and the values for each pair of corresponding labels evaluate to the same value, the records are equal. To take this one step further, data extracted from two (equal or unequal) records with the same label and evaluation are also equal:

```
set Joe to {name:"Joe", age:30, weight:165}  
set Steve to {age:30, name:"Steve", weight:180}  
Joe's age = Steve's age  
-- result: true
```



Chapter 11: AppleScript Operators

Containment operators on records have the same freedoms and restrictions as on lists. One record is said to contain another if the smaller record's labels and values can also be found in the first record. Order of labels and values is not a factor.

```
{brand:"Sony", model:"KV-1029", price:"$399"} ~  
  contains {model:"KV-1029", brand:"Sony"}  
-- result: true
```

Both sides of the operator, however, must be a record. AppleScript can't coerce some values into records:

```
{brand:"Sony", model:"KV-1029", price:"$399"} ~  
  contains "Sony" -- result: Error  
-- (Can't make "Sony" into a record)
```

If your script must know if a certain value is in a record, but the script can't know the label, you can coerce the script to a list, and then use the operator:

```
({brand:"Sony", model:"KV-1029", price:"$399"}  
as list) contains "Sony"  
-- result: true
```

While you can concatenate records, as in:

```
{lastName:"Doe", firstName:"John"} & {age:21}  
(* result: {lastName:"Doe", firstName:"John",  
age:21} *)
```

you have to be careful about combining records with the same labels. AppleScript discards any field value in the right operand that shares a label with the left operand:

```
{lastName:"Doe", firstName:"John"} & ~  
{lastName:"Smith", age:21} -- result:  
-- {lastName:"Doe", firstName:"John", age:21}
```

Booleans and Operators

It's pretty easy to understand Boolean values and things like equivalency operators: each side of the operator must evaluate to the same Boolean value for the expression containing the operator to be **true**:

```
10 > 5 = "b" comes after "a"  
-- result: true (both sides are true)  
  
10 < 5 = "b" comes before "a"  
-- result: true (both sides are false)
```

Notice that in these examples, each of the operands used an expression to achieve some Boolean value. It's true that 10 is greater than 5; it's also true that "b" comes after "a". Therefore, the overall expression tests the equivalency of a **true** on one side and a **true** on the other, yielding a **true** as a final value. Since so many AppleScript operators evaluate to a Boolean, expect to find operators within things like command parameters that require Boolean values—especially **if-then** constructions:

```
set John to {lastName:"Doe", firstName:"John",  
age:21}  
if age of John ≥ 21 then  
  (* statements based on this age group *)  
end if
```

Where Booleans can get confusing for some scripters is when you have to use Boolean operators with Boolean values. Let's look first at the simplest Boolean operator, **not**. This operator is called a **unary** operator, because it requires only one operand. The **not** operator goes in front of a Boolean value to switch it back to the other value (i.e., from **true**



Chapter 11: AppleScript Operators

to **false** or from **false** to **true**). For example:

```
not true
-- result: false
```

```
not (10 > 5)
-- result: false
```

```
not (10 < 5)
-- result: true
```

```
not ("Hard Rock Cafe" ends with "Cafe")
-- result: false
```

Interestingly, if you were to enter the last example into Script Editor without the parentheses, the compiler rewords the expression to read:

```
"Hard Rock Cafe" does not end with "Cafe"
```

When using the **not** operator, it is a good idea to enclose the operand in parentheses. These help the compiler figure out exactly what in the statement you intend to be the operand. It then helps the script reader more fully understand what operand you are reversing.

The **and** operator joins two Boolean values together to arrive at a logical **true** or **false** value based on the results of both operators. This brings up something called a **truth table**, which helps you visualize how the various operands behave with the **and** operator:

Left	Operator	Right	Result
true	and	true	true
true	and	false	false
false	and	true	false
false	and	false	false

You see that only one condition produces a **true** result: both operands must evaluate to **true** . Which side of the **and** operator a **true** or **false** is on makes no difference. Here are some simple examples of each possibility:

```
100 > 0 and "z" comes after "a"
-- result: true
```

```
100 > 0 and "a" comes after "z"
-- result: false
```

```
100 < 0 and "z" comes after "a"
-- result: false
```

```
100 < 0 and "a" comes after "z"
-- result: false
```

In contrast, the **or** operator is more lenient about what it evaluates to **true** . The basis of this operator is that if one or the other (or both) operands is **true**, then it returns a **true**. Its truth table looks like this:



Chapter 11: AppleScript Operators

Left	Operator	Right	Result
true	or	true	true
true	or	false	true
false	or	true	true
false	or	false	false

Therefore, if a **true** exists on either side of the expression, a **true** is the result. Here are **or** operator versions of the previous examples, to show how many more **true** results we get:

```
100 > 0 or "z" comes after "a"  
-- result: true
```

```
100 > 0 or "a" comes after "z"  
-- result: true
```

```
100 < 0 or "z" comes after "a"  
-- result: true
```

```
100 < 0 or "a" comes after "z"  
-- result: false
```

The only way the expression returns **false** is if both operands evaluate to **false**.

Coercing Values—the As Operator

Because operators are so “class conscious,” it is frequently necessary to change the class of a value to make it work in some comparison or marriage of values by operators. We’ve already seen that AppleScript at times automatically tries to coerce a value of a right side operand when the left operand governs the situation:

```
"Vancouver Olympics " & 2010  
-- string & integer  
-- result: "Vancouver Olympics 2010" (string)  
8 + "45"  
-- integer plus a string  
-- result: 53 (integer)
```

But the majority of times when you need to change the class of a value, an operator between the operands isn’t there to help out. Your script may have retrieved a value of one class from a document, but to use that data as a parameter to a command or assign it to a property value, it must be converted to a different class. Here is an example that fails due to an incorrect value class being passed to the **open for access** command:

```
get (choose folder with prompt -  
    "Choose your Data folder:")  
set myPath to result & "Test Data"  
set myFile to (open for access file myPath -  
    with write permission)
```

The above script actually begins to fail in the second statement, which attempts to concatenate a string to the end of an alias value returned by the **choose folder** command. Because the two values are of



Chapter 11:
**AppleScript
Operators**

different value classes, the result of the operation is a list of two disparate items, not a single string containing a path to a file.

AppleScript includes an operator—the **as** operator—to force coercions. This operator requires two operands: on the left the value to be coerced; on the right the name of the class to which you wish the value coerced. Here's how we would apply the operator to our earlier problem:

```
get (choose folder with prompt ¬  
    "Choose your Data folder:")  
set myPath to (result as string) & "Test Data"  
set myFile to (open for access file myPath ¬  
    with write permission)
```

The parentheses are required because it's vital for only the **results** variable to be coerced to a string before becoming a parameter to the **open for access** command.

You should probably realize by now, however, that not every value class can be coerced to any other value class. Some values simply don't have the right stuff to become another class. Use the following table as a guide to possible coercions of the most common value classes (see next page):



Chapter 11:
**AppleScript
Operators**

Coerce From	To	Example	Tips
Integer	Real	5 → 5.0	
Integer	String	5 → "5"	Or text class synonym
Real	Integer	5.0 → 5	Only if real has no fractional part
Real	String	5.1 → "5.1"	Or text class synonym
String	Integer	"5" → 5	Only if string represents an integer
String	Real	"5.1" → 5.1	Only if string represents a real
(any value)	Single List	"joe" → {"joe"}	Resulting list is of one value, which may be concatenated with other lists
Single List	(any value)	{"joe"} → "joe"	Resulting value is same class as item in list
List	String	{"Ice", "T"} → "IceT"	Depends on text item delimiters
Record	List	{x:1, y:2, z:3} → {1, 2, 3}	Labels are dropped
Date	String	date "<date>" → "<date>"	One way only
Alias	String	alias "<path>" → "<path>"	One way only

One coercion pair that needs some explanation is the multiple-item list to a string (or text) value class coercion. The results depend on the current settings of the **text item delimiters** global property.

Consider the following:

```
{2, "Live", "Crew"} as string
-- result: "2LiveCrew"
```

This result came from the default **text item delimiters** property (empty). Notice that each



Chapter 11: AppleScript Operators

item is coerced to a string, and all the strings are concatenated to each other to create one run-on string. But if the property is set to another character, AppleScript inserts that character between items in the list during coercion. Here are some examples:

```
set text item delimiters to space
get {"Now", "is", "the time"} as text
-- result: "Now is the time"

set AppleScript's text item delimiters to "."
get {"Now", "is", "the time"} as text
-- result: "Now.is.the time"
```

Notice that the delimiter character goes only between original items, and not within any item (i.e., the two-worded last item, above).

The **as** coercion operator doesn't work to convert strings to file or alias references, as described in Chapter 9. In those cases, the class name needs to precede the string value representing a folder or file path name, and the resulting expression evaluates to the desired alias or file class. Don't confuse this caution about the **as** operator with the occasional command **as** parameter, which is designed to perform coercion in that format. The parameter only looks like the **as** operator, but it isn't.

String Comparison Aids

In the many examples you've seen for string comparisons, it has probably become quite evident that except for the case of letters, strings are compared character for character, including spaces between words, hyphens, punctuation, and so on. Your script can, however, alter the way string comparisons are made, allowing for more or less leniency as you decree.

The controlling statements that take care of this are the **considering** and **ignoring** statements. The syntax for these statements is as follows:

```
considering attribute ~
[, attribute ... and attribute ] ~
[ but ignoring [, attribute ... ~
and attribute ] ]
[ statement ] ...
end considering

ignoring attribute ~
[, attribute ... and attribute ] ~
[ but considering [, attribute ... ~
and attribute ] ]
[ statement ] ...
end ignoring
```

What's being controlled here is how AppleScript treats the various attributes that all strings have. Here are the string attributes, their normal setting (i.e., if you don't adjust them with these statements), and what they mean:



Chapter 11:
**AppleScript
Operators**

Attribute	Default	Description
case	ignores	Distinguishes uppercase from lowercase letters, applying each character's strict ASCII value for comparisons.
white space	considers	Regards spaces between characters as characters to be compared.
diacriticals	considers	Distinguishes characters with diacritical marks (e.g., á, ô, è) from same letters without the marks.
hyphens	considers	Regards hyphens as characters to be compared.
punctuation	considers	Regards punctuation symbols as characters to be compared.

For example, on its own, AppleScript works like this:

```
"howdy" = "HOWDY"           -- result: true
```

If you want to know if a string is exactly like another, including uppercase versus lowercase letters, you'd invoke the `considering` statement as follows:

```
considering case
    "howdy" = "HOWDY"
end considering               -- result: false
```

You can also combine different attributes with different default behaviors into a single **considering** or **ignoring** statement. For example, here is a progression of results from building a complex **considering** statement:

```
"Chocolate Éclair" = "chocolate éclair"
-- result: false (diacritical É
-- and e don't match)
```



Chapter 11: AppleScript Operators

```
ignoring diacriticals
    "Chocolate Éclair" = "chocolate éclair"
end ignoring
    -- result: true

ignoring diacriticals but considering case
    "Chocolate Éclair" = "chocolate éclair"
end ignoring
    -- result: false (uppercase and
    -- lowercase didn't match)
```

To use these statements reliably, the left operand of a comparison operator inside a **considering** or **ignoring** statement should be a value in the script, as opposed to a reference to an application object. In other words, place the string contents of an application object into a variable, and then use that variable as the left operand in a **considering** or **ignoring** statement operation.

One other attribute, which we covered in Chapter 10's discussion about Apple event timeouts, is the **application responses** attribute. Its default behavior is to consider these responses, thus forcing your script to pause while it waits for the target application object to respond back that it received a command. By setting some statements inside an **ignoring application responses** statement, your script sends the commands and keeps on chugging. You can do this only if you're not expecting any values to come back from the commands, because when your script ignores responses—it really ignores responses.

Operator (and Reference) Precedence

When you begin working with complex expressions that hold a number of operators and a variety of references, it is vital to know in what order AppleScript evaluates those expressions to arrive at its ultimate value. AppleScript assigns different priorities or weights to types of operators in an effort to achieve uniformity in the way it evaluates complex expressions.

In the following expression:

```
4 * 5 + 10          -- result: 30
```

AppleScript uses its scheme of precedence to perform the multiplication before the addition—regardless of where the operators appear in the statement. Therefore, AppleScript first multiplies 4 by 5, and then adds that result to 10, for a result of 30.

But you can alter precedence by a pair of parentheses. Parentheses, as you'll see in the table below, have the highest precedence, telling AppleScript to evaluate what's inside them before doing any other evaluations—even if what's inside is much lower in precedence than other operators in the expression. If we really wanted the earlier expression to add the 10 and 5 together before multiplying that sum by 4, then we would have to force the evaluation of the lower precedence (the addition) ahead of the multiplication:

```
4 * (5 + 10)        -- result: 60
```



Chapter 11:
**AppleScript
Operators**

You see that the parentheses had a great impact on the result of this expression. And if your expression contains parentheses inside a set of parentheses, then the innermost parenthetical expression evaluates first, working evaluation outward from there.

<i>Precedence</i>	<i>Operator</i>	<i>Notes</i>
1	()	From innermost to outermost
2	+ and -	When used with a single value to signify positive or negative
3	's	Object containment (left to right)
4	of in	Object containment (right to left)
5	my its	Object containment (right to left)
6	^	Exponent (right to left)
7	* / and ÷ div mod	Multiplication and division
8	+ -	Addition and subtraction
9	&	Concatenation
10	as	Value coercion
11	< ≤ > ≥ contains	Comparison operators and their synonyms
12	= ≠	Equality
13	not	Unary Boolean negation operator
14	and	Boolean operator
15	or	Boolean operator
16	whose	Filter reference
17	reference to	Object reference

***Table 11.1.** AppleScript Operator & Evaluation Precedence*



Chapter 11:
**AppleScript
Operators**

AppleScript evaluates expressions according to a strict order of precedence. Since expressions can include references to objects (and how any object may be related to another object), it's valuable to know what gets evaluated first in an expression. This may influence how you use parentheses to control expression evaluation. Table 11.1 lists the precedence order for operators (including containment operators) for AppleScript.

Whenever an expression contains more than one operator of the same precedence (possible only with exponent, math, **and**, and **or** operators), AppleScript performs its evaluations from left to right with one exception. Multiple exponent operators in an expression are evaluated from right to left. For example, in the expression:

`4 * 5 * 6`

AppleScript proceeds from left to right, first multiplying 4 by 5, then multiplying 20 by 6. But in the expression:

`2^3^4`

AppleScript first calculates `3^4` (result: **81.0**), and then calculates `2^81.0` to reach an enormous number.

Notice that any math, concatenation, and class coercion is performed prior to any comparison operators. This allows all expressions that can act as operands for these operators to evaluate fully before they are compared.

The key to working with complex expressions—when they're giving you trouble—is to isolate evaluating components, and try them out by themselves if you can. See additional debugging tips in Chapter 13.

Next Stop

Speaking of trouble, the next chapter shows you how to head off potential problems in scripts by trapping and handling errors efficiently. The test of a good scripter is how bulletproof a script is; and that is usually a measure of how well errors are anticipated and handled.



Chapter 12 Error Checking in Scripts

In the rush to get a script to do what you want, it's hard to resist the temptation to take a “damn the torpedoes, full speed ahead” attitude toward the durability of the code. This is especially true when incorporating user interface elements that allow users to interact with the script. In the real world of using computers, users don't always do what you expect them to do, so your scripts must be on the lookout for potential problems and know how to handle them when those problems arrive. That's what error checking is all about.

Why We Forget to Error Check

The process of writing a script (or any program in any language) is an intense one. We focus on a problem and the scripted solution with all our might, completely immersed in the flow of the scripts, the statements, the decisions—all those things that eventually make the script work. Something interesting comes of this intensity: we know the script and its inner workings so well, that we don't take into account something going wrong that we didn't encounter ourselves in the script writing process.

An excellent example of this is using the **display dialog** command to prompt a user to enter some information. The statement may be something like this:

```
display dialog "Enter a number:" -  
    default answer ""
```

If the script goes on from here taking the **text returned** value of the result, such as coercing the number into an integer and applying it to some math or integer parameter of another command, you think



Chapter 12:
**Error Checking
in Scripts**

nothing of it. After all, the instructions in the dialog explicitly tell the user to enter a number. Moreover, when *you* use it, you know what happens in the script afterwards, so you wouldn't think of entering anything but a number.

Thinking like that can only send you, the scripter, to your doom.

Here is a list of some of the things a user could do (usually inadvertently) to botch up your perfect script:

1. Click the Cancel button instead of the OK button.
2. Click the OK button without entering a character.
3. Enter a real value when you expect an integer.
4. Enter the text word for a number (e.g., “two”).
5. Enter a valid integer, but letting the finger slip to some other non-numeric character an instant before pressing Return.
6. Enter a negative number, when your requirement is for a positive one.
7. Enter gobbledygook to see if the script breaks (this one's intentional).

If you haven't protected your script against these possibilities, most of them could present the user with the embarrassing “AppleScript Error” alert that AppleScript displays whenever it can't accept the value presented to it in a statement. The alert is embarrassing, because a user views this error as a sure sign of sloppy programming. Your entire script is now suspect.

This is actually an important point about point-of-view. An inexperienced programmer might consider the problem to be the user's fault because the user didn't follow instructions. But the problem is really the programmer's. Users' expectations are rightfully high. Our job as scripters is to protect the user from making mistakes. The less the user has to think about a script's requirements, the more he or she can think about how the script offers a solution or automation for something that needs doing.



Chapter 12:
**Error Checking
in Scripts**

Anticipating Errors

The list above shows the types of errors a scripter should anticipate users making, whether the errors are made intentionally or not. In the case of the **display dialog** command, the errors break down into three categories:

- ▶ Clicking where you don't expect (the Cancel or other button you scripted).
- ▶ Entering the wrong kind of information.
- ▶ Entering no information.

For something like the **choose file** command (Chapter 7), which displays the open file dialog box, even though you can control the kinds of files that are active in the list, there are still a couple of potential problems:

- ▶ Clicking the Cancel button.
- ▶ Selecting a file that has data unusable by your script.

Recall that the **choose file** command only returns the alias to the file. A succeeding command that opens the file in an application may encounter disk or network problems preventing the **open** command from working properly—another kind of error to worry about for any command that accesses a mounted volume.

AppleScript Errors

When AppleScript encounters an error while running a script, it gathers several pieces of information. If your script is set up to trap errors, then it can extract this information and perhaps offer some suggestions or alternate routes for the user to take. The way you set up a script to trap an error is to place the statement(s) that could generate an error inside a **try** compound statement. The formal syntax for this kind of statement is:

```
try
    [ statementToTest ] ...
on error [ errorMessage ] →
    [ number errorNumber ] →
    [ from offendingObject ] →
    [ to expectedType ] →
    [ partial result resultList ]
        [ global variableID [, variableID ] ... ]
        [ local variableID [, variableID ] ... ]
        [ statementHandlingError ] ...
end [ try | error ]
```

If you anticipate that a command could generate an error, then place the statement containing that command after the **try** statement. If any errors come back, they'll be trapped by the **on error** handler that is built into this **try** statement construction. The handler intercepts the error message that AppleScript normally displays for us in the form of an error alert. At the same time, a number of labeled parameters are passed along, which your error handler can assign to variables. In



Chapter 12: Error Checking in Scripts

most cases, you will need only two of the parameters: the error message (the text of the error message that would appear in the error alert) and the error number (one of the error IDs).

Here is a simple example of providing error handling for a track in an iTunes playlist:

```
tell application "iTunes"
    try
        set oneAlbum to album of track 1 of
        playlist "Dusk"
        on error errMsg number errNum
            if errNum = -1728 then
                -- can't get some reference
                if not (exists playlist
                "Dusk") then
                    display dialog "The
                    "Dusk" playlist cannot be found in iTunes.
                    You may have renamed or deleted it." buttons
                    {"Cancel"} default button "Cancel"
                else
                    display dialog "Make
                    sure there is at least one track in the
                    "Dusk" playlist." buttons {"Cancel"} default
                    button "Cancel"
                end if
            else -- some other error
                display dialog errMsg & "
                Call the Help Desk." buttons {"Cancel"}
                default button "Cancel"
            end if
        end try
        (* statements to continue processing the
        album data *)
    end tell
```

All action takes place inside a **tell** statement directed at iTunes. Its first task is to obtain the

album property of a named playlist. Because we're dealing with an operation that requires that the named playlist exist, the script must take into account that the user may have deleted or renamed the playlist. The **set** command that grabs the property value is the statement that goes in the **try** construction. Should there be any error, execution passes immediately to the **on error** handler within this **try** construction.

In the error handler, we assign two of the parameters (the error message text and the error number) to variables, which we use in different circumstances during error handling. One of the purposes of a well-crafted error handler is to help diagnose the problem and offer a solution to the user. Since we have a good idea that the kinds of errors for the **set** command will have to do with the existence of the playlist, we can perform some additional testing about the error and provide helpful information. Therefore, our first test is to make sure the error is one we know: the -1728 error (see Table 12.1 at the end of this chapter for error numbers), which signifies AppleScript couldn't get something.

