AppleScript Language Guide

For AppleScript 1.3.7

May 5, 1999
Technical Publications

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction

AppleScript is a scripting system that allows you to directly control Macintosh applications, including the Mac OS itself. Instead of using a mouse, keyboard, or other input device to manipulate menus, buttons, and other interface items, you can create sets of written instructions—known as scripts—to automate repetitive tasks, customize applications, and even control complex workflows.

This document is a complete guide to the AppleScript language. It is intended for use with AppleScript version 1.3.4 or later and Mac OS 8.5.1 or later, although some descriptions and examples may work with earlier versions.

This version of the AppleScript Language Guide has been revised to cover new features in AppleScript, to include examples from the Mac OS and the Finder, to improve formatting for online viewing, and to correct errors. For a detailed listing of the changes, see "Document Revision History" (page 391).

This guide should be useful to anyone who wants to write new AppleScript scripts or modify existing scripts. If you are new to AppleScript, however, you might want to start by reading the AppleScript section of the Mac OS Help Center, or by reviewing the introductory materials at the AppleScript website:

<http://www.apple.com/applescript/>

You can also find introductory books on AppleScript at many bookstores and online sites. Macintosh software developers who want to create scriptable applications should refer to *Inside Macintosh: Interapplication Communication*, and to related information available at the Apple Developer website:

<http://www.apple.com/developer/>

This guide describes the AppleScript language in the following chapters:

- "Overview of AppleScript" (page 19) introduces AppleScript and its capabilities and provides an overview of the elements of the AppleScript language.

- "Values and Constants" (page 51) describes AppleScript's value classes, predefined constants, and coercions for converting between value classes.
- "Commands" (page 109) describes the commands available in AppleScript.
- "Objects and References" (page 161) describes objects and their characteristics and explains how to refer to objects in scripts.
- "Expressions" (page 199) describes AppleScript expressions, the operators they use, and how they're evaluated.
- "Control Statements" (page 237) describes statements that control when and how other statements are executed.
- "Handlers" (page 279) describes subroutines, command handlers, error handlers, and the scope of variables and properties in handlers and elsewhere in a script.
- "Script Objects" (page 325) describes how to define and use script objects.
- "The Language at a Glance" (page 349) provides a quick overview of the AppleScript language. It includes a table with links to examples of common scripting tasks.
- "Document Revision History" (page 391) provides a history of changes to this document.

Most sample scripts in this guide demonstrate scriptable features of the Finder, the Mac OS, or applications distributed with the Mac OS, such as the Apple System Profiler. Some examples use AppleWorks, an application suite available from Apple Computer, Inc.

Overview of AppleScript

AppleScript is a dynamic, object-oriented scripting language. Its key feature is the ability to send commands to objects in many different applications, including the Mac OS itself. An **object** is an item, such as a file or folder in a Finder window, a word or paragraph in a text-editing application, or a shape in a drawing application, that can respond to commands by performing actions. AppleScript determines dynamically—that is, whenever necessary—which objects and commands an application recognizes based on information stored in each scriptable application.

In addition to manipulating objects in other applications, AppleScript can store and manipulate its own data, called **values**. Values are simple data structures, such as character strings and real numbers, that can be represented in scripts and manipulated with operators. Values can be obtained from applications or created in scripts.

The building blocks of scripts are **statements**. When you write a script, you compose statements that describe the actions you want to perform. AppleScript provides several kinds of statements that allow you to control when and how statements are executed. These include If statements for conditional execution, Repeat statements for statements that are repeated, and handler definitions for creating user-defined commands.

This chapter provides an overview of AppleScript in the following sections:

- "Conventions Used in This Guide" (page 20) describes conventions you should be familiar with before reading the rest of this guide.
- "What Is AppleScript?" (page 21) provides a brief introduction to AppleScript by answering some frequently asked questions.
- "How AppleScript Works" (page 29) describes the basic mechanisms AppleScript provides to create, compile, and run scripts, and to communicate with and control scriptable applications.

- "Statements" (page 31) provides an overview of the kinds of statements you use to write scripts.
- "Commands and Objects" (page 32) describes the words and phrases you use in AppleScript statements to request actions or results. It also describes the kinds of objects that serve as targets for commands.
- "Dictionaries" (page 34) describes how AppleScript works with applications to determine the words an application understands.
- "Values and Constants" (page 36) describes the types of values and predefined constants AppleScript supports.
- "Expressions" (page 37) describes AppleScript expressions, which are made up of operations and variables.
- "Script Objects" (page 39) describes an advanced feature of AppleScript that lets you define and use objects in scripts.
- "Scripting Additions" (page 40) describes a mechanism for providing AppleScript with additional commands.
- "Dialects" (page 40) describes AppleScript's ability to work with scripts in different representations, though only an English dialect is currently supported.
- "Other Features and Language Elements" (page 41) describes features of the AppleScript scripting language used throughout this guide.

Conventions Used in This Guide

Script samples and script fragments in this guide are accurate as they appear, but you should freely experiment with them and modify them to perform similar tasks or work with different applications.

Glossary terms are shown in **boldface** where they are first defined.

The following conventions are used in syntax descriptions:

`language element`	Plain computer font indicates an element that you type exactly as shown. If there are special symbols (for example, + or &), you also type them exactly as shown.
placeholder	Italic text indicates a placeholder that you replace with an appropriate value. (In some programming languages, placeholders are called nonterminals.)
[optional]	Brackets indicate that the enclosed language element or elements are optional.
(a group)	Parentheses group elements together. If parentheses are part of the syntax, they are shown in bold.
[optional]...	Three ellipsis points (...) after a group defined by brackets indicate that you can repeat the group of elements within brackets 0 or more times.
(a group)...	Three ellipsis points (...) after a group defined by parentheses indicate that you can repeat the group of elements within parentheses one or more times.
a | b | c	Vertical bars separate elements in a group from which you must choose a single element. The elements are often grouped within parentheses or brackets.

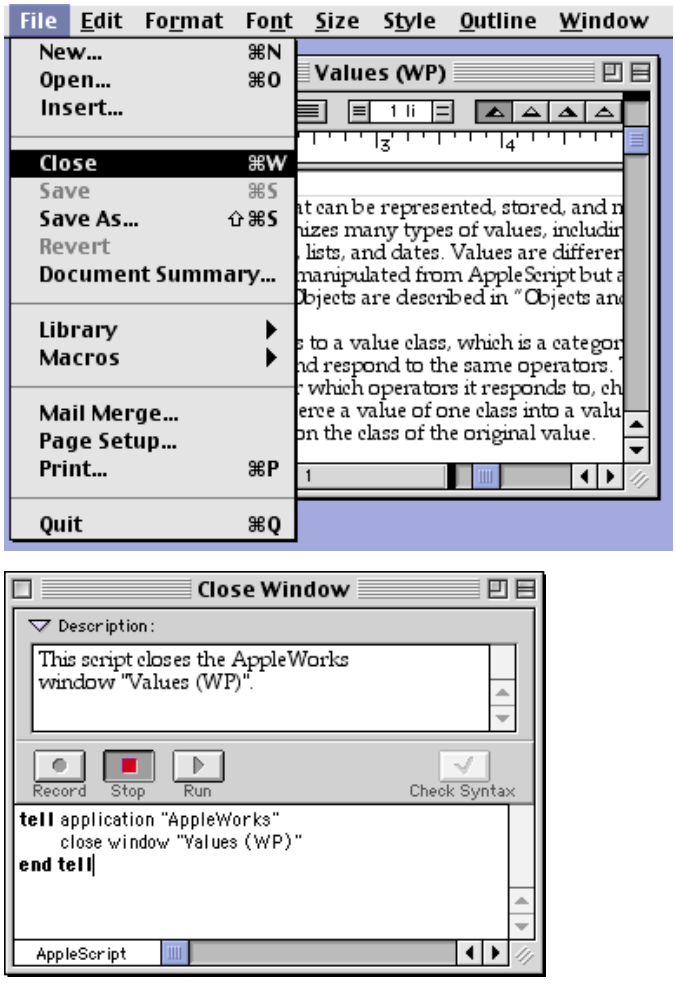
What Is AppleScript?

AppleScript is a scripting language that allows you to control Macintosh computers without using the keyboard or mouse. With AppleScript, you can use a series of English-like instructions, known as a **script,** to control applications, the Finder (or desktop), and many parts of the operating system. For example, Figure 2-1 shows the difference between closing an application window with the mouse and performing the same task with a script.

This guide describes how you can use AppleScript to write scripts. AppleScript shares many features with other scripting, programming, and macro languages. If you've used any of these languages, you'll find AppleScript's English-like terminology easy to learn and use. AppleScript provides the **Script Editor** application to help you create, compile, test, and modify scripts. You can easily create scripts that run as stand-alone applications, as described in "Compiling Scripts With the Script Editor" (page 47). You can also use the Script Editor to

examine the scriptable operations and objects an application supports, as described in "Dictionaries" (page 34). For some applications, you can even record operations into a script to examine how the application implements scriptable features.

Figure 2-1 Closing a window with the mouse and with a script



The following sections provide a brief introduction to the AppleScript scripting language by answering frequently asked questions:

What Makes AppleScript Special?

AppleScript has a number of features that set it apart from both macro programs and scripting languages that control a single program:

- AppleScript uses an English-like terminology that makes it easy for nontechnical users to write scripts. For more experienced users, it is far easier to write powerful scripts with AppleScript and to understand and maintain scripts written by others than it is with a standard programming language.
- You can easily save a script as a stand-alone **script application**—an application whose only function is to run the script associated with it.
- AppleScript makes it easy to refer to data within applications. Scripts can use familiar names to refer to familiar objects. For example, a script can refer to file, disk, and window objects in the Mac OS Finder, or to row, column, and cell objects in a spreadsheet.
- You can control multiple applications from a single script. Although many applications include built-in scripting or macro languages, most of these languages work for only one application. In contrast, you can use AppleScript to control any of the applications that support it. You don't have to learn a new language for each application.
- You can write scripts that control applications on more than one computer. A single script can control any number of applications, and the applications can be on any computer on a given network.

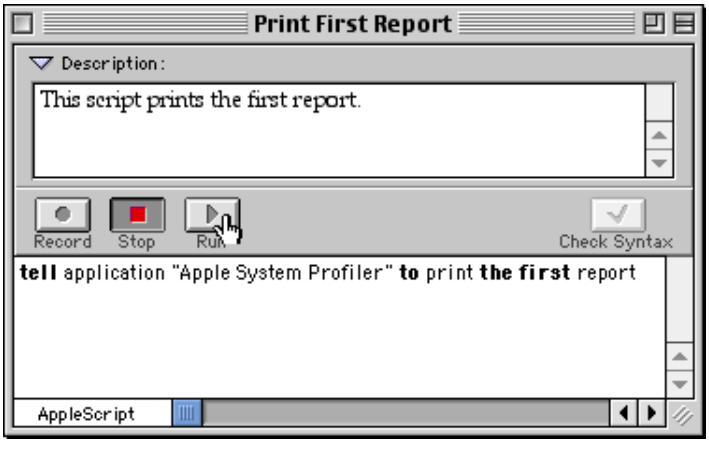
Who Runs Scripts and Who Writes Them?

Anyone who uses a Macintosh computer can run scripts. When you run a script, it performs the operations specified by its statements. Figure 2-2

illustrates two ways to run a script: by double-clicking a script application's icon on the desktop, or by clicking the Run button in the Script Editor window.

Although everyone can run scripts, not everyone needs to write them. One person who is familiar with a scripting language can create sophisticated scripts that many people can use. For example, a scripting specialist in a publishing business can write scripts for everyone in the business to use. Scripts are also sold commercially, included with applications, and distributed over the Internet.

Figure 2-2 Different ways to run a script



What Applications Are Scriptable?

An application that can respond to one or more AppleScript commands is called a **scriptable application.** Not all applications are scriptable, and some applications support only a minimal number of basic commands, such as Open

and Quit. The advertising and packaging for an application usually mention if it is scriptable. The documentation for a scriptable application typically lists the AppleScript words that the application understands. You can also determine if an application is scriptable by attempting to examine its dictionary with the Script Editor application, as described in "Dictionaries" (page 34). Or, for a list of scriptable applications, see the AppleScript website at

<http://www.apple.com/applescript/>

Some scriptable applications are also **recordable.** As you perform actions in a recordable application, the Script Editor can record a series of corresponding instructions in the AppleScript language. To learn how to turn recording on and off, refer to the AppleScript section of the Mac OS Help Center.

Finally, some scriptable applications are also attachable. An **attachable application** is one that can be customized by attaching scripts to specific objects in the application, such as buttons and menu items. The scripts are then triggered by specific user actions, such as choosing the menu item or clicking the button. To determine if an application is attachable, refer to the documentation for the application. For example, the Finder is an attachable application, as described in "Customizing Applications and Automating Workflows" (page 29).

What Can You Do With Scripts?

AppleScript allows you to directly control Macintosh applications, including the Mac OS itself. Instead of using a mouse or other input device to manipulate menus, buttons, and other interface items, you can create sets of written instructions—known as scripts—to automate repetitive tasks, customize applications, and even control complex workflows.

See the following sections for information on

- "Automating Activities" (page 25)
- "Integrating Applications" (page 27)
- "Customizing Applications and Automating Workflows" (page 29)

Automating Activities

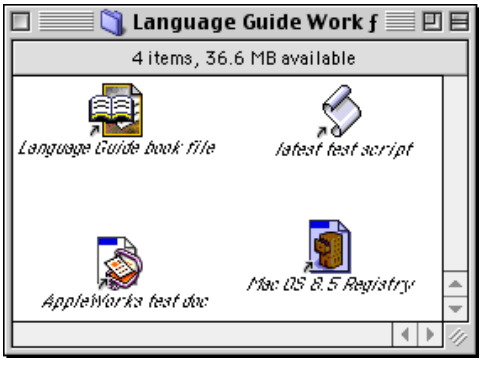
Scripts make it easy to perform repetitive tasks. For example, suppose you are revising a document such as the AppleScript Language Guide. Every time you

sit down to work on the Language Guide, you need to open the following documents:

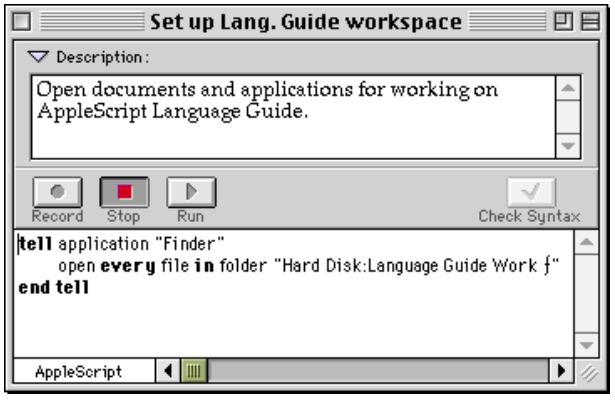
- the FrameMaker file "Language Guide book file," which contains the Language Guide text documents
- the Script Editor script "latest test script," which contains sample scripts to test
- the AppleWorks document "AppleWorks test doc," which you use for testing scripts that open and close documents and windows and work with words and paragraphs
- the FileMaker Pro document "Mac OS 8.5 Registry" (distributed on the AppleScript SDK for developers), which provides information about the scripting terminology available for AppleScript and for scriptable parts of the Mac OS

Instead of individually opening each of these documents, you can put an alias to each document in a folder, then write a simple script that opens the files for you. Figure 2-3 shows the folder containing aliases to your documents and a script that opens them.

Figure 2-3 A script that automates a frequently performed set of steps



Aliases to files used while revising the AppleScript Language Guide

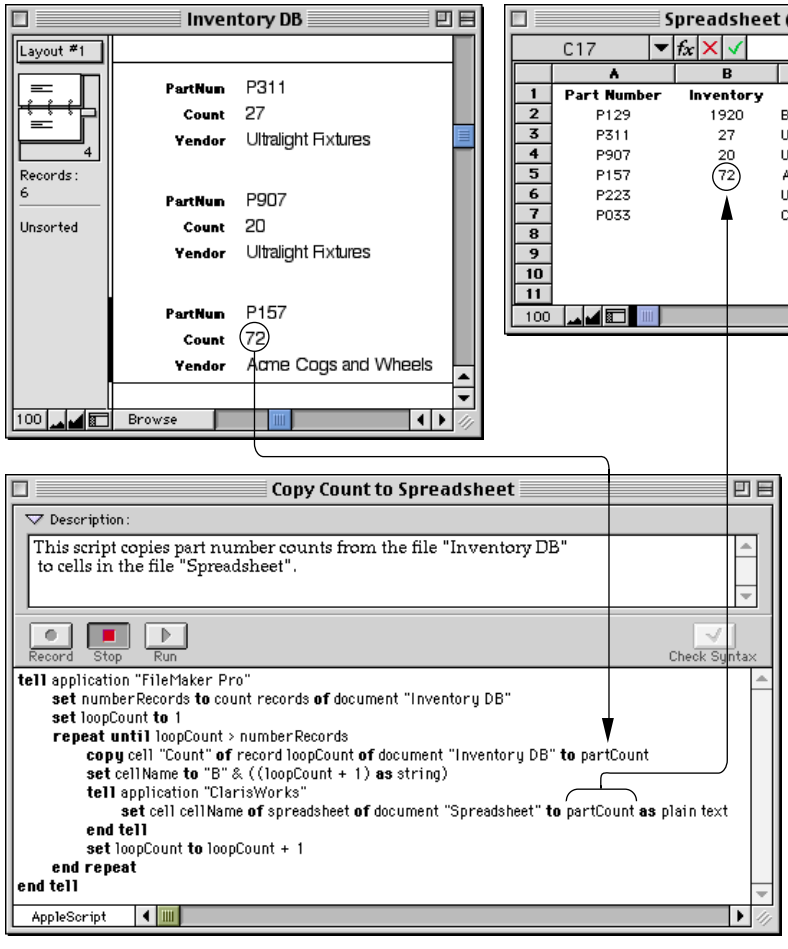


Script that opens work files and applications

Integrating Applications

Scripts are ideal for performing tasks that involve more than one application. A script can send instructions to one application, get the resulting data, and then pass the data on to one or more additional applications. For example, a script can collect information from a database application and copy it to a spreadsheet application. Figure 2-4 shows a simple script that gets a value from the Count cell of an inventory database and copies it to the Inventory column of a spreadsheet. This script is also shown in “Repeat Until” (page 253).

Figure 2-4 A script that copies information from one application to another



In the same way, a script can use one application to perform an action on data from another application. For example, suppose a word-processing application includes a spelling checker and also supports an AppleScript command to check spelling. You can check the spelling of a block of text from any other application by writing a script that sends the AppleScript command and the text to be checked to the word-processing application, which returns the results to the application that runs the script.

If an action performed by an application can be controlled by a script, that action can be also performed from the Script Editor or from any other application that can run scripts. Every scriptable application is potentially a toolkit of useful utilities that can be selectively combined with utilities from other scriptable applications to perform highly specialized tasks.

Customizing Applications and Automating Workflows

Scripts can add new features to applications. To customize an application, you add a script that is triggered by a particular action within the application, such as choosing a menu item or clicking a button. For example, you can customize the Finder by attaching folder action scripts to folders. An attached folder action script is triggered automatically when you open or close a folder, add or remove items from an open folder, or move or resize a folder window.

You can use folder action scripts to perform simple tasks, such as automatically copying files to a backup folder, or more complex tasks, such as automating a workflow that uses multiple applications to perform operations on a graphics file dropped in a folder. For more information on folder actions, see the AppleScript section of the Mac OS Help Center.

Not all applications support adding scripts. The ones that do are responsible for specifying the ways you associate scripts with specific actions.

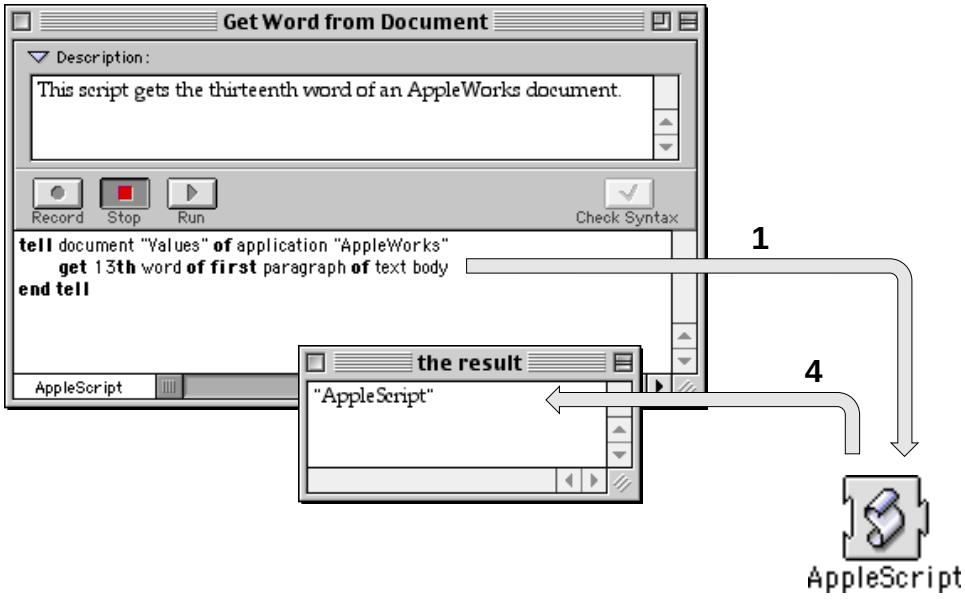
How AppleScript Works

AppleScript works by sending messages, called **Apple events,** to applications. When you write a script, you write one or more groups of instructions called statements. When you run the script, the Script Editor sends these statements to the AppleScript extension, which interprets the statements and sends Apple events to the appropriate applications. Figure 2-5 shows the relationship between the Script Editor, the AppleScript extension, and the application.

Applications respond to Apple events by performing actions, such as inserting text, getting a value, or opening a document. Applications can also send Apple events back to the AppleScript extension to report results. The AppleScript extension sends the final results to the Script Editor, where they are typically displayed in the result window. (You can use the Show Result command to open the result window.)

Figure 2-5 How AppleScript works

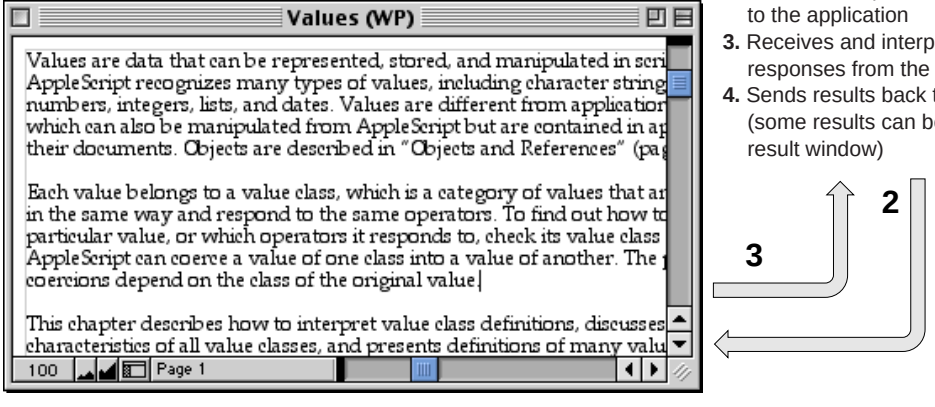
Script Editor
- Compiles and runs scripts
- Displays results



AppleScript extension

1. Interprets statements from the Script Editor
2. Sends corresponding Apple events to the application
3. Receives and interprets Apple event responses from the application
4. Sends results back to the Script Editor (some results can be displayed in the result window)

Application
- Responds to Apple events by performing actions
- Sends Apple events to AppleScript extension



Most scripters don't need to be concerned about Apple events or the AppleScript extension. You just use the AppleScript language to request the actions or results that you want. To find out what scriptable actions an application provides, see "Dictionaries" (page 34).

Statements

Every script is a series of statements. Statements are structures similar to sentences in human languages that contain instructions for AppleScript to perform. When AppleScript runs a script, it reads the statements in order and carries out their instructions. Some statements cause AppleScript to skip or repeat certain instructions or change the way it performs certain tasks. These statements, which are described in Chapter 7, are called **control statements**.

All statements, including control statements, fall into one of two categories: simple statements or compound statements. **Simple statements** are statements such as the following that are written on a single line.

```
tell application "Finder" to close the front window
```

Compound statements are statements that are written on more than one line and contain other statements. All compound statements have two things in common: they can contain any number of statements, and they have the word `end` (followed, optionally, by the first word of the statement) as their last line. The simple statement above is equivalent to the following compound statement.

```
tell application "Finder"
    close the front window
end tell
```

The compound Tell statement includes the lines `tell application "Finder"` and `end tell`, and all statements between those two lines.

A compound statement can contain any number of statements. For example, here is a Tell statement that contains two statements:

```
tell application "Finder"
    set windowName to name of front window
    close front window
end tell
```

This example illustrates the advantage of using a compound Tell statement: you can add additional statements within a compound statement.

Notice that the previous example contains the statement `close front window` instead of `close the front window`. AppleScript allows you to add or remove the word `the` anywhere in a script without changing the meaning of the script. You can use the word `the` to make your statements more English-like and therefore more readable.

Here's another example of a compound statement:

```
if the name of the front window is "Fred" then
    close front window
end if
```

Statements contained in a compound statement can themselves be compound statements. Here's an example:

```
tell application "Finder"
    if the name of the front window is "Fred" then
        close front window
    end if
end tell
```

Commands and Objects

Commands are the words or phrases you use in AppleScript statements to request actions or results. Every command is directed at a target, which is the object that responds to the command. The target of a command is usually an application object. **Application objects** are objects that belong to an application, such as windows, or objects in documents, such as the words and paragraphs in a text document. Commands can also be targeted at **system objects**, which specify objects that belong to the Mac OS, such as a desktop printer, a user or

group object from the Users & Groups control panel, or a theme object from the Appearance control panel.

Each application or system object has specific information associated with it and can respond to specific commands.

For example, in the Finder, window objects understand the Close command. The following example shows how to use the Close command to request that the Finder close the front window.

```
tell application "Finder"
    close the front window
end tell
```

The Close command is contained within a Tell statement. Tell statements specify default targets for the commands they contain. The **default target** is the object that receives commands if no other object is specified or if the object is specified incompletely in the command. In this case, the statement containing the Close statement does not contain enough information to uniquely identify the window object, so AppleScript uses the application name listed in the Tell statement to determine which object receives the Close command.

In AppleScript, you use references to identify objects. A **reference** is a compound name, similar to a pathname or address, that specifies an object. For example, the following phrase is a reference:

```
front window of application "Finder"
```

This phrase specifies a window object that belongs to a specific application. (The application itself is also an object.) AppleScript has different types of references that allow you to specify objects in many different ways. You'll learn more about references in Chapter 5, "Objects and References."

Objects can contain other objects, called **elements.** In the previous example, the front window is an element of the Finder application object. Similarly, in the next example, a file element is contained in a specific folder element, which is contained in a specific disk. You can read more about references to Finder objects in Chapter 5, "Objects and References."

```
file 1 of folder 1 of startup disk
```

Every object belongs to an **object class,** which is simply a name for objects with similar characteristics. Among the characteristics that are the same for the

objects in a class are the commands that can act on the objects and the elements they can contain. An example of an object class is the Folder object class in the Finder. Every folder visible in the Finder belongs to the Folder object class. The Finder's definition of the Folder object class determines which classes of elements, such as files and folders, a folder object can contain. The definition also determines which commands, such as the Close command, a folder object can respond to.

Dictionaries

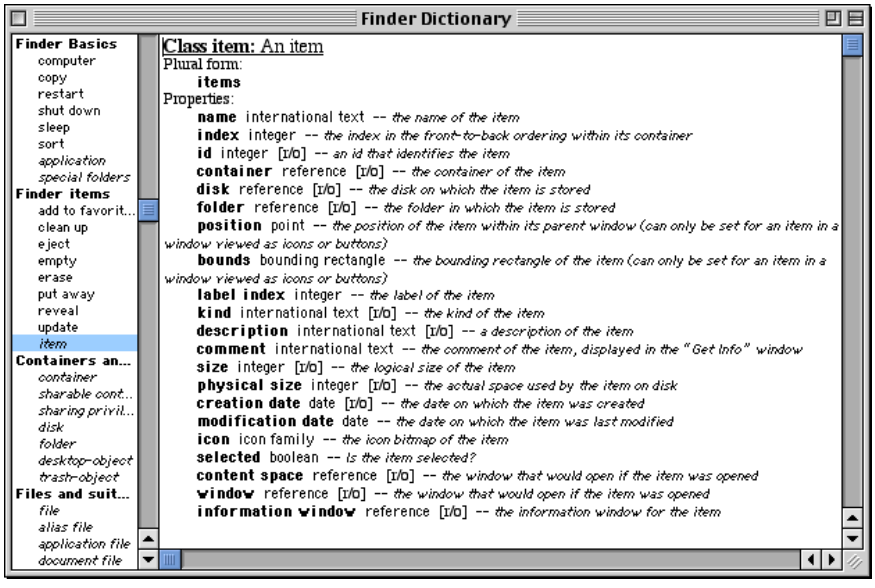
To examine a definition of an object class, a command, or some other word supported by an application, you can open that application's dictionary from the Script Editor. A **dictionary** is a set of definitions for words that are understood by a particular application. Unlike other scripting languages, AppleScript does not have a single fixed set of definitions for use with all applications. Instead, when you write scripts in AppleScript, you use both definitions provided by AppleScript and definitions provided by individual applications to suit their capabilities.