Our first suspect in this case is that iTunes doesn't currently have a playlist by the name "Dusk." We therefore test for the existence of that playlist, just to make sure that a missing playlist is the problem. If the playlist is missing we alert the user in plain language. Otherwise, we suggest that the error has to do with the playlist having no tracks (the script is



Chapter 12:
**Error Checking
in Scripts**

looking for the first track). The one final possibility is that the error is something we couldn't anticipate, in which case, we supply AppleScript's error message in full, along with information about getting help—since any other error surely indicates a more significant problem.

Pay particular attention to the fact that all **display dialog** commands that supply error indications to the user have a single Cancel button—not an OK button. This is quite important as far as the script execution flow is concerned. The Cancel button of any dialog (in addition to sending an error command of its own) automatically stops execution of the script if no further error trapping is performed. This is important, because it guarantees that further script statements after the **try** construction won't execute once the Cancel button is clicked.

Trapping Cancel Buttons

The **display dialog**'s Cancel button isn't the only one to stop script execution if left untouched. The Cancel button on any file or system-oriented dialog (e.g., **choose file**, **choose folder**, **choose file name**) works the same way. But there may be times when your script needs to perform some cleanup work as a result of the user clicking the Cancel button—perhaps resetting a script object property (Chapter 15) or advising a user about the consequences of the cancellation.

Your script can trap the user's click of the Cancel button to perform those cleanup tasks. Again, you use the **try** statement to do it. Clicking the Cancel button generates an error message—but an error message that doesn't trigger the AppleScript Error alert.

The most surefire way to trap for this is to look for an error number of -128. Here's an example of how you might use this feature:



Chapter 12: Error Checking in Scripts

```
tell application "TextEdit"
    try
        open (choose file with prompt
            "Select a file:")
        on error errMsg number errNum
            if errNum = -128 then
                display dialog "You've
                    canceled, so I must stop." buttons {"Cancel"}
                default button 1
            else
                display dialog errMsg & "
                    Call the Help Desk." buttons {"Cancel"}
                default button 1
            end if
        end try
        (* statements to continue processing if
            file opens *)
    end tell
```

In this script's error processing, we look especially for the Cancel button's error number and provide an alert. This alert is for demonstration purposes only: there's no need to tell a user a process has canceled. Instead, the script may want to branch to another subroutine or reset property values. It all depends on what your script is supposed to be doing and what you anticipate the user's expectation would be when clicking the Cancel button of a dialog.

Purposely Generating Errors

AppleScript isn't the only culprit when it comes to generating error messages: you can do it, too. I can think of three excellent reasons why you'd want to generate errors in a script. One would be to help you in writing a script (see Chapter 13 on debugging for more). Another would be to consolidate error handling in a script. A third allows your script to bail out at any point in a script.

To make AppleScript's error alert appear, you send the **error** command, which has practically the same labeled parameters as the ones indicated in the **on error** handler syntax, shown earlier in this chapter. Here's the error command syntax:

```
error [ errorString ] [ number integer ] -
    [ from objectReference ] -
    [ partial result resultList ]
```

The nice feature of this command is that you are in charge of the error message, number, and other parameters. Therefore, you can build your script's own set of error messages and numbers.

When you send the **error** command, AppleScript displays its AppleScript Error alert with whatever *errorString* you send along. With that AppleScript Error window title, the alert is not altogether friendly for casual users. But when the user clicks the Stop button, script execution stops.

Here's a revised version of the previous iTunes script, taking advantage of AppleScript's **error** command:



Chapter 12: Error Checking in Scripts

```
tell application "iTunes"
    try
        set oneAlbum to album of track 1 of
        playlist "Dusk"
        on error errMsg number errNum
            if errNum = -1728 then
                -- can't get some reference
                if not (exists playlist
                "Dusk") then
                    error "The "Dusk"
                    playlist cannot be found in iTunes. You may
                    have renamed or deleted it."
                else
                    error "Make sure there
                    is at least one track in the "Dusk"
                    playlist."
                end if
            else -- some other error
                error errMsg & " Call the
                Help Desk."
            end if
        end try
        (* statements to continue processing the
        album data *)
    end tell
```

For longer scripts, you can even build a separate error subroutine, which consolidates the error handling for multiple **try** statements scattered throughout the script. Several routines in the same script trap errors on their own with the **try** statement. Commands in the error handler redirect some values to a separate subroutine in the script (perhaps named something like **handleErrors**) that is used as a common error display routine for all other subroutines in the script.

Bailing Out of a Script

Even if you don't use a **try-error** construction, your script can cease execution at any point. For example, a deeply nested **if-then** decision may leave the user at a point where the best thing to do is provide some friendly message (with an OK button) and halt any further execution. The line immediately after the **display dialog** command should be:

```
error number -128
```

This is the same message that Cancel buttons in dialogs send. Unless there is a specific trap for this error in your script, AppleScript stops running the script without throwing an execution error alert into the user's face.



Chapter 12:
**Error Checking
in Scripts**

Error Numbers and Messages

You will almost always see negative numbers used for error numbers returned by AppleScript, the System, and scriptable applications. Errors from the same source generally belong to a numeric series. For example, Apple event errors are in the -1700 range; AppleScript errors are in the low positive digit and -2700 ranges; a scriptable application's own errors are generally in the -10,000 range.

The fact that an error that filters back to your script contains both an error number and a plain language message (and sometimes more information) makes it easier to figure out what's happening even when you don't have a reference guide to every error number. While the following suggestion falls under the debugging heading, it's something you can use to help you find out what's happening in a recalcitrant script:

```
try
    (* statement(s) under test *)
on error errMsg number errNum from suspect
    display dialog errMsg & return & errNum
    buttons {"OK"} default button 1
    get suspect
end try
```

This error handler performs triple duty. A **display dialog** shows you both the full text of the error message, plus the error number (in case you want to build a specific trap for it in your script). Lastly, it gets (for the Result pane) a reference to the object that was under consideration when the error

occurred. Usually this information is also in the error message, but this extra parameter sometimes carries additional information that can't be displayed as text in a dialog. When the number of statements under test is large, these bits and pieces of information should help you locate the trouble spot more quickly.

Although the above error handler makes a complete listing of errors somewhat redundant, below are lists of the more common error numbers and their messages:

System Errors

- ▶ -43 File *name* not found.
- ▶ -48 File *name* is already open
- ▶ -50 Parameter error
- ▶ -61 File not open with write permission
- ▶ -108 Out of memory.
- ▶ -128 User canceled.
- ▶ -5000 Network file permission error.
- ▶ -30720 Invalid date and time *string*.

Apple Event Errors

- ▶ -1700 Can't make *expression* into a *className*.
- ▶ -1701 Some parameter is missing for *commandName*.
- ▶ -1708 *Reference* doesn't understand the *commandName* message.



Chapter 12:
**Error Checking
in Scripts**

- ▶ -1712 AppleEvent timed out.
- ▶ -1719 Can't get *reference*. Invalid index.
- ▶ -1723 Can't get *expression*. Access not allowed.
- ▶ -1728 Can't get *reference*.

AppleScript Errors

- ▶ 10 Invalid or unrecognized Apple event
- ▶ -2701 Can't divide *number* by zero.
- ▶ -2753 The variable *variableName* is not defined.

What's Next?

Most of these errors—especially Apple Event and AppleScript errors—should be used as guides for debugging purposes and should not appear or even need to be handled in a script you hand to someone else to use.

Which leads us to the next chapter to cover debugging techniques for AppleScript.



Chapter 13 Debugging Scripts

AppleScript's Script Editor doesn't include a debugger in the traditional sense of the word. If you've programmed in other languages or environments, you probably have the experience of being able to step through program execution, statement-by-statement. Other tools then allow you to examine the values of variables at each step, helping you spot where data isn't in the form you expected when you wrote the program.

Debugging is a critical part of the AppleScript script writing process. But if you're not ready to upgrade to a third-party editor that includes robust debugging features or advance to the AppleScript Studio development environment, then you're restricted to using a few comparatively crude tools: the Result pane, the Event Log, the **display dialog** scripting addition, along with some additional logging commands. Without the help of third-party script editors you cannot step through a script to observe variable values, but we can use the Script Editor's tools at any point to get a reading of intermediate values in either of these venues.

The goal of this chapter is to present some suggestions about how to work through problems you encounter while assembling a script with Script Editor.



Chapter 13:
**Debugging
Scripts**

Script Editor Setup

As you work on a script, you should have the bottom pane of the editor's window open large enough so you can see values and messages that appear there during script execution. After the script pane, the most important area is the Result pane. Take a moment to set things up the way you like and then select **Save as Default** from the **Window** menu to create a default editing window arrangement.

You should also be prepared to open one or two smaller script windows in the same vicinity as the main script window. As your script grows, you will want to test isolated commands and expression evaluations away from the main script. If your script is working with a scriptable application, one of the extra script windows (which can remain untitled and unsaved unless you want to save some fragment for the next scripting session) should have a **tell** statement set up to accept commands directed at the application. This prevents you from having to write a **tell** statement to the application over and over again to check evaluation of things like object references and values returned from application-specific commands.

The third script window is one you'll use to test statements and expression evaluations that are not performed by a scriptable application, but by AppleScript. You must exercise care when isolating statements or fragments like these: if they appear in your script anywhere nested

within a **tell** statement, you should isolate them in the script window you use for testing application-specific statements: even though the application and AppleScript may have statements and objects in common (e.g., **character**, **word**, **paragraph**), the application may treat certain operations differently (or not support them as fully as AppleScript). Context is very important when evaluating expressions.



Chapter 13:
**Debugging
Scripts**

Compile and Execution Errors

No sooner do you start writing a script than you must recognize the difference between a compile-time error and an execution error. They're quite different, and require different debugging tactics.

Unless you're writing an enormous script on a slow Macintosh, you should have Script Editor check the syntax of any completed thought. By that I mean constructions such as **if-then**s, **repeats**, **try** statements, nested **tell** statements, **considering** and **ignoring** statements—any structure that requires an **end** statement.

You can even check the syntax before completely filling in these kinds of balanced structures. Whenever I start such a structure, I immediately enter the **end** statement, as in:

```
repeat with eachItem of longList  
  
end repeat
```

In other words, I type the opening statement, type a return, and then enter the **end** statement. I'm also likely to press the Enter key to perform a syntax check on the outer structure. By knowing that the outer frame works, I can start filling in the space inside the complex statement. I can then check the syntax every few statements if I want confirmation that I'm handling complex object references or command parameters properly.

You know when an error is a compile-time error

when the Syntax Error alert appears (Figure 13.1). If applicable, Script Editor also highlights the word, expression, or statement in the script that is causing the compiler grief—although the problem may be in an earlier line. When you click the OK button, the script will not have reformatted itself in the pretty printing.

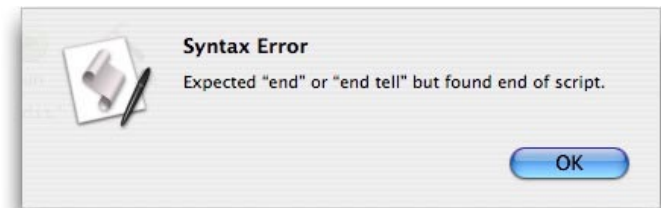


Figure 13.1. A syntax error alert.

The compiler doesn't syntax check application objects to the extent you might expect. For example, if an object's property requires a list value, the compiler doesn't check that the value you're assigning to the property is a list. Therefore, the syntax checker would pass setting a **bounds** property of a window (which usually requires a list value containing four integer values) to a single integer. Only when the script runs does the application protest (although perhaps not with sufficient guidance as to the problem).

Saving an uncompileable script is sometimes necessary when you are mired in a problem, but you need to quit Script Editor. Trying to save a script as



Chapter 13: Debugging Scripts

a compiled script (the default choice) forces Script Editor to compile the script. If compilation fails, you see the same Syntax Error alert as when you attempted to compile the script. The only workable choice for saving an uncompiled script is the Text option in the Save dialog. When you reopen the file later, any pretty printing that may have been in part of the script is gone.

Of course, the only kind of errors you could get during a syntax checking moment are compile-time errors. But if you instruct a newly modified script to run, it first compiles, and then runs. You'll know which kind of error you get by looking at the error alert. A Syntax Error (Figure 13.1) appears if the error occurred during the compilation process; an AppleScript Error alert appears if the problem occurred while actually running the script (Figure 13.2). An AppleScript Error alert means that the script passed the syntax check, but something about the values involved while running the script caused it to break.

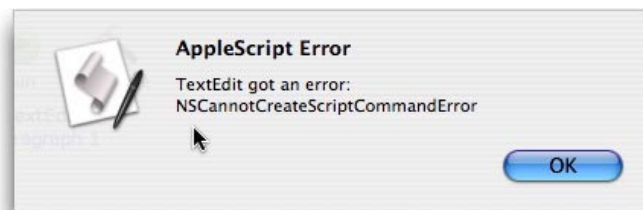


Figure 13.2. A run-time error alert.

Using The Result Pane

As we've demonstrated hundreds of times in examples throughout this book, the Result pane shows the value of the last expression to execute in a script. You can use that behavior to your benefit when you isolate a statement or short series of statements, and you're having trouble with one of the values therein. Similarly, while a script is under construction, you can find the value of the last executable expression in the script, whether it's nested in a compound statement or at the end of the script. It's not unusual to append to a script a variable whose contents you want to view in the Result pane after the script runs.

While the Result pane shows the value of only one expression, you can gang up two or more expressions into a list, which becomes the single expression displayed in the result window. Here's an example:

```
display dialog "Enter your last name:" ->
    default answer ""
set userName to text returned of result
display dialog "Enter your ID number:" ->
    default answer ""
set userID to text returned of result
{userName, userID}
-- result: {"Danny", "345"}
```

The Result pane shows the evaluation of whatever expression is in the last statement.



Chapter 13: Debugging Scripts

Display Dialog

It is fairly common practice to use **display dialog** as a debugging tool to show the contents of intermediate values. Simply insert the command after any statement that influences some value of interest, and pass that value as the parameter to the command.

This command is somewhat risky, however, if you're not sure of the classes of values you want to examine. Remember that the **display dialog** requires a string as an argument that defines what appears in the dialog. Most numbers automatically coerce to strings within this command, so this method works for that kind of data. But if the value you need to inspect happens to be a list, record, or data, your debugging is compounded by the fact that the **display dialog** command will generate an execution error of its own, complaining about the wrong class (type) of parameter.

Event Log Pane

Despite the lack of step-by-step debugging, Script Editor does provide a feedback mechanism for scripters: the *Event Log*. The Event Log pane of a Script Editor window displays the messages being sent to applications outside of AppleScript (i.e., other applications and scripting additions), as well as the resulting data that comes back from a number of commands. The Event Log can therefore assist in run-time debugging.

Once the Event Log pane is open, any script you run in Script Editor will have its events written to the log. Consider the following script:

```
display dialog "Enter a number between 1 and  
10:" default answer ""  
set userValue to (text returned of result) as  
real  
if (userValue < 1) or (userValue > 10) then  
    display dialog "That value is out of  
    range." buttons {"OK"} default button 1  
else  
    display dialog "Thanks for playing."  
    buttons {"OK"} default button 1  
end if
```

Figure 13.3 shows the log created while running the script and entering a "5" in response to the prompt. The most important item to notice about the log is that only events that go outside AppleScript appear in the log. For this script, that means the **display dialog** events, which go to the scripting addition. All internal machinations, such as setting the values of variables or operations involved with conditional



Chapter 13: Debugging Scripts

control structures (**if-then**s, **repeats**), do not appear in the log. If the event returns data of any kind (such as the record that comes back from the **display dialog** scripting addition command), the value appears in the log indented.

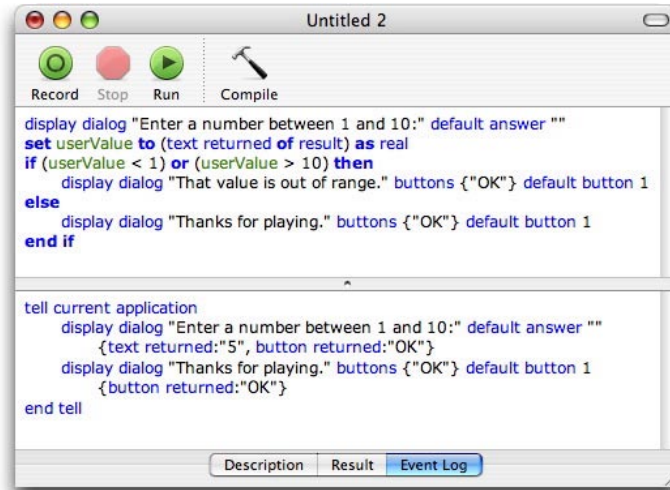


Figure 13.3. The Event Log shows all returned values.

A primary value of the Event Log is that it displays the precise contents of data passed as parameters with commands and the returned values. For example, if a script assembles command parameters from various sources as variables, you can use the Event Log to examine the true content of parameters as they are being sent to the target application. In the following script:

```
set iTLibPath to (path to music folder as
    text) & "iTunes:iTunes 4 Music Library"
set libModDate to date string of (modification
    date of (info for alias iTLibPath))
display dialog "Your iTunes Library was last
    touched on " & libModDate & "."
```

we extract the modification date of the iTunes library file (as a string) and then insert the string as a piece of the parameter sent to the **display dialog** command. Figure 13.4 shows the Event Log for this script. The log shows all the transactions between the script and various scripting additions (**path to**, **info for**, **display dialog**). All variable setting, record-field extraction, coercion, and concatenation is performed within AppleScript, so those items don't appear in the log. But the **display dialog** command's parameter is shown in the log exactly as it is sent to the **display dialog** scripting addition.



Chapter 13: Debugging Scripts

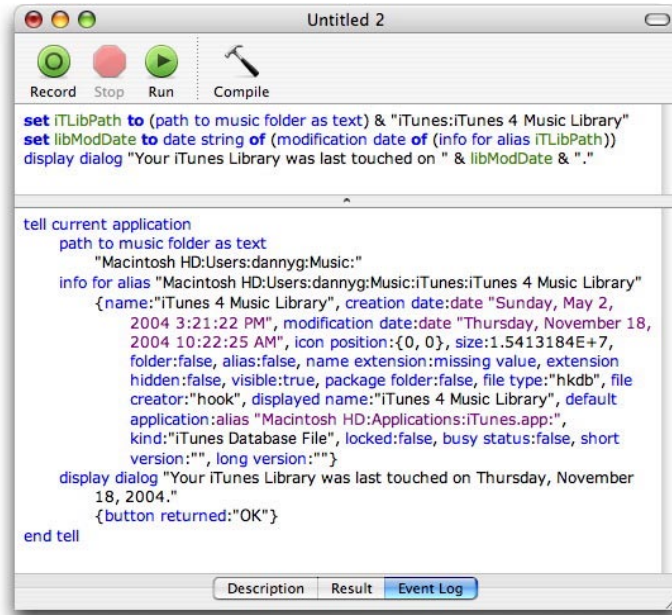


Figure 13.4. Event Log for a script that calls scripting additions.

Event Log Commands

Three scripting addition commands work directly with the Event Log to let you control the log during script execution: **start log**, **stop log**, **log expression**. Two of these commands, however, no longer appear to be supported (in Script Editor 2.0 and Mac OS X 10.3). I'll document them just the same, in case they should return to the AppleScript lexicon sometime in the future.

Unless instructed otherwise, the Event Log

automatically displays events and results whenever the Event Log pane is open. Therefore, if you need to view only a portion of a script in the log, you can turn the log on and off from within the script (logging slows the execution speed of a script, so for a long script, you may want to log only a small portion).

Let's say in the iTunes library modification date script, above, we only want to see the **display dialog** command and its parameter. We can disable the Event Log at the beginning of the script and turn it back on when needed, as in:

```
stop log
set iTLibPath to (path to music folder as
text) & "iTunes:iTunes 4 Music Library"
set libModDate to date string of (modification
date of (info for alias iTLibPath))
start log
display dialog "Your iTunes Library was last
touched on " & libModDate & "."
```

Only the **display dialog** command and its result are logged. When they're supported by your version of Script Editor and Mac OS X, you may insert **stop log** and **start log** commands in as many places as you like within a script.

The third logging command, **log expression**, lets you record the value of variables into the Event Log—and it works in Script Editor 2.0 and Mac OS X 10.3. The **log** command inserts a line into the Event Log (in the form of a comment); the line may be anything you pass as a parameter to the **log**



Chapter 13: Debugging Scripts

command.

In the iTunes library modification date script, above, we can use the **log** command to view the contents of just the **libModDate** variable before sending it as part of the **display dialog** command:

```
set iTLibPath to (path to music folder as text) & "iTunes:iTunes 4 Music Library"
set libModDate to date string of (modification date of (info for alias iTLibPath))
log libModDate
display dialog "Your iTunes Library was last touched on " & libModDate & "."
```

Figure 13.5 shows the contents of the variable value in the Event Log pane. You are not limited to using variables as parameters to the **log** command. Any string expression will also do, including concatenated strings, as in

```
log "The libModDate variable is: " & libModDate
```

This command presents an Event Log entry that looks like this:

```
(*The libModDate variable is:
Thursday, November 18, 2004*)
```

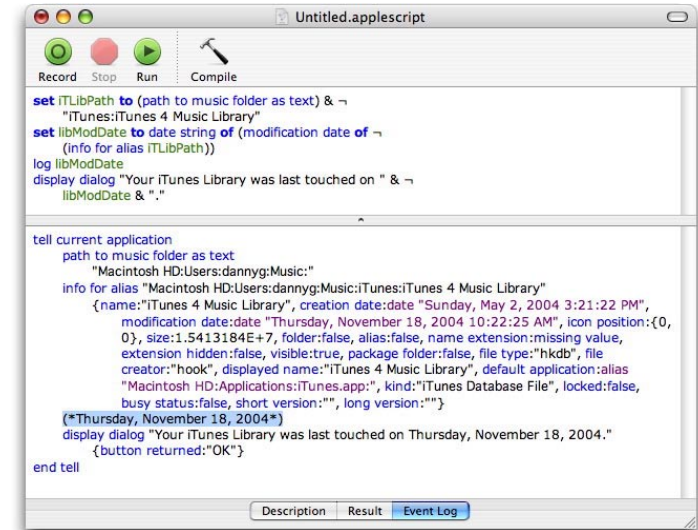


Figure 13.5. Using the Event Log to show explicitly logged data (highlighted for emphasis).

Importantly, the **log** command displays values of other classes. Therefore, if a variable contains things like a date object or a reference to a file or alias object, the Log coerces the value to a string for review purposes. But be careful not to assume that the actual value in the script is also a string.

As one more example, we use the **log** command inside a repeat loop to show the values of random values:



Chapter 13: Debugging Scripts

```
stop log
set counter to 0
set randomValue to 0
repeat while randomValue ≠ 5
    set randomValue to random number 10
    set counter to counter + 1
    log "Loop " & counter & " = random value
    of " & randomValue
end repeat
display dialog "I got a five in " & ~
    counter & " loop(s)."
```

Figure 13.6 shows a typical Event Log for this script, demonstrating how we can view the values of multiple variables with a single **log** command. Other uses include inserting **log** commands at various junctures of a complex **if-then** construction, to log when and with what values script execution passed through those points.

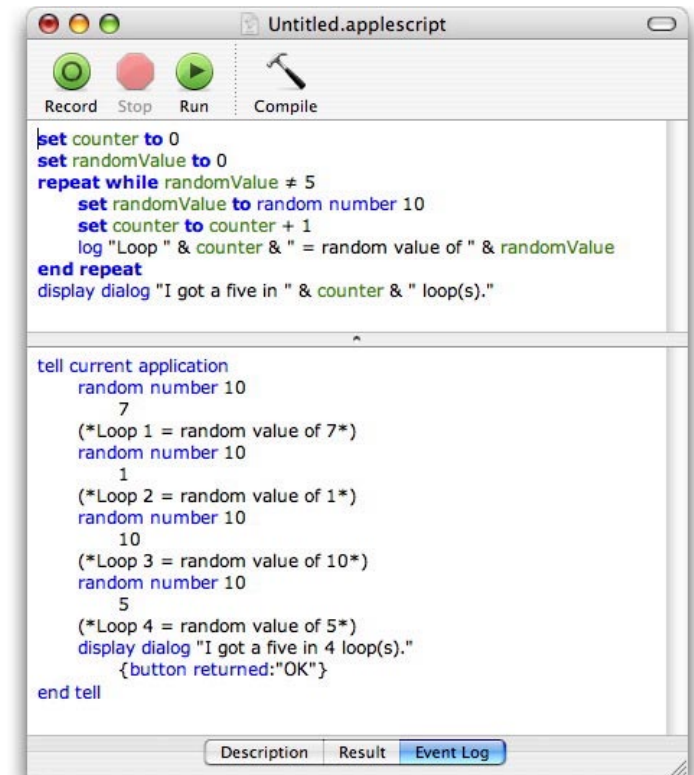


Figure 13.6. Viewing multiple variable values in single log statements.

Without a full debugger, the Event Log is about the best debugging tool that Script Editor has to offer. Third-party editors offer additional debugging tools, which serious scripters should consider.



Chapter 13:
**Debugging
Scripts**

Aural Clues: Beeps and Speech

They're crude debugging tools, but in lieu of a real debugger, system beeps can help you determine if execution is taking a desired path. For example, in an **if-then** construction, a **beep** command (Chapter 6) placed in one of the execution pathways can tell you if execution is actually going that way: run the script and listen for the beep. Use only one beep per test, because you need to be scientific about locating bugs: test for one identifiable thing at a time.

Beeps also help in repeat loops. For example, if you suspect that a repeat loop is exiting earlier than you planned, place a **beep** command as the first line inside the loop. Run the script and count the beep sounds as they play. The number of beeps tells you how many times the loop started executing—somewhere in that last loop, a condition was met that let the loop exit.

Alternatively, you can use the AppleScript **say** command (Chapter 7) to vocalize clues or even values while a script is running. This works best for short values, such as loop counter variables.

Try Statements

As you saw in the previous chapter, the **try** statement and error handler is an equally valuable debugging tool. The error message conveyed to you by AppleScript contains potentially useful information that can help you spot the problem with your script execution.



Chapter 13: Debugging Scripts

“Commenting Out” Lines

If you go on a tear, and write many lines of script without checking them along the way—it’s occasionally an undeniable temptation when you’re working through a complex logic sequence—you may discover that you get some compile errors early in the script. If you want to see if part of the script compiles properly or check the value of a result, you need a way to tell AppleScript to ignore one or more lines of script.

Rather than cut the following lines away, and paste them into another script window as a holding cell, it’s more convenient to comment out the lines you don’t want to compile or execute. If you’re commenting just a line or two, the double-hyphen at the beginning of the line is a quick way to get them out of the line of fire.

But for larger segments, it’s even quicker to surround the group with the `(*...*)` pair of comment symbols. Insert the `(*` (left parenthesis, asterisk) at the beginning of the line that is to be taken out; insert the `*)` (asterisk, right parenthesis) at the end of the group. For example:

```
tell application "TextEdit"
    set windowCount to count of windows
    if windowCount = 0 then
        display dialog "There are no open
windows to arrange." buttons {"Phooey"}
        default button 1
    else
        (*activate
set nextWindowBounds to {1, 39,
```

```
481, 305}
        repeat with i from 1 to windowCount
            set bounds of window i to
nextWindowBounds
            move window i to front
            repeat with j from 1 to 4
                set item j of
nextWindowBounds to (item j of
nextWindowBounds) + 20
            end repeat
        end repeat*)
    end if
end tell
```

In the example above, I commented out the contents of entire **else** statement to make sure the first part of the **if-then-else** statement works. I left the overall structure intact so that the balancing **end** statement is available to permit proper compilation. As you fix bugs, you can move the starting symbol down the script, testing each line or couple of lines as needed.



Chapter 13: Debugging Scripts

A Debugging Demonstration

Let me demonstrate a typical debugging sequence, using a number of the techniques described above. The script's purpose was to work with the Bill of Rights TextEdit document (Chapter 8), setting the “Article” paragraphs to a 20pt font size. While this could be accomplished in a number of ways, I chose the filtered reference. I figured I could have TextEdit bold all paragraphs whose first word was “Article.”

Part 1: Quick-and-Dirty

My first attempt in a rush was the following:

```
tell application "TextEdit"
    set size of paragraphs whose first word =
        "article" to 20
end tell
```

The script compiled without complaint. But an AppleScript error said: “TextEdit got an error: NSCannotCreateScriptCommandError” (Figure 13.7). A portion of the main script statement was highlighted (see Figure 13.8), indicating that the problem wasn't necessarily with the usage of the **set** command or the value being assigned. This long-winded AppleScript error usually indicates that something is wrong with a reference.

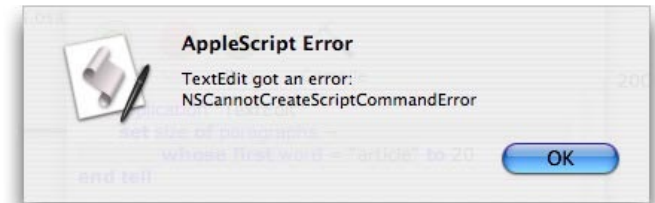


Figure 13.7. The error message

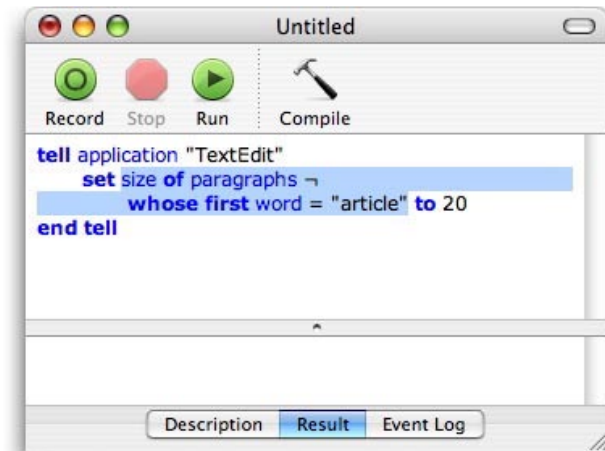


Figure 13.8. The highlighted problem

Because my first stab at the script used a comparatively complex reference (a whose clause, no less), I backed up a bit to see if perhaps I wasn't setting the size correctly. I verified in the TextEdit dictionary that a **paragraph** class object has a **size** property whose value is an integer. So I modified the script to set the size of just the first paragraph:



Chapter 13: Debugging Scripts

```
tell application "TextEdit"
    set size of paragraph 1 to 20
end tell
```

This also compiled, and came up with the same error as before. But this time, the entire statement was highlighted as the problem. Because the **set** command and **size** property are pretty difficult to mess up (assuming I spelled them correctly), the last remaining suspect was the reference to paragraph 1. Perhaps the reference wasn't complete enough for TextEdit.

It's quite common in a scriptable Application that allows multiple windows to need to help the application refer to the proper window or document, even if there is only one window or document showing. Looking through the dictionary again, I found that the **application** class (the object being targeted by the **tell** statement) has two types of elements: the **window** and **document** classes (for some reason, **document** is listed twice). Reference types supported for these elements include index and name. That led me to try referencing the paragraph within one of these elements.

The first try was the more global of the two: the window. Staying with a single paragraph reference, I modified the script to the following:

```
tell application "TextEdit"
    set size of paragraph 1 of window 1 to 20
end tell
```

Compilation was fine, but the script ran into the

same reference problem when running it as it did without the window reference. Time to try the document reference:

```
tell application "TextEdit"
    set size of paragraph 1 of document 1 to 20
end tell
```

Not only did the script compile, but it ran and changed the font size of the first paragraph, as desired. I learned from this experience that TextEdit (and perhaps other applications) required references to an object in a document to be referenced by way of the **document** class.

Part 2: Getting Granular

Time to try the filtered reference again, now that more global issues have been solved. But before getting to that, it would be helpful to verify that the plural reference to paragraphs will retrieve all paragraphs of the document. The dictionary under the **paragraph** class indicates that the plural should be supported, but in AppleScript, seeing is believing. Thus, I run this test in a separate script window to see the results

```
tell of application "TextEdit"
    paragraphs of document 1
end tell
```

The result came back as a list, which was predicted, since I was asking for multiple objects. The beginning of the data looked like this:



Chapter 13: Debugging Scripts

```
-- result: {"ARTICLE I
", "Congress shall make no law respecting an
establishment of religion, or prohibiting the
free exercise thereof; or abridging the
freedom of speech, or of the press; or the
right of the people peaceably to assemble,
and to petition the government for a redress
of grievances.
", "
", "ARTICLE II
", "A well regulated militia, being necessary
to the security of a free state, the right of
the people to keep and bear arms, shall not
be infringed.
", "
", "ARTICLE III,...}
```

Each paragraph's text was its own string item in the list, as I expected. It's a subtle point, but the result showed that every third paragraph in this document was nothing more than a return character.

One more isolated test would help assure me that the way I want to reference the first word of a paragraph would work. I tried the simple way first:

```
tell of application "TextEdit"
  word 1 of paragraph 1 of document 1
end tell
-- result: "ARTICLE"
```

Things were falling into place. With confidence in my references, I constructed a whose clause to confirm that TextEdit would support the kind of reference I wanted. The dictionary indicated I could reference paragraphs “satisfying a test,” so I was optimistic about support for whose clauses.

First, I made sure that the reference would return the paragraphs I wanted, that is, those whose first word was “article”:

```
tell application "TextEdit"
  paragraphs of document 1 whose first word
  is "article"
end tell
-- result: {"ARTICLE I
", "ARTICLE II
", "ARTICLE III
", "ARTICLE IV
", "ARTICLE V
", "ARTICLE VI
", "ARTICLE VII
", "ARTICLE VIII
", "ARTICLE IX
", "ARTICLE X
"}
```

Success! The whose clauses found the desired paragraphs (each paragraph complete with trailing carriage return).

Now the acid test: could a script change a style characteristic of the text in the document directly by addressing a known property of those paragraph class objects? I gave it a try:

```
tell application "TextEdit"
  set size of paragraphs of document 1 whose
  first word is "article" to 20
end tell
```

It worked! All of the article headings zipped up to 20 point text.



Chapter 13: Debugging Scripts

In an earlier incarnation of a scriptable text editor delivered with the Mac, the kind of reference shown above wouldn't necessarily work if a paragraph was just a carriage return. The application choked on referencing the first word of a paragraph containing no words. This, or something like it, could happen in other scriptable applications, so it's worth mentioning how to deal with the problem.

The solution was to perform an additional test to make sure the paragraph wasn't empty before the script looked for that first word. Using the same script from this section to demonstrate, the first logical attempt might be to put the test after the *whose* clause, as in:

```
tell application "TextEdit"
    set size of paragraphs of document
    1 whose first word is "article" and
    whose first word is not "" to 20
end tell
```

This version would not compile, with the error pointing to the second “whose” word. Aha! I was using multiple **whose/where** keywords when apparently just one is needed, as long as each operand of the Boolean **and** expression is, itself, an expression. I removed the second **whose**, and created something not very grammatical, but worth a try:

```
tell application "TextEdit"
    set size of paragraphs of document
    1 whose first word is "article" and it
    is not "" to 20
end tell
```

This compiled OK, but the runtime error continued. Then I remembered that in expressions joined by Boolean operators (**and** in this case), AppleScript evaluates the *left* operand expression first and applies it; if the execution fails, then the right operand isn't even given a chance. That also meant that if the left operand failed gracefully, then the right wouldn't get caught trying to do something it shouldn't. Therefore, I reversed the order of the Boolean operands:

```
tell application "TextEdit"
    set size of paragraphs of document
    1 whose it is not "" and first word is
    "article" to 20
end tell
```

Equally ungrammatical, but AppleScript-syntactically correct, this version performed the task and got around the reference problems. The script ignored paragraphs (*it*) that were empty, and didn't even try to reference word 1 of such paragraphs.

Part 3: Clean It Up

The last step wasn't so much debugging as finding a more compact and grammatical way to express this filtered statement. I began by moving the reference to **document 1** to the **tell** block. This would simplify additional statements that might get added that also addressed elements of the document:

```
tell document 1 of application "TextEdit"
    set size of paragraphs whose first word is
    "article" to 20
end tell
```



Chapter 13:
**Debugging
Scripts**

I could even add some parentheses to aid readability:

```
tell document 1 of application "TextEdit"  
    set size of (paragraphs whose first word  
        is "article") to 20  
end tell
```

The parentheses help a reader recognize that the arguments for the filtered reference extend across the line break.

Lessons Learned

It may seem like a long road to solve a problem, but this is not atypical of the methodologies you will use when errors come up that you don't readily understand. There were a few important lessons from this demonstration:

1. Object references are number 1.
2. Expression evaluation is also number 1.
3. Things in a new application don't always work the way you expect them to.
4. Isolate problems to their narrowest expressions, and check their contents.
5. Error messages don't always tell the complete story, so you have to sometimes figure out what they're really telling you.

Nothing replaces a well-integrated debugging system for a programming language, but unless you use a more sophisticated editing tool, you'll have to improvise with the kind of techniques shown in this chapter.

Next Stop

In the next chapter, we up the power ante by describing AppleScript subroutines and script libraries.



Chapter 14 Using Subroutines, Handlers, and Script Libraries

While some programmers might say that designing and using subroutines is an art, it is truly a skill any active AppleScript scripter can develop. The more you can think of your scripts in terms of subroutines, the more likely you'll be able to reuse chunks of code from one scripting task to another. Over time, you should be able to build libraries of routines that you can either copy and paste into a script, or load directly into a script as it runs. This chapter tries to help you think in terms of subroutines and libraries.

Subroutines and Scripts

For the most part, your script starts execution at the beginning, and runs through the end in a linear, statement-by-statement fashion. Even conditionals—**if-then** and **repeat** constructions—are part of this linear flow if you take one giant step back and view your script as a single running entity.

But you may well encounter scripts that seem to repeat the same sequence of steps in more than one place. I'm not talking about stuff inside a repeat loop, which you intentionally repeat, but rather a few steps you need several times. For example, perhaps your script retrieves some values from a document or table in several places, and you either must check the type of value it is or must perform some kind of unit conversion (e.g., meters to feet) for your script math to work correctly.

There is no particular “strike you down from the heavens” penalty for repeating this kind of code in a script. Yes, the script will be somewhat larger in file size, because the compiler doesn't have artificial intelligence to see the patterns you've created. But



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

as a scripter, you have genuine intelligence, which allows you to see those patterns. Such patterns are a dead giveaway for the possibility of a subroutine.

In place of repetitive code should be a single statement that calls a subroutine. At its most basic level, such a call would look like any AppleScript command. When script execution reaches this call, execution detours to the subroutine code, which is in the same script. Execution continues through the subroutine until it finishes, at which point execution returns to the main script, resuming with the statement immediately following the subroutine call.

Subroutine Concerns

Designing a subroutine entails two interrelated concerns: the construction of the subroutine and the statement that calls the subroutine. In most cases, a script needs to pass some data to the subroutine, and the subroutine returns some information back to the script. As in the hypothetical example of a unit conversion subroutine, your script needs to send the raw value to the subroutine; the subroutine returns the converted value (or perhaps an error if the value was of the wrong type). But when you think about it, the idea of issuing a command and getting some kind of result back is no different than most AppleScript or application commands. You already know how to handle information coming back from such a call.

You can spot a subroutine in any script, because it begins with the word “on” or “to” and, like any AppleScript compound statement, ends with the word “end” (and, optionally, the name of the subroutine).

What may seem confusing at first, however, is that a script does not execute a subroutine unless the subroutine is called. In other words, the “rule” that a script executes from top to bottom has an exception. Thus, the executing script statements and subroutines can coexist in the same script file with no problem.

Let’s look at a simple subroutine that calculates the circumference of a circle of any diameter:



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

```
on circumference(diameter)
  if {real, integer} contains class of
    diameter then
    return (diameter * pi)
  else
    return ""
  end if
end circumference
```

The name of this subroutine is **circumference**, a good choice, since its name also gives a clue about what it does. The subroutine expects to be passed a value for the diameter to be calculated. In the format shown here (there is another format as well), the parameter is labeled with a plain language variable, **diameter**. Any variable identifier, including old standby “x” could be used. Since this subroutine may be called from any source, including user input, it checks to make sure the parameter is something that can be multiplied (a real or integer number), before any attempt to do so. I’ve also combined the action of calculating the circumference (by multiplying the diameter by the value of pi) with the statement that sends the results back to the calling command. This **return** statement is what does it.

In the script that calls this subroutine would be the command name, along with a value to be calculated, set in parentheses:

```
circumference(12)
-- result: 37.699111843078
```

In practice, the command may be part of a **get** command or other statement requiring the calculated value. Here are some possibilities:

```
get circumference(12)

set circum to circumference(12)

display dialog "The circumference of 12\"
is: " & circumference(12) & "\"."
```

Values passed to subroutines as parameters observe the same kind of expression evaluation behavior as you’ve seen with other commands. The following examples are the same:

```
circumference(12)

circumference(6+6)

circumference(x)
-- where x is a variable containing 12

circumference(text returned of (display dialog
"Enter a number 12:" default answer 12) as
real)
```

You see, it’s just like any command. A subroutine is a way to extend the command vocabulary of a script—and it’s all scripter-definable.