Dictionaries tell you which objects are available in a particular application and which commands you can use to control them. You can view an application's dictionary by dropping the application's icon on the Script Editor's icon, or by opening the application with the Script Editor's Open Dictionary command. Figure 2-6 shows the Finder's dictionary, with the Item class displayed. For more information on using the Script Editor, refer to the AppleScript section of the Mac OS Help Center.

To use the words from an application's dictionary in a script, you must indicate which application you want to manipulate. You can do this with a Tell statement that lists the name of the application:

```
tell application "Finder"
    clean up the front window
end tell
```

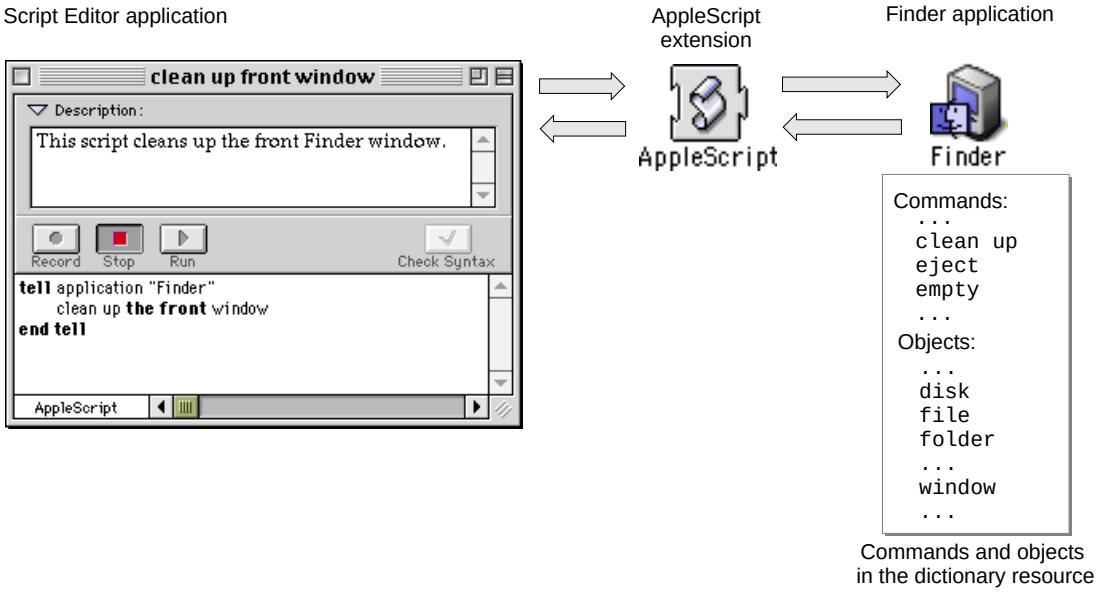
Figure 2-6 The Finder's dictionary, with Item class displayed



When it encounters a Tell statement, AppleScript reads the words in the specified application's dictionary and uses them to interpret the statements in the Tell block. For example, AppleScript uses the words in the Finder dictionary to interpret the Clean Up command in the previous script sample.

When you use a Tell statement or specify an application name completely in a statement, the AppleScript extension gets the dictionary resource for the application and reads its dictionary of commands, objects, and other words. Every scriptable application has a dictionary resource (of resource type `'aete'`) that defines the commands, objects, and other words script writers can use in scripts to control the application. Figure 2-7 shows how AppleScript gets the words in the Finder's dictionary.

Figure 2-7 How the Script Editor accesses the Finder's dictionary



In addition to the terms defined in application dictionaries, AppleScript includes its own standard terms. Unlike the terms in application dictionaries, the standard AppleScript terms are always available. You can use these terms (such as If, Tell, and First) anywhere in a script. This guide describes the standard terms provided by AppleScript.

The words in system and application dictionaries are known as **reserved words.** When defining new words for your script—such as identifiers for variables—you cannot use reserved words.

Values and Constants

A *value* is a simple data structure that can be represented, stored, and manipulated within AppleScript. AppleScript recognizes many types of values, including character strings, real numbers, integers, lists, and dates. Values are fundamentally different from application objects, which can be manipulated

from AppleScript, but are contained in applications or their documents. Values can be created in scripts or returned as results of commands sent to applications.

Values are an important means of exchanging data in AppleScript. When you request information about application objects, it is usually returned in the form of values. Similarly, when you provide information with commands, you typically supply it in the form of values.

A fixed number of specific types of values are recognized by AppleScript. You cannot define additional types of values, nor can you change the way values are represented. The different types of AppleScript values, called value classes, are described in Chapter 3, "Values and Constants."

A constant is a reserved word with a predefined value. AppleScript provides constants to help your scripts perform a variety of tasks, such as retrieving information about an object's properties, performing comparisons, and performing arithmetic operations. Chapter 3, "Values and Constants," describes AppleScript's constants.

Expressions

An **expression** is a series of AppleScript words that corresponds to a value. Expressions are used in scripts to represent or derive values. When you run a script, AppleScript converts its expressions into values. This process is known as **evaluation.**

Two common types of expressions are operations and variables. An **operation** is an expression that derives a new value from one or two other values. A **variable** is a named container in which a value is stored. The following sections introduce operations and variables.

- "Operations" (page 38)
- "Variables" (page 38)

For more information about these and other types of expressions, see Chapter 6, "Expressions."

Operations

The following are examples of AppleScript operations and their values. The value of each operation is listed following the comment characters (--).

```
3 + 4                          --value: 7

(12 > 4) and (12 = 4)          --value: false
```

Each operation contains an **operator.** The plus sign (+) in the first expression, as well as the greater than symbol (>), the equal symbol (=) symbol, and the word `and` in the second expression, are operators. Operators transform values or pairs of values into other values. Operators that operate on two values are called **binary operators.** Operators that operate on a single value are known as **unary operators.** Chapter 6, "Expressions," contains a complete list of the operators AppleScript supports and the rules for using them.

You can use operations within AppleScript statements, such as:

```
tell application "Finder"
    open folder (3 + 2) of startup disk
end tell
```

When you run this script, AppleScript evaluates the expression (`3 + 2`) and uses the result to tell the Finder which folder to open.

Variables

When AppleScript encounters a variable in a script, it evaluates the variable by getting its value. To create a variable, simply assign it a value:

```
copy "Mark" to myName
```

The Copy command takes the data—the string `"Mark"`—and puts it in the variable `myName`. You can accomplish the same thing with the Set command:

```
set myName to "Mark"
```

Statements that assign values to variables are known as **assignment statements.**

You can retrieve the value in a variable with a Get command. Run the following script and then display the result in the Script Editor's result window. (You can use the Show Result command to open the result window.)

```
set myName to "Mark"
get myName
```

The result window shows that the value in `myName` is `"Mark"`, the value you stored with the Set command.

You can change the value of a variable by assigning it a new value. A variable can hold only one value at a time. When you assign a new value to an existing variable, you lose the old value. For example, the result of the Get command in the following script is `"Robin"`.

```
set myName to "Mark"
set myName to "Robin"
get myName
```

AppleScript does not distinguish uppercase letters from lowercase variables in variable names; the variables `myName`, `myname`, and `MYNAME` all represent the same value. For related information, see "Case Sensitivity" (page 45).

Script Objects

Script objects are objects you define and use in scripts. Like application objects, script objects respond to commands and have specific information associated with them. Unlike application objects, script objects are defined in scripts.

Script objects are an advanced feature of AppleScript. They allow you to use object-oriented programming techniques to define new objects and commands. Information contained in script objects can be saved and used by other scripts. For information about defining and using script objects, see Chapter 9, "Script Objects." You should be familiar with the concepts in the rest of this guide before attempting to use script objects.

Scripting Additions

Scripting additions are files that provide additional commands or coercions you can use in scripts. A scripting addition file must be located in the Scripting Additions folder (in the System Folder) for AppleScript to recognize the additional commands it provides.

A single scripting addition file can contain several commands. For example, the Standard Additions scripting addition distributed with AppleScript, includes commands for using the Clipboard, obtaining the path to a file, speaking and summarizing text, and more.

Unlike other commands used in AppleScript, scripting addition commands work the same way regardless of the target you specify. For example, the Beep command, which is part of the standard additions, triggers an alert sound no matter which application you sent the command to.

A **coercion** is software that converts a value from one class to another, such as from an integer value to a real value. Many standard coercions are built in to AppleScript.

For information about the standard additions distributed with AppleScript, see the AppleScript section of the Mac OS Help Center, or see the AppleScript website at

<http://www.apple.com/applescript/>

Dialects

AppleScript is designed so that scripts can be displayed in several different **dialects,** which are representations of AppleScript that resemble human languages or programming languages. However, the English dialect is currently the only dialect supported. The English Dialect file is stored in the Dialects folder, a folder in the Scripting Additions folder in the System Folder.

IMPORTANT
For versions 1.3.7 and earlier of AppleScript, if you move or delete the dialect file or a folder that includes it, AppleScript will not work. ▲

There is currently no plan to support dialects other than English.

Other Features and Language Elements

Previous sections have introduced the key elements of the AppleScript language, including statements, objects, commands, expressions, and script objects. Later chapters in this guide discuss these elements in more detail and describes how to use them in scripts. Before you continue, however, you'll need to know about a few additional elements and features of the AppleScript scripting language that are described in the following sections:

Continuation Characters

A simple AppleScript statement must normally be on a single line. If a statement is longer than will fit on one line, you can extend it by including a **continuation character**, ¬, at the end of one line and continuing the statement on the next. You can enter this character in most text-editing applications by typing Option-L. In the Script Editor, you can also type Option-Return, which inserts the continuation character and moves the insertion point to the next line.

The following statement

```
open the second file of the first folder of the startup disk
```

can appear on two lines:

```
open the second file of the first folder ¬
    of the startup disk
```

IMPORTANT

This document frequently uses the continuation character for sample statements that don't fit on one line on a document page. It also uses the continuation character in some syntax statements to identify an item that, if included, must appear on the same line as the previous item. The continuation character itself is not a required part of the syntax—it is merely a mechanism for including multiple lines in one statement. ▲

The only place a continuation character does not work is within a string. For example, the following statement causes an error, because AppleScript interprets the two lines as separate statements.

```
--this statement causes an error:
open the second file of the first folder of disk "Hard ¬
    Disk"
```

The two dashes (`--`) in the previous example indicate that the first line is a comment. A comment is text that is ignored by AppleScript when a script is run. You add comments to help explain your scripts. For more information, see "Comments" (page 43).

To use a very long string, you can either continue typing without pressing Return, or you can break the string into two or more strings and use the concatenation operator (&) to join them, as in the following example:

```
open the second file of the first folder of disk "Hard" ¬
    & "Disk"
```

For more information about the concatenation operator, see Chapter 6, "Expressions."

Note that to compile any of the statements in this section, you must enclose them in a Tell block, such as the following:

```
tell application "Finder"
    open the second file of the first folder of the startup disk
end tell
```

Comments

You add comments to a script to explain what the script does, you add comments. A **comment** is text that remains in a script after compilation but is ignored by AppleScript when the script is executed. There are two kinds of comments:

- A block comment begins with the characters `(*` and ends with the characters `*)`. Block comments must be placed between other statements. They cannot be embedded in simple statements.
- An end-of-line comment begins with the characters `--` and ends with the end of the line.

You can nest comments, that is, comments can contain other comments.

Here are some sample comments:

```
--end-of-line comments extend to the end of the line;

(* Use block comments for comments that occupy
more than one line *)

copy result to theCount --stores the result in theCount

(* The following subroutine, findString, searches for a string in a list
of AppleWorks word processing files *)

(* Here are examples of
    --nested comments
    (* another comment within a comment *)
*)
```

The following block comment causes an error because it is embedded in a statement.

```
--the following block comment is illegal
tell application "Finder"
    get (* name of *) file 1 of startup disk
end tell
```

Because comments are not executed, you can prevent parts of scripts from being executed by putting them between comment characters. You can use this trick, known as "commenting out," to isolate problems when debugging scripts or temporarily block execution of any parts of a script that aren't yet finished. Here's an example of "commenting out" an unfinished handler:

```
(*
on finish()
   --under construction
end
*)
```

If you later remove "`(*`" and "`*)`", the handler is once again available. Handlers are described in "Handlers" (page 279).

Identifiers

An **identifier** is a series of characters that identifies a value or other language element. For example, variable names are identifiers. The following statement sets the value of the variable `myName` to `"Fred"`.

```
set myName to "Fred"
```

Identifiers are also used as labels for properties and handlers. You'll learn about these uses later in this guide.

An identifier must begin with a letter and can contain uppercase letters, lowercase letters, numerals (0–9), and the underscore character (_). Here are some examples of valid identifiers:

```
Yes
Agent99
Just_Do_It
```

The following are not valid identifiers:

```
C--
Back&Forth
999
Why^Not
```

Identifiers whose first and last characters are vertical bars (|) can contain any characters. For example, the following are legal identifiers:

```
|Back and Forth|
|Right*Now!|
```

Identifiers whose first and last characters are vertical bars can contain additional vertical bars if the vertical bars are preceded by backslash (\) characters, as in the identifier `|This\|Or\|That|`. A backslash character in an identifier must be preceded by a backslash character, as in the identifier `|/ \\ Up \\/ Down|`.

AppleScript identifiers are not case sensitive. For example, the variable identifiers `myvariable` and `MyVariable` are equivalent.

Identifiers cannot be the same as any reserved words—that is, words in the AppleScript language or words in the dictionary of an application named in a Tell statement. For example, you cannot create a variable whose identifier is `file` within a Tell statement to the Finder, because `file` is an object class in the Finder dictionary. In this case, AppleScript returns a syntax error if you use `file` as a variable identifier.

Case Sensitivity

AppleScript is not case sensitive; when it interprets statements in a script, it does not distinguish uppercase from lowercase letters. This is true for all elements of the language.

The one exception to this rule is string comparisons. Normally, AppleScript does not distinguish uppercase from lowercase letters when comparing strings, but if you want AppleScript to consider case, you can use a special statement called a Considering statement. For more information, see “Considering and Ignoring Statements” (page 268).

Most of the examples in this chapter and throughout this guide are in lower-case letters. Sometimes words are capitalized to improve readability. For example, in the following variable assignment, the “N” in `myName` is capitalized

to make it easier to see that two words have been combined to form the name of the variable.

```
set myName to "Robin"
```

After you create the variable `myName`, you can refer to it by any of these names:

```
MYNAME
myname
MyName
mYName
```

However, when you first compile a script that capitalizes a variable name in different ways, AppleScript will convert the capitalization of all occurrences of the variable to match the capitalization of the first occurrence.

Although this guide uses variable names that start with a lower case letter and have initial upper case letters for subsequent words in the name (`myFileName`), some scripters prefer the form `my_file_name`.

When interpreting strings, such as `"Robin"`, AppleScript preserves the case of the letters in the string, but does not use it in comparisons unless directed to do so by a Considering statement. For example, the value of the variable `myName` defined earlier is `"Robin"`, but the value of the expression `myName = "ROBIN"` is `true`.

Abbreviations

The AppleScript language is designed to be intuitive and easy to understand. To this end, it uses familiar words to represent objects and commands and uses statements whose structure is similar to English sentences. For the same reason, it typically uses real words instead of abbreviations. In a few cases, however, AppleScript supports abbreviations for long and frequently used words.

One important example is the abbreviation `app`, which you can use to refer to objects of class application. This is particularly useful in Tell statements. For example, the following two Tell statements are equivalent:

```
tell application "AppleWorks"
    print the front window
end tell
```

```
tell app "AppleWorks"
    print the front window
end tell
```

Compiling Scripts With the Script Editor

When you create or modify a script and then attempt to run or save it as a compiled script or script application, the Script Editor asks AppleScript to compile the script first. To **compile** a script, AppleScript converts the script from the form typed into a Script Editor window (or any script-editing window) to an internal form that AppleScript can execute. The Script Editor also attempts to compile the script when you click the Check Syntax button.

If the script compiles successfully, the Script Editor dims the Check Syntax button and reformats the text of the script according to the preferences set with the Formatting command (in the Edit menu). This may cause indentation and spacing to change, and in some cases may even change the text, but it doesn't affect the meaning of the script. If AppleScript can't compile the script because of syntax errors or other problems, the Script Editor displays a dialog box describing the error or, if you are trying to save the script, allowing you to save the script as a text file only.

For additional information on compiling, see "Double Angle Brackets in Results and Scripts" (page 123).

Debugging Scripts

This section describes several simple techniques you can use to help test and debug scripts. If you write large or complicated scripts, however, you should investigate a commercial script-debugging application from an independent software developer. For information on script debuggers, see the AppleScript website at

<http://www.apple.com/applescript/>

For debugging less complex scripts, try the following techniques:

- Examine the result window to see if the script is producing the desired result. For more information, see "Using Results" (page 121).
- Comment out certain lines in the script until the rest of the script is working correctly. For more information, see "Comments" (page 43).

- Insert Say scripting addition commands to narrate the progress of a script. The Say command speaks the specified text.

```
say "Starting to empty the trash."
tell application "Finder"
    empty trash
end tell
say "Finished emptying the trash."

if fileName is not equal to "Expected FileName" then
    say "Unexpected file name."
end if
```

- Insert a Display Dialog command (another scripting addition command) to display information at a certain point in a script. For an example, see "Version Constant" (page 106).
- Open the Script Editor's Event Log window to display diagnostic information while a script is running. The following paragraphs describe how to use the Event Log.

The Event Log helps you discover and correct errors by showing the results of a script's actions. You open the Script Editor's Event Log window from the Controls menu or by typing Command-E. The window contains two checkboxes. If you check Show Events, all Apple events are logged to the window. If you also check Show Event Results, the value returned from an Apple event is also displayed. (Results will not be returned unless both checkboxes are checked.)

In addition to simply opening the Event Log to view the results of actions taken by your script, you can insert `log` statements at strategic locations in your script. A **log statement** reports the value of one or more variables to the Event Log window.

Suppose you run the following script:

```
tell application "Finder"
    set myFolder to first folder of startup disk
    log (myFolder) --result: (*Claris Emailer Folder*)
end tell
```

With no checkboxes checked, the Event Log window contains just the result of the `log (myFolder)` statement:

```
(*Claris Emailer Folder*)
```

With just the Show Events checkbox checked, the Event Log window contains

```
tell application "Finder"
    get folder 1 of startup disk
    (*Claris Emailer Folder*)
end tell
```

Finally, with both checkboxes checked, the Event Log window contains

```
tell application "Finder"
    get folder 1 of startup disk
        --> folder "Claris Emailer Folder" of startup disk
    (*Claris Emailer Folder*)
end tell
```

In this case, the `log` statement isn't really needed because it returns the same information displayed by checking the Show Event Results checkbox.

You can use `start log` and `stop log` statements to exert finer control over event logging. When the Show Events checkbox is checked, the "log level" count is set to 1. Whenever the log level is greater than 0, the Event Log window shows events. A `start log` statement increments the log level. A `stop log` statement decrements it. If the Show Events checkbox is checked, the following script turns off logging before getting the name of the first folder and turns it on again afterwards:

```
tell application "Finder"
    stop log
    set nameOne to name of first folder of startup disk
    start log
    set nameTwo to name of second folder of startup disk
end tell
```

Log statements can be especially useful when testing a Repeat loop or other control statement. In the following script, the statement `log currentWord` causes the current word to be displayed in the Script Editor's Event Log window each

time through the loop. Once the loop is working correctly, you can comment out or delete the log statement.

```
set wordList to words in "Where is the hammer?"
    --result: {"Where", "is", "the", "hammer"}
repeat with currentWord in wordList
    log currentWord
    if currentWord as text is equal to "hammer" then
        display dialog "I found the hammer!"
    end if
end repeat
```

This script examines a list of words with the *Repeat With (loopVariable) In (list)* form of the Repeat statement, displaying a dialog if it finds the word "hammer" in the list. For more information, see "Repeat With (loopVariable) In (list)" (page 256).

For more information on the Script Editor's Event Log window, see the AppleScript section of the Mac OS Help Center.

Values and Constants

Values are data that can be represented, stored, and manipulated in scripts. AppleScript recognizes many types of values, including character strings, real numbers, integers, lists, and dates. Values are different from application objects, which can also be manipulated from AppleScript but are contained in applications or their documents. Objects are described in "Objects and References" (page 161).

Each value belongs to a **value class**, which is a category of values that are represented in the same way and respond to the same operators. To find out how to represent a particular value, or which operators it responds to, check its value class definition. AppleScript can coerce a value of one class into a value of another. The possible coercions depend on the class of the original value.

This chapter describes how to interpret value class definitions, discusses the common characteristics of all value classes, and presents definitions of many value classes supported in AppleScript. It also describes how to coerce values and describes various types of constant values you can use in scripts.

Value classes and coercions are described in the following sections:

- "Common Value Class Definitions" (page 56) provides detailed definitions for a set of commonly used AppleScript value classes. Each class is described in a separate section. The sections follow the format described in "Using Value Class Definitions" (page 52).
- "Unicode Text and International Text" (page 87) describes two string value classes that you use mainly to transfer data between applications.
- "Unit Type Value Classes" (page 91) provides an overview of the value classes you use to work with measurements of length, area, cubic and liquid volume, mass, and temperature.
- "Coercing Values" (page 97) describes how AppleScript coerces values. It provides a table that shows the available coercions for common value classes.

A **constant** is a reserved word with a predefined value. You can use constants to supply values in scripts. Constants are described in the following sections:

- "Constant" (page 60) describes the value class Constant.
- "Constants" (page 100) describes the different kinds of constants used with AppleScript.

Using Value Class Definitions

Value class definitions contain information about values that belong to a particular class. All value classes fall into one of two categories: **simple values,** such as integers and real numbers, which do not contain other values, or **composite values,** such as lists and records, which do. Each of the value class definitions in this chapter provides information in one or more of the following categories:

- "Literal Expressions" (page 52)
- "Properties" (page 53)
- "Elements" (page 54)
- "Operators" (page 54)
- "Commands Handled" (page 55)
- "Reference Forms" (page 55)
- "Coercions Supported" (page 55)

In addition, some value class definitions end with notes that provide additional information. For an example of a definition that includes each of these categories, see "List" (page 67).

Literal Expressions

A **literal expression** is an expression that evaluates to itself. The "Literal Expressions" section of a value class definition provides examples of how values of a particular class are represented in AppleScript—that is, typical literal expressions for values of that class. For example, in AppleScript and many other programming languages, the literal expression for a string is a

series of characters enclosed in quotation marks. The quotation marks are not part of the string value; they are a notation that indicates where the string begins and ends. The actual string value is a data structure stored in AppleScript.

The following example, from the definition for the List value class, shows a literal expression for a list value, which is a composite-value type.

```
{ "it's", 2, true }
```

As with the quotation marks in a string literal expression, the braces that enclose the list and the commas that separate its items are not part of the actual list value; they are notations that represent the grouping and items of the list. The list class is described fully in "List" (page 67).

Properties

A **property** of a value class is a characteristic that is identified by a unique label and has a single value. The "Properties" section of a value class definition describes the property or properties of the class. Simple values have only one property, called Class, which identifies the class of the value. Composite values have both a Class property and at least one additional property, such as Length or Contents.

For example, the value class Boolean, described in "Boolean" (page 58), is a simple class with just one property, the read-only Class property.

```
class of boolean --result: class
```

However, the class Date, described in "Date" (page 62), is a composite value class that has additional properties. The following example uses the standard scripting addition command Current Date to get the current date, then gets various properties of the date.

```
set theDate to current date
--result: date "Wednesday, March 3, 1999 3:01:44 PM"
weekday of theDate --result: Wednesday
day of theDate --result: 3 (the day of the month)
```

The following example specifies the Length property of a simple list.

```
length of {"This", "list", "has", 5, "items"} --result: 5
```

You can optionally use the Get command to get the value of a specified property. For example:

```
get class of boolean --result: class
```

In most cases, you can also use the Set command to set the additional properties listed in the definitions of composite values. If a property cannot be set with the Set command, its definition specifies that it is read-only.

Elements

Elements of values are values contained within other values. Composite values have elements; simple values do not. For example, the List value class definition, shown in “List” (page 67), contains one element, called an item.

You use references to refer to elements of composite values. For example, the following reference specifies the third item in a list.

```
item 3 of {"To", "be", "great", "is", "to", "be", "misunderstood"}
--result: "great"
```

The “Reference Forms” section of a composite value class definition lists the reference forms you can use to specify elements of composite values.

Operators

You use operators, such as the addition operator (+), the concatenation operator (&), and the equality operator (=), to manipulate values. Values that belong to the same class can be manipulated by the same operators. The “Operators” section of a value class definition lists the operators that can be used with values of a particular class.

For complete descriptions of operators and how to use them in expressions, see “Operations” (page 213).

Commands Handled

Commands are the words or phrases you use in AppleScript statements to request actions or results. Simple values cannot respond to commands, but composite values can. For example, lists can respond to the Count command, as shown in the following example.

```
count {"This", "list", "has", 5, "items"}
--result: 5
```

Each composite value class definition includes a "Commands Handled" section that lists commands to which values of that class can respond.

Reference Forms

A **reference** is a compound name, similar to a pathname or address, that specifies an object or a value. You can use references to specify values within composite values or properties of simple values. You cannot use references to refer to simple values.

The "Reference Forms" section is included in composite value class definitions only. It lists the reference forms you can use to specify elements of a composite value. For complete descriptions of the AppleScript reference forms, see "Objects and References" (page 161)

Coercions Supported

AppleScript can change a value of one class into a value of another class. This is called **coercion.** The "Coercions Supported" section of a value class definition describes the classes to which values of that class can be coerced. For example, the coercions section for the value class List, described in "List" (page 67), notes that any value can be added to a list or coerced to a single-value list.

For more information about coercions, see "Coercing Values" (page 97). For a summary of the coercions supported for commonly used value classes, see Figure 3-3 (page 99).

Common Value Class Definitions

Table 3-1 summarizes common AppleScript value classes and provides links to sections that describe them in detail.

Table 3-1 Common AppleScript value class identifiers

Value class identifier	Description of corresponding value
"Boolean" (page 58)	A logical truth value
"Class" (page 59)	A class identifier
"Constant" (page 60)	A reserved word defined by an application or AppleScript
"Data" (page 61)	Raw data that cannot be represented in AppleScript, but can be stored in a variable
"Date" (page 62)	A string that specifies a day of the week, day of the month, month, year, and time
"File Specification" (page 95)	A collection of data that specifies the name and location on disk of a file that may not yet exist
"Integer" (page 66)	A positive or negative number without a fractional part
International Text	Character data in the form of international text; see "Unicode Text and International Text" (page 87)
"List" (page 67)	An ordered collection of values
"Number" (page 71)	Synonym for either class Integer or class Real
"Real" (page 72)	A positive or negative number that can have a fractional part
"Record" (page 74)	A collection of properties
"Reference" (page 77)	A reference to an object

Table 3-1 Common AppleScript value class identifiers (continued)

Value class identifier	Description of corresponding value
"RGB Color" (page 96)	A collection of three integer values that specify the red, green, and blue components of a color
"String" (page 80)	An ordered series of 1-byte characters
"Styled Clipboard Text" (page 96)	Special text data, retrieved from the Clipboard, that includes style and font information
"Styled Text" (page 84)	Synonym for a special string that includes style and font information
"Text" (page 87)	Synonym for class String
Unicode Text	Character data in the form of Unicode (2-byte) text; see "Unicode Text and International Text" (page 87)
"Unit Type Value Classes" (page 91)	Classes for working with measurements of length, area, cubic and liquid volume, mass, and temperature

Three identifiers in Table 3-1 act only as synonyms for other value classes: Number is a synonym for either Integer or Real, Text is a synonym for String, and Styled Text is a synonym for a string that contains style and font information. You can coerce values using these synonyms, but the class of the resulting value is always the true value class.

For example, you can use the class identifier Text to coerce a date to a string.

```
set x to date "May 14, 1993" as text
class of x
--result: string
```

Although definitions for value class synonyms are provided, they do not correspond to separate value classes. For more information about coercing values using synonyms, see "Coercing Values" (page 97).

Boolean

A value of class **Boolean** is a logical truth value. The most common Boolean values are the results of comparisons, such as `4 > 3` and `WordCount = 5`. The two possible Boolean values are `true` and `false`.

LITERAL EXPRESSIONS

```
true

false
```

PROPERTY

`Class`	The class identifier for the object. This property is read-only, and its value is always `boolean`.

ELEMENTS

None

OPERATORS

The operators that take Boolean values as operands are And, Or, Not, `&`, `=`, and ≠.

The `=` operator returns `true` if both operands evaluate to the same Boolean value (either `true` or `false`); the ≠ operator returns `true` if the operands evaluate to different Boolean values.

The binary operators And and Or take Boolean expressions as operands and return Boolean values. An And operation, such as `(2 > 1) and (4 > 3)`, has the value `true` if both its operands are `true`, and `false` otherwise. An Or operation, such as `(theString = "Yes") or (today = "Tuesday")`, has the value `true` if either of its operands is `true`.