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

Subroutine Scope

One other concern, not mentioned above, affects subroutine calls made inside **tell** statements. As described at length in Chapter 10, the **tell** statement directs all commands nested inside it to the application (or script object) named at the head of the **tell** statement. The problem is that if you call a subroutine inside a **tell** statement, the subroutine call first goes to the application (or script) that is the default object. The application, however, won't have your subroutine's command in its dictionary, and will send back an error saying that the application doesn't understand that command.

To force AppleScript to send the command directly to the current script, rather than to the default object, the subroutine call must refer to the script. The syntax for that is to add the words **of me** after the call or **my** (or **tell me to**) before the call: the script tells AppleScript to send the message to the current script. Here is how this might appear in a TextEdit script that builds a list of circumference values for a range of diameters (assuming the **circumference()** subroutine is also located in this script file):

```
tell application "TextEdit"
    make document at end
    set last paragraph of document 1 to "Dia."
    & tab & "Circumf."
    repeat with i from 1 to 12
        set text of document 1 to text of
        document 1 & return & (i as string) & tab &
        circumference(i) of me
    end repeat
end tell
```

This script builds a new two-column table. The left column, labeled Dia., is filled with the values 1 through 12, using the index value (**i**) of the repeat loop. Tabbed to the right of each diameter value is the circumference value (passing the same index value as a parameter). Notice that we add the words **of me** to direct this command to the current script. Had we not included this direction, we'd get the AppleScript Error message shown in Figure 14.1.

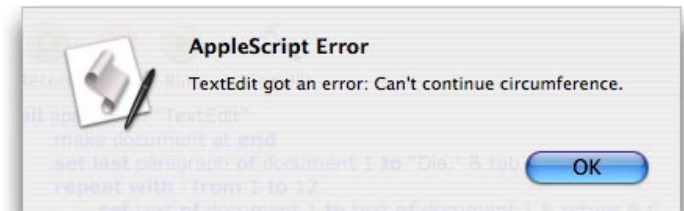


Figure 14.1. Error received without directing a subroutine call to its own script.



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

Subroutine Definitions—Two Types

AppleScript gives us two quite different ways to define a subroutine. The differences are primarily in the ways a subroutine handles parameters passed to it. One version is called a **positional parameter** type; the other is the **labeled parameter** type.

Positional parameters are best for those subroutines that have only one or two parameters or as a way to begin testing a subroutine. Eventually, you should progress to the labeled parameter style, since it reads much better and offers a great deal of flexibility in the calls you make to the subroutine.

We'll start our discussion of subroutine definitions with the positional parameter type, which is the kind we used in the circumference example.

Positional Parameter Subroutines

As its name implies, the positional parameter style of subroutine places great importance on the position, or order, of parameters in a series of parameters. Parameters are placed in a comma-delimited series within a set of parentheses following the subroutine name. The formal syntax is:

```
on | to subroutineName ~
    ([ parameter ] [, parameter ] ...)
        [ global variableID [, variableID ] ... ]
        [ local variableID [, variableID ] ... ]
        [ statement ] ...
end [ subroutineName ]
```

For now, ignore the **global** and **local** variable

parts of the definition. We'll cover them in a separate section, later in this chapter.

Every subroutine begins with either **on** or **to**. Your choice depends solely on which one reads best. Often, the **to** preposition sounds good with a *subroutineName* parameter that is a verb. It's like saying "to do such-and-such, perform the enclosed statements." While all parameters are shown to be optional (and you can have plenty of them), the enclosing parentheses are not. Therefore, even if the subroutine accepts no parameters, the subroutine definition would have an empty set of parentheses, as in:

```
to doSomethingGreat( )
    (* statements that do something great *)
end doSomethingGreat
```

Parameters are variable identifiers that you can use in the enclosed statements of the subroutine. We did this in the circumference subroutine:

```
on circumference(diameter)
    if {real, integer} contains class of
        diameter then
        return (diameter * pi)
    end if
end circumference
```

where "diameter" is used twice for its value.

AppleScript places no limits on the parameter variable. Your subroutine can modify its value and return that modified value, if you like.

Here is a more complex subroutine, which takes a single parameter—the best application of a positional



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

parameter subroutine. This subroutine accepts strings, and removes blank lines between paragraphs:

```
to stripBlanks(textIn)
    set textIn to paragraphs of textIn
    set newText to ""
    repeat with testGraph in textIn
        if (count of testGraph) > 0 then
            if newText is "" then
                set newText to testGraph
            & testGraph
        else
            set newText to newText
            & return & testGraph
        end if
    end repeat
    return newText
end stripBlanks
```

In this subroutine, the first thing we do is convert the incoming string to a list of paragraphs, which will assist in evaluation later down the script. We then initialize a string variable (**newText**), because we'll be concatenating things to it in the repeat loop—we need something there, even if it is blank, to append a string to it.

With the incoming data now in a list, we can use the Repeat In a List form of repeat loop to see if there are any characters in each paragraph. If a paragraph shows any signs of character life, we append it to the **newText** variable (placing a return between paragraphs such that the last paragraph does not end with a return). Paragraphs of length zero are passed over. Finally, the newly assembled string (without

blank lines) is returned to the calling statement.

This example is interesting because it can be called from within a **tell** statement directed at something like TextEdit. Yet, because the subroutine is not in the **tell** statement, it is using AppleScript's text and item parsing, not TextEdit's. In fact, the subroutine doesn't know or care where the source string came from or will go once its blank lines are stripped.

A very important point to remember about positional parameter subroutines is that when you pass multiple parameters, the values you pass must be in the same position as the receiving parameter variables in the subroutine. Here is a subroutine that expects two values:

```
on checkVolumes(startupDisk, volCount)
    set allVolumes to list disks
    if startupDisk is item 1 of allVolumes
        and (count of allVolumes) = volCount then
            return true
        else
            return false
        end if
    end checkVolumes
```

In this case, the first parameter is a string containing a name of a startup disk volume; the second parameter is an integer of the number of volumes expected to be on the Desktop. The calling statement would look something like this:

```
checkVolumes("Macintosh HD", 2)
```

Because the parameters are positional, the "Macintosh HD" string will be assigned to



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

the **startupDisk** parameter variable at the subroutine, just as the integer **2** will be assigned to the **volCount** variable. If you reverse the positions of the parameters in the subroutine call, the wrong values will be assigned to the variables, and you'll have execution errors from here to next Sunday.

Equally important with positional parameters is that all parameters must be specified in a subroutine call, even if their values are to be empty. We've just seen what a valid call would be. If we wanted to pass empty values, we'd have to send empty strings or, for the integer, a zero:

```
checkVolumes("", 2)

checkVolumes("Macintosh HD", 0)

checkVolumes("", 0)
```

None of the following would be accepted:

```
checkVolumes("Macintosh HD",)

checkVolumes(,)

checkVolumes(, 2)
```

In fact, if you try to compile the first two of these last three, the compiler won't even accept them, because it expects an expression after the comma; for the last one, the compiler compiles the parameters to one, dooming the subroutine call to execution error.

Positional Parameter Subroutine Call

We've just been exploring the call that makes a subroutine jump into action. The formal syntax definition is:

```
subroutineName ( [ parameterValue ] ~
                [, parameterValue ] ... )
```

The *subroutineName* must be the same as that of the subroutine you're calling. Case isn't a factor, but spelling is. Subroutine names can be only one word (i.e., no spaces).

As with the subroutine it calls, the call must have a set of parentheses, even if no parameters are being passed. All parameter values must be in a comma-delimited series within the parentheses. Remember that it is the value that gets passed, not the variable name that may be carrying that value. This means that in the subroutine, you can reassign that value to the same or different variable name for use within the subroutine (see below about variable scope of subroutines).

Labeled Parameter Subroutines

While possibly intimidating at first glance, labeled parameter subroutines are truly the way to go for all but the simplest parameter passing. They provide far greater flexibility in design and execution than positional parameter subroutines. They also offer some shortcuts when one or more parameters are Boolean values—things like switches that tell the subroutine what options may be in force each time



Chapter 14:

Using Subroutines, Handlers, and Script Libraries

through the script.

To get the hard part out of the way, here's the formal syntax definition for a labeled parameter subroutine:

```
on | to subroutineName ~
  [ [ of | in ] directParameter ] ~
  [ subroutineParamLabel parameter ] ... ~
  [ given labelName:parameter ~
    [, labelName:parameter ]...
    [ global variableID [, variableID ]...
    [ local variableID [, variableID ]...
    [ statement ] ...
end [ subroutineName ]
```

Again, we'll defer discussion about the **global** and **local** variables to a later section in this chapter.

A *directParameter* is essentially an unlabeled parameter. Like all parameters in this definition, it is a variable identifier which receives the value passed to it by the calling routine. While the **of** and **in** words are optional (and help readability), the only other restriction about the *directParameter* is that it must be the first parameter after the *subroutineName*. If you think about the **display dialog** command, it has a similar setup, with a direct parameter (the string that goes into the dialog message) plus a bunch of optional, labeled parameters:

```
display dialog ~
  "Here's the direct parameter string."
```

The next parameter grouping in this definition (*subroutineParamLabel* parameter) is a bit complex

at first. A *subroutineParamLabel* is one of 24 predefined labels (all in the form of prepositional phrases). You must use one prepositional label per parameter and only one of each type per subroutine. Here is the list of labels:

about	for
above	from
against	instead of
apart from	into
around	on
aside from	onto
at	out of
below	over
beneath	since
beside	through
between	thru
by	under

The purpose of all these prepositional phrases is to let you make readable, meaningful commands to your subroutines. For example, this is how we could make the **checkVolumes()** subroutine example from earlier in this chapter more understandable by utilizing two of these special parameter labels:

```
to checkVolumes against startupDisk for
  volCount
  set allVolumes to list disks
  if startupDisk is item 1 of allVolumes
  and (count of allVolumes) = volCount then
    return true
  else
    return false
  end if
end checkVolumes
```

At the calling end, the statement must also name



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

those labels:

```
checkVolumes against "Macintosh HD" for 2
```

As with other labeled parameters you've seen with various commands, labeled subroutine parameters can be placed in any order. It's the labels that tell AppleScript what values to assign to which parameters in the subroutine. Therefore, the call to the **checkVolumes** subroutine could also be:

```
checkVolumes for 2 against "Macintosh HD"
```

In both cases, the "Macintosh HD" string will be assigned to the **startupDisk** variable in the subroutine, because the "against" label makes that connection; similarly for the "2" integer and the **volCount** variable attached to the "for" label. Because each subroutine's parameters and their interrelationships present a unique set, you can use these labels to make those relationships clearer to the user than just a bland list of values (as with positional parameters).

But if those predefined labels aren't sufficient for your needs, you can add your own labels. The **given** keyword must precede one or more custom labels and their variable identifiers. Here's yet another version of the **checkVolumes** subroutine, this time demonstrating how a custom parameter label may be best of all for this situation:

```
to checkVolumes against startupDisk given  
volumeCount:volCount  
set allVolumes to list disks  
if startupDisk is item 1 of allVolumes
```

```
and (count of allVolumes) = volCount then  
return true  
else  
return false  
end if  
end checkVolumes
```

This definition specifies a custom label, **volumeCount**, but the same **volCount** variable identifier as the other forms. The call to this style would look like this:

```
checkVolumes against -  
"Macintosh HD" given volumeCount:2
```

Definitions and calls allow for multiple custom labels after the **given**, each one separated by a comma. For example:

```
to checkVolumes given disk:startupDisk, -  
volumeCount:volCount  
...  
end checkVolumes
```

which would be called by:

```
checkVolumes given disk:"Macintosh HD", -  
volumeCount:2
```

As you can see, it's possible to create any mixture of direct, predefined labeled, and custom labeled parameters in a subroutine definition. Use whatever makes the most grammatical sense to you. Don't be afraid to use something awkward while under construction and then change it later on. Just be sure to make the same change to the statement that calls the subroutine.



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

Caution: If you are creating a labeled parameter subroutine that has only one parameter, AppleScript requires that the preposition for that single parameter be anything except **of**. This is due to the ambiguity caused by object containment syntax, which makes liberal use of this preposition.

Labeled Parameter Subroutine Call

We've already seen a variety of labeled parameter subroutine calls in the previous section. But such calls have further powers to simplify the passing of parameters consisting of Boolean values. To understand this, let's look at the formal definition of these calls:

```
subroutineName [ [ of | in ] ~
    directParameter ] ~
    [ subroutineParamLabel parameterValue ] ... ~
    [ given labelName:parameter ~
    [, labelName:parameter ]... ] ~
    [ with labelForTrueParam ~
    [, labelForTrueParam, ... and ~
    labelForTrueParam ] ] ~
    [ without labelForFalseParam ~
    [, labelForFalseParam , ... and ~
    labelForFalseParam ] ]
```

Everything is the same through the **given** types of parameters. This means that this form adheres to the same predefined labels as well as allowing custom labels to match the ones in the subroutine definition shown earlier in the chapter.

But if the subroutine has one or more labeled parameters whose values are Booleans (**true** or **false**), then you can pass those Boolean values by simply listing the parameter labels in the **with** and **without** clauses. Any label following the **with** preposition is passed a **true** value from the calling statement; any label following the **without** preposition is passed a **false** value.

As an example, I've created a subroutine that checks for the class of any value. To take data into the subroutine, it has one direct parameter (the value to be tested) and four labeled parameters, each representing a class, and each requiring a Boolean value. The subroutine uses these labeled parameters as switches to know which test(s) to perform on the value being passed in. Here is the subroutine:

```
on classCheck of valueIn given stringClass:
    testString, numberClasses: testNumbers,
    listClass:testList, recordClass:testRecord
    if testString then
        set stringResult to (class of
valueIn is string)
    else
        set stringResult to true
    end if
    if testNumbers then
        set numbersResult to ({real,
integer} contains class of valueIn)
    else
        set numbersResult to true
    end if
    if testList then
        set listResult to (class of valueIn
is list)
```



Chapter 14:
**Using
 Subroutines,
 Handlers,
 and Script
 Libraries**

```

else
    set listResult to true
end if
if testRecord then
    set recordResult to (class of
valueIn is record)
else
    set recordResult to true
end if
return stringResult and numbersResult and
listResult and recordResult
end classCheck

```

As the subroutine runs, it performs a test of the value against any of the four classes whose labeled parameter comes in as **true**. Each test, itself, returns a **false** value only if the test is requested and it fails. The final Boolean expression (in the **return** statement) calculates the combined Boolean values of all four possible tests; if any one fails, a **false** value returns to the calling statement.

The real demonstration point of this subroutine is the way it's called. The long-winded way would be something like this:

```

classCheck of 100 given stringClass:true, ~
    numberClasses:true, listClass:false, ~
    recordClass:false

```

But we can pass those **true**s and **false**s by incorporating them into **with** and **without** clauses:

```

classCheck of 100 with stringClass and ~
    numberClasses without listClass ~
    and recordClass

```

The subroutine assigns **true** to **testString**

and **testNumbers**, while assigning **false** to **testList** and **testRecord**. It's a kind of shorthand in the calling statement that addresses just the labels. While the form makes the statement appear in more natural language, it does force the script reader to mentally perform the Boolean assignments.

To include more than two parameters with the **with** or **without** clauses, string together all but the last two with comma delimiters, followed by the **and** operator for the last one (with no comma after the next-to-last item). For example, to call the **classCheck** subroutine to test for only one class, the call would be:

```

classCheck of 100 with numberClasses ~
    without listClass, stringClass ~
    and recordClass

```

One final point about labeled subroutine calls: while the order of the parameters (except for the direct parameter) can be willy-nilly, every parameter defined in the subroutine must be in the call as well. Failure to have a label-for-label match results in an execution error.



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

Subroutine Parameters—By Value and By Reference

When you pass a value to a subroutine, the impact that the subroutine has on the value varies somewhat with the class of the passed value. The rules are pretty simple, however. Whenever the value is a list, record, or other type of object class, statements in the subroutine work on the original object that otherwise exists outside of the subroutine. Experienced programmers would call this *passing a parameter by reference*. For all other value types (including properties of records or objects), the subroutine does not directly impact the value that exists outside of the subroutine. Experienced programmers call this *passing a parameter by value*.

As often happens in AppleScript, a couple of simple examples speak thousands of words. To start, we'll devise a subroutine that receives a list value and modifies one of the items in it. Then we'll assign a list to a variable, pass that variable to the subroutine, and see what happens to the original list:

```
on tweakList(theList)
    set item 2 of theList to 5
end tweakList

set x to {1, 2, 3}
tweakList(x) -- call to subroutine
x
-- result: {1, 5, 3}
```

In other words, the subroutine received the parameter in the form of a reference to the list stored

in x. Manipulation of list items occurred even from within the subroutine.

Next, we'll rewrite the script so that the subroutine receives some value as a parameter, and changes that value within the subroutine (without returning any value):

```
on tweakList(theVal)
    set theVal to 5
end tweakList

set x to {1, 2, 3}
tweakList(item 2 of x) -- call to subroutine
x
-- result: {1, 2, 3}
```

The difference here is that by passing an item of the list as the parameter, the script essentially disconnects the passed value from its original source: the list. Changes made to the passed value occur independently of the “life” of the list outside of the subroutine, and the original list remains untouched.



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

Subroutine Variables

In the formal definitions for both types of subroutines, you saw references to parameters labeled as **global** and **local** variables. The reason they are mentioned in subroutine definitions is that variable behavior becomes somewhat skewed by subroutines.

Until now, we were comfortable with the fact that a local variable defined anywhere in a script made that variable available to any statement below it in the script. As it turns out, subroutines are like little protected enclaves within a script: unless directed otherwise, a variable defined outside a subroutine is not alive inside the subroutine; conversely, any variable defined in a subroutine dies after the subroutine returns. Let's look at some examples to explain:

First, we'll define a variable as a string outside a subroutine and assign an integer to a variable with the same name inside the subroutine:

```
set myValue to "Gumby"
on varTest ( )
    set myValue to 25
    return myValue
end varTest
varTest ( )
-- result: 25
myValue
-- result: "Gumby"
```

We reused a variable (**myValue**) name inside a subroutine. If it weren't for the subroutine,

reassigning the variable to an integer would have simply replaced the string with the integer. But since the integer assignment was inside the subroutine, the outer variable remains untouched.

To make the same variable apply to both inside and outside the subroutine, we use the **global** statement inside the subroutine:

```
set myValue to "Gumby"
on varTest ( )
    global myValue
    set myValue to 25
    return myValue
end varTest
varTest ( )
-- result: 25
myValue
-- result: 25
```

The **global** statement tells the subroutine to use the **myValue** variable from outside (if it's there). Therefore, in this construction, the **set** command inside the **varTest ()** subroutine actually replaces the string value with the integer. After the subroutine ends, the global-ness of the variable persists: there is only one **myValue** variable in this entire script. If the variable had not been declared prior to the subroutine, it would have been initialized within the subroutine and its existence and value would extend after the subroutine.

As you saw, there was no need in the first example to declare a local variable within the subroutine: its default behavior made any new variable a local variable. But that changes when a **script property** has



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

been defined in the same script. A script property is like an object's property: a value that is stored with the script object (see Chapter 15 for more about script objects and their properties). The point is, however, that a script property is global in scope by its nature. For example:

```
property myValue:"Gumby"
on varTest ( )
    set myValue to 25
    return myValue
end varTest
varTest ( )
-- result: 25
myValue
-- result: 25
```

In other words, the global-ness of a property pervades even subroutines. Here, the **set** command in the **varTest ()** subroutine is actually adjusting the value of the script property, changing it from a string to an integer.

To prevent this while using the same variable identifier as a property (probably not a good idea, anyway), you must declare a local variable in the subroutine:

```
property myValue:"Gumby"
on varTest ( )
    local myValue
    set myValue to 25
    return myValue
end varTest
varTest ( )
-- result: 25
myValue
-- result: "Gumby"
```

The **local** declaration keeps the property of the same name out of the subroutine altogether. Once the subroutine ends, the local variable and its value are discarded, leaving the field open for the property to reign once again.



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

Recursion

AppleScript allows subroutines to call themselves: a process known as *recursion*. The legendary factorial calculation offers a good example:

```
on factorial(n)
    if n > 0 then
        return n * (factorial(n - 1))
    else
        return 1
    end if
end factorial
```

As AppleScript performs this subroutine round and round, it essentially nests the intermediate values for evaluation after the last time through. Be sure to test recursive subroutines carefully, because you may overpower the limits of memory (specifically, the stack that tracks all the pending operations) if you're not careful.

Turning Subroutines into Libraries

Everything we've said about subroutines up to this point implied that subroutines exist in the same scripts as the statements that call them. There's nothing wrong with this, except that once you've defined some generic subroutines, there are more efficient ways to get their powers into multiple scripts without copying and pasting them into those scripts. You can make a subroutine (or better yet, a collection of related subroutines) into a **script library**.

A script library is nothing more than a subroutine that has been saved as a compiled script (either editable or run-only). To bring the power of a script library into another script, you use the **load script** command. Once loaded, your script talks to the library as if it were another application (via a **tell** statement) and as if its handlers were commands belonging to that application.

In the Chapter 14 folder of the companion scripts collection is a library of subroutines for performing some paragraph sorting and blank line stripping from big blocks of text. The four subroutines are:

```
on sortLines of textIn given sortOrder:s
```

sorts lines (paragraphs) of text in either "ascending" or "descending" order.

```
to stripBlanks(textIn)
```

strips away blank lines between paragraphs. It can be used by itself, and is called by sortLines in its work.



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

```
on classCheck of valueIn given ~  
    stringClass:testString, ~  
    numberClasses:testNumbers, ~  
    listClass:testList, recordClass:testRecord
```

tests a value against a series of classes specified as arguments—called by other routines in the library.

```
on listToParagraph of newLines given ~  
    sortOrder:s
```

reconverts lists of paragraphs to return-delimited text, according to “ascending” or “descending” sorting order—called by other routines in the library.

This library is saved as “paragraphLib.” Most libraries have “Lib” (rhymes with jibe) at the ends of their file names to help us distinguish a library from an ordinary script.

A library consists strictly of subroutines, and perhaps some properties. There is no other executing script in a library, because it doesn’t run like a script. Other scripts call the subroutines into action (which may call other subroutines in the library), but there are no other freestanding statements to execute.

Storing Libraries

Since a script must load a library by its pathname, things can get pretty tricky if you plan to hand over the script and library to other users. Your script can’t possibly know the eventual pathname to files. While your script could ask the user to locate the library file the first time (and store the path as a property), there is another method that works correctly the first time: insist that the library file be stored in the *Scripting Additions* folder (even though the paths differ for Mac OS 9 and X). Your library will share the company of any scripting additions on that user’s Macintosh. In fact, copy *paragraphLib.spt* from the *Handbook Scripts/Chapter 14* folder to your Scripting Additions folder now to experiment with the following scripts.

The benefit here is that your script can use the **path to** command to determine the path to the *Scripting Additions* folder. It’s a simple task to append the rest of the path to your library. For example, to get the path to *paragraphLib.spt* in *Scripting Additions*, use the following statements:

```
get (path to scripting additions as string) ~  
    & "paragraphLib.spt"  
set graphLib to load script file result
```

or combine the two statements as:

```
set graphLib to load script file ~  
    ((path to scripting additions as string) ~  
    & "paragraphLib.spt")
```

Loading the script object assigns the script object to



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

the variable **graphLib**. Your script can now grab some text and have the library sort it. Try it yourself with the script below. First load a list of months, numbers, whatever into the front-most document of **TextEdit**. Then enter and run the following script:

```
get (path to scripting additions as string) &
    "paragraphLib.spt"
set graphLib to load script file result
set theText to text of (document 1 of
    application "TextEdit")
tell graphLib
    get sortLines of theText given
        sortOrder:"ascending"
end tell
-- result: (sorted string)
```

By loading that script as a script object, we've put the power of dozens of lines of code into this script without the enormous overhead those lines would add to this script. In the **tell** statement, we talk to the script object and send it a command as if it were a full fledged scriptable application. In fact, if the library had properties, we could get and set its properties just as we do for an application's object.

Open the *paragraphLib.spt* script with Script Editor, and study it for the many examples of subroutine definitions and calls. You'll find a smorgasbord of subroutine and parameter styles, each one tailored to the kind of data and calls it expects.

Handlers in Attachable Applications

Although we don't have many AppleScript-attachable applications, the Finder is attachable to some extent. AppleScript provides a mechanism that allows you to write handlers that become inextricably connected with the application. For example, let's say that an application allowed you to attach a script to a document object. You could write handlers for that object that respond to commands that object receives from other scripts or other objects in the application.

The formal syntax of this kind of handler is nearly identical to that of a labeled parameter subroutine, with the exception of the predefined labels:

```
on | to commandName [ [ of ] ~
    directParameter ] ~
    [ [ given ] paramLabel: parameter ~
        [, paramLabel: parameter ]... ]
        [ global variableID [, variableID ]... ]
        [ local variableID [, variableID ]... ]
        [ statement ] ...
end [ commandName ]
```

Command handlers can be written to respond not only to user-defined commands, but also to the application's commands. In other words, if you want the object to perform some special processing in lieu of a regular command, the command handler would trap the command before it reaches the application dictionary.

In the case of attachable folders in the Mac OS X Finder, attachability is facilitated by a series of



Chapter 14:
**Using
Subroutines,
Handlers,
and Script
Libraries**

five AppleScript commands that the system send whenever one of the corresponding five user actions occurs:

- ▶ Adding items to a folder
- ▶ Removing items from a folder
- ▶ Opening a folder window
- ▶ Closing a folder window
- ▶ Moving the screen position of a folder window

In other words, if you want to use any of these actions to trigger a script, you write a handler that responds to the desired command. You can see an example of this usage in Chapter 7's discussion of the Folder Action Commands.

If the syntax of an attachable command handler looks like that of an **on error** handler described in Chapter 12, you're correct. The error handler in a **try** statement is merely a pre-defined variation of this labeled parameter subroutine. In the error handler's case, the labels (e.g., **number**, **from**, **to**, **partial result**) are predefined. You can assign global and local variables in exactly the same way as you may need for any subroutine. The same rules apply about the lack of variable persistence into and out of an error handler as for any subroutine.

Next Stop

In the next chapter, we take the final leap, creating script objects that can become droplet applications and agents.



Chapter 15 Script Properties, Objects, and "Agents"

This chapter assumes that you have a good grasp of all that has come before. If this book were used in an AppleScript classroom course, this chapter would be part of the advanced topics section. While the material may be complex, the rewards for working your way through can be high. We start with a detailed look at script properties, which some might regard as super variables. Then we look at script objects, which build on your exposure to subroutines. Finally, we see how we can make script objects into programs once commonly called “agents”—little helpers running in the background, to alert us to things happening behind the scenes.

Script Properties

While the subject of script properties could have been covered in Chapter 9’s discussion of variables, I feel it makes more sense here, because properties are even more useful in script objects, which we’ll cover later in this chapter. Properties are also useful in subroutines.

An application’s objects, as AppleScript dictionaries reveal, have properties, whose values we can view (and sometimes modify). These were the properties like **artist** of an iTunes music track or the **color** of a paragraph’s text. Pretty straightforward AppleScript stuff.

But you can also assign one or more properties to a script. The formal syntax for doing this is:

```
property | prop propertyLabel:initialValue
```

PropertyLabel is the name for the property (just as “bounds” is the name of one window property), whereas *initialValue* is the value the script assigns to the property when the script is initialized (more on that in a moment). This property statement can



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

go anywhere in a script, and the property label and its value may be accessed in any statement after this declaration. By convention, properties are usually initialized at the top of a script because it provides some helpful guidance to someone reading the script.

With one exception (discussed later), property declarations are made at the outermost level of a script (i.e., not nested in any **tell** statement or subroutine). In this form, a property behaves like an ultra-global variable. We saw evidence of this in the last chapter, where a global property's value was available and modifiable inside even the normally exclusive domain of the subroutine. Recall this example:

```
property myValue:"Gumby"
on varTest( )
    set myValue to 25
    return myValue
end varTest
varTest( )
-- result: 25
myValue
-- result: 25
```

While the property, **myValue**, was initially set to a string value, it was modified within the subroutine and stayed that way after the subroutine ended.

Property "Persistence" in Compiled Scripts

Notice that I said an initial value is assigned to a property when the script containing the declaration initializes. That implies that the value can change, which it can. But how long the property maintains its

modified value, and under what circumstances the value persists from session to session is a squirrely subject. Let's take it one step at a time.

- 1) Start by creating a demonstration script that has a property named **beanCount**, whose goal is to maintain a count of the number of times the script runs:

```
property beanCount:0
set beanCount to beanCount + 1
display dialog "Since the last
initialization, this script has run " &
beanCount & " time(s)."
```

When you first compile this script, it sets the value of **beanCount** to zero. The first time you run the script, the value of **beanCount** increases by 1, and shows itself that way in the dialog. Click the Run button again, and the value increases to 2.

- 2) Add a blank line to the end of the script and click the Compile button to re-compile the script. Run the script, and you see the value has reset itself. This is good in a way because it means each time you modify a script, it resets property values, so you can begin testing from the same point each time.
- 3) Run the script until the dialog says you've run it 5 times.
- 4) Save the script as a compiled script in the *[user]/Library/Scripts* folder so that you will be able to run the script from the Script menu of the Mac menubar. Save the script with the name "Bean Counter.scpt". *Saving the script saves the current*



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

value of the property with it.

- 5) Close the script window.
- 6) Now run the script from the Scripts menu. The dialog indicates that the counter picked up where you had left off. Run it a few times until the counter reaches 8.
- 7) Re-open the Bean Counter script in Script Editor (find it quickly via the **Open Recent** choice of the **File** menu).
- 8) Do not make any changes to the script, but click the Run button. The message indicates the counter has, once again, restarted, displaying a value of 1. *Opening the script in Script Editor re-initializes the property in the copy running in Script Editor.*
- 9) Now run the saved version through the Script menu once more. The counter should now read 9. The instance of the script open in Script Editor is separate from the one saved in the Scripts Folder.
- 10) Close the Script Editor version without saving, and run the saved version again via the Script menu. It continues counting upward from 9.
- 11) Re-open the *Bean Counter.scpt* file once more in Script Editor, and run it twice until the counter reads 2.
- 12) Choose **Save** from the **File** menu.
- 13) Close the script window.

- 14) Finally, run the saved script via the Script menu. Because the edited version overwrote the previous saved version, the counter should continue upward from the last count it had, 2.

You can see from this experience that properties have different persistence patterns when the script is run from Script Editor or run by way of the Script menu (or other invocation, such as from a Mac OS X cron job or shell script via the **osascript** command, described in Chapter 7). Scripts that rely on the persistence of property values should be invoked through external triggers, rather than loading them into Script Editor for running.

A related technique allows you to run a property-laden script the first time in Script Editor to establish initial values that the script will use when invoked later by the Scripts menu or shell script call. After you create and test the script, save it in the location from which it will ultimately be invoked. Provide a prefatory section of the script that tests for the existence of the property value. If the value is missing the script can ask the user to supply the value, and then saves the script, thus preserving the initialization value for the next time the script is invoked.

The following is a segment that asks the user to provide a folder path name so that the script can operate at some later time, unattended (invoked by a shell script):



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

```
-- store the reference to the folder
property folderPath : ""
if folderPath is "" then
    set folderPath to (choose folder
with prompt "Locate the "data"
Folder:") as string
    tell application "Script Editor"
        save document 1
    end tell
    display dialog "Folder path
saved." buttons {"OK"}
    error number -128
end if
```

After the alert advises the scripter that the path has been saved, the **error** command causes the script to exit before executing the statements that are to be run when invoked externally.

Property "Persistence" in Script Applications

A script saved as an application behaves the same way with respect to script property persistence. As long as you continue to invoke the script application externally, such as by double-clicking its icon or choosing it from the Script menu, property values will persist from session to session. Opening up the application in Script Editor causes the copy running in Script Editor to reinitialize the property values. Closing the Script Editor version without saving preserves the original settings; saving the Script Editor version overwrites previous settings in the script application version.

A global variable's value also persists from one script execution to the next:

```
global x
```

Unlike a property definition, however, a global variable declaration does not let you initialize the variable with a starting value. That must be done with another statement elsewhere in the script.

Another method of achieving data persistence is to save property data in a text document. You can use the file-related scripting addition commands (Chapter 7) to open, write, read, and close a data file which you update with property values as needed. Each time the script runs, it reads the current value in the file to set the property value.

We'll come back to properties again once we cover the basics of script objects.



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

Script Objects

You aren't limited to using objects defined in applications or AppleScript, itself. You can create your own objects—objects whose names, properties, and execution behaviors are entirely under your control. You may design script a object to be part of a larger script or as a script unto itself.

Unless you've worked with object-oriented systems before, the concept of script objects may be hard to grasp at first. Often, the hard part is making the jump from a series of AppleScript statements to a tangible object. In truth, script objects are artificial entities. In some ways they remind me of make believe friends that children concoct in their imaginations.

Let's create one of these imaginary friends. His name Mac. His behavior is limited to responding to one command: telling us the version of iTunes running on the machine. Mac is smart enough to take in the name of the person who is asking, and make that name part of the response. Mac also has a property, which tracks the name of the last person to ask for iTunes version information. Here is the definition of Mac as a script object:

```
script Mac
    property lastPerson : ""
    to sayiTunesVersion to someone
        set lastPerson to someone
        get short version of (info for file
            ((path to applications folder) as string) &
            "iTunes.app"))
        return someone & ", I am currently
            running " & "iTunes version " & result & "."
    end script
```

```
end sayiTunesVersion
end script
```

To get Mac to respond, we must send a command it understands (**sayiTunesVersion**) with whatever parameter it expects (someone's name). In the same script, for instance, we can issue this command:

```
tell Mac to sayiTunesVersion to "Patrick"
(* result: "Patrick, I am currently running
iTunes version 4.7." *)
```

Notice the use of the **tell** statement. This is just like directing a command to an application. We can use the syntax talking directly to the script object by name when the object and calling statement are in the same script (we'll see what differences are required if the script object is in a separate script object file later).

Mac also has the **lastPerson** property, which is assigned the value passed along as a parameter. To access this property, we use the same kind of reference as a property of an application object:

```
get lastPerson of Mac
-- result: "Patrick"
-- (after running the above call)
```

We can then combine accesses to commands and properties in a script that talks to Mac:



Chapter 15: Script Properties, Objects, and "Agents"

```
display dialog "Please enter your name:"
  default answer ""
if text returned of result ≠ "" then
  set someone to text returned of result
  if lastPerson of Mac = someone then
    display dialog "What, you again?"
  else
    tell Mac to sayiTunesVersion to
  someone
  display dialog result
end if
end if
```

The formal syntax description of a script object is as follows:

```
script [ scriptObjectVariable ]
  [ property | prop ~
  propertyLabel:initialValue ] ...
  [ handlerDefinition ] ...
  [ statement ] ...
end [ script ]
```

The ellipses (...) after each type of placeholder indicates that you can define any number of those kinds of items. Therefore, you can have multiple properties and/or handler definitions in a script object. In the example about Mac, we've seen how all these elements work, except for the statement placeholder. For that, we should talk more about running script objects.

Running Script Objects

To best understand the variety of responses to the **run script** command (Chapter 7), start with the following script object, and save it to your Desktop

as a compiled script named *Mac.scpt* (or copy it from *Handbook Scripts/Chapter 15* to the Desktop):

```
script Mac
  property lastPerson : ""
  to sayiTunesVersion to someone
    set lastPerson to someone
    get short version of (info for file
      ((path to applications folder) as string) &
      "iTunes.app"))
    return someone & ", I am currently
  running " & "iTunes version " & result & "."
  end sayiTunesVersion
  -- next line added for demonstration
  display dialog "Mac" just ran."
end script
display dialog "Please enter your name:"
  default answer ""
if text returned of result ≠ "" then
  set someone to text returned of result
  if lastPerson of Mac = someone then
    display dialog "What, you again?"
  else
    tell Mac to sayiTunesVersion to
  someone
  display dialog result
end if
end if
```

When you run this script manually in Script Editor, AppleScript is, essentially, sending the **run** command to this script. A script's default behavior in such cases is to execute every executable line in the script. In this case, nothing in the upper script object definition runs until it is called by the statements in the lower group. Since only the **sayiTunesVersion** handler is called, the



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

freestanding statement (**display dialog** "**Mac just ran.**") does not execute when the **run** command is sent to the script as a whole.

Now open a new, small Script Editor window, and enter the following script, which sends the **run** command to the script file:

```
run script file (((path to Desktop) ~  
as string) & "Mac.scpt")
```

When you run this script, it sends a single **run** command to the script file (which can be open or closed). This is just like clicking the **run** button for the Mac script in Script Editor.

If you run this little script multiple times and enter the same name each time, you'll notice something else that proves a point about properties: the value of the **lastPerson** property doesn't persist. That's because each **run** command initializes the script object—an action that automatically resets properties defined in a compiled script. Of course, you may design a script that just performs something on its own, and you don't have to worry about updating properties. In such a case, this **run script** construction suits you fine.

Load and Run

Now modify the small script to perform the following:

```
repeat 3 times  
    run script file (((path to Desktop) as  
string) & "Mac.scpt")  
end repeat
```

Even if you enter the same name each of the three times, Mac doesn't remember the last person's name, because, as before, the **run script** command keeps initializing the object each time through the repeat loop. If, however, you want to let the script run while keeping the **lastPerson** variable up to date, you must first load the script into a variable, and then work with the script object:

```
set MacScript to load script file (((path to  
Desktop) as string) & "Mac.scpt")  
repeat 3 times  
    run MacScript  
end repeat
```

When you run this script and enter the same name each time, you get the "What, you again?" alert after the second attempt, because the **run** command (instead of a **run script** command) is sent repeatedly to an open script object. Only when you run this smaller script again, which re-loads a copy of the saved Mac script object, does the **lastPerson** value start from empty again.

In case you want the script object to remember the last value of the property, you can summon the powers of the **store script** scripting addition.



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

The command, as described in Chapter 7, requires a path name to the stored file. Here's one way to do it:

```
set scriptPath to (((path to Desktop) as  
string) & "Mac.scpt")  
set MacScript to load script file scriptPath  
repeat 3 times  
    run MacScript  
end repeat  
store script MacScript in scriptPath replacing  
yes
```

The last command stores the current state of the script object as modified during the final time through the repeat loop, replacing the original version that contained the empty property. Thus, when you run the script a second time and enter the same name for the first request as the last request of the previous run, the script recognizes the name as a repeat requester.

Now, what about the statement in the Mac script object definition that's supposed to display a dialog that says it's running? This line (and any other statements not part of a handler definition within the script object) executes only when the object—not the script—receives a **run** command. Therefore, to see that line execute, we'd have to modify our calling script as follows:

```
set MacScript to load script file -  
    (((path to Desktop) as string) & "Mac.scpt")  
tell Mac of MacScript to run
```

In other words, we first load a copy of the entire script into a variable (**MacScript**), and then direct

the **run** command to the script object, Mac. None of the handlers in Mac execute, because they haven't been called. Nor do the statements below the script object execute, because the **run** command was directed to the Mac object, not the entire script.

If your head is still swimming, re-read this section and try the examples yourself. It does make sense in light of AppleScript's manner of directing commands to specific objects.



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

Advanced Object-Oriented Techniques

AppleScript provides facilities for inheritance and delegation among script objects. Using some of the script objects in earlier examples, we'll explore these concepts.

Inheritance

A script object may inherit the properties and commands of another script by establishing a special *parent* property. A parent property, whose formal syntax is:

```
property | prop ~
    parent : scriptObjectVariable
```

tells AppleScript that a script object shares everything that the parent has. A script object containing a **parent** property is called a *child*. A child can have only one parent, whereas a parent may have any number of children (the parent has no designation in its script signifying that it is a parent or who its children are—all the work stems from the child's declaration of a **parent** property).

If we start with our Mac script object:

```
script Mac
    property lastPerson : ""
    to sayiTunesVersion to someone
        set lastPerson to someone
        get short version of (info for file
            ((path to applications folder) as string) &
            "iTunes.app"))
        return someone & ", I am currently
```

```
    running " & "iTunes version " & result & "."
    end sayiTunesVersion
end script
```

we can create another object in the same script file that is a child of **Mac**. We'll call that script object **myComputer**. It looks like this:

```
script myComputer
    property parent:Mac
end script
```

Any command or property request we send to the **myComputer** object is carried out by the **Mac** object. It's as if the **myComputer** object were:

```
script myComputer
    property lastPerson : ""
    to sayiTunesVersion to someone
        set lastPerson to someone
        get short version of (info for file
            ((path to applications folder) as string) &
            "iTunes.app"))
        return someone & ", I am currently
    running " & "iTunes version " & result & "."
    end sayiTunesVersion
    display dialog "Mac just ran."
end script
```

Therefore, if the calling statement reads:

```
tell myComputer to sayiTunesVersion to "Joe"
    (* result: "Joe, I am currently running
        iTunes version 4.7." *)
```

the reply comes from the handler in **Mac**, because **myComputer** hands it off to its parent.

The advantage to this mechanism is that the **myComputer** object can now have some of its



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

own properties and handlers in addition to those of the **Mac** object, but **myComputer** doesn't have to replicate any of **Mac**'s items. By declaring the **parent** property, the child has added all that functionality to itself.

It also means that if the child needs a slight variation of a handler or property that is contained in the parent, the child's script can intercept the command before it reaches the parent. For example:

```
script myComputer
  property parent : Mac
  to sayiTunesVersion to someone
    get short version of (info for file
      ((path to applications folder) as string) &
      "iTunes.app"))
    return "Hey, bub, it's " & result
  & "."
  end sayiTunesVersion
end script
```

With this version, if we say:

```
tell myComputer to sayiTunesVersion to "Joe"
-- result: "Hey, bub, 4.7."
```

the command is caught by the handler in **myComputer**.