The unary Not operator changes a `true` value to `false` or a `false` value to `true`.

For additional information on these operators, see “Operations” (page 213).

COERCIONS SUPPORTED

AppleScript supports coercion of a Boolean value to a single-item list.

Class

A value of class **Class** is a class identifier. A class identifier is a reserved word that specifies the class to which an object or value belongs. The Class property of an object contains a class identifier value.

LITERAL EXPRESSIONS

```
string
integer
real
boolean
class
```

PROPERTY

`Class`	The class identifier for the object. This property is read-only, and its value is always `class`.

ELEMENTS

None

OPERATORS

The operators that take class identifier values as operands are `&`, `=`, ≠, and As.

The operator As takes a value of one class and coerces it to a value of a class specified by a class identifier. For example, the following statement coerces a string into the corresponding real number, `1.5`:

```
"1.5" as real --result: 1.5
```

For more information about coercing values, see “Expressions” (page 199).

COERCIONS SUPPORTED

AppleScript supports coercion of a class identifier to a single-item list.

Constant

A value of class **Constant** is a reserved word defined by AppleScript or an application in its dictionary. Applications define sets of values that can be used for parameters of a particular command. For example, the value of the `saving` parameter of a Close command must be one of the three constants `yes`, `no`, and `ask`, where `saving no` means do not save any changes, `saving yes` means save without asking, and `saving ask` means ask the user whether to save.

For more information on the use of constants in AppleScript, see "Constants" (page 100).

LITERAL EXPRESSIONS

```
yes
no
ask
plain
bold
italic
```

For a complete listing of the constants AppleScript provides, see Table A-3 (page 358) and Table A-2 (page 355).

PROPERTY

`Class`	The class identifier for the object. This property is read-only, and its value is always `constant`.

ELEMENTS

None

OPERATORS

The operators that take values of class Constant as operands are `&`, `=`, ≠, and As.

COERCIONS HANDLED

AppleScript supports coercion of a constant to a single-item list.

Starting in version 1.3.7, AppleScript supports coercion of a constant to a string.

NOTES

Constants are not strings, and they must not be surrounded by quotation marks.

You cannot define your own constants; constants can be defined only by applications and AppleScript.

Data

A value of class **Data** is data returned by an application (in response to a command) that does not belong to any of the other value classes defined in this chapter. A value of class Data is raw data that can only be stored in a variable. For more information on raw data, see "Raw Data in Parameters" (page 120).

PROPERTY

`Class`	The class identifier for the object. This property is read-only, and its value varies depending on the application.

ELEMENTS

None

OPERATORS

The operators that can take values of class Data as operands are = and ≠.

COERCIONS SUPPORTED

AppleScript supports coercion of a Data value to a single-item list.

Date

A complete **Date** value specifies the day of the week, the date (month, day of the month, and year), and the time; if you provide only some of this information, AppleScript fills in the missing pieces with default values. You can get and set properties of a Date value that correspond to different parts of the date and time information.

You can specify Date values in many different formats. The format always begins with the word `date` followed by a string (within quotation marks) containing the date and time information. You can spell out the day of the week, month, or date. You can also use standard three-letter abbreviations for the day and month.

When you compile a script, AppleScript displays date and time values according to the format specified in the Date & Time control panel.

For more information on arithmetic operations you can perform on dates and times, see "Date-Time Arithmetic" (page 233). For a description of how AppleScript handles century's end dates such as the year 2000, see "Working With Dates at Century Boundaries" (page 235).

LITERAL EXPRESSIONS

The following expressions show several options for specifying a date.

```
date "7/25/53, 12:06 PM"

date "8/9/50, 12:06"

date "8/9/50, 17:06"

date "7/16/70"

date "12:06"

date "Sunday, December 12, 1954 12:06 pm"
```

PROPERTIES

`Class`	The class identifier for the object. This property is read-only, and its value is always `date`.
`Day`	An integer that specifies the day of the month of a date value.
`Weekday`	One of the constants `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, `Sunday` or `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, `Sun`.
`Month`	One of the constants `January`, `February`, `March`, `April`, `May`, `June`, `July`, `August`, `September`, `October`, `November`, `December` or `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, `Dec`.
`Year`	An integer specifying the year; for example, `1993`.
`Time`	An integer that specifies the number of seconds since midnight of the date value; for example, `2700` is equivalent to 12:45 *AM* (2700 = 45 minutes times 60 seconds).
`Date String`	A string that consists of the date portion of the date value; for example, `"Saturday, February 27, 1999"`.
`Time String`	A string that consists of the time portion of the date value; for example, `"3:20:24 PM"`.

ELEMENTS

None

OPERATORS

The operators that take Date values as operands are &, +, –, =, ≠, >, ≥, <, ≤, Comes Before, Comes After, and As. In expressions containing >, ≥, <, ≤, Comes Before, or Comes After, a later time is greater than an earlier time. The following operations on Date values with the + and – operators are supported:

date + *timeDifference*
`--result:` *date*

date - *date*
`--result:` *timeDifference*

date - *timeDifference*
`--result:` *date*

where *date* is a Date value and *timeDifference* is an Integer value specifying a time difference in seconds. To simplify the notation of time differences, you can also use one or more of these of these constants:

`minutes`	60
`hours`	60 * minutes
`days`	24 * hours
`weeks`	7 * days

Here's an example:

```
date "September 13, 1998" + 4 * days + 3 * hours + 2 * minutes
--result: date "Thursday, September 17, 1998 3:02:00 AM"
```

For more information about the way AppleScript operators treat Date values, see "Date-Time Arithmetic" (page 233).

REFERENCE FORMS

You can refer to properties of a Date value using the Property reference form. When you compile a script, AppleScript displays date and time values according to the format specified in the Date & Time control panel. The following statements access various date properties:

```
set theDate to current date --using scripting addition command
--result: date "Saturday, February 27, 1999 3:37:50 PM"
weekday of theDate --result: Saturday
day of theDate --result: 27
month of theDate --result: February
year of theDate --result: 1999
time of theDate --result: 56270 (seconds since 12:00:00 AM)
time string of theDate --result: "3:37:50 PM"
date string of theDate --result: "Saturday, February 27, 1999"
```

For more information on compiling dates, see the Notes section below.

If you want to specify a time relative to a date, you can do so by using `of`, `relative to`, or `in`, as shown in the following examples.

```
date "2:30 am" of date "March 3, 1999"
--result: date "Wednesday, March 3, 1999 2:30:00 AM"
```

```
date "Nov. 19, 1999" relative to date "3PM"
--result: date "Friday, November 19, 1999 3:00:00 PM"

date "1:30 pm" in date "March 3, 1999"
--result: date "Wednesday, March 3, 1999 1:30:00 PM"
```

If you set the Day property of a date to a value that does not fit into the month, the date rolls over to the next month:

```
set myDate to date "Thursday, February 4, 1999 12:00:00 AM"
set day of myDate to 37
myDate --result: date "Tuesday, March 9, 1999 12:00:00 AM"
```

COERCIONS SUPPORTED

AppleScript supports coercion of a Date value to a single-item list or a string.

NOTES

When you compile a script, AppleScript displays Date values in a format similar to the one shown in the following example, regardless of the format you use when you type the date. The compiled version includes the full name of the day of the week and month and no leading zeros for the date. The actual format is based on the settings in the Date & Time control panel. The following notes and examples assume the Date & Time control panel is set for 12-hour time, not 24-hour time.

```
date "Friday, January 3, 1992 12:05:00 PM"
```

If you don't specify a complete date, day, and time when typing a Date value, AppleScript fills in information as needed. If you don't specify the date information, AppleScript uses the date when the script is compiled. If you don't specify the time information, 12:00 AM (midnight) is the default. If you omit AM or PM, AM is the default; however, if you specify 12:00 without AM or PM, 12:00 PM is the default. If you specify a time using 24-hour time, AppleScript converts it to the equivalent time using AM or PM (when the Date & Time control panel is set for 12-hour time); for example, 17:00 is equivalent to 5:00 PM.

The following example shows how AppleScript fills in a default time property when the specified date doesn't include the time:

```
time string of date "March 3, 1999"
--result: "12:00:00 AM"
```

To get the current date, use the scripting addition command Current Date:

```
set theDate to current date
if (weekday of theDate) = Saturday then
    display dialog "I shouldn't be working today!"
end if
```

For more information about the Current Date and Display Dialog commands, and about the other standard scripting addition commands distributed with AppleScript, see the following website:

<http://www.apple.com/applescript/>

Integer

A value of class **Integer** is a positive or negative number without a fractional part.

LITERAL EXPRESSIONS

```
1
2
-1
1000
```

PROPERTY

`Class`	The class identifier for the object. This property is read-only, and its value is always `integer`.

ELEMENTS

None

OPERATORS

The Div operator always returns an Integer value as its result. The +, -, *, Mod, and ^ operators return values of type Integer or Real.

The operators that can have Integer values as operands are +, -, *, ÷ (or /), Div, Mod, ^, =, ≠, >, ≥, <, and ≤.

COERCIONS SUPPORTED

AppleScript supports coercion of an Integer value to a single-item list, a real number, or a string.

You can also coerce an integer using the synonym Number, but the class of the resulting value remains unchanged:

```
set x to 7 as number
class of x --result: integer
```

NOTES

The largest value that can be expressed as an integer in AppleScript is ±536870909, which is equal to $\pm(2^{29} - 3)$. Larger integers (positive or negative) are converted to real numbers (expressed in exponential notation) when scripts are compiled.

List

A value of class **List** is an ordered collection of values. The values contained in a list are known as **items.** Each item can belong to any class.

LITERAL EXPRESSIONS

A list appears in a script as a series of expressions contained within braces and separated by commas. For example, the following statement defines a list that contains a string, an integer, and a Boolean:

```
{ "it's", 2, true }
```

Each list item can be any valid expression. The following list has the same value as the list in the previous example, because each of the expressions it contains has the same value as the corresponding expression in the previous example:

```
{ "it" & "'s", 1 + 1, 4 > 3 }
```

An **empty list** is a list containing no items. It is represented by a pair of empty braces:

```
{}
```

PROPERTIES

`Class`	The class identifier for the value. This property is read-only, and its value is always `list`.
`Length`	An integer containing the number of items in the list. This property is read-only.
`Rest`	A list containing all items in the list except the first item.
`Reverse`	A list containing all items in the list, but in the opposite order.

ELEMENT

`Item`	A value contained in the list. Each value contained in a list is an item. You can refer to values by their item numbers. For example, `item 2 of {"soup", 2, "nuts"}` is the integer `2`. To specify items of a list, use the reference forms listed in "Reference Forms" later in this definition.

OPERATORS

The operators that can have List values as operands are &, =, ≠, Starts With, Ends With, Contains, Is Contained By.

For detailed explanations and examples of how AppleScript operators treat lists, see "Operators That Handle Operands of Various Classes" (page 220).

COMMANDS HANDLED

You can count the items in a list with the Count command. For example, the value of the following statement is `6`:

```
count {"a", "b", "c", 1, 2, 3}
--result: 6
```

You can also count elements of a specific class in a list. For example, the value of the following statement is `3`:

```
count integers in {"a", "b", "c", 1, 2, 3}
--result: 3
```

Another way to count the items in a list is with a Length property reference:

```
length of {"a", "b", "c", 1, 2, 3}
--result: 6
```

REFERENCE FORMS

Use the following reference forms to refer to properties of lists and items in lists:

- *Property*. For example, `class of {"this", "is", "a", "list"}` specifies `list`.
- *Index*. For example, `item 3 of {"this", "is", "a", "list"}` specifies `"a"`.
- *Middle*. For example, `middle item of {"this", "is", "a", "list"}` specifies `"is"`.
- *Arbitrary*. For example, `some item of {"soup", 2, "nuts"}` might specify any of the items in the list.
- *Every Element*. For example, `every item of {"soup", 2, "nuts"}` specifies `{"soup", 2, "nuts"}`.
- *Range*. For example, `items 2 thru 3 of {"soup", 2, "nuts"}` specifies `{2, "nuts"}`.

You cannot use the Relative, Name, ID, or Filter reference forms. For example, the following reference, which uses the Filter reference form on a list, is not valid.

```
the items in {"this", "is", "a", "list"} whose first ¬
    character is "t"
--result: not a valid reference
```

For more information on the Filter reference form, see "Filter" (page 173).

COERCIONS SUPPORTED

AppleScript supports coercion of a single-item list to any value class to which the item can be coerced if it is not part of a list.

AppleScript also supports coercion of an entire list to a string if all items in the list can be coerced to a string. The resulting string concatenates all the items, separated by the current value of the AppleScript property Text Item Delimiters. This property defaults to an empty string, so the items are simply concatenated. For more information on Text Item Delimiters, see "AppleScript Properties" (page 210).

```
{5, "George", 11.43, "Bill"} as string
--result: "5George11.43Bill"
```

Individual items in a list can be of any value class, and AppleScript supports coercion of any value to a list that contains a single item. Concatenated values of any class can also be coerced to a list:

```
5 & "George" & 11.43 & "Bill" as list
--result: {5, "George", 11.43, "Bill"}
```

NOTES

You can use the concatenation operator (&) to merge or add values to lists,. For example:

```
{"This"} & {"is", "a", "list"} --result: {"This", "is", "a", "list"}
```

The concatenation operator merges the items of the two lists into a single list, rather than making one list a value within the other list.

For large lists, it may be more efficient to use the Copy or Set command to insert an item directly into a list. The following script creates a list of 10,000 integers in

about a second (the required time will vary depending on the speed of the computer used and may vary depending on the version of AppleScript):

```
set bigList to {}
set bigListRef to a reference to bigList
set numItems to 10000
set t to (time of (current date)) --Start timing operations.
repeat with n from 1 to numItems
    copy n to the end of bigListRef
end
set total to (time of (current date)) - t --End timing.
total --result: 1 second
```

For more information on working efficiently with large lists, see the example section in "The A Reference To Operator" (page 203).

Number

The class identifier **Number** is a synonym for Integer or Real; it describes a positive or negative number that can be either of class Integer or of class Real.

LITERAL EXPRESSIONS

```
1
2
-1
1000

10.2579432
1.0
1.
```

Any valid literal expression for an Integer or a Real value is also a valid literal expression for a Number value.

PROPERTY

`Class`	The class identifier for the object. This property is read-only, and its value is always either `integer` or `real`.

ELEMENTS

None

OPERATORS

Because values identified as values of class Number are really values of either class Integer or class Real, the operators available are the operators described in the definitions of the "Integer" (page 66) or "Real" (page 72) value classes.

COERCIONS SUPPORTED

You can use the class identifier Number to coerce any value that can be coerced to a Real value or an Integer value. However, the resulting value class is always either Integer or Real:

```
set x to 1.5 as number
class of x --result: real
```

Real

Values that belong to the class **Real** are positive or negative numbers that can include a decimal fraction, such as 3.14159 and 1.0.

LITERAL EXPRESSIONS

```
10.2579432
1.0
1.
```

As shown in the third example, a decimal point indicates a real number, even if there is no fractional part.

Real numbers can also be written using exponential notation. A letter `e` is preceded by a real number (without intervening spaces) and followed by an integer exponent (also without intervening spaces). The exponent can be either positive or negative. To obtain the value, the real number is multiplied by 10 to the power indicated by the exponent, as in these examples:

```
1.0e5 --equivalent to 1.0 * 10^5, or 100000
1.0e+5 --same as 1.0e5
1.0e-5 --equivalent to 1.0 * 10^-5, or .00001
```

PROPERTY

`Class`	The class identifier for the object. This property is read-only, and its value is always `real`.

ELEMENTS

None

OPERATORS

The ÷ and `/` operators always return Real values as their results. The `+`, `-`, `*`, Mod, and `^` operators return Real values if either of their operands is a Real value.

The operators that can have Real values as operands are `+`, `-`, `*`, ÷ (or `/`), Div, Mod, `^`, =, ≠, >, ≥, <, and ≤.

COERCIONS SUPPORTED

AppleScript supports coercion of a Real value to a single-item list or a string. AppleScript supports coercion of a Real value to an Integer value only if the Real value has no fractional part.

AppleScript also supports coercion of a Real value using the synonym Number, but the class of the resulting value remains unchanged.

```
set x to 1.5 as number
class of x --result: real
```

NOTES

Real numbers that are greater than or equal to 10,000.0 or less than or equal to 0.0001 are converted to exponential notation when scripts are compiled. The largest value that can be evaluated (positive or negative) is 1.797693e+308.

Record

A value of class **Record** is an unordered collection of properties. Like the properties of application objects, each property has a label, and the properties of a record are distinguished from each other by their label. There can be only one property with a particular label in any record.

LITERAL EXPRESSIONS

Records appear in scripts as series of properties contained within braces and separated by commas. Each property has a label. Following the label is a colon, and following the colon, the value of the property. For example, the record

```
{ name:"Steve", height:74.5, weight:175 }
```

contains three properties: Name (a string), Height (a real number), and Weight (an integer). The values assigned to properties can belong to any class.

AppleScript evaluates expressions in a record before using the record in other expressions. For example, the following record is equivalent to the previous one.

```
{ name:"Steve", height:76 - 1.5, weight:150 + 25 }
```

PROPERTIES

In addition to the properties that are specific to each record, two properties are common to all records:

`Class` The class identifier for the object. For most records, the value of the Class property is `record`.

The Class property of a record can be modified—it is not read-only. For example, an application that edits text could define a special record to specify the styles (such as `bold` and `underline`) of text objects. The value of the Class property for these records, as illustrated in the following example, is the class identifier Text Style Info.

```
tell application "AppleWorks"
    -- Get text style from open document.
    style of text body of document 1
end tell
```

Running the previous script produces the following result:

```
{class:text style info, on styles:{plain}, off styles:{italic, underline,
outline, shadow, condensed, expanded, strikethrough, superscript,
subscript, superior, inferior, double underline}}
```

`Length` An integer containing the number of properties in the record. This property is read-only.

If you define a Class property explicitly in a record, the value you define replaces the implicit Class property `record` described above.

OPERATORS

The operators that can have records as operands are `&`, `=`, `≠`, Contains, and Is Contained By.

For detailed explanations and examples of how AppleScript operators treat records, see "Operators That Handle Operands of Various Classes" (page 220).

COMMANDS HANDLED

You can count the properties in a record with the Count command. For example, the value of the following statement is `2`.

```
count {name:"Robin", mileage:4000}
--result: 2
```

Another way to count the properties in a record is with a Length property reference. For example, the value of the following reference is 3.

```
length of {name:"Robin", mileage:8000, city:"Sunnyvale"}
--result: 3
```

REFERENCE FORMS

The only reference form you can use with records is the Property reference form. For example, the following reference specifies the Mileage property of a record.

```
mileage of {name:"Robin", mileage:8000, city:"Sunnyvale"}
--result: 8000
```

You cannot refer to properties in records by numeric index. For example, the following reference, which uses the Index reference form on a record, is not valid.

```
item 2 of { name:"Robin", mileage:8000, city:"Sunnyvale" }
--result: not a valid reference
```

COERCIONS SUPPORTED

AppleScript supports coercion of records to lists; however, all property labels are lost in the coercion and the resulting list cannot be coerced back to a record.

NOTES

To specify a particular property of a record, you give its name. For example, if you assign the record to a variable, as in

```
copy { name:"Steve", height:70.5, weight:165 } to writer
```

you can then get the value of the Name property with the expression

```
name of writer --result: "Steve"
```

A property of a record can contain a value of any class. You can change the class of a property simply by assigning a value belonging to another class.

After you define a record, you cannot add additional properties to it. You can, however, concatenate records. For more information, see "Concatenation" (page 229).

Reference

A value of class **Reference** is a reference to an object. A reference can refer to an application object such as a window or file, or to an AppleScript object such as an item in a list or a property in a record. You can create a value of class Reference by using the A Reference To operator. In addition, applications can return references in response to commands.

A value of class Reference is different from the value of the object to which a reference refers. For example, the reference `docNameRef` in the following script refers to a name object (`name of document 1 of application "AppleWorks"`) whose value is a string (such as `"April Report"`).

```
tell application "AppleWorks"
    set docNameRef to a reference to the name of the first document
        --result: name of document 1 of application "AppleWorks"
    docNameRef as string --result: "April Report"
end tell
```

If you change the name of the report to "Revised April Report" and run this script again, the result of the reference will be the same (`name of document 1 of application "AppleWorks"`), but the value will change (`"Revised April Report"`).

The difference between a value of class Reference and the object it refers to is analogous to the difference between an address and the building it refers to. The address is a series of words and numbers, such as "1414 Maple Street," that identifies the location of the building. It is distinct from the building itself. If the building is replaced with a new building at the same location, the address remains the same.

A value of class Reference created with the A Reference To operator is a structure within AppleScript that refers to (or points to) a specific object.

```
tell application "AppleWorks"
    set docRef to a reference to the first document
        --result: document 1 of application "AppleWorks"
    name of docRef --result: "New Report"
end tell
```

In this script, the reference `docRef` refers to the first document of the application AppleWorks, which happens to be named "New Report". However, the object that `docRef` points to can change. If you open a second AppleWorks document called "Second Report" and run this script again, it will return the name of the newly opened document, "Second Report".

You can instead create a direct reference to the document "New Report":

```
tell application "AppleWorks"
    set docRef to a reference to document "New Report"
        --result: document "New Report" of application "AppleWorks"
    name of docRef --result: "New Report"
end tell
```

If you run this script after opening a second document, it will still return the name of the original document, "New Report". You can also use the `alias` form to refer to a file whose name or location may change. For more information, see "References to Files" (page 191).

Values of class Reference are similar to pointers in other programming languages, but unlike pointers, references can refer only to objects. Using a reference can sometimes be much more efficient than using an object directly, as shown in the example in the Notes section in "List" (page 67). For related information about using values of class Reference, see "The A Reference To Operator" (page 203).

LITERAL EXPRESSIONS

```
set itemRef to a reference to item 3 of {1, "hello", 755, 99}
    --result: item 3 of {1, "hello", 755, 99}
set newTotal to itemRef + 45 --result: 800

a reference to the name of the first report
```

PROPERTIES

`Class`	The class identifier for the object. This property is read-only, and its value is always `reference`.
`Contents`	The value of the object to which the reference refers. The class of the value depends on the reference. For information about how to use the Contents property, see "The A Reference To Operator" (page 203).

ELEMENTS

None

OPERATORS

The A Reference To operator returns a reference as its result. This operator is described in "The A Reference To Operator" (page 203).

COERCIONS SUPPORTED

The application to which an object specified by a reference belongs determines whether the value of the object can be coerced to a desired class.

NOTES

A reference can function as a reference to an object or as an expression whose value is the value of the object specified in the reference. When a reference is the direct parameter of a command, it usually functions as a reference to an object, indicating to which object the command should be sent. In most other cases, references function as expressions, which AppleScript evaluates by getting their values.

The reference `front window of application "Apple System Profiler"` in the following example functions as a reference to an object. It identifies the object to which the Close command is sent.

```
close front window of application "Apple System Profiler"
```

On the other hand, the reference `name of the first report` in the following example functions as a reference expression:

```
tell application "Apple System Profiler"
    set reportNameString to name of the first report
end tell
```

When AppleScript executes this script, it gets the value of the reference `name of the first report`—a string—and then stores it in the variable `reportNameString`.

The following script shows an AppleWorks application command, the Make command, which returns a reference:

```
tell application "AppleWorks"
    -- Create a new document and get a reference to it.
    set docRef to (make new document at beginning ¬
        with properties {name:"New Report"})
    --result: document "New Report" of application "AppleWorks"
end tell
```

String

A value of class **String** is a character string (an ordered series of characters) in AppleScript. For information on additional string value class types, see “Unicode Text and International Text” (page 87).

LITERAL EXPRESSIONS

Strings in scripts are always surrounded by quotation marks, as in these examples:

```
"string"
"Rolling along, stringing a song"
"Pennsylvania 68000"
```

To include quotation marks in a string, you must use the two-character sequence, `\"`. For more information, see “Special Characters in Strings” later in this section.

PROPERTIES

`Class`	The class identifier for the object. This property is read-only, and its value is always `string`.
`Length`	The number of characters in the string.

ELEMENTS

Strings can have character, word, paragraph, and text elements.

The elements of a string may be different from the character, word, paragraph, and text objects of applications.

Character	A single character contained in the string.
Paragraph	A series of characters beginning immediately after either the first character after the end of the preceding paragraph or the beginning of the string and ending with either a return character or the end of the string.
Text	A continuous series of characters, including spaces, tabs, and all other characters, within a string (see "Notes" later in this section).

Word	A continuous series of characters that contains only the following types of characters:
	
	letters (including letters with diacritical marks)
	digits
	nonbreaking spaces
	dollar signs, cent signs, English pound symbols, or yen symbols
	percent signs
	commas between digits
	periods before digits
	apostrophes between letters or digits
	
	hyphens (but not minus signs [Option-hyphen] or dashes [Option-Shift-hyphen]).
	
	Here are some examples of words:
	
	`read-only`
	`he's`
	`v1.0`
	`$99.99`
	`12c-d`

This definition of a word applies to English text in the Roman script system. Words in other languages are defined by the script system for each language if the appropriate script system is installed.

OPERATORS

The operators that can have strings as operands are `&`, `=`, `≠`, `>`, `≥`, `<`, `≤`, Starts With, Ends With, Contains, Is Contained By, and As.

For detailed explanations and examples of how AppleScript operators treat strings, see "Operators That Handle Operands of Various Classes" (page 220).

REFERENCE FORMS

You can use the following reference forms to refer to elements of strings:

- *Property*. For example, `class of "This is a string"` specifies `string`.
- *Index*. For example, `word 3 of "This is a string"` specifies `"a"`.
- *Middle*. For example, `middle word of "This is a string"` specifies `"is"`.
- *Arbitrary*. For example, `some word of "This is a string"` might specify any of the words in the string.
- *Every Element*. For example, `every word of "This is a string"` specifies `{"This", "is", "a", "string"}`.
- *Range*. For example, `words 2 thru 3 of "This is a string"` specifies `{"is", "a"}`.

You cannot use the Relative, Name, ID, or Filter reference forms.

SPECIAL CHARACTERS IN STRINGS

The backslash (`\`) and double-quote (`"`) characters have special meaning in strings. If you want to include either of these characters in a string, you must use the equivalent two-character sequence:

Backslash character	`\\`
Double-quote character	`\"`

The tab and return characters can be included in strings, or they can be represented by equivalent two-character sequences:

Tab character	`\t`
Return character	`\r`

When a string containing any of the two-character sequences is displayed to the user (as, for example, in a dialog box), the sequences are converted. For example, the string

```
"item 1\t1\ritem 2\t2"
```

is displayed in a dialog box as

```
item 1      1
item 2      2
```

STRING CONSTANTS

AppleScript defines three constants for string values:

Constant	Value
`space`	`" "`
`tab`	`"\t"`
`return`	`"\r"`

COERCIONS SUPPORTED

If a string represents an appropriate number, AppleScript supports coercion of the string to an integer, a number, or a real number. Similarly, any integer, number, or real number can be coerced to a string. AppleScript also supports coercion of a string to a single-item list and coercion of a list whose items can all be coerced to strings to a single concatenated string. Starting with version 1.3.7, AppleScript supports coercion of a constant, such as Monday or January, to a string.

NOTES

There is no limit on the length of strings except the memory available in the computer.

To get a contiguous range of characters within a string, use the text element. For example, the value of the following statement is the string `"y thi"`.

```
get text 3 thru 7 of "Try this at home"
--result: "y thi"
```

The result of a similar statement using the character element instead of the text element is a list.

```
get characters 3 thru 7 of "Try this at home"
--result: {"y", " ", "t", "h", "i"}
```

You cannot set the value of an element of a string. For example, if you attempt to change the value of the first character of the string `"Boris"` as shown in the following example, you'll get an error.

```
set myName to "Boris"
set character 1 of myName to "D"
--results in an error, because you cannot set the values of
--elements of strings
```

However, you can modify the name by getting the last four characters and concatenating them with `"D"`:

```
set myName to "boris"
set myName to "D" & (get text 2 through 5 of myName)
--result: "Doris"
```

Styled Text

The class identifier **Styled Text** is a synonym for a string that includes style and font information.

LITERAL EXPRESSIONS

The only difference between a value of class String and a value of class Styled Text is that the latter can include (but is not required to include) style and font information. Thus any valid literal expression of class String is also valid as class Styled Text.

PROPERTIES

`Class`	The class identifier for the object. This property is read-only, and its value is always `string`.
`Length`	The number of characters in the string.

ELEMENTS

Styled text has the same character, word, paragraph, and text elements as a string.

OPERATORS

Because values identified as Styled Text values are really values of class String, the operators available are the operators described in the definition of class String: `&`, `=`, `≠`, `>`, `≥`, `<`, `≤`, Starts With, Ends With, Contains, Is Contained By, and As.

For detailed explanations and examples of how AppleScript operators treat strings, see "Operators That Handle Operands of Various Classes" (page 220).

REFERENCE FORMS

You can use the same reference forms with styled text that you can use with strings: Property, Index, Middle, Arbitrary, Every Element, and Range. For details, see "String" (page 80).