Delegation

A child can not only intercept commands that would otherwise make their way to the parent, but the child can trap a command under some circumstances, and let it pass to the parent under others. For this to work, the child's version of the handler must contain a **continue** statement.

The formal definition of the continue statement is:

```
continue  commandName  parameterList
```

where the parameters are the command and parameters required by the parent command's handler. All parameters must be in the same form as the handler (i.e., matching positional parameters or labeled parameters as the case may be). We can look at an example from the same **Mac/myComputer** parent/child relationship from above. In this case, we modify the child so that it delegates the **sayiTunesVersion** command to the parent if the name of the user is Steve (after all, it could be Steve Jobs or Steve Wozniak, and we want the script objects to be polite); anyone else gets the rude version from the child:



Chapter 15:

Script Properties, Objects, and "Agents"

```
script myComputer
  property parent : Mac
  to sayiTunesVersion to someone
    if {"Steve", "Steven"} contains
someone then
      continue sayiTunesVersion to
someone
    else
      get short version of (info
for file ((path to applications folder) as
string) & "iTunes.app"))
      return "Hey, bub, it's " &
result & "."
    end if
  end sayiTunesVersion
end script
```

The call then goes out:

```
tell myComputer to sayiTunesVersion to "Joe"
-- result: "Hey, bub, 4.7."
```

or

```
tell myComputer to sayiTunesVersion to "Steve"
(* result: "Steve, I am currently running
iTunes version 4.7." *)
```

Notice that we didn't have to make any changes to the **Mac** parent script object. It's all in the child. We can even be more subversive by slightly altering the parameters delegated to the parent:

```
script myComputer
  property parent : Mac
  to sayiTunesVersion to someone
    if {"Steve", "Steven"} contains
someone then
      continue sayiTunesVersion to
someone & ", sir"
    else
      get short version of (info
for file ((path to applications folder) as
string) & "iTunes.app"))
      return "Hey, bub, it's " &
result & "."
    end if
  end sayiTunesVersion
end script
```

Instead of continuing with only the name "Steve" or "Steven," we pass along an added note of respect to the **Mac** object's handling of the command via the **continue** command. Therefore:

```
tell myComputer to sayiTunesVersion to "Steve"
(* result: "Steve, sir, I am currently
running iTunes version 4.7." *)
```



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

Creating Droplets

A droplet is an informal term for any application that runs when you drag and drop one or more Finder items (files and folders) onto the application's own icon. In some ways, a droplet behaves like an agent (described more fully below), because by dropping an item onto your script application, you essentially hand off control to the script application, and let it do whatever it is programmed to do.

What makes a script application (i.e., a script saved as an application) a droplet is an **on open** handler in the script. The act of dropping a Finder item on a droplet causes the Macintosh to send an **open** message to the droplet. It is incumbent upon your application to have an **on open** handler that traps the **open** message.

When Script Editor saves a script as an application, the icon it assigns to the script is shown in Figure 15-1(left). But if Script Editor detects an **on open** handler in the script, it assigns a different icon, as shown in Figure 15-1(right). The difference is in the down arrow on the icon, indicating that the application is designed as a droplet. Of course, it's up to your script to do something with the information that comes with the **open** message (a list of files or folders).



Figure 15-1. Regular script application icon (left) and droplet icon (right).

To demonstrate, below is the script listing for the application *Show Creator & Type* (see next page). The script application file is in the *Handbook Scripts/Chapter 15* folder. This application lets you drag any number of files or folders to it, and it displays back (one at a time) the four-character creator and file type signatures that the Finder uses. This can be a helpful tool for scripters who need to find out the file type for a choose file dialog parameter under construction.



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

Here's the script:

```
-- dropping a file(s) on a script application
-- sends an open command along with a list
-- of aliases for the dragged file(s). This
-- handler executes upon the drop.
on open fileList
    tell application "Finder"
        -- work through each item dragged
        -- with repeat in list form
        repeat with oneItem in fileList
            -- set Boolean for a folder flag
            set isFolder to (kind of oneItem is "folder")
            set fileName to name of oneItem
            if fileName is not "" then
                -- we got something, so assemble and
                -- display appropriate messages
                if isFolder then
                    -- short-circuit for a folder
                    display dialog "" & fileName & " is a folder." buttons
{"Okeedokee"} default button 1
                else
                    set details to return & "Creator: " & oneItem's creator type &
return & "File Type: " & oneItem's file type
                    display dialog "Info for " & fileName & ":" & details buttons
{"Thanks"} default button 1
                end if
            end if
        end repeat
    end tell
end open
```



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

The script consists of an **on open** handler. Notice that the parameter that went with the **open** message to trigger the script is a list of alias references. Although we could have used the **info for** scripting addition to extract the pertinent information, we demonstrate some features of the scriptable Finder here to entice you to look further into the Finder's AppleScript dictionary.

Inside the **tell** block, we establish a repeat loop to work through each item in the list (in case there are multiple items). Because we need to investigate properties of each item, the Repeat In List format of loop is the most efficient. Each item is handed to the **oneItem** variable for further processing. Folders don't have creator or file types, so if the user drags a folder among the items to the droplet, an appropriate message displays that information. For files, however, the **name**, **creator type**, and **file type** property values are assembled into a multiple-lined string that is shown via **display dialog**. A sample readout of a file imported by iTunes appears in Figure 15.2.



Figure 15.2. Example dialog produced by Show Creator & Type droplet.

We save this script as an application. Make sure that all checkboxes in the Save dialog are unchecked.

To run the application, drag any Finder object to the droplet icon. The application zooms open like any application, showing its name in the menu-bar briefly. After a few seconds, a dialog displays the findings of the open handler. When you click the dialog's button, the application quits.

You can assign properties to these applications, and their values (if updated in the course of running) will be saved to the application. For example, you could record behind the scenes how many times a particular shared script application was used on the network by creating a counter property that increments each time the script opens (put the incrementer into the open handler).



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

Agents

The idea of software agents—processes that operate in the background to monitor conditions and alert you when something of interest occurs—were all the rage back in the early 1990s. You don't hear the term much anymore, but the concept of background processing is more popular than ever, especially now that servers and personal computers have enough processing speed to do this extra stuff without bogging down the main work we need to accomplish.

AppleScript Folder Actions are agent-like in the way they work. With Mac OS X, it is also practical to compose AppleScript scripts that perform meaningful, but otherwise hidden work, and trigger that work from a timer-based feature of the underlying Unix foundation. These so-called cron jobs let advanced users assign tasks that need to be triggered automatically at fixed times, whether it be every five minutes or once a month.

Reminder Alarm Agent

To demonstrate how to create an agent out of an AppleScript script application, I've written an elementary daily reminder application. Its purpose is to beep and display a message at any number of pre-set times during the day. These aren't appointment reminders, per se, unless you have a standing appointment at the same time every day. These are best thought of as reminders for things you may forget to do every day, such as check your e-mail, listen to the radio for a specific stock market

report, or take medication. Note that this application assumes that the agent is run at the start of each day (something that a Unix cron job could do with a call to **osascript**).

This application screams for a user interface, but to keep things simple for the demonstration, we use a TextEdit document as a permanent repository for alarm times and messages. While we could store it all in a script property, using TextEdit gives you easier access to editing and testing various values.

The data file format is simple. Each line of the document is a single entry consisting of three or more AppleScript words. The first word is the time's numbers; the second word is the AM or PM designation (this example is obviously scripted for U.S. time formats, but could be modified for others). Remaining words in each line comprise the message that is to be displayed when the alarm reminder goes off. Figure 15.3 shows an Alarm Data document with sample data in it.

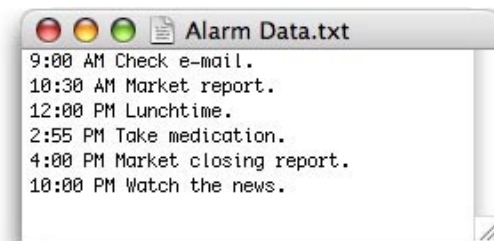


Figure 15.3. Alarm data document.



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

You can study the script in more detail for data parsing and assembly, but I want to highlight the elements that make this script an agent, and let it run in the background. Here's the script:

```
property alarmList : {}

on run
    tell application "TextEdit"
        open (choose file with prompt "Open
Alarm Data.txt")
        set alarmData to text of document 1
        close document "Alarm Data.txt"
    end tell
    repeat with i from 1 to number of
paragraphs of alarmData
        set oldDelims to AppleScript's text
item delimiters
        set AppleScript's text item
delimiters to {space}
        set oneEntry to {"", ""}
        set item 1 of item 1 of oneEntry to
date ((text items 1 thru 2 of paragraph i of
alarmData) as string)
        set item 2 of item 1 of oneEntry to
(text items 3 thru -1 of paragraph i of
alarmData) as string
        set AppleScript's text item
delimiters to oldDelims
        set alarmList to alarmList &
oneEntry
    end repeat
    repeat until alarmList = {} or item 1 of
item 1 of alarmList > (current date)
        set alarmList to rest of alarmList
    end repeat
    if alarmList = {} then
        display dialog "All alarm times
have passed for today." buttons {"Bye"}
```

```
default button "Bye"
        quit
    end if
end run

on idle
    if alarmList ≠ {} then
        if item 1 of item 1 of alarmList ≤
(current date) then
            activate me
            display dialog item 2 of
item 1 of alarmList buttons {"OK"} default
button "OK"
            set alarmList to rest of
alarmList
            return 0
        end if
    else
        quit
    end if
    return 5
end idle

on quit
    set alarmList to {}
    continue quit
end quit
```

The script consists of a single property and three handlers. **AlarmList** is the property that stores the current list of pending alarms. This value is set in the **run** handler.

The Run Handler

When you double-click this script application (or add it to your list of startup items in your Accounts preference pane), the script receives a **run** message.



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

Inside this handler, the script opens the *Alarm Data.txt* document with TextEdit to yank out a copy of the data.

Cycling through the paragraphs of the data, the script assembles a list of lists (thus the complex-looking parsing going on here). Given the sample data shown in Figure 15.3, this is what the data would look like if all records are in **alarmList**:

```
{{date "Friday, November 19, 2004 9:00:00 AM",  
"Check e-mail."}, {date "Friday, November 19,  
2004 10:30:00 AM", "Market report."}, {date  
"Friday, November 19, 2004 12:00:00  
PM", "Lunchtime."}, {date "Friday, November  
19, 2004 2:30:00 PM", "Take medication."},  
{date "Friday, November 19, 2004 4:00:00 PM",  
"Market closing report."}}
```

When dealing with dates and times, as this script does, you must be careful about the way date values are treated. While our data file specifies only times, AppleScript inserts any missing data when coercing the data to a date value. If you supply a time only, AppleScript supplements it with the current date. This happens to work great for this application, but it may trip up some other application you work on.

In assembling the alarm list, this application assumes the data is already in chronological order. In a real alarm application, there would be a user interface to enter alarms at any date and time in the future, plus a sorting mechanism to place the data in chronological order. Part of the list assembly procedure is to test whether the list has anything left in it—perhaps the

user starts the application late in the day. Rather than occupy memory (although only a small amount for a script application) for nothing pending, we alert the user that all alarms have passed for today, and send a **quit** message, which I'll get to in a moment.

We've seen **run** handlers in other script applications earlier in this chapter, but normally this means that the script runs and then automatically quits when it finishes. To prevent that automatic quitting, you must save this kind of agent script application with the Stay Open checkbox checked in the Save File dialog box. Opening an application saved this way makes it go through its run handler (if it has one), but then the application stays open, showing itself in the Mac OS X Dock and **Alt-Tab** rotation list of running applications. When the user selects the script application, a bare minimum of menus (mostly disabled) is available.

Idle Handler

Now, we must make sure our script checks the system clock against the top item in **alarmList**. To trigger that check, our script includes an **idle** handler, which intercepts the **idle** message that every Stay Open script application receives from the system. This message is sent only to script applications, not scripts running in Script Editor.

Most agents you create don't need to perform their tasks once every fraction of a second, as they would if they responded to every **idle** message. You might



Chapter 15:
**Script
Properties,
Objects, and
"Agents"**

notice a slowdown in overall machine performance. A special version of the **return** statement inside an **idle** handler sends a special instruction to the System, advising it how long to wait before sending this particular script application the next **idle** message.

In our script, we set the timer to 5 seconds (the integer after the **return** statement indicates how many seconds until the next idle message is sent to the application). That means for our application, the timer could be as much as 4.9 seconds slow. How accurate you need an application like this is up to you, but keep possible system degradation in mind if you shorten the interval.

Inside the **idle** handler, we compare the current time against the first item (the time) of the first item (entire alarm item) of the alarm list. If the current time is later, then we activate the script application so it takes precedence over whatever else is going on at the moment, and displays the alarm message. We also send an immediate **return** message for a couple of reasons. If another item is set to the same time, you want it to appear immediately after dismissing the alert for the first item, instead of waiting 5 seconds. Also, since we quit the application when the last alarm has rung for the day, you want that to happen immediately, as well.

Quit Handler

In both the **run** and **idle** handlers, we issue a **quit** command. This is actually the same **quit** command that the application gets if the user chooses Quit from the Reminder Alarm App menu (or shuts down the Mac). We have a **quit** handler in this script to reset the **alarmList** property to empty before the application quits. This way, we know the property will be empty the next time the user starts the application. If we didn't need any special processing when the script quits, we could omit this handler.

Notice, however, that the **quit** handler also contains a **continue** statement. If we didn't have this statement here, the **quit** message would never reach the application. Therefore, if you have a **quit** handler, be sure to let the message continue.

Summary

This chapter is chock full of three-pocket-protector-level techniques and applications for AppleScript. While simple scripting is fun and productive, the promise of creating standalone applications that integrate the best programming you can do with the powers of scriptable applications should stimulate your imagination. In the next and final chapter, we'll equip you with tips on how to approach a new scriptable application.



Chapter 16 Scripting Third-Party Applications

Although the previous chapters have spent time scripting in several applications—TextEdit, iTunes, BBEdit, Entourage, Excel, and wee bit of the Finder—it is unfortunate that learning how to script one application doesn't necessarily prepare you for scripting other products, even in the same category. Therefore, a script you write for TextEdit won't work in BBEdit or probably any other scriptable text editor. Each of these products has defined a different way of working with objects, and has a different set of commands. Still, there are similarities in the way you approach any scriptable application to learn its features and quirks. That is the focus of this chapter.

Knowing the Program

I can't emphasize enough how important it is to be very familiar with an application you intend to script. By this I don't necessarily mean the scriptable part of the program, but rather the way users normally interact with the program and data stored in its documents. Very often, a program has a lexicon of its own to denote parts of documents or processes. These words are then often carried over to the scripting commands and objects that are defined in the product's scripting dictionary.

As an example, Microsoft Excel defines an object called the **workbook**, which may contain numerous **worksheet** objects. If you don't understand how Excel workbooks work when manually using the program, you will have great difficulty managing multiple worksheets or multiple workbook files under AppleScript. Each of these objects must be part of references to data entry or retrieval of cells if you have any hope of transferring information to the desired locations in the Excel document files.

The basic recommendation, therefore, is to be well



Chapter 16:
**Scripting
Third-Party
Applications**

acquainted with the depths of a product you intend to script before you begin scripting. If you must risk scripting a new product in a hurry, then it will definitely be worth your time to sit through whatever tutorial comes along with the program to learn about the program as a user. It should acquaint you with at least the basic terminology of the program, which you may then recognize in the scripting dictionary.

Approaching a New Program

Because every program's scripting is different, it takes a bit of exploration to become familiar with scripting possibilities for the program. To that end, I've assembled a sequence of questions for which I seek answers each time I start to fiddle with a new scriptable application.

Is the Program Recordable?

As mentioned elsewhere in this book, script recording is a two-edged sword. On the one hand it lets someone who doesn't know much about scripting or the scripting of a particular application automate a manual process easily. But the downside is that such scripts can't be very smart—they can't make decisions or repeat operations in a truly iterative way (in a repeat loop).

Script recording, however, is still a good way to learn some of the basics about a program's objects. Turn on the recorder from the script editor, switch to the program, and perform several common operations, such as creating a new document, entering some data, copying and pasting data from within the document, changing a font style of a selection, opening a sample file, and saving a test file. Then examine the script and look especially for object references about the document and the data you entered.

Not every program's recorded scripts provide the precise scripting verbiage you would use in truly



Chapter 16:
**Scripting
Third-Party
Applications**

scripted actions—even though what the recorder captures will work fine. The difference is that very often the recorder captures actions in a rudimentary way as it watches what you do. For example, instead of a typical script action of blasting a number into a spreadsheet cell, the recorded method may show the cell being physically selected first, and the entry going into the selection. Yes, this works, and you may continue to replicate the recorded syntax in your own scripts, but such a method is slower and more verbose than a comparable action performed solely by scripting.

What is the Application's Object Model?

Once I determine the recordability of a product, the next task is to inspect the scripting dictionary to see how its object model matches my conception of the program as a user. It's easy to see if the program's designers have defined object classes for the dictionary by opening up each suite and expanding the list of classes. At this point in my perusal, I'm not concerned about the specifics. Just the fact that a product includes application-specific object classes in its dictionary is generally a positive sign—provided the dictionary is accurate.

I am often suspicious, however, if the dictionary indicates support only for the Core or Standard Apple event suites. This suspicion comes from a few examples of products that show standard object support in the dictionary, yet the objects are of little or no value to the product (e.g., text

objects in a graphics program, and nothing else). A better situation, however, is when the dictionary includes one or more additional suites, either of a generic nature (e.g., the Text Suite) or specific to the program. When I see Table and Charting suites in Excel, or the extensive iTunes suite in iTunes (lots of classes and commands), I'm hopeful that the program's authors gave their AppleScript implementation some thought.

Is the Object Model One That You Already Know?

Just because a program has defined object classes, even in specific event suites, doesn't mean that the objects behave the same way as the same kind of objects do in other programs. For example, two table-based applications might define a **cell** object, but the way your scripts must refer to these cells in each program could vary significantly: one may prefer to point to the cell as **cell x of row y**; another may prefer a format for the cell reference as **cell "RxCy"** where the "R" stands for "row," and the "C" for "column." Unfortunately, you can't always tell this from the dictionary (see about getting stuff into and out of a document, below). But you can examine an object's definition and see if it has the same kind of properties you know for similar objects in other programs you know.

As much as I would like to see all programs of a category maintain a common denominator of object and command definitions, I don't recommend



Chapter 16:
**Scripting
Third-Party
Applications**

shunning a program because its dictionary is foreign to you. More important is whether the program does what you need, and whether the tasks you wish to automate can be scripted. The fact that a dictionary may bear new definitions and formats only means that there will be some extra learning time involved prior to successfully scripting the application.

How Extensive is the Dictionary?

In virtually every example I've seen of a large scripting dictionary, the programming team has given significant thought to the scripting implementation. At the same time, however, you can't always equate a substantial dictionary with good scriptability. Microsoft Word X, for example, provided a fairly large dictionary. Unfortunately, there was precious little of it that actually worked. Instead, you found commands that let AppleScript invoke macros that you were supposed to create first within Word, and then trigger from AppleScript. In general, when I see a **do script** command (or similar) in a dictionary, I suspect the developers of being more dedicated to their proprietary internal scripting or macro language than to the true spirit of AppleScript.

Does the Program Support “Whose” Clauses?

One of the most valuable constructions in the AppleScript syntax is the filtered reference (Chapter 8), otherwise known as whose clauses. A whose

clause allows a script to ask a program to work on a number of objects that meet certain criteria, such as all records of a database whose State field is “MA”. When an application supports whose clauses, it can save lines and lines of scripting or figuring out how to make the program select objects by its own mechanism (e.g., an AppleScript **find** command tied to the program's internal search engine).

The dictionary usually reveals whether the programs will respond to whose clauses. When viewing the details of container object classes (documents, windows, etc.), a list of elements will usually indicate how those elements may be addressed. If the listing includes items available by “satisfying a test,” then the authors have likely implemented at least some level of whose clauses.

If a program supports whose clauses, it is mostly likely to be on common objects that can be represented in the plural form and have numerous properties—**documents, paragraphs, records**, and so on. To verify whose clause support, first create a document in the application that has known data and settings that you can test for. Make sure that the contents of at least two objects have something in common.

Next, make sure the program responds to the every element reference (a prerequisite for filtered references) by getting a property of all items in the document. Sometimes merely naming the object returns the object's contents, as in



Chapter 16: Scripting Third-Party Applications

```
every paragraph of document 1 -- TextEdit
every record -- FileMaker Pro
every item of startup disk -- Finder
```

Other times, you may need to specifically request the value of the object to see understandable data, as in

```
Value of every Cell of Range "R1C1:R10C10"
-- Excel
```

In this last example, we limited the excursion to a range of 100 cells, since getting every cell of a worksheet is time consuming and can overrun the ability of the Result pane to show the results.