SPECIAL CHARACTERS AND STRING CONSTANTS

You can use the same special characters, constants, and coercions with styled text that you can use with strings. For details, see "String" (page 80). Note that literal string constants do not include style and font information; in other words, they are not styled text.

COERCIONS SUPPORTED

You can use the same coercions with styled text that you can use with strings: coercion to an integer, number, real number, or single-item list, and coercion of a list of strings to a single concatenated string.

You can use the class identifier Styled Text to coerce any string to styled text. However, the resulting value is always of class String.

NOTES

AppleScript itself provides no commands for directly manipulating styled text. To change the style or font information for a styled text value, you must work with an application that knows how to manipulate styled text. However, AppleScript does preserve style and font information when copying text objects from applications to scripts and vice versa.

For example, you can use a script like this to obtain styled text, manipulate it, and copy it back into an AppleWorks document:

```
tell application "AppleWorks"
    -- Get text from open document.
    set myText to text body of document "Report"
    -- Add some information at the end.
    set myText to myText & return & "The End."
    -- Select all the current text in the document and replace it.
    select text body of document "Report"
    set selection of document "Report" to myText
    close document "Report" saving ask
end tell
```

Because AppleWorks returns styled text when it returns the text from a document, you don't need to coerce the returned text to styled text. The style and font of the text are preserved both when it is copied to the variable `myText` and when it is concatenated with a return character and the string `"The End."` The modified text that is inserted back into the document consists of the original text with its original style and font, a return character to move to a new line, and the unstyled text, `"The End."`, which appears in the style and font of the text immediately preceding it.

Styled text also contains information about the form in which the text is written. If you copy non-Roman text to a variable in a script as styled text, AppleScript preserves the original text information even though the Script Editor may not

be able to display it correctly. If you then copy the text to an application that can handle the text in its original form, the text is displayed correctly. For related information, see “Unicode Text and International Text” (page 87).

Text

You can use the class identifier **Text** as a synonym for the identifier String—for example, in coercions:

```
"A string" as string = "A string" as text --result: true
```

However, the class of a string is always `string`:

```
set myThing to "A string"
class of myThing --result: string
set otherThing to myThing as text
class of otherThing --result: string
```

Unlike the class identifier Number (which is a synonym for either Real or Integer) or Styled Text (which denotes a string that includes font and style information), the class identifier Text is precisely equivalent to a single class identifier—String.

Unicode Text and International Text

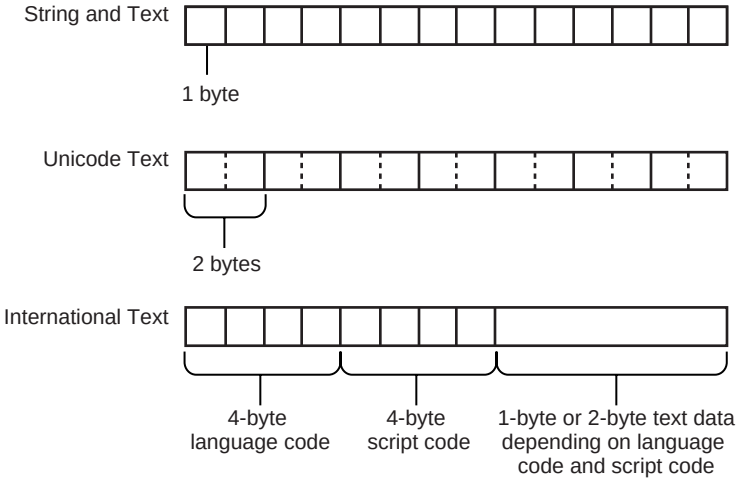
In addition to the string value classes described in “String” (page 80), “Styled Text” (page 84), and “Text” (page 87), AppleScript provides partial support for the following string types:

- **Unicode Text**: An ordered series of 2-byte Unicode characters. (**Unicode** is an international standard that uses a 16-bit encoding to uniquely specify the characters and symbols for all commonly used languages.)
- **International Text**: An ordered series of bytes, beginning with a 4-byte language code and a 4-byte script code that together determine the format of the bytes that follow. (International text can be obtained only from a Macintosh computer that has a language kit installed.)

You can use the Unicode Text, International Text, and String classes in any script and they do not need to be enclosed in a Tell block. These classes all represent text data, though in different formats, as shown in Figure 3-1. You typically use the Unicode Text and International Text classes to get information from and send information to applications that work with these types of text.

Because the different string value classes store data in different formats, the size in bytes of a string can vary from the number of characters it contains. Comparisons between Unicode Text, International Text, and String values are not likely to be useful. You cannot determine the Length property of a value stored as Unicode Text, or access its Character, Word, or Paragraph elements (as you can with other string types, including International Text).

Figure 3-1 Formats for the String (or Text), Unicode Text, and International Text value classes

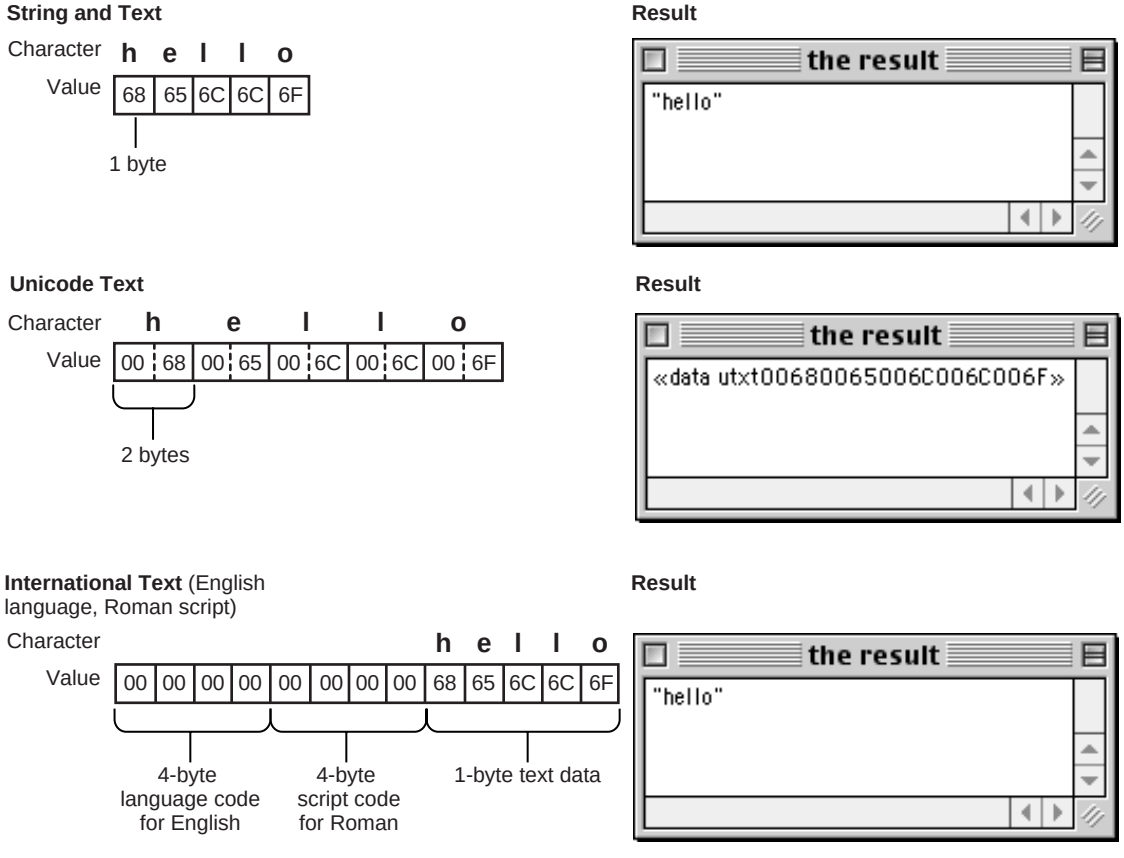


AppleScript provides limited options for displaying Unicode Text and International Text:

- The Script Editor can display Unicode Text only as raw data, as shown in Figure 3-2. For more information on raw data, see “Raw Data in Parameters” (page 120).

- The Script Editor displays International Text according to the language and script of the text and depending on whether a language kit is installed. For example, the Script Editor can display International Text in the English language and Roman script as a simple string, as shown in Figure 3-2. However, it cannot display simplified Chinese characters unless you have installed the appropriate language kit.

Figure 3-2 How the Script Editor displays String, Unicode Text, and International Text data



AppleScript provides coercions among the Unicode Text, International Text, and String (or Text) classes. For example, if your script gets a value of type Unicode Text from an application that supports Unicode, you can coerce the value to String to see it in a readable format. However, because the String, Unicode Text, and International Text classes store data differently, as shown in Figure 3-1 (page 88), and because there are differences in the text data each can represent, information may be lost in some coercions. As shown in Figure 3-3 (page 99), coercions from Unicode Text to International Text or String, and from International Text to String, may result in lost information. For example, the String class cannot represent 2-byte Chinese characters (from either International Text or Unicode Text values) in its 1-byte character format.

For an overview of AppleScript coercion, see "Coercing Values" (page 97).

The following script statements demonstrate how to initialize a string as Unicode Text and then coerce it to a standard String value:

```
set myString to "hello" as Unicode text
--result: «data utxt00680065006C006C006F»
myString as string
--result: "hello"
```

The following script retrieves text data from an open AppleWorks word processing document, adds to it, and replaces the existing text with the combined text. Because the original text is stored as Chinese characters (entered with the Chinese Language Kit), AppleWorks returns the text as International Text, so the script doesn't need to coerce it to International Text. After the script completes, the document contains the original Chinese characters, followed by "The End." in the current language and script (English Roman). If the appropriate language kit installed, AppleScript can display Chinese characters in its result window.

```
tell application "AppleWorks"
    -- Get text (Chinese characters) from open document.
    set myText to text body of document "Chinese Text"
    -- Add some information at the end in English.
    set myText to myText & return & "The End."
    -- Select all the current text in the document and replace it.
    select text body of document "Chinese Text"
    set selection of document "Chinese Text" to myText
end tell
```

If you use the & operator to combine two values of Unicode Text, the result is a list, not a string:

```
set greetingString to "Hello" as Unicode text
    --result: «data utxt00480065006C006C006F»
set nameString to " Bob" as Unicode text
    --result: «data utxt00200042006F0062»
set combinedString to greetingString & nameString
    --result:
    -- {«data utxt00480065006C006C006F», «data utxt00200042006F0062»}
set combinedString to (greetingString as string) ¬
                    & (nameString as string)
    --result: "Hello Bob"
```

Unit Type Value Classes

AppleScript provides Unit Type value classes for working with measurements of length, area, cubic and liquid volume, mass, and temperature. Unit Type classes are simple values that do not contain other values and have only a single property, the Class property. You can use the Unit Type classes in any script and they do not need to be enclosed in a Tell block.

AppleScript provides coercions from Unit Type to String and Number (Real or Integer) and from String, Real, or Integer to Unit Type. You can also coerce between Unit Types in the same category, such as inches to kilometers (length) or gallons to liters (liquid volume). As you would expect, there is no coercion between categories, such as from gallons to degrees Centigrade.

Note that AppleScript did not implement the quarts and degrees Kelvin Unit Types or support coercions from miles to other types until version 1.3.7.

For an overview of value coercion in AppleScript, see “Coercing Values” (page 97).

AppleScript Unit Types by Category

This section lists the Unit Type value classes AppleScript provides.

Length

centimetres
centimeters
feet
inches
kilometres
kilometers
metres
meters
miles
yards

Area

square feet
square kilometres
square kilometers
square metres
square meters
square miles
square yards

Cubic Volume

cubic centimetres
cubic centimeters
cubic feet
cubic inches
cubic metres
cubic meters
cubic yards

Liquid Volume

gallons
litres
liters
quarts

Weight

grams
kilograms
ounces
pounds

Temperature

degrees Celsius
degrees Fahrenheit
degrees Kelvin

Working With Unit Type Value Classes

This section provides sample script statements for working with Unit Type value classes.

The following statements calculate the area of a circle with a radius of 7 yards, then coerce the area to square feet (`pi` is an AppleScript-defined constant).

```
set circleArea to (pi * 7) as square yards
    --result: square yards 21.991148575129
circleArea as square feet
    --result: square feet 197.920337176157
```

The following statements set a variable to a value of 5.0 square kilometers, then coerce it to various other units of area:

```
set theArea to square kilometers 5.0
    --result: square kilometers 5.0
theArea as square miles
```

```
    --result: square miles 1.930510798581
theArea as square meters
    --result: square meters 5.0E+6
```

You can also coerce a value from square meters to a real number or an integer:

```
set theArea to square meters 5.0
    --result: square meters 5.0
theArea as real
    --result: 5.0
theArea as integer
    --result: 5
```

However, you cannot coerce an area measurement to a Unit Type in a different category:

```
set theArea to square meters 5.0
    --result: square meters 5.0
theArea as cubic meters
    --result: error
theArea as degrees Celsius
    --result: error
```

The following statements demonstrate coercion of a Unit Type to a String, and from a String to a Unit Type:

```
set myValue to pounds 2.2   --result: pounds 2.2
myValue as string           --result: "2.2"
"2.2" as kilograms          --result: kilograms 2.2
```

Other Value Classes

AppleScript defines the following value classes to provide basic support for some common data types. While AppleScript provides little built-in support for coercing these classes or manipulating them directly, the classes are useful for working with applications and the Clipboard.

■ “File Specification” (page 95)

- "RGB Color" (page 96)
- "Styled Clipboard Text" (page 96)

File Specification

The File Specification class specifies the name and location on disk of a file that may not yet exist. You can obtain a file specification from the New File scripting addition command distributed with AppleScript, or from an application command that returns a file specification. The following statements use the New File command, which displays a standard system dialog to obtain a filename and location from the user.

```
set fileSpec to new file default name "New Report"
class of fileSpec --result: file specification
```

The default name displayed is "New Report" and the default directory is the current directory. (The current directory is typically the directory where the application was launched, the directory where the application last opened or saved a previous document, or another directory specified by the application. The current directory may be affected by settings in the General Controls control panel.)

The user can specify any filename and location. This statement stores the returned file specification, which describes the name and location specified by the user, in the variable `fileSpec`. Depending on what the user specifies, the result in the Script Editor's result window is something like the following:

```
file "Hard Disk:Desktop Folder:New Report"
```

You can coerce a file specification to a string, which results in a string containing the full path name to the file:

```
fileSpec as string --result: "Hard Disk:Desktop Folder:New Report"
```

Beyond coercing the pathname to a string, you cannot use AppleScript to directly access or manipulate the information in a file specification. However, you can obtain a file specification from, or pass a file specification to, a scripting addition or application that knows how to work with a file specification. For a full example that obtains a file specification and passes it to an application, see "Specifying a File by File Specification" (page 194).

RGB Color

The RGB Color value class represents a collection of three integer values that specify the red, green, and blue components of a color. You can coerce a list of three integer values into an RGB color if each of the values is from 0 to 65535. In fact, AppleScript reports the class of an RGB Color value as List:

```
set myGreenRGB to {0, 65535, 0} as RGB color --result: {0, 65535, 0}
class of myGreenRGB --result: list
```

You can get or set individual values in an RGB color by accessing the items as you would those in any list:

```
set myRGB to {500,25000,500} as RGB color --result: {500, 25000, 500}
set myGreenValue to second item of myRGB --result: 25000
set item 3 of myRGB to 12000
myRGB --result: {500, 25000, 12000}
```

You can use the RGB Color value class to obtain RGB colors from, or supply RGB colors to, applications that work with RGB colors. For example, a graphic object in an AppleWorks drawing document has Fill Color and Pen Color properties that are RGB colors.

Styled Clipboard Text

The Styled Clipboard Text value class represents text data from the Clipboard that includes style and font information. Although you can't coerce this value class to any other class or display it in its native format, you can use it to pass styled text between applications that work with styled text.

The following script copies all of the text, consisting of the one word "Hello" in outline font, from the document `"Hello with style"` to the Clipboard. It then gets the contents of the clipboard and displays the class of those contents.

```
tell application "AppleWorks"
    set myText to text body of document "Hello with style"
    activate -- Required for Clipboard commands.
    -- Next two lines use scripting addition commands.
    set the clipboard to myText
    set myClipboardText to the clipboard as scrap styles
```

```
    --result: «data styl0001000000000000F000A001008A0000C000000000000»
    class of myClipboardText --result: styled Clipboard text
end tell
```

As shown in the script, getting styled text from the Clipboard results in a value of type Styled Clipboard Text, which AppleScript can only display as raw data enclosed in chevrons (double-angle brackets). Although you cannot coerce styled clipboard text to styled text or to a string, you can store it in a variable and pass it to applications that work with styled text.

For more information on raw data, see "Double Angle Brackets in Results and Scripts" (page 123).

Coercing Values

This section describes how you coerce values from one class to another. For specific information on coercing Unit Type values, see "Unit Type Value Classes" (page 91). For information on coercing Unicode Text and International Text values, see "Unicode Text and International Text" (page 87). For information on coercing additional classes, see "Other Value Classes" (page 94).

Coercing is the process of converting a value from one class to another. AppleScript coerces values in one of two ways:

- in response to the As operator
- automatically, when a value is of a different class than was expected for a particular command or operation

The ability to coerce a value from one class to another is either a built-in function of AppleScript or a capability provided by a scripting addition command. All of the coercions described in this section are built into AppleScript, so you can use them in any script and they do not need to be enclosed in a Tell block.

The As operator specifies a particular coercion. You can use the As operator to coerce a value to the correct class before using it as a command parameter or operand. For example, the following statement coerces the integer `2` into the string `"2"` before storing it in the variable `myString`:

```
set myString to 2 as string
```

Similarly, this statement coerces the string `"2"` to the integer `2`, so that it can be added to the other operand, `8`:

```
"2" as integer + 8
```

If you provide a command parameter or operand of the wrong class, AppleScript automatically coerces the operand or parameter to the expected class, if possible. For example, when AppleScript executes the following repeat statement, it expects an integer for the number of times to repeat the enclosed display dialog command (a scripting addition command).

```
repeat "2" times
    display dialog "Hello"
end repeat
```

If you pass a string, as in this example, AppleScript attempts to coerce the string to an integer. If you pass a string that AppleScript can't coerce to an integer, such as `"many"`, it reports an error.

Not all values can be coerced to all other classes of values. Figure 3-3 summarizes the coercions that AppleScript supports for commonly used value classes. To use the figure, find the class of the value to be coerced in the column at the left. Search across the table to the column labeled with the class to which you want to coerce the value. If there is a square at the intersection, then AppleScript supports the coercion.

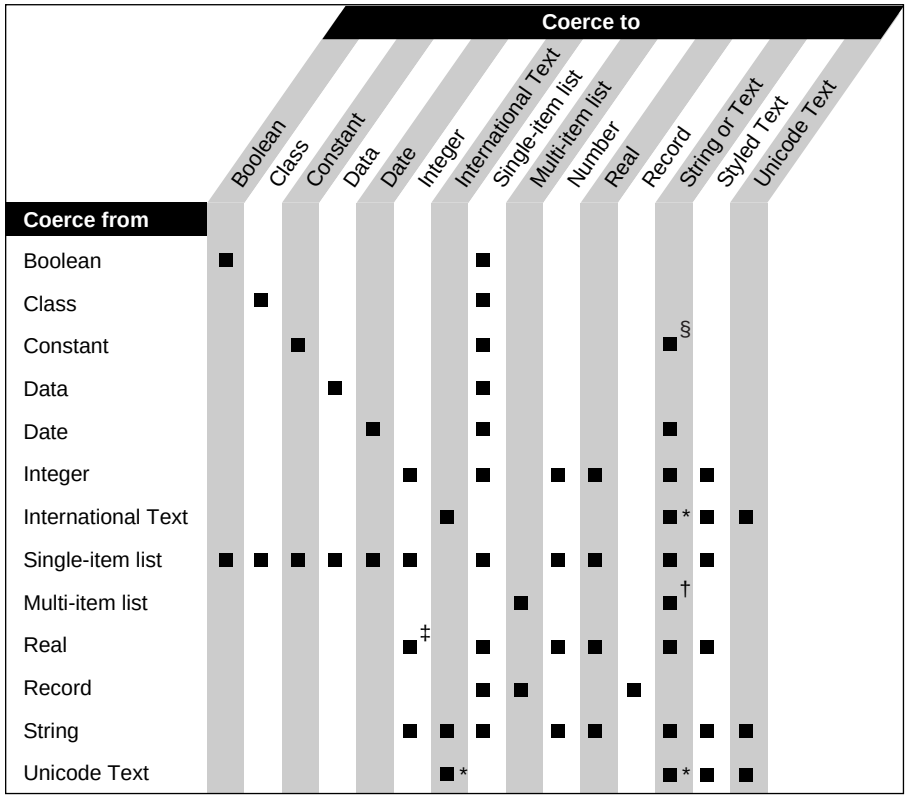
Reference values are not included in the table because the contents of the reference determine whether the value specified by a reference can be coerced to a desired class.

For more information about each coercion, see the corresponding value class definitions in this chapter.

Note
When coercing strings to values of class Integer, Number, or Real or vice versa, AppleScript uses the current settings in the Numbers control panel for decimal and thousands to determine what separators to use in the string.

When coercing strings to values of class Date or vice versa, AppleScript uses the current settings in the Date & Time control panel for date and time format. ◆

Figure 3-3 Coercions supported by AppleScript



* Some information may be lost in performing these coercions.
† Only a list whose items can all be coerced to strings can be coerced to a string.
‡ Only a real value that has no fractional part can be coerced to an integer.
§ Available with AppleScript version 1.3.7.

Three of the identifiers mentioned at the top of Figure 3-3 act only as synonyms for other value classes: “Number” is a synonym for either “Integer” or “Real,” “Text” is a synonym for “String,” and “Styled Text” is a synonym for a string that contains style and font information. You can coerce values using these synonyms, but the class of the resulting value is always the appropriate value class, not the synonym. Here are some examples:

```
set x to 1.5 as number
class of x
--result: real


set x to 4 as number
class of x
--result: integer

set x to "Hello" as text
class of x
--result: string
```

Constants

A **constant** is a reserved word with a predefined value. AppleScript provides constants to help your scripts perform a variety of tasks, such as retrieving properties, performing comparisons, and performing arithmetic operations. Constants are defined in several ways, including as part of the AppleScript language (the Boolean constants `true` and `false`), as global properties of AppleScript (`tab`, `space`, `pi`, `return`), and as individual value classes with predefined values (`Monday` through `Sunday` and `January` through `December`).

Knowing how constants are defined is less important, however, than knowing which constants are available and how to use them. The following sections list constants you use for specific tasks, show how to use the constants in a script, and provide links to related information:

- "Text Style Constants" (page 106)
- "Version Constant" (page 106)

Arithmetic Constants

AppleScript supplies several real and integer constants you can use in arithmetic calculations:

- `pi`

  The arithmetic constant 3.14159265359. Described Table A-3 (page 358).

  ```
  set circleArea to pi * 7 --result: 21.991148575129
  ```

- `minutes`, `hours`, `days`, `weeks`

  The number of seconds, respectively, in a minute, hour, day, and week. You use these constants when working with dates and times. See also "Date" (page 62), "Date and Time Constants" (page 102), and "Date-Time Arithmetic" (page 233).

  ```
  date "September 13, 1998" + 4 * days + 3 * hours + 2 * minutes
  --result: date "Thursday, September 17, 1998 3:02:00 AM"
  ```

Boolean Constants

AppleScript supplies the Boolean constants `true` and `false`, shown in Table A-2 (page 355), to evaluate the results of Boolean operations, such as Greater Than, Less Than, and Is Equal To. For additional information, see "Boolean" (page 58).

```
set mustFileReturn to (myIncome > 0)
if mustFileReturn is equal to true then
    -- Statements to file return.
end if
```

Considering and Ignoring Attributes

AppleScript defines the attributes `application responses`, `case`, `diacriticals`, `expansion`, `hyphens`, `punctuation`, and `white space` for comparisons that use Considering or Ignoring statements. These attributes are described in Table A-2

(page 355). They specify whether to consider or ignore specific characteristics when performing an evaluation.

```
"Hello Bob" = "HelloBob" --result: false
ignoring white space
    "Hello Bob" = "HelloBob" --result: true
end considering
```

For more information and examples, see "Considering and Ignoring Statements" (page 268).

Date and Time Constants

AppleScript supplies several constants you can use when working with date and time values:

- `minutes`, `hours`, `days`, `weeks`

  The number of seconds, respectively, in a minute, hour, day, and week.

  ```
  date "September 13, 1998" + 4 * days + 3 * hours + 2 * minutes
  --result: date "Thursday, September 17, 1998 3:02:00 AM"
  ```

- `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, `Sunday` or `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, `Sun`

  The days of the week. Use the identifier `weekday` to extract a day of the week from a date.

  ```
  set theDate to current date --using scripting addition command
  --result: date "Saturday, February 27, 1999 3:37:50 PM"
  weekday of theDate --result: Saturday
  ```

- `January`, `February`, `March`, `April`, `May`, `June`, `July`, `August`, `September`, `October`, `November`, `December` or `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, `Dec`

  The months of the year. Use the identifier `month` to extract a month from a date.

  ```
  month of theDate --result: February
  ```

For more information on working with date and time values, see "Date" (page 62) and "Date-Time Arithmetic" (page 233).

Miscellaneous Script Constants

AppleScript defines the constants `anything`, `current application`, `it`, `me`, `missing value`, `my`, and `result` to help perform various tasks in your scripts. The constants `it`, `me`, and `my` are described in "Using it, me, and my in Tell Statements" (page 242).

- `anything`

  The constant `anything` is rarely used in a script, though you might choose to do something like the following:

```
set myVariable to anything
    -- perform operations that might change the value of myVariable
if myVariable is equal to anything then
    -- perform certain steps, knowing the variable never got changed
else
    -- perform other steps if the variable got changed
end if
```

  Contrary to what you might think, the Boolean statement `if myVariable is equal to anything` evaluates to `true` only if `myVariable` is specifically equal to the constant `anything`. That is, comparing the contents of a variable to `anything` doesn't guarantee a `true` result.

  You may also see `anything` as a parameter to an application command or to a scripting addition command. Here, the meaning is different, and does indicate that the command accepts any kind of input value for that parameter. For an example, see the With Data parameter of the Make command in the AppleWorks dictionary. "Dictionaries" (page 34) describes how to examine a dictionary.

- `current application`

  The `current application` is either the default target application or whatever application is currently set as a script's parent property. You can make any application the current application for a script or script object simply by declaring it as a parent property. Any subsequent command in the script for which the script doesn't have a handler is passed to the application you declare as the parent, and subsequent occurrences of the constant `current application` refer to that application.

For example, the following script declares the Finder as its parent property, then sends commands that close the Finder's frontmost window and return the application's name:

```
property parent: application "Finder"
close front window
tell current application to return my name --result: "Finder"
```

For more information, see "The Parent Property and the Current Application" (page 341).

- `missing value`

  The `missing value` constant is a placeholder for missing information. For example, if your script asks the Network Setup Scripting application for the zone of every connection, you might get back a list that showed the actual zone for every AppleTalk connection, but the item `missing value` for every TCP/IP connection because TCP/IP connections do not include a zone. If the current setup had two AppleTalk connections and two TCP/IP connections, the resulting list might look something like the following:

  ```
  {"4th Floor South", "4th Floor North", missing value, missing value}
  ```

  You could then perform operations such as the following, using the `missing value` constant:

  ```
  if item 3 of myZoneList = missing value then
      --do one thing if the connection does not report a zone
  else
      --do something else if the connection reports a zone
  end if
  ```

- `result`

  AppleScript stores the result of a command in the predefined variable `result`. The value remains there until the next command is executed. If a command does not return a result, the value of `result` is undefined.

  ```
  tell application "Finder"
      count files in folder "Apple Extras" of startup disk
      set numFiles to result -- Save result in numFiles variable.
  end tell
  ```

  For more information, see "Using the Predefined Result Variable" (page 122)

Save Option Constants

AppleScript defines the constants `yes`, `no`, and `ask` for use with the Close command. These constants are described in Table A-2 (page 355).

The following script creates a new document, inserts some text into it, and closes the document, asking the user whether to save it.

```
tell application "AppleWorks"
    -- Create a new document and insert some previously gathered text.
    make new document at beginning with properties ¬
        {name:"New Report"}
    -- Create a selection and replace it with the gathered text.
    select text body of document "New Report"
    set selection of document "New Report" to myText
    close document "New Report" saving ask
end tell
```

The term `saving ask` means ask the user whether to save, while `saving no` means do not save any changes to the closed document and `saving yes` means save without asking.

String Constants

AppleScript defines the single-character string constants `return`, `space`, and `tab` to represent, respectively, a return character, a space character, and a tab character. You can use them with the concatenation operator (&) to add them to a string, or you can use them in comparison operations.

```
set addressString to return & "66601 Colton Blvd." & return ¬
    & "Oakland, CA 94611" & return
--result: two-line address, starting on a new line
```

These string constants are also listed in Table A-3 (page 358).

Text Style Constants

You can use the following constants, also listed in Table A-2 (page 355), to specify text style characteristics:

```
all caps, all lowercase, bold, condensed, expanded, hidden, italic,
outline, plain, shadow, small caps, strikethrough, subscript,
superscript, underline
```

The following script gets the text style from the text body of an open AppleWorks word processing document:

```
tell application "AppleWorks"
    -- Get style of text from open document.
    style of text body of document "Draft Report"
end tell

--result:
{class:text style info, on styles:{plain}, off styles:{italic, underline,
outline, shadow, condensed, expanded, strikethrough, superscript,
subscript, superior, inferior, double underline}}
```

Depending on the application, you can get and set text style characteristics for individual characters, words, paragraphs, or the entire text.

Version Constant

You can use AppleScript's `version` property to check the current version of AppleScript. The following script checks for a version greater than or equal to version 1.3.4 before performing any other operations.

```
if (version as string) ≥ "1.3.4" then
    -- Perform any operations that depend on the
    -- presence of AppleScript version 1.3.4 or later.
end if
```

Version is a property of other applications as well as AppleScript, so to access the AppleScript version inside a Tell block to another application, you must use the phrase `AppleScript's version` or `version of AppleScript`:

```
tell application "AppleWorks"
    version --result: "5.0v1" (version of AppleWorks)
    if (AppleScript's version as string) ≥ "1.3.4" then
    -- Perform any operations that depend on the
    -- presence of AppleScript version 1.3.4 or later.
    end if
end tell
```

For a description of the global variable `AppleScript`, see "AppleScript Properties" (page 210).

The following script checks for version 8.5 or later of the Finder, and puts up a warning if it isn't present (using the Display Dialog scripting addition command).

```
tell application "Finder"
    --check to make sure System 8.5 or later is installed
    set theVersion to (get the version) as number
    if theVersion is less than 8.5 then
        beep
        display dialog ¬
            "This script requires MacOS 8.5 or later." ¬
            buttons {"Cancel"} default button 1 with icon 0
    end if
end tell
```

Commands

A command is a word or series of words used in AppleScript statements to request an action. Every command is directed at a target, which is the object that responds to the command. The target is usually an application object, but it can also be a script object or a user-defined subroutine or value in the current script.

Not all commands can be used with all types of targets. When you use a command to request an action, you must choose a command that works with the target you want to manipulate. You must also be sure to specify the target correctly. Several factors, including the direct parameter you provide with a command and whether or not the command is included in a Tell statement, can determine the target of a command.

Commands are described in the following sections:

- "Types of Commands" (page 110) describes the types of commands you use with AppleScript. It also discusses targets of commands and summarizes which types of commands work with which types of targets.
- "Using Command Definitions" (page 115) describes the details of using commands and command definitions.
- "Using Parameters" (page 118) describes how to coerce parameters, how to use parameters that specify locations, and how to deal with raw data in parameters.
- "Using Results" (page 121) describes how to view the result values generated by AppleScript commands and how to use the predefined result variable.
- "Double Angle Brackets in Results and Scripts" (page 123) explains what it means when you see double angle brackets («») in a script or a result.
- "Command Definitions" (page 127) provides detailed descriptions of the five AppleScript commands and of many standard application commands.

Types of Commands

The following sections describe the different types of commands and their targets. There are four types of commands you can use in AppleScript to request actions:

- "Application Commands" (page 110)
- "AppleScript Commands" (page 112)
- "Scripting Addition Commands" (page 112)
- "User-Defined Commands" (page 114)

This chapter provides detailed descriptions of all AppleScript commands and many standard application commands. It provides a brief overview of scripting additions and user-defined commands and provides pointers to additional information on these types of commands.

Each time you use a command, you specify the **target,** or recipient, of the command. Potential targets include application objects, script objects, the current script, and the current application. In some cases you specify the target explicitly by including it in a Tell statement or supplying it as a direct parameter. In other cases you specify the target implicitly.

Application Commands

Application commands are commands that cause actions in scriptable applications. The target of an application command is an application object or a script object. Different application objects respond to different commands. You can determine which commands an application supports by examining the application's dictionary. You can view an application's dictionary by dropping the application's icon on the Script Editor's icon, or by opening the application with the Script Editor's Open Dictionary command. Table 5-1 (page 163) and Table 5-2 (page 164) show excerpts from the dictionaries for the Finder and for AppleWorks, respectively. For related information, see "Dictionaries" (page 34).

There are two ways to specify an object as the target of a command: in the direct parameter of the command or in a Tell statement that contains the command.

The **direct parameter** is a value, usually a reference, that appears immediately after a command and specifies the target of the command. Not all commands have a direct parameter. If a command can have a direct parameter, the command's definition says so.

For example, in the following statement, the reference `last file of window 1 of application "Finder"` is the direct parameter of the Duplicate command:

```
duplicate last file of window 1 of application "Finder"
```

A Tell statement is a statement that specifies a default target for all commands contained within it. If a command is contained within a Tell statement, the direct parameter is optional. If you leave out the direct parameter, AppleScript uses the default target specified in the Tell statement. For example, the Duplicate command in the following Tell statement has the same effect as the Duplicate command in the previous example:

```
tell last file of window 1 of application "Finder"
    duplicate
end tell
```

Similarly, if you specify a reference incompletely in the command line, AppleScript uses the default target specified in the enclosing Tell statement to complete the reference. For example, the following statement is equivalent to both of the previous examples:

```
tell window 1 of application "Finder"
    duplicate last file
end tell
```

Remember that you can make your script statements look more like written English if you choose. For example, the following statement is the same, syntactically, as the first Duplicate statement above:

```
duplicate the last file of the first window of application "Finder"
```

For a definition of script objects, as well as information on sending application commands to script objects, see "Script Objects" (page 325).

AppleScript Commands

AppleScript commands are commands that are built into the AppleScript language. They act on values in scripts. The target of an AppleScript command is a value in the current script, which is usually specified in the direct parameter of the command.

There are only five AppleScript commands: Get, Set, Count, Copy, and Run. All of these commands except the Copy command can also function as application commands. For the Count, Get, Run, and Set commands, if the direct parameter is a value, then the command functions as an AppleScript command. If the direct parameter is an application object, the command functions as an application command.

For example, the following Count command functions as an AppleScript command because the direct parameter is a value (a list):

```
count {"How", "many", "items", "in", "this", "list"}
```

The following Count command functions as an application command because the direct parameter is an application object:

```
count the files in the first window of application "Finder"
```

For more examples of how to use Copy, Count, Get, Run, and Set, see the command definitions later in this chapter.

Scripting Addition Commands

Scripting additions are files that provide additional commands you can use in scripts. Each scripting addition can contain one or more command handlers. If a scripting addition is located in the Scripting Additions folder (in the System Folder), the command handlers it provides are available for use by any script whose target is an application on that computer.

For information about the standard scripting addition commands distributed with AppleScript, see the AppleScript section of the Mac OS Help Center, or see the following website:

<http://www.apple.com/applescript/>

Target

Like the target of an application command, the target of a scripting addition command is always an application object or a script object. If the script doesn't explicitly specify the target with a Tell statement, AppleScript sends the command to the default target application, which is usually the application running the script (for example, the Script Editor).

A scripting addition command performs its action only after the command has been received by a target application. Unlike application commands, scripting addition commands always work the same way regardless of the application to which they are sent.

For example, the scripting addition command Display Dialog displays a dialog box that can include text, one or more buttons, an icon, and a field in which the user can type text. In the script that follows, the target of the Display Dialog command is the Finder application. When the script runs, the Finder passes the command to the scripting addition's handler for the Display Dialog command, which displays the dialog box.

```
tell application "Finder"
    display dialog "What's your name?"
end tell
```

In the next example, the Display Dialog command is not enclosed in a Tell statement, nor does it have a direct parameter, so its target is the Script Editor (or whatever application runs the script). When you run the script, the Script Editor passes the command to the scripting addition's handler for the Display Dialog command, which displays the dialog box in the Script Editor's layer (that is, in front of any other Script Editor windows that may be open), while the Script Editor is still the active application.

```
set theCount to number of files in front window of application "Finder"
if theCount > 500 then
    display dialog "You have exceeded your file limit."
end
```

If you specify a script object as the target of a scripting addition command, the script object either handles the command itself (potentially modifying it) or passes the command to the default target application. For more information about scripting additions and script objects, see "Using Continue Statements to Pass Commands to Applications" (page 339).

Name Conflicts

Each scripting addition that contains command handlers has its own dictionary, which lists the reserved words—including the command names, parameter labels, and in some cases object names—used to invoke the commands supported by the scripting addition. You can view a scripting addition's dictionary by dropping the scripting addition's icon on the Script Editor's icon, or by opening the scripting addition with the Script Editor's Open Dictionary command.

If a scripting addition dictionary includes words that are also part of an application dictionary, then you cannot use those words within Tell statements to that application.

IMPORTANT

Every event handler, class definition, and even enumeration that is defined in the dictionary of a scripting addition is global to AppleScript. If a common term such as "search" (or any other term) is used in a scripting addition, that effectively preempt its use in any scriptable application. If you are having trouble using a term with an application, check whether that term is also used in a scripting addition and, if necessary, remove that scripting addition. ▲

User-Defined Commands

User-defined commands are commands that trigger the execution of collections of statements, called subroutines, elsewhere in the same script. The target of a user-defined command is the **current script,** that is, the script from which the command is executed.

There are two ways to specify the current script as the target of a user-defined command. Outside of a Tell statement, simply use the command to specify the current script as its target. For example, suppose that `minimumValue` is a command defined by the user in the current script. The handler for the `minimumValue` command is a subroutine that returns the smaller of two values. The target of the `minimumValue` command in the following example is the current script:

```
set theCount to minimumValue(12,105)
```

Inside a Tell statement, use the words `of`, `me`, or `my` to indicate that the target of the command is the current script and not the default target of the Tell statement. For example, the following sample script shows how to call the `minimumValue` subroutine from within a Tell statement:

```
tell application "Finder"
    set fileCount to count files in front window
    set myCount to my minimumValue(fileCount, 100)
    --do something with up to the first 100 files…
end tell
```

Without the word `my` before the `minimumValue` command, AppleScript would send the `minimumValue` command to the Finder, resulting in an error.

"Handlers" (page 279) describes the syntax for defining and invoking subroutines such as `minimumValue` in more detail.

Note
You can also define subroutines in script objects. The target of a user-defined command whose subroutine is defined in a script object is the script object. For information about defining and invoking subroutines in script objects, see "Script Objects" (page 325). ◆

Using Command Definitions

Each of the command definitions in this chapter provides information about what a command does and how to use it in a script. The information is divided into the following categories:

- "Syntax" (page 116)
- "Parameters" (page 116)
- "Result" (page 117)
- "Examples" (page 117)

In addition, some command definitions provide information about errors. For an example of a definition that includes each of these categories, see the command "Get" (page 141).

Syntax

Each command definition begins with a **syntax description,** which is a template for using the command in a statement. Syntax descriptions use the same typographic conventions used elsewhere in this guide: plain computer font indicates a language element you must type exactly as shown; italic text indicates a placeholder you must replace with an appropriate value; brackets indicate the enclosed language element or elements are optional; three ellipsis points indicate you can repeat the preceding element or elements one or more times; and vertical bars separate elements from which you must choose a single element.

To create a "Move" (page 148) command statement from the Move syntax description, you must replace *referenceToObject* with a reference to the object to move and *referenceToLocation* with a reference to the location to which to move it. For example:

```
move file "Bob" of startup disk to folder "Joe" of startup disk
```

The term `startup disk` is described in "Viewing a Result in the Script Editor's Result Window" (page 121). References are described in "Objects and References" (page 161).

The use of the continuation character (¬) in a syntax description, such as that of the application command "Make" (page 146), indicates that the following language element must be placed on the same line as the previous element. The continuation character itself is not a required part of the syntax—it is merely a mechanism for extending a statement to include the next line.

Parameters

Parameters are values that are included with a command. The "Parameters" section of a command definition lists the parameters of a particular command and the information you need to use them correctly.

Many commands include a *direct parameter* that specifies the object of the action. If a command includes parameters other than the direct parameter, they are identified by labels. Parameters that are identified by labels are called **labeled parameters.** The direct parameter immediately follows the command; labeled parameters can be listed in any order. For example, the following is the syntax for the "Move" (page 148) command:

`move` *referenceToObject* `to` *referenceToLocation*

The Move command has a direct parameter (*referenceToObject*) that specifies the object to move and a labeled parameter (whose label is `to`) that specifies where to move the object. An actual Move statement looks like the following:

```
move currentReport to ReportFolder
```

Each parameter value must belong to a particular class, which is listed in its description in the command definition. For the Move command, the direct parameter belongs to the class reference. Its value, a reference, is a phrase that identifies the object to be moved. The `to` parameter also belongs to the class reference. It specifies the location to which to move the object. References are described in "Objects and References" (page 161).

Parameters can be required or optional. **Required parameters** must be included with the command; **optional parameters** need not be. Optional parameters are enclosed in brackets in syntax descriptions. For optional parameters, the description in the "Parameters" section specifies a default value that is used if you don't include the parameter.

For more information about direct parameters, see "Application Commands" (page 110). For more information about using parameters, see "Using Parameters" (page 118).

Result

Many, but not all, commands return results. The **result** of a command is the value generated when the command is executed. The "Result" section of a command definition tells whether a result is returned, and if so, lists its class. For example, the result of an AppleScript "Count" (page 134) command is an integer that specifies the counted number of elements of a specified class.

For more information about results, see "Using Results" (page 121).

Examples

Each command definition includes one or more short examples demonstrating how to use the command. For example, see the definition for the application command "Move" (page 148).

Errors

Commands can return error messages as well as results. An **error message** is a message that is returned by an application, AppleScript, or the operating system if an error occurs during the handling of a command. The "Errors" section of a command definition, if present, lists errors that are likely to be returned by a particular command. This information can help you decide if you need to write *error handlers* to respond to the error messages that are returned. Error handlers are described in "Handlers" (page 279).

Some "errors" are not the result of abnormal conditions but are the normal way you get information about what happened during command execution. For example, you use the Choose File scripting addition command to ask the user to choose a file. When AppleScript executes this command, it displays a dialog box similar to the one you get when you choose Open from the File menu. If the user presses the Cancel button in the dialog box, AppleScript returns error number –128 and the error string `"User canceled"`. Your script must handle this error for script execution to continue.

For a complete description of handling errors that occur during script execution, see "Handlers" (page 279).

Using Parameters

The following sections provide information for working with parameters:

- "Coercion of Parameters" (page 118) describes issues involved in converting a parameter from one class to another.
- "Parameters That Specify Locations" (page 119) describes situations in which a command uses a parameter to specify data insertion or replacement.
- "Raw Data in Parameters" (page 120) describes how AppleScript displays data that doesn't belong to any of the basic value classes.

Coercion of Parameters

If a parameter doesn't belong to the right class, it may be possible to coerce it, that is, to change it into a value of another class. For example, you can coerce an integer such as 2 to the corresponding string `"2"` using the As operator:

```
2 as string
--result: "2"
```

AppleScript performs some coercions, including the previous one, automatically. For example, the following statement uses the Copy command to set the name of a folder.

```
tell application "Finder"
    copy 12 to name of folder 1 of disk "RAM disk"
end tell
```

Because a folder name is a string, the direct parameter of the Copy command should also be a string. When AppleScript executes this script, it automatically coerces the integer 12 to the string `"12"` and uses that string to set the name of the first folder.

The coercions that AppleScript can perform are listed in "Values and Constants" (page 51). Applications can also perform additional coercions, such as coercions for classes that are specific to an application. These coercions are listed in the documentation for the application. Scripting additions, which are described in "Scripting Addition Commands" (page 112), can also perform coercions.

Parameters That Specify Locations

Many commands have parameters that specify locations. A location can be either an insertion point or another object. An **insertion point** is a location where an object can be added. An object, when used as a location parameter, is an object to be replaced by another object.

For example, in the following statement, the `to` parameter specifies the location to which to move the first word. The value of the `to` parameter of the Move command is the reference `before word 10 of text body`, which is an insertion point.

```
tell front document of application "AppleWorks"
    move word 1 of text body to before word 10 of text body
end tell
```

The following example gets a reference to a document's selected text, then moves the first word of the document to the location of the selected text. The

value of the `to` parameter of the Move command is an object, `selectedText`, which is replaced with `word 1`.

```
tell application "AppleWorks"
    tell document "Simple"
        set selectedText to selection
        move word 1 of text body to selectedText
    end tell
end tell
```

Phrases such as `word 1` and `before word 10` are called index references and relative references, respectively. These kinds of references specify locations. For more information about these kinds of references, see "Index" (page 177) and "Relative" (page 185).

Raw Data in Parameters

Some application commands return values that do not belong to any of the normal AppleScript value classes. For example, the Edit Graphic command supported by some graphics applications returns values that belong to the class Data, which is described in "Data" (page 61). In its result window, the Script Editor displays values of class Data between double angle brackets. You can store such data in variables and send them as parameters to other commands. For example, if it's necessary to use two different applications to edit a graphic, you can send the data value returned by one Edit Graphic command as the direct parameter of another Edit Graphic command.

The Script Editor also displays unicode text values in the result window as raw data. The following example shows how the string "hello" is displayed as unicode text.

```
set myString to "hello" as Unicode text
--result: «data utxt00680065006C006C006F»
```

If an application returns values of class Data, its documentation should say so.

For information on other places where AppleScript uses double angle brackets, see "Double Angle Brackets in Results and Scripts" (page 123). For more information on unicode text, see "Unicode Text and International Text" (page 87).

Using Results

The following sections describe how you work with the results generated when AppleScript executes a command:

- "Viewing a Result in the Script Editor's Result Window" (page 121)
- "Using the Predefined Result Variable" (page 122)

For related information, see "The Return Statement" (page 281).

Viewing a Result in the Script Editor's Result Window

The result of a command is the value generated when the command is executed. You can display the result of a command in the Script Editor by using the Show Result command from the Controls menu to open the result window. For example, if you run the following script,

```
tell application "Finder"
    duplicate folder "Apple Extras" of startup disk
end tell
```

and then choose Show Result in the Script Editor, you'll see a value similar to

```
folder "Apple Extras copy" of disk "Hard Disk" of application "Finder"
```

The term `startup disk` is one of several special folder and disk names that the Finder understands. Others include `apple menu items folder`, `control panels folder`, `desktop`, `extensions folder`, `fonts folder`, `preferences folder`, and `system folder`. You can read more about scripting the Finder in the AppleScript section of the Mac OS Help Center.

Some commands return a result as a value. For example, the Count command in the following statement returns a value: the number of files (not including folders or files enclosed in folders) in the specified folder.

```
tell application "Finder"
    count files in folder "Apple Extras" of startup disk
end tell
```

You can use this statement anywhere a value is required by enclosing the statement in parentheses. For example, the following statement sets the value of `numFiles` to the value returned by the Count command.

```
tell application "Finder"
    set numFiles to ¬
        (count files in folder "Apple Extras" of startup disk)
end tell
```

For more information on how to use the Script Editor, see the AppleScript section of the Mac OS Help Center.

Using the Predefined Result Variable

In addition to displaying the result of a command in the result window, AppleScript puts the result into a predefined variable called `result`. The value remains there until the next command is executed. If the next command does not return a result, the value of `result` is undefined. The following two commands show how to use the `result` variable to set the value of `numFiles` to the value returned by the Count command:

```
tell application "Finder"
    count files in folder "Apple Extras" of startup disk
    set numFiles to result
end tell
```

When a direct parameter specifies more than one object, the result is a list that contains a value for each object that was handled. Here is an example of a command whose result is a list:

```
tell application "Finder"
    get name of every file in folder "Apple Extras" of startup disk
end tell
```

The result is a list of strings, one for each file (not including folders or files enclosed in folders). Depending on the contents of the Apple Extras folder, the list looks something like the following:

```
{"Key Caps", "Language Register", "Register with Apple"}
```

The first string is the name of the first file, the second string is the name of the second file, and so on.

Double Angle Brackets in Results and Scripts

When you type English language script statements in a Script Editor script window, AppleScript is able to compile the script because the English terms are described either in the terminology built into the AppleScript language or in the dictionary of an available scriptable application or scripting addition. When AppleScript compiles your script, it converts it into an internal executable format, then reformats the text as described in "Compiling Scripts With the Script Editor" (page 47).

When you open, compile, edit, or run scripts with the Script Editor, you may occasionally see terms enclosed in **double angle brackets,** or chevrons («»), in a script window or in the result window. For example, you might see the term `«event sysodlog»` as part of a script. You will typically see text enclosed in chevrons for one of three reasons:

- AppleScript can't reformat a term in English in a script window because the term is not part of the AppleScript language and no dictionary that defines the term is available. This situation is described in "When a Dictionary Is Not Available" (page 123).
- AppleScript can't display data in the data's native format in the result window. This situation is described in "When AppleScript Displays Data in Raw Format" (page 125).
- You intentionally entered the chevrons (by typing Option-Backslash and Shift-Option-Backslash). This situation is described in "Entering Script Information in Raw Format" (page 125).

When a Dictionary Is Not Available

AppleScript uses double angle brackets in a Script Editor script window when it can't identify a term or can't display a value directly. The first word within the double angle brackets can be any of the following: `event`, `property`, `class`, `data`, `preposition`, `keyform`, `constant`, or `script`. The second word varies depending on the context.

AppleScript can not display a term in English if isn't a part of the AppleScript language and it isn't defined in an application or scripting addition dictionary that is available when the script is opened or compiled. That usually happens for one of two reasons:

- A required application or scripting addition dictionary isn't physically present when the script is opened or compiled (possibly because the script was compiled on one machine and opened on another).
- A required application or scripting addition dictionary is available but doesn't support a term used in the script (most likely because the dictionary is from an older version of the application or scripting addition).

As an example of a missing dictionary, suppose you create a script that uses the Display Dialog scripting addition command, then open the script when the Standard Additions scripting addition (which includes the Display Dialog command) is not present. AppleScript replaces the words `display dialog` in the script with `«event sysodlog»`. In this case, you should make sure the Standard Additions scripting addition is present in the Scripting Additions folder (which is located in the System folder) before attempting to compile or run the script.

As an example of a missing term, suppose you create the following script, which uses the `delay` scripting addition command, available starting with Mac OS 8.5:

```
display dialog "Ready to test your patience?"
set myDate to current date
delay 4
display dialog "4 second delay on " & (date string of myDate) & "."
```

This script displays a dialog, waits for the user to dismiss it, then delays for four seconds before displaying a second dialog that includes the current date. If you save this script as text and compile it on a machine running Mac OS 8.1, the Script Editor will display the following:

```
display dialog "Ready to test your patience?"
set myDate to current date
«event sysodela» 4
display dialog "4 second delay on " & (date string of myDate) & "."
```

AppleScript converts the term `delay 4` to `«event sysodela» 4` because the Delay scripting addition command is not available with Mac OS 8.1. Without the Delay term in an available dictionary, AppleScript doesn't have an English

language term to display. The script compiles, but won't run correctly on the machine with Mac OS 8.1 because the Delay command isn't available. If you save the compiled script, then move it to the original machine with Mac OS 8.5, it will run correctly. If you recompile the script on the original machine, AppleScript converts `«event sysodela»` back to `delay 4`.

For related information, see "Entering Script Information in Raw Format" (page 125).

When AppleScript Displays Data in Raw Format

Double angle brackets can also occur in results. For example, if the value of a variable is a script object named `Joe`, AppleScript represents the script object as shown in this script:

```
script Joe
    property theCount : 0
end script

set x to Joe
x
--result: «script Joe»
```

For more information about script objects, see "Script Objects" (page 325).

Similarly, if the value of a variable is of class Data and the Script Editor can't display the data directly in its native format, it uses double angle brackets to enclose both the word `data` and a sequence of numerical values that represent the data. Although this may not visually resemble the original data, the data's original format is preserved. You can treat the data like any other value, except that you can't view it directly in its native format in a Script Editor window. For an example, see "Raw Data in Parameters" (page 120).

Entering Script Information in Raw Format

You can enter double angle brackets, or chevrons («»), directly into a script by typing Option-Backslash and Shift-Option-Backslash. There are several reasons you might choose to do this:

- You're creating a script at home to use on a work machine that is running a newer version of the Mac OS or a newer version of a scriptable application or scripting addition. Drawing from the example in "When a Dictionary Is Not Available" (page 123), suppose you want to use the scripting addition command Delay, but are running Mac OS 8, which doesn't support that command. At home, you can write `«event sysodela» 4` in your script. When you open the script at work, AppleScript converts the term to `delay 4`. A problem with this approach is that you can't check the validity of your script until you try it on the target machine.
- You know that an application supports a certain Apple event but it doesn't supply terminology for the event in its dictionary. For example, if you're a developer creating a scriptable application, you may want to test a feature you've added to the code but not yet added to the application's dictionary.

You can also use AppleScript to insert chevrons into a script, using the following steps:

1. Create a script using standard terms compiled against an available application or scripting addition.
2. Save the script as text and quit the Script Editor.
3. Remove the application or scripting addition from the computer.
4. Open the script again and compile it.
5. When AppleScript asks you to locate the application or scripting addition, specify a file that doesn't contain a terminology.

The script will compile successfully, but the Script Editor will display the script with chevron format for any terms that rely on a missing dictionary. (To recompile the script again and supply the correct dictionary, save it as text and quit the Script Editor again, then open and recompile it, specifying the correct application or scripting addition.)

There are several situations in which you might want to recompile a script this way.

- You're creating a script that will target an application on a remote machine. AppleScript doesn't currently allow you to compile against the dictionary of an application or scripting addition on a remote machine. If you compile using a local copy of the application, you are not only using its dictionary (which may be the same as for the remote application), you are also targeting the local application. However, you can write and compile the script with the local version of the application, insert chevrons as described above so that

the target is no longer the local application, then type in the remote machine target, as described in “References to Remote Applications” (page 196).

- You want a Tell block loop variable to loop over items (such as disk drives) on a remote machine. As described in the previous item, you can compile your script using a local copy of the application, insert chevrons as described above, then type in the remote machine target.

Sending Raw Apple Events From a Script

The section “Entering Script Information in Raw Format” (page 125) describes how you can use double angle brackets (or chevrons) to enter raw information directly into a script. You enter these symbols («») by typing Option-Backslash and Shift-Option-Backslash.

Using chevrons, you can directly enter a term such as `«event sysodlog»` (equivalent to `display dialog`) in a script. If the Display Dialog command *is* available, AppleScript will convert `«event sysodlog»` to `display dialog` when you compile the script. The term `«event sysodlog»` is actually the raw form for an Apple event with event class `'syso'` and event ID `'dlog'`. You can use raw syntax to enter and execute events (even complex events with numerous parameters) when there is no dictionary to support them. However, this guide does not provide detailed documentation for raw syntax.

Command Definitions

The sections that follow are in alphabetical order by command name and provide definitions for both AppleScript commands and standard application commands. The general features of these types of commands are described in “Types of Commands” (page 110). The command type is noted in the brief command description at the beginning of each definition.

Table 4-1 summarizes the standard application-only commands described in this chapter and provides links to sections that describe the commands in detail. Table 4-2 does the same for the AppleScript commands defined in this chapter. Some of the commands in Table 4-2 are also implemented as application commands.

For more information about the standard scripting addition commands distributed with AppleScript, see the AppleScript section of the Mac OS Help Center. For definitions of commands provided by other scripting additions, see the documentation for those scripting additions.

The application commands defined in this chapter are standard application commands supported by most scriptable applications. The definitions in this chapter describe how these commands work in most applications. Individual applications can extend or change the way the standard application commands work.

Application dictionaries list application commands by suites. Each **suite** defines the Apple event constructs (object class definitions, descriptor types, and so on) needed for performing a particular type of scriptable activity. The suite categories include the Standard suite, the Internet suite, the Text suite, and so on. Different applications may support different commands in the Standard suite, but all support the Open, Print, Quit, Run, and Reopen commands, which were formerly in a separate Required suite.

Note
If an application supports just the four required commands it is not considered scriptable and it doesn't need a dictionary. If it supports the required commands in the standard way and supports additional commands as well, it needn't include the required commands in its dictionary. In either case, the required commands will be available to scripters (they will compile and run with the Script Editor). ◆

Many applications also define their own suite of more specialized commands. The application's dictionary provides definitions of all commands supported by the application (with the exception noted above for the four required commands). Check the appropriate application dictionary before using application commands. You can view the dictionary of an application or scripting addition by dropping its icon on the Script Editor's icon, or by

opening the application or scripting addition with the Script Editor's Open Dictionary command.

Table 4-1 Standard application-only commands

Command	Description
Must be supported by all applications	
"Launch" (page 143)	Launches an application without invoking its standard startup procedures. This command is handled differently than other commands in this table. Refer to the definition for details.
"Open" (page 149)	Opens one or more files.
"Print" (page 150)	Prints one or more objects.
"Quit" (page 151)	Terminates an application.
"Reopen" (page 152)	Brings an already-open application to the front and re-invokes its standard startup procedures.
"Run" (page 154)	Launches an application and invokes its standard startup procedures.
Commonly supported by scriptable applications	
"Close" (page 130)	Closes one or more objects.
"Count" (page 134)	Counts elements of a particular class in an object.
"Delete" (page 137)	Deletes one or more objects.
"Duplicate" (page 138)	Copies an object or objects to a new location.
"Exists" (page 139)	Determines if an object exists.
"Get" (page 141)	Returns the value of an object.
"Make" (page 146)	Creates a new object.
"Move" (page 148)	Moves an object or objects.
"Save" (page 156)	Saves an object to a file.
"Set" (page 157)	Assigns a value to an object.