Once you know that the every element reference works, it's time to try a filtered reference by adding a valid whose clause. A whose clause must focus on a property that certain objects have in common, such as contents, text format, or any other property setting. Here are examples that build on the ones above:

```
-- TextEdit
every paragraph of document 1 -
    where it contains "Fred"
-- FileMaker Pro
every record whose cell "State" contains "CA"
-- Finder
every item of startup disk -
    whose kind is "folder"
-- Microsoft Excel
Value of every Cell of Range "R1C1:R10C10" -
    whose font is "Geneva"
```

Of all the tests, above, only the one for Excel fails, meaning that it is unlikely you'll be able to use whose clauses in Excel. Indeed, the dictionary

entries for elements bears this out, because none are listed as satisfying a test. Knowing that a particular program does not support whose clauses is valuable knowledge early on, so you won't waste time trying to write them later and wondering what the problem is with your script.

How Do You Get Stuff Out of a Document?

A vital operation for many scripting tasks is retrieval of information from a document. Retrieving information generally entails the ever-popular **get** command, so the true focus is on accurate ways of referring to objects and their information.

To the conscientious scripter, the dictionary information listed as elements of object classes often contains valuable information (if the developer has been so kind as to provide that information). For example, look at the listing of elements for the FileMaker Pro 7 table class in Figure 16.1.

Class table: A FileMaker Pro table
Plural form:
tables
Elements:
field by name, by numeric index, before/after another element, as a range of elements, satisfying a test, by ID
record by name, by numeric index, before/after another element, as a range of elements, satisfying a test, by ID
cell by name, by numeric index, before/after another element, as a range of elements, satisfying a test, by ID

Figure 16.1. FileMaker Pro elements of a table class

The description after each object name indicates the kinds of references your scripts can use for the object. A record of a table, for example, can be



Chapter 16:
**Scripting
Third-Party
Applications**

signified by its name or ID, by its positional count, by its location relative to another object, by a range reference, and a filtered reference (“satisfying a test”). Therefore, any of the following references would be valid:

record 3 of table 2

record before last record of table 2

records 10 thru 20 of table 2

records of table 2 whose ID < 4

Some program’s class definitions, however, don’t give you enough information about addressing them from a script. Excel is a particularly good example. First of all, if you come from any other table-based environment, you think of the cell as the basic information object. Excel, however, utilizes the range object as the primary object for data access, even if the range is of a single cell. Not only that, but the syntax for ranges is different from that which you tend to use (and which Excel, itself uses by default) when working with ranges in formulas. For example, if you manually type a formula into cell A4 to sum the three cells above it, the formula reads

=SUM (A1 : A3)

with “A1:A3” representing the range of three cells starting at A1. Although recent versions of Excel accept scripted references in that form, you can see in intermediate results that the program prefers the RaCb:RcCd nomenclature for a range. There is no

indication of this in the dictionary, but you do get a hint of this by inspecting scripts you record. But the point is that the dictionary definition for the Range object lists only the “name” reference, which in Excel’s instance means either a name previously defined for the range or by its RaCb:RcCd format. Therefore, the range object consisting of the three cells described above would be:

Range "R1C1:R3C1"

You can also get clues about referencing by watching the Result pane of Script Editor when you enter a simple reference. For example, in Excel, if you enter a statement that gets an object, the program’s preferred way of referencing the object appears in the Result pane, as shown in Figure 16-2. As you can see, it may take a combination of experimentation and script recording to find out how a program refers to its basic objects.



Chapter 16: Scripting Third-Party Applications

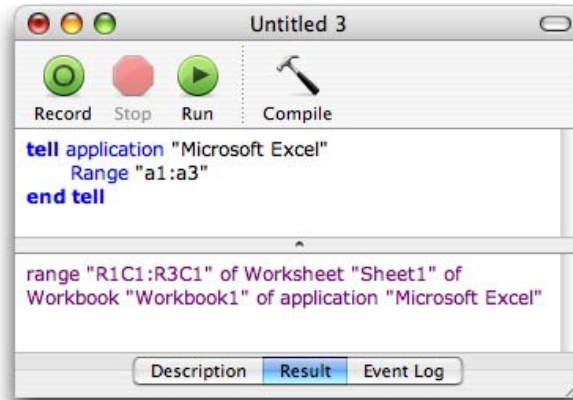


Figure 16.2. Use the Result pane to find the preferred referencing syntax

Referencing an object is important, but what we're really interested in is the data that is contained in those objects—that's what we're scripting about, anyway. Once you can successfully refer to a data object, it's time to see what the default reference returns. In other words, enter the equivalent of a **get** statement that points to an object holding some kind of known data (i.e., data you've entered into a sample document). In many programs, the default reference returns the contents of the object. For example:

```
paragraph 1 of document 1 -- TextEdit

record 1 -- FileMaker Pro

cell 1 of record 1 -- FileMaker Pro

folder 1 of startup disk -- Finder
```

All of these return data in a form that is perfectly natural: a TextEdit string; an AppleScript list of field values for FileMaker Pro; a string of a cell's contents from FileMaker Pro; and a reference to the first folder in sorting order of the startup disk in the Finder.

But not every default reference returns just the object's value. Consider Excel's default returned value for a worksheet cell:

```
tell application "Microsoft Excel"
    Cell "B3"
end tell
(* result: range "R3C2" of Worksheet
   "Sheet1" of Workbook "Workbook1" of
   application "Microsoft Excel" *)
```

Instead of the contents, we get a reference to a single-cell range within the worksheet. To get the contents of the cell, we have to look deeper into the dictionary, where we see a property of a range defined as:

Value anything

That's all it tells us. It turns out, however, that the **value** property (or other property name incorporating the word "value") is a common way to extract the data from an object whose default reference doesn't return its data. For Excel, then, the data extraction would be:

```
tell application "Microsoft Excel"
    value of Cell "B3"
end tell
(* result: 2050 [or whatever
   value is displayed] *)
```

FileMaker Pro actually can return data with both



Chapter 16:
**Scripting
Third-Party
Applications**

types of references:

```
every cell of record 1
```

```
cellValue of every cell of record 1
```

Both forms return an AppleScript list, with each cell's value a string item within the list.

How You Get Stuff Into a Document?

The effort you exert in learning how to retrieve information from a scriptable application's document goes a long way toward helping you enter information into documents. Primary concerns are selecting the desired command, knowing how to refer to objects that are to receive the information, and formatting the information you're about to insert into the document.

On the command side of things, it's a relatively short list of possibilities, depending on the object and the program. The most common commands for entering new information are:

```
set
```

```
copy
```

```
make
```

```
create
```

This list is deceiving, because the first two are identical in their action (just different in the order of their parameters) and, generally speaking, a program

will support one of the last two, but rarely both. You cannot, however, assume that programs supporting the **make** or **create** commands define them in the same manner, so it's a good idea to check the dictionary for parameters in each program.

Setting (or copying) data involves doing so to an existing object or insertion point, depending on the kind of document you're working with and the application's support for the necessary constructs. For example, the BBEdit text editor lets you set any potential text cursor location (insertion point) to some new text value, thereby inserting text into that point. The following script places the text "Howdy" at the beginning of the document:

```
tell window 1 of application "BBEdit"  
    set insertion point before text 1 to "Howdy"  
end tell
```

For some other kinds of documents, you use similar **set** or **copy** constructions directed at the same object (or property of the object) from which you use the **get** command to read data. Also, an application may define its own variation of the **set** command for the sake of its internal clarity (i.e., to distinguish the command from a standard **set** command). For example, one way to plug a value into a FileMaker Pro cell is to use FileMaker's **set data** command as follows:

```
tell application "FileMaker Pro"  
    set data of cell "State" to "OH"  
end tell
```




Chapter 16:
**Scripting
Third-Party
Applications**

The above command puts text into a specific cell of the current record showing in the database window (whether or not the application is active).

But when an application's default reference for its objects does not yield the contents of the object (e.g., when retrieving we have to get the **value** or **cellValue** of the object), a script must set the appropriate property for that object. Therefore, in Excel, to insert a number into a spreadsheet cell, the construction would be like this:

```
set value of Range "R4C5" to 5959
```

It is vital to know the format of data your script needs to insert into a document. For example, setting the value of three consecutive fields of a database will undoubtedly require that the data for each field be separated in some fashion. To know how best to format the data, retrieve some sample data of a similar range first. In Excel, for instance, if we retrieve the value of all cells in a range, we get a list:

```
tell application "Microsoft Excel"
  get Value of Range "A1:A3"
end tell
-- result: {"Fred"}, {"Ginger"}, {"Dennis"}}
```

Therefore to set the value of those cells to different values, the script must format the data the same way:

```
tell application "Microsoft Excel"
  set Value of Range "A1:A3" to {"Nick"},
  {"Nora"}, {"Asta"}}
end tell
```

Another method of entering data into a document is, when supported by the application, to create a new object and assign data to that new object at the same time. That's where the **make** or **create** commands come in. Sending the commands without any additional parameters generally creates a new, blank object. But to send some data along, you should check the dictionary entry of the **make** or **create** commands for **with data** and/or **with properties** labeled parameters. In addition, there may be a parameter that lets you specify where in the document the new object should go (usually with an **at** parameter label). Let's take these parameters one at a time to see how they work.

The **with data** parameter is the key one, since the data you pass with this parameter is the data that generally goes into the new object. The format of the data must be in the same format as you'd extract from that object. For instance, you can instruct TextEdit to create a new paragraph at the start of the document with the words "New Paragraph" in it:

```
tell document 1 of application "TextEdit"
  make new paragraph at beginning with data
  ("New Paragraph" & return)
end tell
```

(Note: Through at least version 1.3 of TextEdit, the AppleScript dictionary mistakenly calls the **at** parameter optional, when, in truth, it is required for the **make** command.)

A FileMaker Pro record, on the other hand, expects



Chapter 16:
**Scripting
Third-Party
Applications**

its data to be in an AppleScript list, as in:

```
tell application "FileMaker Pro"
    create new record with data {"John",
    "Doe", "Acme Products", "Continuing"}
end tell
```

Data for each cell is in the same order as the fields in the layout.

The **with properties** parameter can let you pass additional information about the object you're creating. Any settable property of the object can be adjusted as it's being created. For example, in TextEdit, you may wish to set some font properties, as in:

```
tell document 1 of application "TextEdit"
    make new paragraph at beginning with data
    ("New Paragraph" & return) with properties
    {size:20, font:"Courier"}
end tell
```

A final parameter (optional in some programs, required in others) specifies where the object is to go. Failure to indicate the location usually places the new object at whatever location would receive keystrokes or a new object from the program's **New** menu (when the menu item applies to content items). But the **at** parameter lets your script control the insertion spot. Relative references almost always work, such as:

```
at end
at beginning
after <objectReference>
before <objectReference>
```

Appending a new object to the end of the document

(or as the last record of a database) requires the **at end** variation. In most cases, creating a new object like this inserts the new object, perhaps disturbing only the sequence of existing objects. But be sure to test this out on known data first: some variations might overwrite the object before which or after which the new object is to go (most likely a bug in the scriptable program).

As you can see, the **make** and **create** commands can be rather powerful constructions, depending on the program's support for them.

How Do You Massage Stuff Inside a Document?

As you dig more deeply into scripting data within a document, you get further away from any commonality that scriptable applications may share. In the majority of cases, each program defines a number of commands and object classes that are unique to itself, causing scripters to have to learn the scripting vocabulary and idiosyncrasies of each scriptable application.

Of course, the category of pseudo-scriptable applications—the ones that expect you to do serious scripting in the program's own scripting or macro language—have a lot in common. In addition to support of just a handful of common commands and objects (if that much), such applications usually contain scripting dictionary definitions for two essential commands:



Chapter 16:
**Scripting
Third-Party
Applications**

```
do script  
evaluate
```

The first command, **do script**, runs a script or macro written in the program's own language. Somewhere in the architecture of the language is a way to assign a name to the script (it may appear in a customizable menu, for instance). That name (as a string value) becomes the parameter for the **do script** command. You must run tests on calling these macros with the **do script** command, because processes that take a long time may require the AppleScript timeout setting be adjusted upward to accommodate the macro's execution.

Occasionally, a macro will supply or return some data, much like a subroutine does. In such cases, the value comes back as the result of the **evaluate** command. Again, the parameter for the command is most likely the name of an internal macro or script in the program's own language. To capture the value, assign the result to a variable in your own script, such as

```
set returnedValue to evaluate "ProgramMacro"
```

You must also exercise care in the format of the returned value. If you are accustomed to receiving AppleScript list objects or records for multiple-piece data, it's more than likely that the program's internal scripting language doesn't return data in those formats. Therefore, check carefully for the format of the data, and write AppleScript code to get it the way you want it.

For true object scripting within a document, it is important to keep accurate track of the objects you're working with. In the case of the Finder, for example, virtually every command returns the reference to the object just manipulated, so you can perform another task on that object with the comfort that you have its most recent and accurate reference. Not all programs are that friendly, so as you begin working on a script that controls objects in documents, I suggest you activate the program during script construction so you can observe the process taking place. In many cases, the problem is one of context: the object you want to work on isn't available or has a new reference as a result of a previous action.

The vast differences in working with the specifics of scriptable applications cause difficulty when trying to get help from other people who know AppleScript (in case you're either intimidated by or disappointed with the product support from the program publisher). Unless your colleagues have had direct experience with the very program you're stuck in, it's unlikely that they'll be able to help you with specific questions. But if you hook up with online scripting forums, there's a good chance someone will have experience with all but the most off-beat applications.



Chapter 16:

**Scripting
Third-Party
Applications**

Onward and Upward

This chapter and all the ones before it should have provided you with a more than adequate introduction to the fundamentals of the AppleScript language and the Macintosh scripting environment. I hope I have achieved the goal of equipping you to explore additional scriptable applications, including the Finder. There is still much to learn about the individual programs you intend to script, but once the language basics are in your head, you can focus on the special requirements of individual programs.

May your scripting future be a bright one.



Appendix A AppleScript Quick Reference

AppleScript Commands

activate *referenceToApplication*
activate application ~
"SteveHD:Applications:Microsoft Excel" ~
of machine "eppc://192.168.1.122"

copy *expression to variableOrReference*
copy paragraph 1 of ~
document "Sample 6.1" to oneGraph

count [of] *directParameter* ~
[each *className*]
count {1, 1.5, 2} each real
-- result: 1

[get] *expressionOrReference*
get words 1 thru 3 of document "Preamble"
-- result: {"We", "the", "people"}

launch *variableOrReference*
launch application "TextEdit"

put *expression into variableOrReference*
put paragraph 1 of ~
document "Sample 6.1" into oneGraph

run *variableOrReference*
run application "TextEdit"

set *variableOrReference to expression*
set oneGraph to paragraph 1 of ~
document "Sample 6.1"

Scripting Addition Commands

ASCII character *integer0to255*
ASCII character 65
-- result: "A"

ASCII number *characterAsString*
ASCII number "G"
-- result: 71

beep [*numberOfBeeps*]
beep 4

choose application [with title ~
windowTitle] ~
[with prompt *promptString*] ~
[as *appOrAliasClass*] ~
[multiple selections allowed *Boolean*]
choose application with title ~
"Application Chooser" with ~
prompt "Pick a program:" as alias
-- result: application "Microsoft Excel"

choose color [default color ~
RGBColorSpec]
choose color default color ~
{65535, 65535, 65535}
-- result: {26143, 63756, 21232}



Appendix A: AppleScript Quick Reference

```
choose file    [ with prompt ~
    promptString ] ~
    [ of type fileTypeList ] ~
    [ default location fileOrFolderAlias ] ~
    [ invisibles Boolean ] ~
    [ multiple selections allowed Boolean ]
    choose file with prompt ~
        "Select one or more documents:" ~
        default location (path to desktop) ~
        invisibles false of type {"XLS8"} ~
        with multiple selections allowed
    -- result: alias "HD:Documents:budget.xls"
```

```
choose file name [ with prompt ~
    promptString ] ~
    [ default name fileName ] ~
    [ default location fileOrFolderAlias ]
    set myFile to new choose file name ~
        with prompt "Store information in:" ~
        default name "Backup"
    -- result: alias "MacHD:Documents:Backup"
```

```
choose folder [ with prompt ~
    promptString ] ~
    [ default location fileOrFolderAlias ] ~
    [ invisibles Boolean ] ~
    [ multiple selections allowed Boolean ]
    choose folder with prompt ~
        "Select a destination folder:"
    -- result: alias "Macintosh HD:Letters:"
```

```
choose from list itemList ~
    [ with prompt promptString ] ~
    [ default items itemList ] ~
    [ OK button name string ] ~
    [ cancel button name string ] ~
    [ multiple selections allowed Boolean ] ~
    [ empty selection allowed Boolean ]
```

```
choose from list ~
    {"MacOS", "Windows", "Linux", "Unix", ~
    "Other"} default items ~
    {"MacOS"} with prompt ~
    "What is your favorite operating system?"
    -- result: "MacOS"
```

```
choose URL    [ showing servicesList ]
    [ editable URL Boolean ]
    (servicesList: Web servers, FTP Servers,
    Telnet hosts, File servers, News servers,
    Directory services, Media servers, Remote
    applications)
    choose URL showing ~
        {Web servers, FTP Servers}
    -- result: "afp://192.168.1.102"
```

```
close access fileReference
    close access myFileRef
```

```
clipboard info
    clipboard info
    (* result: {{styled Clipboard text, 22},
    {string, 4}, {uniform styles, 144},
    {Unicode text, 8},
    {«class RTF », 238}} *)
```

```
current date
    get current date
    (* result: date "Friday, October 29,
    2004 4:23:26 PM" *)
```

```
delay         seconds
    delay 3
```

```
display dialog string ~
    [ default answer string ] ~
    [ buttons buttonList ] ~
    [ default button integer | string ] ~
    [ with icon integer | string ] ~
    [ giving up after seconds ]
```



Appendix A: AppleScript Quick Reference

```
display dialog "What is your name?" ~
    default answer ""
    (* result: {text returned:"David",
        button returned:"OK"} *)
display dialog ~
    "What color would you like?" ~
    buttons {"Red","Green","Blue"} ~
    default button 1 with icon 1
    -- result: {button returned:"Blue"}
```

```
do shell script    commandString ~
    [ as returnValueClass ] ~
    [ administrator privileges Boolean ] ~
    [ password passwordString ] ~
    [ altering line endings Boolean ]
do shell script ~
    "sudo apachectl restart" ~
    with administrator privileges ~
    password "notmydog"
```

```
get eof    fileReference
get eof myFile
    -- result: 233 (length of file)
```

```
get volume settings
get volume settings
    (* result: {output volume:22, input
        volume:missing value, alert
        volume:67, output muted:false} *)
```

```
info for    fileOrFolderReference
info for alias ~
    "HD:Applications:iTunes.app"
    (* result: {name:"iTunes.app",
        creation date:date "Friday, April
        25, 2003 11:20:06 PM", modification
        date:date "Sunday, October 31, 2004
        12:30:21 AM", icon position:{0, 0},
        size:2.7769953E+7, folder:true,
        alias:false, name extension:"app",
        extension hidden:true, visible:true,
        package folder:true, file
        type:"APPL", file creator:"hook",
        displayed name:"iTunes",
```

```
kind:"Application", short
version:"4.7", long version:"iTunes
4.7, Copyright 2000-2004 Apple
Computer, Inc.", bundle
identifier:"com.apple.iTunes"} *)
```

list disks

```
list disks
    (* result: {"Hard Disk", "Backup HD",
        "U2-Vertigo"} *)
```

list folder folderReference ~

```
[ invisibles    Boolean ]
list folder alias ~
    "MacHD:Users:jane:Desktop:"
    (* result: {".DS_Store", ".FBCIndex",
        ".FBCLockFolder", ".tcshrc",
        "display.pdf", ...} *)
```

load script aliasReference

```
load script alias "HD:System Folder:Script
Libraries:StringLib"
    -- result: <<script>>
```

log expression

```
log myVar
```

mount volume nameOrURL ~

```
[ as user name    userName ] ~
[ with password    password ]
mount volume ~
    "afp://192.168.1.104/TomsLaptopHD"
    -- result: file tomspb:
```

open for access fileOrAliasReference ~

```
[ write permission Boolean ]
open for access file "Test File" with ~
    write permission
    -- result: 9496 (file reference number)
```