Table 4-2 lists the AppleScript commands defined in this chapter. Some of these commands are also implemented as.application commands

Table 4-2 AppleScript and application commands

Command	Description
"Copy" (page 132)	Assigns a value to a variable or object.
"Count" (page 134)	Counts the elements of a composite value.
"Get" (page 141)	Returns the value of an expression.
"Run" (page 154)	Executes statements other than handler and property definitions in a script object definition.
"Set" (page 157)	Assigns a value to a variable or an object.

Another AppleScript command, the Error command, is described in "Try Statements" (page 259).

Close

Close is an application command that requests the closing of one or more objects, usually application windows or documents.

SYNTAX

```
close referenceToObject [ saving in referenceToFile ] [ saving saveOption ]
```

PARAMETERS

referenceToObject

A reference to the object or objects to close, usually application windows or documents.
Class: Reference

referenceToFile A reference of the form `file` *nameString* or `alias` *nameString* (see "Notes" below).
Class: Reference
Default value: The file in which the object was last saved. If the object hasn't been saved before, the application follows its normal procedure when first saving a new file, which is typically to ask whether to save the document and, if saving, to let the user specify a name and destination for the file.

saveOption A parameter that specifies whether to save an object that has been modified before closing it. The constant `yes` specifies that the object should be saved. The constant `no` specifies that the object should not be saved. The constant `ask` specifies that the user must be asked whether or not to save the object. See "Notes" below for more information.
Class: Constant
Default value: The default value is `ask`, unless you specify a file in which to save the object, in which case the default value is `yes`.

RESULT

None

EXAMPLES

```
tell application "AppleWorks"
    close front window saving in file "Hard Disk:Documents:Report"¬
        saving yes
end tell

tell application "Finder"
    close front window
end tell
```

NOTES

To specify the name (*nameString*) of a file in which to save the object, use a string of the form "*Disk*:*Folder1*:*Folder2*:. . . :*Filename*"; for details, see "References to Files" (page 191). You can also specify a string with only a filename ("*Filename*").

In this case, the application attempts to find the file in the current directory. If it can't find the specified file, the application creates a file with the specified name in the current directory or, if the name includes a path, then in the directory specified by the path. (The current directory is typically the directory where the application was launched, the directory where the application last opened or saved a previous document, or another directory specified by the application. The current directory may be affected by settings in the General Controls control panel.)

The Close command will typically only save a file if it is "dirty" (has been modified since it was last saved). When the save option is `ask`, the Close command asks, for each modified document, whether to save the document. Depending on how an the application implements the Close command, the command may save over an existing file with the same name as the specified file without asking.

Copy

The Copy command is an AppleScript command that makes a copy of one or more values and stores the result in one or more variables. For information on differences between the Copy command and the Set command, see "Data Sharing" (page 206). The Set command is described in "Set" (page 157).

To copy within an application, you should use the command "Duplicate" (page 138). To copy between applications, performing the equivalent of the Edit menu's Copy command, you typically use the Clipboard-related scripting addition commands that are part of AppleScript's Standard Additions. For information about the standard scripting addition commands distributed with AppleScript, see the AppleScript section of the Mac OS Help Center.

As shown in the syntax definition, `put` and `into` are synonyms for `copy` and `to`. When you compile a script, `put` and `into` are automatically changed to `copy` and `to`.

SYNTAX

`( copy | put )` *expression* `( to | into)` *variablePattern*

PARAMETERS

expression The expression whose value is to be assigned. If *expression* is a reference or a list or record of references, AppleScript gets the values of the objects specified by the references.
Class: Any class

variablePattern
The name of the variable in which to store the value, or a list of variable patterns, or a record of variable patterns.
Class: Identifier, list, or record

RESULT

If the Copy command is used to assign a value to a variable, the result is the value that was stored in the variable. If the command is used to copy an object, the result is a reference to the copied object.
Class: Varies

EXAMPLES

The following example copies a string to the variable `myOccupation`:

```
copy "writer" to myOccupation
```

The next example gets a reference to a window, copies it to another reference, then gets the window's name form the copied reference:

```
set windowRef to a reference to window 1 of application "Finder"
copy windowRef to currentWindow
name of currentWindow --result: “Script testing folder”
```

In addition to copying a value to a single variable or object, you can copy patterns of values to patterns of variables. For example, this script copies the position of the front window to a list of two variables:

```
tell application "Finder"
    copy position of front window to {x, y} --result: {13, 47}
end tell
```

Since the Finder returns `position of front window` as a list of two integers, the preceding example copies the first item in the list to `x` and the second item in the list to `y`.

Patterns copied with the Copy command can also be more complex. For example:

```
set x to {8, 94133, {firstName:"John", lastName:"Chapman"}}
copy x to {p, q, {lastName:r}}
return(p, q, r) --result: {8, 94133, "Chapman"}
```

As this example demonstrates, the properties of a record need not be given in the same order and need not all be used when you copy a pattern to a pattern, as long as the patterns match.

The use of the Copy command with patterns is similar to the use of the Set command with patterns. For information about the Set command, see (page 157).

NOTES

For more information about using the Copy command to create or change the values of variables, see “Variables” (page 200).

Count

The Count command can function as an AppleScript command or an application command. The AppleScript command counts the number of elements of a particular class in a list, record, or string. The application command counts the number of elements of a particular class in an object or objects.

APPLESCRIPT COMMAND SYNTAX

`count [ [ each | every ]` *className* `|` *pluralClassName* `( in | of ) ]` *compositeValue*

`number of [` *className* `|` *pluralClassName* `( in | of ) ]` *compositeValue*

APPLICATION COMMAND SYNTAX

`count [ each | every ]` *className* | *pluralClassName* `[ ( in | of )` *referenceToObject* `]`

`number of` *className* | *pluralClassName* `[ ( in | of )` *referenceToObject* `]`

PARAMETERS

className The class name of the elements to be counted. If you use the term `each` or `every`, you should use only the singular form of the class name, even though in some cases AppleScript or an application may handle the plural form when it is used incorrectly. The elements of lists, records, and strings are listed in the value class definitions in "Values and Constants" (page 51). The elements of application objects are listed in their object class definitions in the application dictionary.
Class: Class identifier
Default value: Item for lists, records, and application objects; Character for strings (see "Notes")

pluralClassName
The plural class name of the elements to be counted. You should use the plural form of the class name when appropriate, even though in some cases AppleScript or an application may handle the singular form when it is used incorrectly.The elements of lists, records, and strings are listed in the value class definitions in "Values and Constants" (page 51).
Class: Class identifier
Default value: Item for lists, records, and application objects; Character for strings (see "Notes")

compositeValue
An expression that evaluates to a composite value whose elements are to be counted.
Class: List, record, reference, or string

referenceToObject
A reference to the object or objects whose elements are to be counted. If you do not specify this parameter, the application counts the elements in the default target of the Tell statement.
Class: List, record, reference, or string

RESULT

The result of the AppleScript command is an integer that specifies the number of elements of a specified class in a composite value.

The result of the application command is either an integer or a list of integers. See "Notes" for details.

Class: Integer or list of integers

EXAMPLES

In the following example, *compositeValue* is a list. The command does not explicitly specify a class of elements to count, so AppleScript counts all the items in the list.

```
count {"Yes", "No", "Maybe", 4, 5, 6}
--result: 6
```

In this example, *className* is `integers` and *referenceToObject* is a list of strings and integers. The following statements count first the integers in the list, then the strings:

```
count the integers in {"Yes", "No", "Maybe", 4, 5, 6}
--result: 3

count the strings in {"Yes", "No", "Maybe", 4, 5, 6}
--result: 3
```

This example shows another way to count the integers in the list:

```
count each integer in {"Yes", "No", "Maybe", 4, 5, 6}
--result: 3
```

You can use the term `number of` as well to count items in a list:

```
number of integers in {"Yes", "No", "Maybe", 4, 5, 6}
--result: 3
```

In the following example, `every file of disk 1` evaluates to a list of files. The Finder counts the files in the list.

```
tell application "Finder"
    count every file of disk 1
end tell
--result: number of files on first disk
```

The following statement is equivalent to the previous example:

```
tell application "Finder"
    count files of disk 1
end tell
```

In the following example, *referenceToObject* is `windows of application "Finder"`, which is a list of windows. The Finder counts the windows in the list.

```
count of windows of application "Finder"
```

NOTES

If you use the Count command on a string without specifying the class to be counted, AppleScript counts the characters. Consider the following two examples.

```
count "This is a string"
--result: 16

count words in "This is a string"
--result: 4
```

If you count a list or record without specifying a class, the Count command counts the items:

```
count {1, 2, 5, 3} --result: 4
count {name: Jun, height: 72, weight: 200} --result: 3
```

Delete

Delete is an application command that deletes one or more objects.

SYNTAX

```
delete referenceToObject
```

PARAMETER

referenceToObject
A reference to the object or objects to be deleted.
Class: Reference

RESULT

None

EXAMPLES

```
tell application "Finder"
    delete file "old report" of startup disk
end tell

tell application "Desktop Printer Manager"
    delete desktop printer "Spot"
end tell

tell document "Simple" of application "AppleWorks"
    delete words 1 through 5 of text body
end tell
```

Duplicate

Duplicate is an application command that copies one or more objects and inserts them either at a location specified in the command or at the location following the copied object or objects.

SYNTAX

```
duplicate referenceToObject [ to newLocation ]
```

PARAMETERS

referenceToObject

A reference to the object or objects to be duplicated.
Class: Reference

newLocation

The new location for the object.
Class: Reference
Default value: If you do not specify a new location, the object is inserted at the location immediately following the object specified in the direct parameter.

RESULT

A reference to the new object.

Class: Reference

EXAMPLE

The following example tells the Finder to duplicate a file from the startup disk to a folder on the startup disk. If a file with the same name already exists, it is not replaced.

```
tell application "Finder"
    duplicate first file of startup disk ¬
        to first folder of startup disk replacing no
end tell
```

Exists

Exists is an application command that determines whether the object specified by a reference exists.

SYNTAX

referenceToObject `exists`

`exists` *referenceToObject*

PARAMETER

referenceToObject
A reference to the object or objects to find.
Class: Reference

RESULT

The result is `true` if all of the objects referred to by *referenceToObject* exist, `false` if one or more of the objects referred to by *referenceToObject* do not exist.
Class: Boolean

EXAMPLES

The following example checks whether a folder exists before attempting to count the files in the folder.

```
tell application "Finder"
    if folder "Apple Extras" of startup disk exists then
        count files in folder "Apple Extras" of startup disk
    end if
end tell
```

The next example deletes the first paragraph from a report document, if the paragraph exists.

```
tell document “My Report” of application "AppleWorks"
    if paragraph 1 of text body exists then
        delete paragraph 1 of text body
    end if
end tell
```

Get

The Get command can function as an AppleScript command or an application command. The AppleScript command returns the value of an expression. The application command returns the value of an object. In both cases, the command assigns the value returned to the predefined variable `result`, which is described in "Using the Predefined Result Variable" (page 122).

APPLESCRIPT COMMAND SYNTAX

```
[ get ] expression [ as className ]
```

APPLICATION COMMAND SYNTAX

```
[ get ] referenceToObject [ as className ]
```

PARAMETERS

expression An expression whose value is to be returned in the `result` variable.
Class: Any AppleScript expression

className A class identifier that specifies the desired value class for the returned data.
Class: Class
Default value: The default value class for the object or objects

referenceToObject
A reference to an object whose value is to be returned in the predefined `result` variable, which is described in "Using the Predefined Result Variable" (page 122).
Class: Reference

RESULT

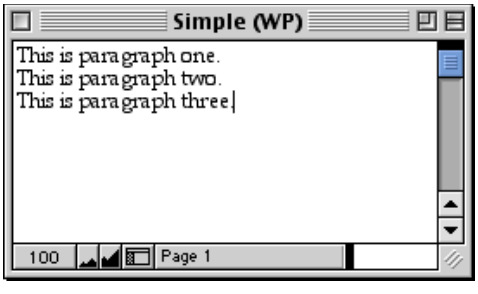
If no error is generated, the result is the value of the specified reference or expression.

If the *referenceToObject* parameter specifies a single object only (such as `word 1` or `the last word`), the result is a single value. If the specified object doesn't exist, for example, if the reference is `word 12` and there are fewer than 12 words in the specified container, AppleScript generates an error.

If the *referenceToObject* parameter uses a whose clause (such as `the words whose first character is "B"`) to specify the object or objects to get, the result is always a list. If the specified objects don't exist (for example, if the reference is `the words whose first character is "B"` and there are no words that begin with "B"), the result is an empty list.

Class: The class specified by the *className* parameter or a list of values of that class. The application can use this parameter to determine what kind of data to return. For example, an application may return text in various formats. Note that if necessary, AppleScript attempts to coerce the result into the class specified by this parameter.

Figure 4-1 The AppleWorks document "Simple"



EXAMPLE

```
tell document "Simple" of application "AppleWorks"
    get paragraph 3 of text body
end tell
--result (for doc in Figure 4-1): "This is paragraph three."
```

NOTES

The word `get` in the Get command is optional because AppleScript automatically gets the value of expressions and references when they appear in scripts.

For example, the following statements are equivalent:

```
item 1 of {"Hi,", "how", "are", "you?"}
--result: "Hi,"

get item 1 of {"Hi,", "how", "are", "you?"}
--result: "Hi,"
```

The following statements are also equivalent:

```
tell application "AppleWorks"
    word 1 of text body of document "Introduction"
end tell

tell application "AppleWorks"
    get word 1 of text body of document "Introduction"
end tell
```

ERROR

Error number	Error message
–1728	`Can't get <reference>.`

Launch

Launch is an application command. If an application is not already running, sending it a Launch command launches it without sending a Run command. (If the application is already running, the Launch command has no effect.) This allows an application to open without performing its usual startup procedures, such as opening a new window or, in the case of a script application, running the script. For example, you can use the Launch command when you don't want a script to cause an application open and close visibly.

Although the target of a Launch command is always an application, the Launch command is actually handled by the Finder. Unlike the other application commands defined in this chapter, it doesn't need to be explicitly supported by applications and doesn't appear in any application's dictionary.

SYNTAX

```
launch [ referenceToApplication ]
```

PARAMETER

referenceToApplication
A reference of the form `application` *nameString* (see "Notes"). This parameter is optional if the Launch command is used within an appropriate Tell statement.
Class: Reference

RESULT

None

EXAMPLES

```
launch application "SimpleText"

tell application "SimpleText"
    launch
end tell
```

NOTES

To specify the name (*nameString*) of an application to launch, use a string of the form "*Disk*:*Folder1*:*Folder2*:. . . .:*ApplicationName*"; for details, see "References to Applications" (page 194). You can also specify a string with only an application name ("*ApplicationName*"). In this case, AppleScript attempts to find the application using the Desktop Database maintained by the Finder.

AppleScript sends an implicit Run command whenever it begins to execute a Tell statement whose target is an application that is not already open. This can

cause problems with applications such as SimpleText that normally perform specific tasks on startup, such as opening a new window. Consider the following example:

```
tell application "SimpleText"
    open file "Hard Disk:Status Report"
end tell
```

Before AppleScript tells SimpleText to open the file Status Report, it sends SimpleText an implicit Run command. If the application is not already open, the Run command causes SimpleText not only to launch but also to perform its usual startup tasks, including opening an untitled window. Therefore, running this script opens two windows: an untitled window and a window for the file Status Report.

If you don't want AppleScript to send an implicit Run command when it launches an application as the result of a Tell statement, use the Launch command explicitly at the beginning of the statement:

```
tell application "SimpleText"
    launch
    open file "Hard Disk:Status Report"
end tell
```

In this case, AppleScript launches the application without sending it a Run command, and the application opens only a window for the requested document.

The Launch command is particularly useful for scriptable control panels such as the Appearance Manager. For example, if the Appearance Manager isn't already running, either visibly or in the background, the following script will cause it to run and show the Appearance Manager control panel:

```
tell application "Appearance"
    set themeName to name of theme 1
    display dialog "The name of the first theme is " & themeName
end tell
```

However, in the same circumstances, the following script displays the theme information without causing the Appearance Manager to show its user interface (the Appearance Manager control panel). That's because the first line in the Tell

block is a `launch` command, which causes AppleScript to launch the Appearance Manager without sending it a Run command.

```
tell application "Appearance"
    launch
    set themeName to name of theme 1
    display dialog "The name of the first theme is " & themeName
end tell
```

For similar reasons, it is sometimes important to use the Launch command before sending the Run command to a script application. For more information, see "Calling a Script Application From a Script" (page 310). For information about Run handlers, see "Run Handlers" (page 303).

Make

Make is an application command that creates a new object. The command can include values for properties of the object, for the data of the object, or both.

SYNTAX

```
make [new] className at referenceToLocation                          ¬
    [ with properties                                                ¬
        { propertyLabel:propertyValue [, propertyLabel:propertyValue ]...}]  ¬
    [ with data dataValue ]
```

The continuation characters (¬) indicate that the command must all be on one line (or on a line extended with continuation characters). They are not a required part of the syntax.

PARAMETERS

className The class of the object to be created.
Class: Class identifier

referenceToLocation
The location at which to create the new object.
Class: Reference

propertyLabel	The name of a property whose value is to be set for the new object. *Class:* String
propertyValue	The value to assign to the property. *Class:* The value class of the property, as specified in the application dictionary definition of the object class being created, or a value that can be coerced into the class of the property *Default value:* The default value of the property, as specified in the application dictionary definition of the object class being created
dataValue	The value to assign to the object. *Class:* The default value class of the object, or a value that can be coerced into the default value class. Default value classes of objects are listed in the "Default Value Class Returned" sections of the dictionary definitions of the objects. *Default value:* None

RESULT

A reference to the newly created object.

Class: Reference

EXAMPLE

The following example creates a document named "New Report" in the current directory. (The current directory is typically the directory where the application was launched, the directory where the application last opened or saved a previous document, or another directory specified by the application. The current directory may be affected by settings in the General Controls control panel.)

```
tell application "AppleWorks"
    make new document at beginning with properties {name:"New Report"}
end tell
```

Move

Move is an application command that moves an object or objects.

SYNTAX

```
move referenceToObject to referenceToLocation
```

PARAMETERS

referenceToObject
A reference to the object or objects to move.
Class: Reference

referenceToLocation
A reference to the location to *which* to move the object or objects.
Class: Reference

RESULT

A reference to the object that was moved.

Class: Reference

EXAMPLE

```
tell application "Finder"
    move file "StdLog" of startup disk ¬
        to folder "Saved Error Logs" of startup disk
end tell

tell document 1 of application "AppleWorks"
    move word 1 of text body to before paragraph 2 of text body
end tell
```

Open

Open is an application command that opens a file or files.

SYNTAX

```
open referenceToFile
```

PARAMETER

referenceToFile

A reference of the form `file` *nameString* or `alias` *nameString*, or a list of such references (see "Notes").
Class: Reference or list of references

RESULT

None

EXAMPLES

Open a single file.

```
tell application "AppleWorks"
    open file "Hard Disk:New Products:Mammoth:Product Intro"
end tell
```

Open a list of files.

```
tell application "AppleWorks"
    open {file "Hard Disk:Letters:Offer", ¬
        file "Hard Disk:Letters:Acceptance"}
end tell
```

Ask the Finder to open an Apple System Profiler report file. This is the equivalent of double-clicking on a file icon in the Finder, and launches the application if it isn't already running.

```
tell application "Finder"
    open the file "Control Panel Report" of disk "Hard Disk"
end tell
```

NOTES

To specify the name (*nameString*) of a file to open, use a string of the form "*Disk*:*Folder1*:*Folder2*:. . . :*Filename*"; for details, see "References to Files" (page 191). You can also specify a string with only a filename ("*Filename*"). In this case, the application attempts to find the file in the current directory. (The current directory is typically the directory where the application was launched, the directory where the application last opened or saved a previous document, or another directory specified by the application. The current directory may be affected by settings in the General Controls control panel.)

If the file or files specified by *referenceToFile* is already open, it remains open and, typically, becomes the frontmost window (if it was not already).

Print

Print is an application command that prints one or more objects.

SYNTAX

`print` *referenceToObject*

PARAMETER

referenceToObject
A reference to the object or objects to print—typically one or more files, documents, or windows.
Class: Reference or list of references

RESULT

None

EXAMPLES

Print an open document:

```
tell application "Apple System Profiler"
    print report "Control Panel Report"
end tell
```

The next script tells the Finder to print two AppleWorks documents. This is the equivalent of selecting the two documents in the Finder and choosing Print from the File menu.

```
tell application "Finder"
    print {file "Hard Disk:Letters:Offer", ¬
        file "Hard Disk:Letters:Acceptance"}
end tell
```

NOTES

To specify the name of a file to print, use the term `file` or `alias` followed by a string of the form "*Disk*:*Folder1*:*Folder2*:. . . :*Filename*"; for details, see "References to Files" (page 191). You can also specify a string with only a filename ("*Filename*"). In this case, the application attempts to find the file in the current directory. (The current directory is typically the directory where the application was launched, the directory where the application last opened or saved a previous document, or another directory specified by the application. The current directory may be affected by settings in the General Controls control panel.)

Quit

Quit is an application command that terminates an application. If no optional parameters are specified, the Quit command has the same result as choosing the Quit menu item in the application.

SYNTAX

`quit` *referenceToApplication* `[ saving` *saveOption* `]`

PARAMETERS

referenceToApplication
A reference of the form `application` *nameString*, where *nameString* is a string that matches the name of the application you want to quit as it is listed in the Application menu.
Class: Reference

saveOption A constant that specifies whether to save documents that have been modified before quitting. The possible values are `yes`, `no`, and `ask`. The value `yes` specifies to save the documents. The value `no` specifies not to save the documents. The value `ask` specifies to ask the user whether or not to save the documents. See "Notes" below for more information.
Class: Constant
Default Value: `ask`

RESULT

None

EXAMPLES

The first example below causes the application to quit without saving any modified documents. The second example asks before saving any modified documents.

```
tell application "AppleWorks"
    quit saving no
end tell

quit application "AppleWorks" saving ask
```

Reopen

Reopen is an application command that reactivates a running application. It has the same effect as double-clicking on the application's icon in the Finder when the application is already running. Because it may not be obvious to a user that anything happened after double-clicking an application icon when no windows

for that application are open, the application may choose to open an untitled window or perform some other operation to make its activation known. If an application is not already running, sending it a Reopen command has the same effect as running the application—see "Run" (page 154).

Each application determines how it will implement the Reopen command. Some applications may perform their usual startup procedures, such as opening a new window, while others perform no additional operations.

SYNTAX

```
Reopen [ referenceToApplication ]
```

PARAMETER

referenceToApplication
A reference of the form `application` *nameString* (see "Notes"). This parameter is optional if the Reopen command is used within an appropriate Tell statement.
Class: Reference

RESULT

None

EXAMPLES

```
Reopen application "SomeAppName"

tell application "SomeAppName"
    Reopen
end tell
```

NOTES

To specify the name (*nameString*) of an application to reopen, use a string of the form "*Disk*:*Folder1*:*Folder2*:. . . .:*ApplicationName*"; for details, see "References to Applications" (page 194). You can also specify a string with only an application

name (“*ApplicationName*”). In this case, AppleScript attempts to find the application using the Desktop Database maintained by the Finder.

Run

The Run command can function as an AppleScript command or an application command.

The application Run command launches an application if it is not already running. The application must be on a local or mounted volume. If the application is already running, then the effect of the Run command depends on the application. Some applications are not affected; others, such as SimpleText, repeat their startup procedures each time they receive a Run command.

The AppleScript Run command acts on script objects. It causes statements in a script object definition, other than handler and property definitions, to be executed. You do not use the Run command to directly execute individual script statements or to execute AppleScript or scripting addition commands.

APPLESCRIPT COMMAND SYNTAX

```
run [ scriptObjectVariable ]
```

APPLICATION COMMAND SYNTAX

```
run [ referenceToApplication ]
```

PARAMETERS

scriptObjectVariable
A variable identifier whose value is a script object. This parameter is optional if the Run command is used within an appropriate Tell statement.
Class: Script

referenceToApplication
A reference of the form `application` *nameString* (see "Notes"). This parameter is optional if the Run command is used within an appropriate Tell statement.
Class: Reference

RESULT

The AppleScript Run command returns the result, if any, returned by the specified script object's Run handler.

The application Run command doesn't return a result.

EXAMPLES

```
run application "SimpleText"

tell application "SimpleText"
    run
end tell
```

You do not use the Run command to directly execute individual script statements or to execute AppleScript or scripting addition commands:

```
run (save) --WRONG!
run (beep) --WRONG! (Don't try this!)
```

For more information, including additional examples, see the sections listed under Notes for this command.

NOTES

To specify the name (*nameString*) of an application to run, use a string of the form "*Disk*:*Folder1*:*Folder2*:. . . .:*ApplicationName*"; for details, see "References to Applications" (page 194). You can also specify a string with only an application name ("*ApplicationName*"). In this case, if the application is not already running, AppleScript attempts to find the application using the Desktop Database maintained by the Finder.

AppleScript sends an implicit Run command whenever it begins to execute a Tell statement whose target is an application that is not already open. This can

cause problems with applications such as SimpleText that normally perform specific tasks on startup, such as opening a new window. To launch an application without invoking its usual startup behavior, use the Launch command, as described in "Launch" (page 143). For information about using the Run and Launch commands with script applications, see "Calling a Script Application From a Script" (page 310).

For information about Run handlers, see "Run Handlers" (page 303). For information about using the Run command with script objects, see "Script Objects" (page 325).

Save

Save is an application command that saves an object or objects.

SYNTAX

```
save referenceToObject [ in referenceToFile ]
```

PARAMETERS

referenceToObject
A reference to the object or objects to be saved.
Class: Reference

referenceToFile A reference of the form `file` *nameString* or `alias` *nameString* that specifies the file in which to save the objects (see "Notes").
Class: Reference
Default value: The file in which the object was last saved. If the object has not been saved before, the application creates a new file.

RESULT

None

EXAMPLE

```
tell application "AppleWorks"
    save document "Stupendous" in file ¬
    "Hard Disk:Documents:Large Documents:Elephantine"
end tell
```

NOTES

To specify the name (*nameString*) of a file in which to save the specified object or objects, use a string of the form "*Disk*:*Folder1*:*Folder2*:. . . :*Filename*"; for details, see "References to Files" (page 191). You can also specify a string with only a filename ("*Filename*"). In this case, the application attempts to find the file in the current directory. (The current directory is typically the directory where the application was launched, the directory where the application last opened or saved a previous document, or another directory specified by the application. The current directory may be affected by settings in the General Controls control panel.)

If you use the form `file` *nameString* and the specified file is not present in the specified location, the application creates a file with the specified name in that location. If you use the form `alias` *nameString* and the specified file is not present in the specified location, the script won't compile.

The Save command will replace, without asking, an existing file with the same name as the specified file (if one exists).

Set

The Set command can function as an AppleScript command or an application command. The AppleScript command assigns one or more values to one or more variables. It can also be used to share data among lists, records, or script objects—see the Notes section below and "Data Sharing" (page 206). The application command sets the values of one or more objects.

APPLESCRIPT COMMAND SYNTAX

`set` *variablePattern* `to` *expression*

expression `returning` *variablePattern*

APPLICATION COMMAND SYNTAX

`set` *referencePattern* `to` *expression*

expression `returning` *referencePattern*

PARAMETERS

variablePattern The name of the variable in which to store the value, or a list of variable patterns, or a record of variable patterns.
Class: Identifier, list, or record

expression The expression whose value or values are to be assigned. If *expression* is a reference or a list or record of references, AppleScript gets the values of the objects specified by the references.
Class: For a variable, any class.

referencePattern
A reference to the location whose value is to be set, or a list of reference patterns, or a record of reference patterns.
Class: Reference, list, or record

RESULT

The value assigned.

EXAMPLES

You can use the Set command to set a variable to any value:

```
set x to 5
--result: 5

set myList to { 1, 2, "four" }
--result: {1, 2, "four"}
```

```
tell application "AppleWorks"
    set x to word 1 of text body of front document
end tell
```

These two statements are equivalent:

```
set x to 3
3 returning x
--result: 3
```

Similarly, the following examples are equivalent:

```
tell front document of application "AppleWorks"
    set x to word 1 of text body
end tell

tell front document of application "AppleWorks"
    word 1 of text body returning x
end tell
```

In addition to setting a variable to a single value, you can set patterns of variables to patterns of values. For example, this script sets a list of two variables to the position of the front window.

```
tell application "Finder"
    set {x, y} to position of front window
end tell
```

Since the Finder returns `position of front window` as a list of two integers, the preceding example sets `x` to the first item in the list and `y` to the second item.

Patterns set with the Set command can also be more complex. Here are some examples:

```
set x to {8, 94133, {firstName:"John", lastName:"Chapman"}}
set {p, q, r} to x
(* now p, q, and r have these values:
                p = 8
                q = 94133
                r = {firstName:"John", lastName:"Chapman"} *)
```

```
set {p, q, {lastName:r}} to x
(* now p, q, and r have these values: p = 8
                                      q = 94133
                                      r = "Chapman" *)
```

As the last example demonstrates, the properties of a record need not be given in the same order and need not all be used when you set a pattern to a pattern, as long as the patterns match.