Appendix A: AppleScript Quick Reference

```
open location URLText ~
[ error reporting Boolean ]
open location "http://apple.com"

offset of containedString in ~
containerString
offset of "people" in "We the people"
-- result: 8

path to applicationOrFolder ~
[ from domain ] ~
[ as class ] ~
[ folder creation Boolean ]
(applicationOrFolder constants: apple
menu [items folder], application support
[folder], applications folder, control
panels [folder], control strip modules
[folder], current application, current
user folder, desktop [folder], desktop
pictures folder, documents folder,
editors [folder], extensions [folder],
favorites folder, Folder Action scripts,
fonts [folder], frontmost application,
help [folder], home folder, internet
plugins folder, keychain folder, library
folder, me, modem scripts [folder],
movies folder, music folder, pictures
folder, preferences [folder], printer
descriptions [folder], printer drivers
[folder], printmonitor [folder], public
folder, scripting additions [folder],
scripts folder, shared documents [folder],
shared libraries [folder], shutdown items
folder, sites folder, speakable items,
startup disk, startup items [folder],
stationery [folder], system folder, system
preferences, temporary items [folder],
trash [folder], users folder, utilities
folder, voices [folder])
path to library folder from user domain
(* result: alias "Macintosh HD:Users:
fred:Library:" *)
```

```
random number [ number ] ~
[ from bottomNumber to topNumber ] ~
[ with seed number ]
random number from 3 to 30
-- result: 12
```

```
read fileReference ~
[ from startInteger ] ~
[ for numberOfBytes | to endInteger ~
| until includedDelimiter | ~
before excludedDelimiter ] ~
[ as dataType | class ] ~
[using delimiter[s] delimiterList ]
read myFile from 201 for 100
(* result: [100 characters of
text from file] *)
```

```
round realNumber [ rounding ~
up | down | toward zero | ~
to nearest | as taught in school ]
round 35.74 rounding down
-- result: 35
```

```
run script variableOrReference ~
[ with parameters parameterList ] ~
[ in scriptingComponent ]
run script alias ~
"HD:Library:Scripts:dateRecord.scpt"
(* result: [value returned by
script object, if any] *)
```

```
say text [ displaying text ] ~
[ using voiceName ] ~
[ waiting until completion Boolean ] ~
[ saving to aliasOrFileRef ]
say "I am not a number." using "Trinoids"
```



Appendix A:
AppleScript
Quick Reference

scripting components

```
scripting components
-- result: {"JavaScript", "AppleScript"}
```

set eof *fileReference* to *integer*

```
set eof myFile to 512
```

set the clipboard to ~

```
variableOrReference
set the clipboard to myText
```

set volume *integer0thru7* ~

```
[ output volume integer0thru100 ] ~
[ input volume integer0thru100 ] ~
[ alert volume integer0thru100 ] ~
[ output muted Boolean ]
set volume output volume 50 ~
alert volume 80
```

start log

```
start log
```

stop log

```
stop log
```

store script *scriptObject* ~

```
in aliasReference ~
[ replacing ask | yes | no ]
store script transactionLog in alias ~
"Server:Shared Objects:Transaction Log" ~
replacing yes
```

summarize *textOrTextFileRef* ~

```
[ in sentenceCount ]
summarize ~
"HD:Documents:Meeting Minutes" in 2
-- result: [Sherlock-summarized string]
```

system attribute ~

```
[ environmentVarName ] [ has integer ]
system attribute "SHELL"
-- result: "/bin/tcsh"
```

the clipboard [as class]

```
get the clipboard
-- result: [Clipboard contents]
```

time to GMT

```
(time to GMT)/hours
-- result: -8.0
```

write *expression* to *fileReference* ~

```
[ for numberOfBytes ] ~
[ starting at integer ] ~
[ as class ]
write "Fred" to myFile starting at eof
```



Appendix A:
AppleScript
Quick Reference

Folder Action Commands

adding folder items to -
aliasReference after receiving -
aliasReferenceList

closing folder window for -
aliasReference

moving folder window for -
aliasReference from *rectangle*

opening folder *aliasReference*

removing folder items from -
aliasReference after losing -
aliasReferenceList

Reference Forms

Property References

[the] *propertyLabel* <of *objectOrRecord*>
bounds of window 1

Indexed References

[the] *className* [*index*] *index* -
 <of *objectOrItem*>
paragraph 12 of first document

[the] (*first* | *second* | *third* | -
fourth | *fifth* | *sixth* | *seventh* | -
eighth | *ninth* | *tenth*) *className* -
 <of *objectOrItem*>
third paragraph of document 1

[the] *index*(*st* | *nd* | *rd* | *th*) -
className <of *objectOrItem*>
the 234th word

[the] (*last* | *front* | *back*) -
className <of *objectOrItem*>
first word

Relative References

[*className*] (*before* | [in] *front of*) -
baseReference
word before paragraph 3

[*className*] (*after* | [in] *back of* | -
behind) *baseReference*
window behind window 1
after word 1 of document 1



Appendix A:
AppleScript
Quick Reference

Name References

```
className [ named ] nameString ~
    <of objectOrItem>
        window "Layer 2"
```

ID References

```
className id IDvalue <of Object>
    window id 9029
```

Middle Element Reference

```
middle className <of objectOrItem>
    middle word of paragraph 1
```

Arbitrary Element References

```
some className <of objectOrItem>
    some item of {20, 40, "sixty", 80, 100}
```

Every Element References

```
every className <of objectOrItem>
    every cell of row 12 of table 1
```

```
pluralClassName <of objectOrItem>
    words of paragraph 1
```

Range References

```
className startIndex ( thru | through ) ~
    stopIndex
    word 1 thru 3 of paragraph 1
```

```
pluralClassName startIndex ~
    ( thru | through ) stopIndex
    words 1 thru 3 of paragraph 1
```

```
every className from boundaryReference1 ~
    to boundaryReference2
    every character from word 1 to word 2
```

```
pluralClassName from boundaryReference1 ~
    to boundaryReference2
    characters from word 1 to word 2
```

Filtered References

```
reference whose | where BooleanExpression
    every word whose first character is "s"
    every word where it is "sea"
```

File References

```
file pathname
    file "HD:Correspondence:Memo to Mike"
```

```
alias pathname
    alias "HD:Correspondence:Memo to Mike"
```



Appendix A:
AppleScript
Quick Reference

Value Classes

<i>Class Name</i>	<i>Example</i>
boolean	true
class	string
constant	pi
data	«data»
date	date "Tuesday, November 9, 2004 3:53:02 PM"
file specification	
integer	25
international text	"Steve"
list	{2, "Michael", 34.5}
number	25 or 25.0
real	25.0
record	{name:"Joe", age:32, weight:166}
reference	word 1 of document 1 of application "TextEdit"
rgb color	{65535, 0, 65535}
string	"Steve"
styled clipboard text	
styled text	
text	"Steve"
unicode text	"Steve"



Appendix A:
AppleScript
Quick Reference

Date Class Properties

<i>Class</i>	<i>Example</i>	<i>Description</i>
Boolean	true	A logical true or false
integer	233	A positive or negative whole number (no decimals allowed)
list	{1,2,3}	A series of other values—of any class— arranged in a specific order inside curly braces
real	2.5	A positive or negative number with decimal fraction
string	"Howdy"	A series of characters inside quote marks

List Class Properties

<i>Property</i>	<i>Value Class</i>	<i>Description</i>
rest of	list	items in the list except for the first one
reverse	list	items in the list in reverse order
length	integer	number of items in the list



Appendix A:
AppleScript
Quick Reference

Unit Type Classes

Length

centimeters
centimetres
feet
inches
kilometers
kilometres
meters
metres
miles
yards

Area

square feet
square kilometers
square kilometres
square meters
square metres
square miles
square yards

Weight

grams
kilograms
ounces
pounds

Cubic Volume

cubic centimeters
cubic centimetres
cubic feet
cubic inches
cubic meters
cubic metres

Liquid Volume

gallons
liters
litres
quarts

Temperature

degrees Celsius
degrees Fahrenheit
degrees Kelvin



Appendix A:
AppleScript
Quick Reference

Control Statements

Tell

```
tell reference to statement
tell reference
    [ statement ] ...
end [ tell ]
```

If-Then-Else

```
if BooleanExpression then statement
```

```
if BooleanExpression [ then ]
    [ statement ] ...
end [ if ]
```

```
if BooleanExpression [ then ]
    [ statement ] ...
else
    [ statement ] ...
end [ if ]
```

```
if BooleanExpression [ then ]
    [ statement ] ...
else if BooleanExpression [ then ]
    [ statement ] ... ...
end [ if ]
```

```
if BooleanExpression [ then ]
    [ statement ] ...
else if BooleanExpression [ then ]
    [ statement ] ... ...
else
    [ statement ] ...
end [ if ]
```

Repeat

```
repeat
    [ statement ] ...
end [ repeat ]
```

```
repeat numberOfTimes [ times ]
    [ statement ] ...
end [ repeat ]
```

```
repeat until BooleanExpression
    [ statement ] ...
end [ repeat ]
```

```
repeat while BooleanExpression
    [ statement ] ...
end [ repeat ]
```

```
repeat with counterVariable from  $\neg$ 
    startValue to stopValue [ by stepValue ]
    [ statement ] ...
end [ repeat ]
```



Appendix A:
AppleScript
Quick Reference

```
repeat with loopVariable in listReference
    [ statement ] ...
end [ repeat ]
```

Exit

```
exit [ repeat ]
```

Try-Error

```
try
    [ statementToTest ] ...
on error [ errorMessage ] ¬
    [ number errorNumber ] ¬
    [ from offendingObject ] ¬
    [ to expectedType ] ¬
    [ partial result resultList ]
    [ global variableID [, ¬
variableID ] ... ]
    [ local variableID [, ¬
variableID ] ... ]
    [ statementHandlingError ] ...
end [ try | error ]
```

Timeout

```
with timeout [ of ] numberOfSeconds ¬
    seconds [s]
    [ statement ] ...
end [ timeout ]
```

Transaction

```
with transaction [ sessionID ]
    [ statement ] ...
end transaction
```

Considering

```
considering attribute ¬
    [ , attribute ... and attribute ] ¬
    [ but ignoring ¬
    [ , attribute ... and attribute ] ]
    [ statement ] ...
end considering
(attributes: case, white space, diacriticals,
hyphens, punctuation, application responses)
```

Ignoring

```
ignoring attribute ¬
    [ , attribute ... and attribute ] ¬
    [ but considering ¬
    [ , attribute ... and attribute ] ]
    [ statement ] ...
end ignoring
```



Appendix A:
AppleScript
Quick Reference

Considering & Ignoring Attributes

<i>Attribute</i>	<i>Default</i>	<i>Description</i>
case	ignores	Distinguishes uppercase from lowercase letters, applying each character's strict ASCII value for comparisons.
white space	considers	Regards spaces between characters as characters to be compared.
diacriticals	considers	Distinguishes characters with diacritical marks (e.g., á, ô, è) from same letters without the marks.
hyphens	considers	Regards hyphens as characters to be compared.
punctuation	considers	Regards punctuation symbols as characters to be compared.



Appendix A:
AppleScript
Quick Reference

Operators

Connubial Operators

Syntax	Name	Operands	Results
+	Plus	Integer, Real	Integer, Real
-	Minus	Integer, Real	Integer, Real
*	Multiply	Integer, Real	Integer, Real
/ or ÷ (Option-/)	Divide	Integer, Real	Integer, Real
div	Integral Division	Integer, Real	Integer
mod	Modulo	Integer, Real	Integer, Real
^	Exponent	Integer, Real	Real
&	Concatenation	All (See below)	List, Record, String
as	Coercion	(See below)	(See below)
[a] ref[erence] [to]	A Reference To	Reference	Reference



Appendix A:
AppleScript
Quick Reference

Comparison Operators

Syntax	Name	Operands	Results
= is equal[s] [is] equal to	Equal	All	Boolean
≠ (Option=) is not isn't isn't equal [to] is not equal [to] does not equal doesn't equal	Not equal	All	Boolean
> [is] greater than comes after is not less than or equal [to] isn't less than or equal [to]	Greater than	Date, Integer, Real, String	Boolean
< [is] less than comes before is not greater than or equal [to] isn't greater than or equal [to]	Less than	Date, Integer, Real, String	Boolean
>= ≥ (Option->) [is] greater than or equal [to] is not less than isn't less than does not come before doesn't come before	Greater than or equal to	Date, Integer, Real, String	Boolean
<= ≤ (Option-<) [is] not less than or equal [to] is not greater than isn't greater than does not come after doesn't come after	Less than or equal to	Date, Integer, Real, String	Boolean



Appendix A:
AppleScript
Quick Reference

Containment Operators

Syntax	Name	Operands	Results
contain[s]	Contains	List, Record, String	Boolean
does not contain doesn't contain	Does not contain	List, Record, String	Boolean
is contained by	Is contained by	List, Record, String	Boolean
is not contained by isn't contained by	Is not contained by	List, Record, String	Boolean
start[s] with	Starts with	List, String	Boolean
begin[s] with			
end[s] with	Ends with	List, String	Boolean

Boolean Operators

Syntax	Name	Operands	Results
and	And	Boolean	Boolean
or	Or	Boolean	Boolean
not	Not	One Boolean	Boolean



Appendix A:
AppleScript
Quick Reference

Operator & Evaluation Precedence

Precedence	Operator	Notes
1	()	From innermost to outermost
2	+ and -	When used with a single value to signify positive or negative
3	's	Object containment (left to right)
4	of in	Object containment (right to left)
5	my its	Object containment (right to left)
6	^	Exponent (right to left)
7	* / and ÷ div mod	Multiplication and division
8	+ -	Addition and subtraction
9	&	Concatenation
10	as	Value coercion
11	< ≤ > ≥ contains	Comparison operators and their synonyms
12	= ≠	Equality
13	not	Unary Boolean negation operator
14	and	Boolean operator
15	or	Boolean operator
16	whose	Filter reference
17	reference to	Object reference



Appendix A:
AppleScript
Quick Reference

Subroutines and Handlers

Positional Parameters

```
subroutineName ( [ parameterValue ] ↵
    [, parameterValue ] ... )

on | to subroutineName ( [ parameter ] ↵
    [, parameter ] ... )
    [ global variableID ↵
    [, variableID ] ... ]
    [ local variableID ↵
    [, variableID ] ... ]
    [ statement ] ...
end [ subroutineName ]
```

Labeled Parameters

```
subroutineName [ [ of | in ] ↵
    directParameter ] ↵
    [ subroutineParamLabel ↵
    parameterValue ] ... ↵
    [ given labelName:parameter ↵
    [, labelName:parameter ] ... ] ↵
    [ with labelForTrueParam ↵
    [, labelForTrueParam, ... and ↵
    labelForTrueParam ] ] ↵
    [ without labelForFalseParam ↵
    [, labelForFalseParam , ... and ↵
    labelForFalseParam ] ]
(labelNameParameters: about, above, against,
apart from, around, aside from, at, below,
```

beneath, beside, between, by, for, from,
 instead of, into, on, onto, out of, over,
 since, through, thru, under)

```
on | to subroutineName [ [ of | in ] ↵
    directParameter ] ↵
    [ subroutineParamLabel parameter ] ... ↵
    [ given labelName:parameter ↵
    [, labelName:parameter ] ... ]
    [ global variableID [, ↵
    variableID ] ... ]
    [ local variableID [, ↵
    variableID ] ... ]
    [ statement ] ...
end [ subroutineName ]
```

Return

return expression

Command Handler

```
on | to commandName [ [ of ] ↵
    directParameter ] ↵
    [ [ given ] paramLabel: parameter ↵
    [, paramLabel: parameter ] ... ]
    [ global variableID [, ↵
    variableID ] ... ] ↵
    [ local variableID [, variableID ] ... ]
    [ statement ] ...
end [ commandName ]
```



Appendix A:

AppleScript Quick Reference

Miscellaneous

Script Objects

```
script [ scriptObjectVariable ]  
    [ property | prop ~  
        propertyLabel: initialValue ] ...  
    [ handlerDefinition ] ...  
    [ statement ] ...  
end [ script ]
```

Child Script Objects

```
script [ scriptObjectVariable ]  
    [ property | prop ~  
        parent: parentScriptObject ] ...  
    [ handlerDefinition ] ...  
    [ statement ] ...  
    [ continue parentHandler ] ...  
end [ script ]
```

Property Definition

```
property | prop propertyLabel:initialValue
```

Droplet Handler

```
on open itemList  
    [ statement ] ...  
end open
```

“Agent” Handlers

```
on run  
    [ statement ] ...  
end run
```

```
on idle  
    [ statement ] ...  
end idle
```

```
on quit  
    [ statement ] ...  
end quit
```

Constants

```
days  
hours  
it  
me  
minutes  
pi  
result  
return  
space  
tab  
weeks
```



Appendix A:
AppleScript
Quick Reference

AppleScript Objects

application
machine
zone
paragraph
word
character
item
text item

AppleScript Global Property

text item delimiters default: {""}



Appendix B ASCII Table

ASCII	Character	Name	ASCII	Character	Name
0	NUL	Null	15	SI	Shift In
1	SOH	Start of Heading	16	DLE	Data Link Escape
2	STX	Start of Text	17	DC1	Device Control 1
3	ETX	End of Text	18	DC2	Device Control 2
4	EOT	End of Transmission	19	DC3	Device Control 3
5	ENQ	Enquiry	20	DC4	Device Control 4
6	ACK	Acknowledge	21	NAK	Negative Acknowledge
7	BEL	Bell	22	SYN	Synchronous Idle
8	BS	Backspace	23	ETB	End of Transmission Block
9	HT	Horizontal Tab	24	CAN	Cancel
10	LF	Line Feed	25	EM	End of Medium
11	VT	Vertical Tab	26	SUB	Substitute
12	FF	Form Feed	27	ESC	Escape
13	CR	Carriage Return	28	FS	File Separator
14	SO	Shift Out	29	GS	Group Separator



Appendix B:
ASCII Table

ASCII	Character	Name	ASCII	Character	Name
30	RS	Record Separator	51	3	
31	US	Unit Separator	52	4	
32	SP	Space	53	5	
33	!	Exclamation Point	54	6	
34	"	Double Quote	55	7	
35	#	Number Sign	56	8	
36	\$	Dollar Sign	57	9	
37	%	Percent Sign	58	:	Colon
38	&	Ampersand	59	;	Semicolon
39	'	Apostrophe	60	<	Less Than
40	(	Open Parenthesis	61	=	Equals
41	)	Close Parenthesis	62	>	Greater Than
42	*	Asterisk	63	?	Question Mark
43	+	Plus	64	@	Commercial At
44	,	Comma	65	A	
45	-	Hyphen	66	B	
46	.	Period	67	C	
47	/	Slant	68	D	
48	0	Zero	69	E	
49	1		70	F	
50	2		71	G	



Appendix B:
ASCII Table

ASCII	Character	Name
72	H	
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	
79	O	
80	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	
91	[	Open Bracket
92	\	Reverse Slant

ASCII	Character	Name
93	]	Close Bracket
94	^	Circumflex
95	_	Underline
96	‘	Grave Accent
97	a	
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	



Appendix B:
ASCII Table

ASCII	Character	Name
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	Open Brace
124		Vertical Line
125	}	Close Brace
126	~	Tilde
127	DEL	Delete



License Agreement

This is a legal agreement between you and SpiderWorks, LLC, a Virginia Limited Liability Corporation, covering your use of this electronic book and related materials (the “Book”). Be sure to read the following agreement before using the Book. BY USING THE BOOK, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT USE THE BOOK AND DESTROY ALL COPIES IN YOUR POSSESSION.

Unauthorized distribution, duplication, or resale of all or any portion of this Book is strictly prohibited. No part of this Book may be reproduced, stored in a retrieval system, shared or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

By using the Book, you acknowledge that the Book and all related products constitute valuable property of SpiderWorks and that all title and ownership rights to the Book and related materials remain exclusively with SpiderWorks. SpiderWorks reserves all rights with respect to the Book and all related products under all applicable laws for the protection of proprietary information, including, but not limited to, intellectual properties, trade secrets, copyrights, trademarks and patents.

The Book is owned by SpiderWorks and is protected by United States copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. Therefore, you must treat the Book like any other copyrighted material. The Book is licensed, not sold. Paying the license fee allows you the right to use one copy of the

Book on your own personal computer. You may not store the Book on a network or on any server that makes the Book available to anyone other than yourself. You may not rent, lease or lend the Book, nor may you modify, adapt, translate, copy, or scan the Book. If you violate any part of this agreement, your right to use this Book terminates automatically and you must then destroy all copies of the Book in your possession.

The Book and any related materials are provided “AS IS” and without warranty of any kind and SpiderWorks expressly disclaims all other warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Under no circumstances shall SpiderWorks be liable for any incidental, special, or consequential damages that result from the use or inability to use the Book or related materials, even if SpiderWorks has been advised of the possibility of such damages. In no event shall SpiderWorks’s liability exceed the license fee paid, if any.

Copyright 2005 SpiderWorks, LLC. “SpiderWorks” is a trademark of SpiderWorks, LLC. Macintosh is a trademark of Apple Computer, Inc. Microsoft Windows is a trademark of Microsoft Corporation. All other third-party names, products and logos referenced within this Book are the trademarks of their respective owners. All rights reserved.