The use of the Set command with patterns is similar to the use of patterned parameters with subroutines, which is described in "Subroutines With Positional Parameters" (page 296).

NOTES

If you use the Set command to set a variable to a list, record, or script object, the variable shares data with the original list, record, or script object. If you change the data of the original, the value of the variable also changes. Here's an example of how this works:

```
set myList to { 1, 2, 3 }

set yourList to myList

set item 1 of myList to 4
```

The result of these statements is that item 1 of both `myList` and `yourList` is `4`.

Data sharing promotes efficiency when using large data structures. Rather than making copies of shared data, the same data can belong to multiple structures. When one structure is updated, the others are automatically updated.

IMPORTANT

To avoid data sharing for lists, records, and script objects, you typically copy these items with the AppleScript command "Copy" (page 132), rather than use the Set command. ▲

Only data in lists, records, and script objects can be shared; you cannot share other values. Moreover, you can share data only on the same computer, and the shared structures must all be in the same script.

Objects and References

This chapter describes how to interpret object class definitions and how to use references to specify objects.

Objects are the things in applications, the Mac OS, or AppleScript that can respond to commands by performing actions. For example, application objects are objects stored in applications and their documents. Usually, they are identifiable items that users can manipulate in applications, such as windows, words, characters, and paragraphs in a text-editing application. Objects can contain data, in the form of values, properties, and elements, that can change over time.

Each object belongs to an object class, which is a category for objects that have similar characteristics and respond to the same commands. You can get detailed information about the object classes an application supports by examining the application's dictionary. To examine the dictionary, you drop the application's icon on the Script Editor icon, or open the application with Script Editor's Open Dictionary command. Table 5-1 and Table 5-2 show a portion of the Window class definitions from the application dictionaries of, respectively, the Finder and AppleWorks.

To refer to an object from a script, you use a reference, which is a compound name, similar to a path or address, that identifies an object or groups of objects.

Objects and references are described in the following sections:

- "Object Class Definitions" (page 162) describes the kind of information you can expect to find in an object class definition and where to obtain these definitions.
- "References" (page 165) defines a reference, describes complete and partial references, and explains how to use containers in references.
- "Reference Forms" (page 169) lists reference forms you can use to identify objects and provides sections that describe each reference form in detail.

- "Using the Filter Reference Form" (page 187) describes the optional filter form you can add to a reference to specify only the objects that match one or more conditions.
- "References to Files and Applications" (page 190) describes how to specify files and applications, both locally and on remote machines.

Most objects are contained in applications, but you can also create another type of object, called a script object, that can be stored in scripts or saved in files. For information about script objects, see "Script Objects" (page 325).

Object Class Definitions

An object class definition describes the common features of objects that belong to the class. For example, all document objects created by the AppleWorks application have certain properties (such as a name property) and elements (such as a window element) in common. An application's dictionary describes the classes that the application supports. You can view an application's dictionary by dropping the application's icon on the Script Editor's icon, or by opening the application with the Script Editor's Open Dictionary command.

Table 5-1 shows a sample object class definition for a window object, excerpted from the Finder dictionary. The definition contains three types of information: the plural form for the class, its elements (none in this case), and its properties. Each property description includes the name, the class, and a description. A property description should also state whether the property is read only.

The components of an object class definition are described in the following sections:

- "Properties" (page 164)
- "Element Classes" (page 165)
- "Default Value Class Returned" (page 165)

For more information about how AppleScript works with application dictionaries, see "Dictionaries" (page 34). For additional examples of class definitions, see the value class definitions in "Values and Constants" (page 51).

Different applications can support the same class in different ways. Table 5-2 shows a window class definition excerpted from the AppleWorks application. The window class defined by AppleWorks uses the same plural form as the

Finder's window class and shares some properties, such as name, bounds, and zoomed. However, the AppleWorks version has elements, such as panes, that the Finder does not, and there are several properties the two classes do not share.

Table 5-1 Window class definition from the Finder dictionary

Plural form windows

Elements:

None

Properties:

Name	Class	Description
position	point	The upper left position of the window
bounds	rectangle	The boundary rectangle for the window
titled	boolean [r/o]	Does the window have a title bar?
name	international text [r/o]	The name of the window
index	integer	The number of the window in the front-to-back layer ordering
closeable	boolean [r/o]	Does the window have a close box?
floating	boolean [r/o]	Does the window have a title bar?
modal	boolean [r/o]	Is the window modal?
resizable	boolean [r/o]	Is the window resizable?
zoomable	boolean [r/o]	Is the window zoomable?
zoomed	boolean [r/o]	Is the window zoomed?

(some properties not shown)

As you would expect, these applications define window classes that support the different ways they work with windows. AppleWorks uses windows to let users enter and manipulate data in word processing documents, spreadsheet cells, database records, and so on. The Finder uses windows to display disks, folders, and files, and to let users manipulate these items. When you perform script operations on a window in one of these applications, you have access to

most of the operations a user can perform and, in some cases, to capabilities that a user cannot perform with the program's user interface.

Table 5-2 Window class definition from the AppleWorks dictionary

Plural form windows

Elements:

Name	Description
split	by numeric index, as a range of elements, satisfying a test
pane	by numeric index, as a range of elements, satisfying a test

(some elements not shown)

Properties:

Name	Class	Description
name	international text [r/o]	The window's name
document	document [r/o]	The document associated with the window
bounds	bounding rectangle	The boundary rectangle for the window
zoomed	boolean	Is the window zoomed?
position	point	Upper left coordinates of the window
scale	real	The viewing scale (zoom percentage)
ruler	ruler	The ruler for this window

(some properties not shown)

Properties

A *property* of an object is a characteristic that has a single value, such as the name of a window or the font of a character. Properties of an object are distinguished from each other by their unique labels. For example, the Window class definition shown in Table 5-2 includes the Position property. Its unique label is the label Position. The definition also lists the class to which each property belongs. For example, the class of the Position property is Point,

indicating that the value of the property is a two-dimensional point. Because the Position property is not marked read only, you can set its value. The class of a property can be a simple value class like String or Number, a composite class like the Point class, or a more complex object class, perhaps with properties of its own.

Element Classes

Elements are objects contained within an object. The element classes listed in an object class definition indicate the kinds of elements objects of that class can contain. An object can contain many elements or none, and the number of elements that it contains may change over time. For example, it is possible for a Paragraph object to contain no words. At a later time, the same paragraph might have many words.

Most application and system objects can contain elements. The Finder's Window definition in Table 5-1 has no elements, but the AppleWorks Window definition in Table 5-2 includes Panes, Splits, and other elements.

Default Value Class Returned

Each object has a value. For example, the value of a Word object is a string that may include style and font information. You can get the value of a system or application object by sending it a Get command or by simply referring to it in a script. If the Get command doesn't specify a value class for the value returned, the default value class is used. For many object classes, the default value is a reference to an object of that class type. For example, the default value of a Window object based on the definition in either Table 5-1 or Table 5-2 is a reference to a Window object. References are described in detail in "References" (page 165).

References

A **reference** is a phrase that specifies one or more objects. You use references to identify objects within applications. An example of a reference is

```
tell application "Finder"
    file 1 of folder 1 of startup disk
end tell
```

which specifies the first file in the first folder of the startup disk.The term `startup disk` and other special terms understood by the Finder are described in "Viewing a Result in the Script Editor's Result Window" (page 121).

A reference describes what type of object you're looking for, where to look for the object, and how to distinguish the object from other objects of the same type. These three types of information—the *class*, or type; the *container*, or location; and the *reference form*, or distinguishing information—allow you to specify any object of an application.

In general, you list the class and distinguishing information at the beginning of a reference, followed by the container. In the previous example, the class of the object is `file`. The container is the phrase `folder 1 of startup disk`. The distinguishing information (the reference form) is the combination of the class, `file`, and an index value, `1`, which together indicate the first file.

References allow you to identify objects in a flexible and intuitive way. Just as there might be several ways to identify an object on the desktop, AppleScript has different reference forms that allow you to specify the same object in different ways. For example, here's another way to specify the first file in the first folder of the startup disk:

```
tell application "Finder"
    file before file 2 of first folder of startup disk
end tell
```

Because the first file in the first folder of the startup disk can change (if files or folder are added or deleted or have their name changed, or if the startup disk is changed), you may need to use a more specific reference:

```
tell application "Finder"
    file "SimpleText" of folder "Applications" of disk "Hard Disk:"
end tell
```

For detailed information on specifying files, see "References to Files" (page 191).

To write effective scripts, you should be familiar with AppleScript's reference forms and know how to use containers and reference forms to identify the

objects you want to manipulate. These topics are described throughout this chapter.

Containers

A **container** is an object that contains one or more objects or properties. In a reference, the container specifies where to find an object or a property. To specify a container, use the word `of` or `in`, as in the following statement

```
folder "Utilities" of disk "Hard Disk:"
```

A container can be an object or a series of objects. In a series, the smallest object is listed first, followed by the larger objects that contain it. Use the word `of` or `in` to separate each object from its larger, containing object. In the following example, the body of text is contained in a larger document object, paragraph three is contained in the larger text object, and word four in the larger paragraph object.

```
tell application "AppleWorks"
    word 4 in paragraph 3 of text body of front document
end tell
```

You can also use the possessive form (`'s`) to specify containers. If you use the possessive form, list the container before the object it contains. In the following example, the container is `first window`. The object it contains is a Name property.

```
tell application "AppleWorks"
    first window's name
end tell
```

All properties and elements have containers. The previous example specified the Name property of a window, which is contained in a window object. Similarly, the following example specifies the Style property, which is contained in a text object.

```
tell application "AppleWorks"
    style of text body of front document
end tell
```

Complete and Partial References

A **complete reference** has enough information to identify an object or objects uniquely. For a reference to an application object to be complete, its outermost container must be the application itself, as in

```
version of application "Finder" --result: "8.5"
```

In contrast, **partial references** do not specify enough information to identify an object or objects uniquely; for example:

```
delete file 1 of disk 4
```

When AppleScript encounters a partial reference, it attempts to use the default target specified in the Tell statement to complete the reference. The default target of a Tell statement is the object that receives commands if no other object is specified. For example, the following Tell statement tells the Finder to delete the first file of the fourth disk, using the previous partial reference.

```
tell application "Finder"
    delete file 1 of disk 4
end tell
```

Similarly, the following Tell statement tells the front document of the application AppleWorks to get the style of its text.

```
tell document 1 of application "AppleWorks"
    get style of text body
end tell
```

Tell statements can contain other Tell statements, called nested Tell statements. When AppleScript encounters a partial reference in a nested Tell statement, it tries to complete the reference starting with the innermost Tell statement. If that does not provide enough information, AppleScript uses the direct object of the next Tell statement, and so on. For example, the following Tell statement is equivalent to the previous Finder example.

```
tell application "Finder"
    tell file 1 of disk 4
        delete
    end tell
end tell
```

This example works because all of the nested statements target the same application, the Finder. For information on restrictions in using nested Tell statements, see "Tell Statements" (page 240).

Reference Forms

A **reference form** is the syntax, or rule, for writing a phrase that identifies an object or group of objects. For example, the Index reference form allows you to identify an object by its number, as in

```
word 5 of paragraph 10
```

AppleScript includes other reference forms for identifying objects in applications. Table 5-3 summarizes the reference forms you can use to identify objects and provides links to sections that describe them in detail. Each section includes a brief explanation of a reference form, a syntax summary, and examples of how to use the reference form to specify application objects. The Filter reference form is described in more detail in "Using the Filter Reference Form" (page 187).

Table 5-3 Reference forms

Reference form	Purpose
"Arbitrary Element" (page 170)	Specifies an arbitrary object in a container (rarely used)
"Every Element" (page 171)	Specifies every object of a particular class in a container
"Filter" (page 173)	Specifies every object in a particular container that matches conditions specified by a Boolean test expression

Table 5-3 Reference forms (continued)

Reference form	Purpose
"ID" (page 174)	Specifies an object by its ID property
"Index" (page 177)	Specifies the position of an object with respect to the beginning or end of a container
"Middle Element" (page 179)	Specifies the middle object in a container (rarely used)
"Name" (page 180)	Specifies an object by its Name property
"Property" (page 182)	Specifies a property of an application object, a record, a script object, or a date
"Range" (page 183)	Specifies a series of objects
"Relative" (page 185)	Specifies the position of an object in relation to another object in the same container

Arbitrary Element

The **Arbitrary Element** reference form specifies an arbitrary object in a container. If the container is a value (such as a list), AppleScript chooses an object at random (that is, it uses a random-number generator to choose the object). If the container is an application object, it is up to the application to choose an object. It can choose a random object or any object at all. This form is rarely used.

SYNTAX

```
some className
```

where

className is the class identifier for the desired object.

EXAMPLES

The following examples return a random file from a disk and a random word from a document. Because an arbitrary element is, by its nature, random, this form is rarely useful in processing groups of files, words, or other objects.

```
tell application "Finder"
    some file of startup disk
end tell

tell application "AppleWorks"
    some word of text body of front document
end tell
```

Every Element

The **Every Element** reference form specifies every object of a particular class in a container.

SYNTAX

`every` *className*

pluralClassName

where

className is a singular class name (such as word or paragraph).

pluralClassName is the plural form (such as `words` or `paragraphs`) defined by AppleScript or an application. The plural form of an object class name has the same effect as the word `every` before an object class name. Plural forms are listed in application dictionaries.

VALUE

The value of an Every Element reference is a list of the objects in the container. If the container does not contain any objects of the specified class, the list is an empty list. For example, the value of the expression

```
every word of {1, 2, 3}
```

is the empty list:

```
{}
```

EXAMPLES

The following example assigns a string to the variable `myString`, and then uses the Every Element reference form to specify every word contained in the string.

```
set myString to "That's all, folks"
every word of myString
```

The value of the reference `every word of myString` is a list with three items:

```
{"That's", "all", "folks"}
```

The following reference specifies the same list:

```
words of myString
```

The following example specifies a list of all the files in the Control Panels folder.

```
tell application "Finder"
    every file in control panels folder
end tell
```

The following specifies the same list as the previous example.

```
tell application "Finder"
    files in control panels folder
end tell
```

NOTES

If you specify an Every Element reference as the container for a property or object, the result is a list containing the specified property or object for each object of the container. The number of items in the list is the same as the number of objects in the container. For example, the value of the reference

```
length of every word
```

is a list such as

```
{ 2, 3, 6 }
```

The first item in the list is the length of the first word, the second item is the length of the second word, and so on.

Filter

The **Filter** reference form specifies all objects in a container that match one or more conditions specified in a Boolean expression. The Filter reference form specifies application objects only. It cannot be used to filter AppleScript objects: lists, records, or strings. For more information, see "Using the Filter Reference Form" (page 187).

SYNTAX

referenceToObject `( whose | where )` *Boolean*

where

referenceToObject is a reference that specifies one or more objects.

Boolean is any Boolean expression.

The words `whose` and `where` have the same meaning.

EXAMPLES

The following are some examples of references that use the Filter reference form. For additional examples, see "Using the Filter Reference Form" (page 187).

The following example specifies a list of file references for all files in the Control Panels folder with file type `'APPL'`.

```
tell application "Finder"
    every file in control panels folder whose file type is "APPL"
end tell
```

The following example specifies all application windows that do not match a given name:

```
tell application "AppleWorks"
    every window whose name is not "Old Report (WP)"
end tell
```

NOTES

Except for the Every Element reference form, the application returns an error if no objects pass the test or tests. For the Every Element reference form, the application returns an empty list, {}, if no objects pass the test or tests.

To specify a container after a filter, you must enclose the filter and the reference it applies to in parentheses. For example, the parentheses around `words 1 thru 5 whose first character = "M"` in the following reference are required because the container `of paragraph 5` follows the filter.

```
tell application "Finder"
    (files whose file type is "APPL") in control panels folder
end tell
```

ID

The **ID** reference form specifies an object by the value of its ID property. You can use this reference form only for objects that have an ID property.

SYNTAX

className `id` *IDvalue*

where

className is the class identifier for the specified object.

IDvalue is the value of the object's ID property.

EXAMPLES

```
tell application "Finder"
    id of every disk --typical result: {-1, -2, -3}
    id of disk "Hard Disk" --typical result: -2
end tell
```

NOTES

Although ID properties are most often integers, an ID property can belong to any class. An application that includes ID properties must guarantee that the IDs are unique within a container. Some applications may also provide additional guarantees, such as ensuring the uniqueness of an ID among all objects.

The value of an ID property is not modifiable. It does not change even if the object is moved within the container. This allows you to save an object's ID and use it to refer to the object for as long as the object exists. In some scripts you may wish to refer to an object by its ID, rather than by a property such as its name, which may change. The following two examples demonstrate the advantage of using an ID when creating a new folder and renaming it.

The next script generates an error because it gets a reference to a newly created folder, renames the folder, then tries to use the reference (which still refers to the original name) to open the folder:

```
tell application "Finder"
    set newFolder to make new folder at startup disk
        --result: folder "untitled folder" of startup disk
        --          of application "Finder"
    set name of newFolder to "New Folder Name"
    open newFolder
        --result: ERROR, because the folder has been renamed but the
        --  reference still refers to the original "untitled folder"
end tell
```

The following script successfully creates a new folder, renames it, and opens it because it uses a folder ID, which doesn't change when you change the folder's name:

```
tell application "Finder"
    set newFolder to make new folder at startup disk
        --result: folder "untitled folder" of startup disk
        --          of application "Finder"
    set newFolderID to id of newFolder
        --result: an ID, such as 27568
    set name of newFolder to "New Folder Name"
    open folder id newFolderID of startup disk
        --result: Successfully opens new folder named "New Folder Name"
        --          using the folder ID, which is unique and permanent.
end tell
```

Instead of using a name, you could keep track of a file or folder by its index, using statements such as `open file 1` or `open the first folder`. However, because file and folder indexes are ordered alphabetically, they are likely to refer to a different item after renaming.

You can use an ID to keep track of a file as well as a folder, though because the Finder doesn't currently support opening a file by its file ID, you have to use an approach similar to the one shown in the next script:

```
tell application "Finder"
    set fileID to the id of the first file of disk "Hard Disk"
        --The script executes some command that, either directly or
        -- indirectly, causes file 1 to be renamed as "NewName".
    set fileName to the name of the first file ¬
        of disk "Hard Disk" whose id is fileID
        --result: "NewName"
    open file fileName of disk "Hard Disk" -- Opens original file.
end tell
```

A still better way to keep track of files and folders is to use an alias reference, as described in "References to Files" (page 191).

Applications are not required to support ID properties. To find out if or how an application uses ID properties, see the documentation for the application.

Index

The **Index** reference form specifies an object or a location by describing its position with respect to the beginning or end of a container. For related information, see “Relative” (page 185).

SYNTAX

className `[ index ]` *integer*

integer`(st | nd | rd | th )` *className*

```
( first | second | third | fourth | fifth | sixth |
     seventh | eighth | ninth | tenth ) className
```

```
( last | front | back ) className
```

where

className is the class identifier of the object being specified.

integer is an integer that describes the position of the object in relation to the beginning of the container (if *integer* is a positive integer) or the end of the container (if *integer* is a negative integer).

The forms `first`, `second`, and so on are equivalent to the corresponding integer forms (for example, `second word` is equivalent to `2nd word`). For objects whose index is greater than 10, you can use the forms `12th`, `23rd`, `101st`, etc. (Note that any integer followed by any of the suffixes listed is valid; for example, you can use `11rd` to refer to the eleventh object.)

The `front` form (for example, `front window`) is equivalent to *className* `1` or `first` *className*. The `last` and `back` forms (for example, `last word` and `back window`) refer to the last object in a container. They are equivalent to *className* `-1`.

EXAMPLES

Many of the examples in this section require an enclosing Tell statement, targeting the Finder or a word processing application such as AppleWorks, to compile.

Each of the following references specifies the first file on the startup disk:

```
file 1 of the startup disk
file index 1 of the startup disk
the first file of the startup disk
```

The following references specify the second word from the beginning of the third paragraph.

```
word 2 of paragraph 3
2nd word of paragraph 3
second word of paragraph 3
```

The following references specify the last word in the third paragraph.

```
word -1 of paragraph 3
last word of paragraph 3
```

The following reference specifies the next-to-last word in the third paragraph.

```
word -2 of paragraph 3
```

The following example contains two references. The first is a reference to the fourth word of the document called Introduction. The second is a reference to the last insertion point in the same document.

```
tell application "AppleWorks"
    move word 4 of text body of document "Introduction" to ¬
        end of text body of document "Introduction"
end tell
```

The following reference specifies the first file in the first folder in the startup disk:

```
tell application "Finder"
    file 1 of folder 1 of startup disk
end tell
```

NOTES

An Index can be volatile. Changing some other property of the object may change its index, as well as the index of other like objects. For example, after deleting `file 4` in a folder, the file no longer exists. But there may still be a `file 4`—the file that was formerly `file 5`. After `file 4` is deleted, any files with an index higher than 4 will also have a new index. So the object an Index reference refers to can change.

For a unique, persistent reference to an object, you can use the "ID" (page 174) reference form, if the application supports it for the object class you are working with. For example, the Finder supports IDs for files, folders, disks, and so on.

A still better way to keep track of a file is to use an alias reference, as described in "References to Files" (page 191).

Middle Element

The **Middle Element** reference form specifies the middle object of a particular class in a container. This form is rarely used.

SYNTAX

`middle` *className*

where

className is the class identifier for the specified object.

EXAMPLES

```
tell application "AppleWorks"
    middle paragraph of text body of front document
end tell

middle item of {1, "doughnut", 33} --result: "doughnut"
middle item of {1, "doughnut", 22, 33} --result: "doughnut"
middle item of {1, "doughnut", 11, 22, 33} --result: 11
```

NOTES

AppleScript calculates the middle object with the expression `((n + 1) div 2)`, where `n` is the number of objects and `div` is the integer division operator. If there is an even number of objects in the container, the result is rounded down. For example, the middle word of a paragraph containing twenty words is the tenth word.

Name

The **Name** reference form specifies an object by name.

SYNTAX

className `[ named ]` *nameString*

where

className is the class identifier for the specified object.

nameString is the value of the object's Name property (see "Notes").

EXAMPLES

The following statements identify an object by its name:

```
close document "Report" -- document identified by its name property

close window named "Help" -- window identified by its name property
```

Note the distinction between a statement that specifies the actual name of an object, a statement that specifies an object by name, and a statement that specifies a reference to a name object:

```
--Specify name of file:
tell application "Finder"
    name of first file in extensions folder
end tell
--result: "LaserWriter 8" (a name string)
```

```
--Specify file by name:
tell application "Finder"
     file "LaserWriter 8" in extensions folder
end tell
--result:
-- file "LaserWriter 8" of folder "Extensions" of folder "System 8.5.1"
-- of startup disk of application "Finder" (a file object reference)

--Specify reference to name of file:
tell application "Finder"
    a reference to name of first file in extensions folder
end tell
--result: name of file 1 of extensions folder of application "Finder"
-- (a name object reference)
```

NOTES

In some applications, it is possible to have multiple objects of the same class in the same container with the same name. For example, if there are two drives named "Hard Disk", the following statement is ambiguous (at least to the reader):

```
tell application "Finder"
    file 1 of disk "Hard Disk"
end tell
```

In such cases, it is up to the application to determine which object is specified by a Name reference.

For applications and files, the *nameString* parameter can be a string of the form "*Disk*:*Folder1*:*Folder2*:. . .*FileName*"; for details, see "References to Files and Applications" (page 190).

For more information about Name properties of specific types of objects, see the definitions for object classes provided by the AppleScript documentation or the application's documentation.

Property

The **Property** reference form specifies a property of an application object, a script object, a record, or a date.

SYNTAX

propertyLabel

where

propertyLabel is the label for the property.

EXAMPLES

The following example is a reference to the Name property of the front window. It lists the label for the property (`name`) and its container (`front window`).

```
name of front window
```

The following example is a reference to the UnitPrice property of a record. A record is an AppleScript value that consists of a collection of properties. For more information about records, see "Values and Constants" (page 51). The label of the property is UnitPrice and the container is the record.

```
UnitPrice of {Product:"Super Snack", UnitPrice:0.85, Quantity:10}
```

NOTES

Property labels are listed in object class definitions in application dictionaries. Because a property's label is unique among the properties of an object, the label is all you need to distinguish a property from all the other properties of the object. Unlike other reference forms, there is no need to specify the class of the object.

Range

The **Range** reference form specifies a series of objects of the same class in the same container. You can specify the objects with a pair of indexes (such as `words 12 thru 24`) or with a pair of boundary objects (such as `words from paragraph 3 to paragraph 5`).

SYNTAX

`every` *className* `from` *boundaryReference1* `to` *boundaryReference2*

pluralClassName `from` *boundaryReference1* `to` *boundaryReference2*

className startIndex `( thru | through )` *stopIndex*

pluralclassName startIndex `( thru | through )` *stopIndex*

where

className is a singular class ID (such as `word` or `paragraph`).

pluralclassName is the plural class identifier defined by AppleScript or an application (such as `words` or `paragraphs`).

boundaryReference1 and *boundaryReference2* are references to objects that bound the range. The range includes the boundary objects. You can use the reserved word `beginning` in place of *boundaryReference1* to indicate the position before the first object of the container. Similarly, you can use the reserved word `end` in place of *boundaryReference2* to indicate the position after the last object in the container.

startIndex and *stopIndex* are the indexes of the first and last object of the range (such as `1` and `10` in `words 1 thru 10`).

VALUE

The value of a Range reference is a list of the objects in the range. If the specified container does not contain all of the objects specified in the range, an error is returned. For example, the following reference results in an error.

```
words 1 thru 3 of {1, 2, 3}
--results in an error
```

EXAMPLES

In the following example, the phrase `folders 3 thru 4` is a range reference that specifies a list of two folders in the container `startup disk`.

```
tell application "Finder"
    folders 3 thru 4 of startup disk
end tell
--typical result: (depends on contents of startup disk)
--{folder "AppleScript" of startup disk of application "Finder", ¬
--folder "Apple Extras" of startup disk of application "Finder"}
```

In the following example, `files` is a reference that is a synonym for `every file`, and the phrase `folders 3 thru 4 of startup disk` is a container reference made up of the range reference `folders 3 thru 4` and the container `startup disk`.

```
tell application "Finder"
    files of folders 3 thru 4 of startup disk
end tell
```

To get the result, AppleScript first gets the value of the container, which is a list of two folders on the startup disk. It then gets every file in each of the folders, which results in a single list of file names. The actual files in the list depend on the contents of the startup disk.

NOTES

If you specify a Range reference as the container for a property or object, as in

```
name of files 2 thru 3 of startup disk
```

the result is a list containing the specified property or object for each object of the container. The number of items in the list is the same as the number of objects in the container. For example, the value of the reference in this example might be

```
{"ASP memory report, 3/3/99", "BBEdit 5.0.2 Update.img"}
```

The first item in the list is the name of the second file on the startup disk, and the second item is the name of the third file.

To refer to a contiguous series of characters—instead of a list—when specifying a range of text objects, use the text element. Text is an element of most text objects and is also an element of AppleScript strings.

For example, compare the values of the following references.

```
words 1 thru 4 of "We're all in this together"
--result: {"We're", "all", "in", "this"}

text from word 1 to word 4 of "We're all in this together"
--result: "We're all in this"

text of words 1 thru 4 of "We're all in this together"
--result: {"We're", "all", "in", "this"}
```

Relative

The **Relative** reference form specifies an object or a location by describing its position in relation to another object, known as the base, in the same container.

SYNTAX

[*className*] (before | [in] front of) *baseReference*

[*className*] (after | [in] back of | behind) *baseReference*

where

className is the class identifier of the specified object. If you leave out this parameter, AppleScript assumes you want an insertion point.

baseReference is a reference to the base object.

The `before` and `in front of` forms, which are equivalent, refer to the object immediately preceding the base object. The `after`, `in back of`, and `behind` forms are equivalent and refer to the object immediately after the base.

The following forms refer to insertion points:

```
beginning | front

end | back
```

The `beginning` and `front` forms are equivalent and refer to the first insertion point of the container (`insertion point 1`). The `end` and `back` forms are equivalent and refer to the last insertion point of the container (`insertion point -1`).

Although terms such as `beginning` and `end` sound like absolute positions, they are relative to the existing contents of a container (that is, before or after the existing contents).

EXAMPLES

The two references in the following Tell block specify the same file by identifying its position relative to another file on a disk.

```
tell application "Finder"
    file before file 3 of startup disk
    file in front of file 3 of startup disk
end tell
```

The following example contains three references. The first two are Index references that specify the front document and the first word. The third is a Relative reference that specifies the insertion point before the third paragraph. The command moves the first word to the insertion point before the third paragraph.

```
tell front document of application "AppleWorks"
    move word 1 of text body to before paragraph 3 of text body
end tell
```

The following example moves the first word of a document named Introduction to the last insertion point of the document, then moves the last word of the document to the first insertion point (effectively undoing the previous move).

```
tell application "AppleWorks"
    move word 1 of text body of document "Introduction" ¬
        to end of text body of document "Introduction"
```

```
    move last word of text body of document "Introduction" ¬
        to beginning of text body of document "Introduction"
end tell
```

The following example is the same as the previous example, except that it uses `in front` and `in back` instead of `beginning` and `end`.

```
tell application "AppleWorks"
    move word 1 of text body of document "Introduction" ¬
        to in back of text body of document "Introduction"
    move last word of text body of document "Introduction" ¬
        to in front of text body of document "Introduction"
end tell
```

NOTES

You can specify only a single object with the Relative form. You can use the form to specify an object that is either before or after the base object.

If it is possible for the specified object to contain the base object (as in the expression `paragraph before word 99`), the reference does not specify the container but instead specifies the object immediately before or after the container of the base object. For example, the expression `paragraph before word 99` specifies the paragraph immediately before the paragraph containing the ninety-ninth word.

All applications allow you to specify a base object belonging to the same class as the desired object (such as `window in back of window "Big"`). Not all allow you to specify a base of a different object class (such as `word before figure 1`). The possible base classes for a particular class are up to each application.

Using the Filter Reference Form

When specifying one or more objects contained in an application object, you can use the Filter reference form to include an optional filter. A **filter** restricts the objects you specify to those that match one or more conditions.

Note
To compile the examples in this section, you must include them in a Tell block. File and window examples assume a Finder Tell block; paragraph examples assume an AppleWorks Tell block. If the condition specified by a statement isn't satisfied, running the script may return {} (an empty list) or cause an error. For error-handling information, see "Handlers" (page 279). ◆

Compare this reference without a filter

```
every file of extensions folder
```

to the same reference with a filter:

```
every file of extensions folder whose creator type is "OMGR"
```

The first reference specifies all the files in the Extensions folder in the System folder. The second reference, which includes the filter `whose creator type is "OMGR"`, specifies all the files in the same container whose creator type is `"OMGR"`. Files that do not pass this test are filtered out.

In effect, a filter reduces the number of objects in the container. Instead of specifying every window in the Finder, the following reference specifies every word of a smaller container, the windows that are currently zoomed.

```
every window whose zoomed is true
```

The following statement is equivalent to the previous one:

```
windows where zoomed is true
```

To determine the objects in the smaller container, the application applies the filter to all of the objects of the specified class in the specified container—in this case, the windows that are currently zoomed. Adding a whose clause can significantly increase the time it takes to evaluate a reference because it forces the application to test every object of the specified type.

Within a filter, the predefined variable `it` refers to the object currently being tested. For example, in the following reference, the word `it` refers to each paragraph in the document Product Intro.

```
second paragraph of text body of document "Product Intro" ¬
    where it contains "dynamo"
```

The filter, `where it contains "dynamo"`, is applied to each paragraph in the document, resulting in a smaller container whose paragraphs all contain the string `"dynamo"`. The reference specifies the second paragraph of that smaller container.

A Filter reference includes one or more tests. Each **test** is a Boolean expression that compares a property or element of each object being tested, or the objects themselves, with another object or value. Table 5-4 shows some Filter references, the Boolean expressions they contain, and what is being tested in each reference.

Table 5-4 Boolean expressions and tests in Filter references

Filter reference	Boolean expression	What is being tested
`windows whose zoomed is true`	`zoomed is true`	The zoomed property of each window
`windows whose name isn't "Hard Disk"`	`name isn't "Hard Disk"`	The name property of each window
`files whose creator type is "OMGR"`	`creator type is "OMGR"`	The creator type property of each file

A test can be any Boolean expression (such as `files where 1 < 2`), but only those that actually test objects or their contents are useful for filtering objects.

To include more than one test in a filter, link the tests with Boolean operators, as in

```
windows whose zoomed is true and floating is false
```

Here the Boolean operator And indicates that each file must pass both tests to be included in the smaller container.

```
windows whose zoomed is true or floating is true
```

Here the Boolean operator Or indicates that the files can pass either test to be included in the smaller container.

Because each test is a Boolean expression, it can also include the Boolean operator Not. For example, the following reference refers to only those files that aren't zoomed and aren't named Hard Disk.

```
windows whose zoomed is false and not its name is "Hard Disk"
```

The expression `its name is "Hard Disk"` is a valid Boolean expression, and applying the Boolean Not operator to it, as in

```
not (its name is "Hard Disk")
```

inverts the value of the expression, so that a `true` value becomes `false`, and a `false` value becomes `true`.

A more elegant way to apply the Boolean Not operator to the expression `its name is "Hard Disk"` is

```
its name isn't "Hard Disk"
```

The expression `its name isn't "Hard Disk"` is a **synonym** for the expression `not its name is "Hard Disk"`. AppleScript supports synonyms for many of its operators. Using a synonym doesn't change the meaning of an expression, but it can make the expression easier to read. Operators and synonyms are listed in Chapter 6, "Expressions."

References to Files and Applications

You can specify a file or an application as a parameter to many application and scripting addition commands. You can even target files and applications on remote machines connected to a network. You can specify file, alias, application, machine, and zone objects using either the Name reference form or the A Reference To form.

File and application references are described in the following sections:

- "References to Files" (page 191)
- "References to Applications" (page 194)

References to Files

You can use either of these forms to refer to any file:

`file` *nameString*

`alias` *nameString*

where

nameString is a string of the form "*Disk*:*Folder1*:*Folder2*:. . . :*Filename*" that specifies exactly where the file is stored or a string that consists of the file's name only. *Disk* specifies the disk on the local computer on which the application is stored, *Folder1*:*Folder2*:. . . specifies the sequence of folders that you would have to open to find the application on the local computer, and *fileName* specifies the name of the file. File systems in the Mac OS don't normally distinguish uppercase letters from lowercase letters in filenames, although applications such as AppleWorks may distinguish case in document, window, or other object names.

If *nameString* consists of the file's name only, AppleScript attempts to locate the file in the current directory for the application from which the script is being run (for example, Script Editor). The **current directory** is the folder or volume whose contents you can see when you choose Open, Save, or a related command from the application's File menu. The current directory is typically the directory where the application was launched, the directory where the application last opened or saved a previous document, or another directory specified by the application. The current directory may be affected by settings in the General Controls control panel.

Specifying a File by Name or Pathname

To be sure that a command acts on the correct file, specify the entire pathname, including the names of the volume and the entire sequence of folders that you would have to open to find the file.

If you use a reference of the form `file` *nameString*, AppleScript doesn't attempt to locate the file until the script is actually run. When AppleScript executes the statement that accesses the file, the file must exist in the specified folder (or, if only a filename was provided, in the current directory) for AppleScript to locate it. Some commands, such as the Save command, create a file with the specified name in the specified location if it doesn't already exist. Some commands, such

as the Close and Save commands, may replace an existing file with the same name as the specified file (if one exists).

```
tell application "Finder"
    open file "Hard Disk:June Sales"
end tell
```

A disadvantage of specifying a file by name or pathname is that if the user moves the file or renames the file or a folder in its pathname, AppleScript won't be able to find the file and the script will not complete correctly. For a more robust approach, see "Specifying a File by Alias" (page 193).

For a sample script that shows how a script application can handle pathnames of files dropped on it, see "Open Handlers" (page 305).

Specifying a File by Reference

To save a reference of the form `file` *nameString* in a variable, you can use the A Reference To operator as shown in the example that follows.

```
tell application "Finder"
    set fileRef to a reference to file "Hard Disk:June Sales"
    --result: file "Hard Disk:June Sales" of application "Finder"
end tell
tell application "AppleWorks"
    open fileRef
end
```

When you specify a file with a file reference, the file must exist at the time a statement that uses the reference is executed.

You cannot coerce a filename string to a file reference. For example, you cannot replace the second line of the previous script with the following line:

```
set fileRef to ("Hard Disk:June Sales" as file) --result: error!
```

However, you can perform a similar coercion with the alias form, as described in "Specifying a File by Alias" (page 193).

Specifying a File by Alias

If you use a reference of the form `alias` *nameString*, AppleScript creates an **alias** for the file—that is, a representation of the file, much like an alias icon on the desktop, that identifies the file no matter where it is located. AppleScript attempts to locate the file whenever you compile the script—that is, whenever you modify the script and then attempt to check its syntax, save it, or run it.

AppleScript treats an alias like a value that can be stored in a variable and passed around within a script. You don't need to use the A Reference To operator. For example, this script first saves an alias in the variable `fileRef`, then uses the variable in a Tell statement that opens the file.

```
set fileRef to alias "Hard Disk:June Sales"
tell application "AppleWorks"
    open fileRef
end
```

If you save this script as a script application or compiled script, rename the file "June Sales" or move it to another location, then run the script again, the script still works correctly and opens the original file.

You can coerce a filename string to an alias. For example, you can replace the first line of the previous script with the following line:

```
set fileRef to ("Hard Disk:June Sales" as alias)
```

However, you cannot perform a similar coercion with the file form, as described in "Specifying a File by Reference" (page 192).

Differences Between Files and Aliases

The difference between the forms `file` *nameString* and `alias` *nameString* is apparent when the file in question is located on a remote computer. If you use the form `file` *nameString*, AppleScript doesn't attempt to locate the file until you actually run the script. If you use the form `alias` *nameString*, AppleScript also attempts to locate the file whenever you compile the script, requiring appropriate access privileges and possibly a password each time.

Specifying a File by File Specification

You can use a file specification to refer to a file that may not yet exist. The File Specification value class is described in "File Specification" (page 95). You can obtain a file specification from the New File scripting addition command distributed with AppleScript, or from an application command that returns a file specification.

You might use a file specification when you want to let a user specify a filename and location for a file that may not exist, but that you will create or save at a later time. For example, your script could use the first statement below to obtain a file specification by calling the New File scripting addition, which displays a standard system dialog to obtain a filename and location from the user. (The second statement just displays the class of the returned value.) In this case, the script supplies a default name of "New Report":

```
set fileSpec to new file default name "New Report"
class of fileSpec --result: file specification
```

Suppose your script has opened a new AppleWorks document named "Untitled 1" and stored that name in a variable called `currentDocument`. The document has not yet been saved to disk, but the script has executed the statement shown above to get a file specification for the file. At a later point, your script could use the following Tell statement to save the document:

```
tell application "AppleWorks"
    save document currentDocument in fileSpec
end tell
--result: "Untitled 1" renamed and saved as "New Report".
```

References to Applications

You can use this form to refer to any application:

```
application applicationNameString                                    ¬
    [ of machine computerName        [ of zone AppleTalkZoneName ] ]
```

where

applicationNameString is either a string of the form `"`*Disk*`:`*Folder1*`:`*Folder2*`:` . . . `:`*ApplicationName*`"` that specifies where the application is stored on the local

computer or a string that consists of the name of the application. *Disk* specifies the disk on the local computer on which the application is stored, *Folder1*:*Folder2*:. . . specifies the sequence of folders that you would have to open to find the application on the local computer, and *ApplicationName* specifies the name of the application. If it is located on a remote computer, the application must be running and *applicationNameString* must be the name of the application as listed in the Application menu on that computer. AppleScript doesn't distinguish uppercase letters from lowercase letters in application names.

computerName (a string) is the Macintosh Name assigned in the File Sharing control panel of the computer on which the specified application is running. This portion of the reference is required if the application is located on a remote computer.

AppleTalkZoneName (a string) is the name of the zone, if any, in which the specified remote computer is located. The name must appear in the list of AppleTalk Zones displayed in the Chooser.

After a script is compiled, a reference to an application on the local computer identifies the application no matter where it is located on that computer. This behavior resembles the behavior of an alias. However, a reference to an application on a remote computer won't compile unless the application is running and several other conditions are met; see "References to Remote Applications" (page 196) for details.

The actions you can perform on a specific application depend on the way the application that created the file defines an application object. AppleScript always locates the application as described in the sections that follow, but uses the definition in the application's dictionary to determine the characteristics of the object, such as its properties and the commands it can handle.

References to Local Applications

You can specify an application on the local computer with a string of the form "*Disk*:*Folder1*:*Folder2*:. . . .:*ApplicationName*" that specifies the application's exact location. If AppleScript can't find the application in that location, it displays a directory dialog box asking where the application is located.

You can also specify an application on the local computer with only the application's name ("*ApplicationName*"). In this case, AppleScript attempts to find an application of that name among currently running applications. If the application isn't running, AppleScript attempts to locate it in the current

directory. If the application isn't in the current directory, AppleScript displays a directory dialog box asking where the application is located. If the name of the application you select is different from the name specified in the script, the name in the script changes to match the name of the application you select.

When you run a script on the same computer on which it was compiled (that is, on which it was last run or saved, or had its syntax checked), AppleScript finds the application you specified in the original script even if you have moved it or changed its name. If the application has been removed, AppleScript searches for another version of the same application.

As with aliases, it is often convenient to store a reference to an application in a variable:

```
set x to application "AppleWorks"
tell x to quit
```

If you save this script as a script application or compiled script, move the AppleWorks application to another location, change its name, then open the script again, the name `"AppleWorks"` in the script changes to reflect the application's new name, and the script still works correctly.

References to Remote Applications

If an application is on a remote computer, you must specify the application's name as it would be listed in the Application menu, the name of the computer, and, if necessary, the name of the zone in which the computer is located:

```
quit application "AppleWorks" ¬
    of machine "Otto's 2nd Best Server" of zone "Customer Service"
```

For a script to send commands to a remote application, the following conditions must be satisfied:

- The specified remote application must be running. AppleScript doesn't open applications on remote computers.
- The computer that contains the application and the computer on which the script is run must be connected to a network.
- Program linking (set with the File Sharing control panel) must be enabled.

- Access for the user (set with the Users & Groups control panel) must be provided.
- The application must allow remote program linking (set by selecting the application, choosing Sharing from the File menu, and selecting the checkbox labeled “Allow remote program linking”).

For information about these menus and control panels, see the user’s guide for your Macintosh computer.

The following script sends commands to an application on a remote computer, including a Quit command when the script is finished:

```
tell application "AppleWorks" of ¬
    machine "Paula's Mac" of zone "Publications"
        open file "Hard Disk:Reports:Status Report"
        -- statements to perform operations on the report
        close document "Status Report" saving ask
    quit -- quit AppleWorks
end tell
```

CHAPTER 6

Expressions

An *expression* is any series of AppleScript words that has a value. You use expressions to represent or derive values in scripts. When AppleScript encounters an expression, it converts it into an equivalent value. This is known as *evaluation*.

The simplest kinds of expressions, called literal expressions, are representations of values in scripts. For more information on literal expressions, including examples, see Chapter 3, "Values and Constants."

This chapter describes expressions in the following sections:

- "Results of Expressions" (page 200) describes how you can use the Script Editor to evaluate an expression and display its value.
- "Variables" (page 200) describes how to create and use variables. Topics covered include reference variables, data sharing, the scope of variables, and the predefined variables available in AppleScript.
- "Script Properties" (page 208) describes script properties, which are named containers for values that you can use in the same way you use variables.
- "AppleScript Properties" (page 210) describes global properties of AppleScript, which you can use in any script. Some properties act as constants, but you can modify AppleScript's Text Item Delimiters, which AppleScript uses in performing various string operations.
- "Reference Expressions" (page 212) describes compound expressions that refer to objects in applications, and which you can use to represent values in scripts.
- "Operations" (page 213) describes expressions that use operators (such as addition or concatenation) to derive values from other values.

Results of Expressions

The result of any expression is its value.You can use the Script Editor to display the result of an expression by typing an expression on a line by itself and running the script. AppleScript returns the value of the expression. Here's an example:

1. **Open the Script Editor if it is not already open.**
2. **Type the following expression in the editor subwindow:**

   ```
   3 + 4
   ```

3. **Click the Run button in the Script Editor window.**

   This causes AppleScript to evaluate the expression and show the result, `7`, in the result window.

4. **Choose Show Result from the Controls menu.**

   If the result window is behind another Script Editor window, this will bring it to the front.

Variables

A **variable** is a named container in which to store a value. When AppleScript encounters a variable in a statement, it evaluates the variable by getting its value. Variables are contained in a script, not in an application, and their values are normally lost when you close the script that contains them. If you need to keep track of variable values that are persistent even after you close a script or shut down your computer, use properties instead of variables. See "Script Properties" (page 208) for more information.

Unlike variables in many other programming languages, AppleScript variables can hold values of any class. For example, you can use the following sequence of assignment statements to set `x` to a string value, an integer value, and finally a Boolean value:

```
set x to "Title"    --result: “Title”
set x to 12         --result: 12
set x to true       --result: true
```

The name of a variable is a series of characters, called an identifier, that you specify when you create the variable.

Creating Variables

To create a variable in AppleScript, assign it a value. There are two commands for doing this:

- Set
- Copy

With the Set command, list the variable name first, followed by the value you want to assign:

```
set myName to "Paula"
```

With the Copy command, list the value first, followed by the variable name:

```
copy "Paula" to myName
```

Statements like these that assign values to variables are called *assignment statements*.

The variable name is a series of characters called an identifier. AppleScript identifiers are not case sensitive—for example, the variables `myname`, `myName`, and `MYNAME` all represent the same value. The rules for specifying identifiers are listed in “Identifiers” (page 44).

You can use an expression instead of a value in an assignment statement. AppleScript evaluates the expression and assigns the resulting value to the variable. For example, the following statement creates a variable called `myNumber` whose value is the integer 17.

```
set myNumber to 5 + 12
```

A variable can also get its value from a reference. In this case, AppleScript gets the value of the object specified in the reference and assigns it to the variable.

For example, the following statement gets the value of the first word of the document called Report—a string—and stores it in a variable called `myWord`. The next three examples use the document shown in Figure 4-1 (page 142):

```
tell application "AppleWorks"
    set myWord to word 1 of text body of document "Simple"
end tell
--result: “This”
```

To create a variable whose value is the reference itself instead of the value of the object specified by a reference, use the A Reference To operator, which is described in more detail in “The A Reference To Operator” (page 203):

```
tell application "AppleWorks"
    set myWord to a reference to word 1 of text body of document "Simple"
end tell
--result:
-- word 1 of text body of document "Simple" of application "AppleWorks"
```

You can use the Copy command instead of the Set command to assign either a value or a reference to a variable. The following example assigns a value:

```
tell application "Finder"
    copy name of first file of startup disk to firstFileName
end tell
--result (depends on disk contents): "ASP Control Panel Report"
```

The results of the two types of assignment statements are the same in all cases except when the value being assigned is a list, record, or script object. The Copy command makes a new copy of the list, record, or script object, and the Set command creates a variable that shares data with the original list, record, or script object. For more information, refer to “Data Sharing” (page 206).

Using Variables

To use the value of a variable in a script, include the variable in a command or expression. For example, the first statement in the following example creates a variable, called `myName`, whose value is the string `"Steve"`. The second statement uses the variable `myName` in place of a string as the `default answer` parameter of the Display Dialog scripting addition command.

```
set myName to "Steve"
display dialog "What is your name?" default answer myName
```

If you assign a new value to a variable, it replaces the old value. The following script shows what happens when you assign a new value. It uses the Display Dialog command to display the values. Try running this script:

```
set myName to "Steve"
display dialog ("The value of myName is now " & myName) ¬
    buttons "Sure Is" default button 1
set myName to "John"
display dialog ("The value of myName is now " & myName) ¬
    buttons "You Betcha" default button 1
```

The first Display Dialog command displays the value stored by the first assignment statement (the string `"Steve"`). The next Display Dialog command displays the value after the second assignment statement (the string `"John"`).

The A Reference To Operator

To create a variable whose value is a reference instead of the value of the object specified by a reference, use the A Reference To operator. Here's an example:

```
set myWindow to a reference to window "ASP Control Panel Report" ¬
    of application "Apple System Profiler"
```

The value of the variable `myWindow` is the reference

```
window "ASP Control Panel Report" of application "Apple System Profiler"
```

After you create a variable whose value is a reference, you can use it in a script anywhere a reference is required. When AppleScript executes the statement containing the variable, it replaces the variable with the reference. For example, when AppleScript executes the statement

```
tell myWindow
    get name --result: "ASP Control Panel Report"
end tell
```

it replaces the variable `myWindow` with the reference `window "ASP Control Panel Report" of application "Apple System Profiler"`.

SYNTAX

```
[a] ( ref [to] | reference to ) reference
```

where *reference* is a reference to an object.

EXAMPLES

As indicated in the syntax description, there are many ways to shorten expressions containing A Reference To. For example, all of these expressions are equivalent:

```
set myWindow to a reference to the window "Report" ¬
    of the application "Apple System Profiler"

set myWindow to reference to window "Report" ¬
    of application "Apple System Profiler"

set myWindow to a ref window "Report" ¬
    of application "Apple System Profiler"

set myWindow to ref window "Report" ¬
    of application "Apple System Profiler"
```

By using the abbreviation `app`, you can even fit the statement on one line:

```
set myWindow to ref window "Report" of app "Apple System Profiler"
```

As always, it's up to you to decide how conversational to make your script statements.

NOTES

You can use the A Reference To operator to avoid large copy operations. For example, suppose that you want to transfer a large image from one application to another application or script. If your script calls on the first application to create a copy of the image, then passes the copy on to the second application or

script, the copy may require a large block of memory. Instead, your script can ask for a reference to the image, pass the reference, and let the application transfer the data directly.

You can also use the A Reference To Operator to access items in a large list efficiently. For example, the following script required 26 seconds to access all the items in a list of 4,000 integers (the time may vary based on the computer used and the version of AppleScript):

```
-- bigList is a list of 4,000 integers
set numItems to 4000
set t to (time of (current date))
repeat with n from 1 to numItems
    item n of bigList
end repeat
set total to (time of (current date)) - t
total --result: 26 seconds (depends on computer and AppleScript version)
```

The following script performs the same 4,000 access operations in approximately one second, using the A Reference To operator:

```
-- bigList is a list of 4,000 integers
set bigListRef to a reference to bigList
set numItems to 4000
set t to (time of (current date))
repeat with n from 1 to numItems
    item n of bigListRef
end repeat
set total to (time of (current date)) - t
total --result: 1 second (time may vary)
```

After you create a reference with the A Reference To operator, you can use the Contents property to get the value of the object that it refers to. The Contents property is the value of the object specified by a reference. For example, the result of getting the Contents property in the following script is the window reference itself.

```
set myWindow to a reference to window ¬
    "ASP Control Panel Report" of application "Apple System Profiler"
contents of myWindow
--result:
window "ASP Control Panel Report" of application "Apple System Profiler"
```

The following example gets a reference to a window's name:

```
set myWindow to a reference to name of window ¬
    "ASP Control Panel Report" of application "Apple System Profiler"
--result: name of window "ASP Control Panel Report"
--          of application "Apple System Profiler"
```

Getting the Contents property of the name reference returns a name string:

```
contents of myWindow --result: "ASP Control Panel Report"
```

For more information on the Contents property, see "Reference" (page 77).

Data Sharing

Data sharing allows you to create two or more variables that share the same list, record, or script object data; it can be used to promote efficiency when working with large data structures. Only data in lists, records, and script objects can be shared; you cannot share other values. In addition, the shared structures must all be on the same computer.

To create a variable that shares data with another variable whose value is a list, record, or script object, use the Set command. For example, the second Set command in the following example creates the variable `yourList`, which shares data with the previously defined variable `myList`.

```
set myList to { 1, 2, 3 }
set yourList to myList          --this command creates yourList,
                                --which shares data with myList
set item 1 of myList to 4
get yourList --result:{ 4, 2, 3}
```

If you update `myList` by setting the value of its first item to 4, then the value of both `myList` and `yourList` is `{4, 2, 3}`. Rather than having multiple copies of shared data, the same data belongs to multiple structures. When one structure is updated, the other is automatically updated.

To avoid data sharing for lists, records, and script objects, use the Copy command instead of the Set command. The Copy command makes a copy of the list, record, or script object. Changing the value of the original structure

does not change the value of the new copy. Here's an example of using Copy instead of Set to create the variable `yourList`.

```
set myList to { 1, 2, 3 }
copy myList to yourList          --this command makes a copy of
                                 --mylist
set item 1 of myList to 4
get yourList --result: { 1, 2, 3 }
```

If you update `myList`, the value of `yourList` is still `{1, 2, 3}`.

Scope of Variables

The **scope** of a variable determines where else in a script you may refer to the same variable. The scope depends on where you declare a variable and whether you declare it as global or local.

After you define a **global variable** in a script, you can make subsequent references to the same variable either at the top level of the script or in any of the script's subroutines. After you define a **local variable,** you can make subsequent references to the same variable only at the same level of the script at which you defined the variable.

AppleScript assumes that all variables defined at the top level of a script or within its subroutines are local unless you explicitly declare them as global. For more detailed information and examples of the use of variables in subroutines, see "Recursive Subroutines" (page 286).

You can also declare variables within script objects. The scope of variables in a script object is limited to that script object. For more information, see "Scope of Script Variables and Properties" (page 311).

Predefined Variables

Predefined variables are variables whose values are supplied by AppleScript. You can use them in scripts without setting their values. Predefined variables are global—that is, you can use them anywhere in a script.

For a summary of AppleScript's predefined variables, see "Constants" (page 100) and "The Language at a Glance" (page 349).

Note
Although AppleScript does not prevent you from setting the values of predefined variables, you should treat predefined variables as constants—that is, you should never change their values. ◆

Script Properties

A **script property** is a labeled container for a value that you can use in much the same way you use a variable. The value of a script property persists until you recompile the script that contains it, and you can easily set the property's initial value without resetting it each time the script is run. You can accomplish the same thing with a global variable, but it is usually more convenient to use a property for this purpose.

The following sections provide information on script properties:

- "Defining Script Properties" (page 208)
- "Using Script Properties" (page 209)
- "Scope of Script Properties" (page 210)

Note
The description of script properties provided here assumes that you are using the Script Editor application supplied with AppleScript. Other script editors might not support persistence of script properties. If you are using a different script editor, check its documentation to see how it handles script properties. ◆

Defining Script Properties

The syntax for defining a script property is

```
( prop | property ) propertyLabel : initialValue
```

where

propertyLabel is an identifier. The rules for specifying identifiers are listed in "Identifiers" (page 44).

initialValue is the value that is assigned to the property when you first run the script that contains the property or when you save it or check its syntax.

After you define a script property, you change its value the same way you change variable values: with the Set or Copy command. You can get a script property value using the Get command or by using it in an expression.

Using Script Properties

To see how script properties work, try running the following script, which contains a script property called `theCount`.

```
property theCount : 0
set theCount to theCount + 1
display dialog "The value of theCount is: " & theCount ¬
    as string
```

The first time you run the script, the value of `theCount` is set to 0. The Set command adds one to `theCount`, and the Display Dialog scripting addition command displays the value of `theCount`, which is 1.

Now run the script again. The Set command adds 1 to the value of `theCount` (which is still one because it has not been reset), and the Display Dialog command reports a value of 2. If you run the script a third time, the value of `theCount` is 3, and so on.

Now save the script as a compiled script. Close the script, and then open and run it without making any changes. The value of `theCount` is one more than it was before you closed the script.

Finally, recompile the script. (You can do this by making an insignificant change, such as adding a space at the end of a line, and clicking the Check Syntax button.) The value of `theCount` is set to the initial value in the property definition. The Display Dialog command reports a value of 1.

Scope of Script Properties

Like the scope of a variable, the scope of a script property determines where else in a script you may refer to the same property identifier. The scope of a property in turn depends on where you declare it.

You can declare a property at the top level of a script or at the top level of a script object. If you declare it at the top level of a script, a property identifier is visible throughout the script. If you declare it at the top level of a script object, a property identifier is visible only within that script object. After declaring a property, you can use the same identifier as a separate variable only if you first declare it as a local variable.

For detailed information and examples of the use of properties in subroutines, see "Scope of Script Variables and Properties" (page 311).

AppleScript Properties

You can use the global variable `AppleScript` to get properties of AppleScript itself rather than properties of the current target. You can refer to this global variable from any part of any script. AppleScript provides the following global properties you can use as constants: `pi`, `return`, `space`, `tab`, and `version`. These values are described in "Arithmetic Constants" (page 101), "String Constants" (page 105), and "Version Constant" (page 106). You should not attempt to change the values for these constants.

The Text Item Delimiters property consists of a list of strings used as delimiters by AppleScript when it coerces lists to strings or gets text items from strings. AppleScript currently uses only the first delimiter in the list.

You can get and set the current value of AppleScript's Text Item Delimiters. Normally, AppleScript doesn't use any delimiters. For example, if AppleScript's text delimiters have not been explicitly changed. the script

```
{"bread", "milk", "butter", 10.45} as string
```

returns the following result:

```
"breadmilkbutter10.45"
```

For printing or display purposes, it is usually preferable to set the text delimiters to something that's easier to read. For example, the script

```
set AppleScript's text item delimiters to {", "}
{"bread", "milk", "butter", 10.45} as string
```

returns this result:

```
"bread, milk, butter, 10.45"
```

The Text Item Delimiters property also allows you to extract individual names from a pathname. For example, the script

```
set AppleScript's text item delimiters to {":"}
get last text item of "Hard Disk:CD Contents:Release Notes"
```

returns the result `"Release Notes"`.

If you change the Text Items Delimiters property in the Script Editor, it remains changed until you restore its previous value or until you quit the Script Editor and launch it again. If you change the Text Items Delimiters in a script application, it remains changed in that application until you restore its previous value or until the script application quits; however, the delimiters are not changed in the Script Editor or in other script applications you run.

You may want to use an error handler to reset the Text Item Delimiters property to its former value if an error occurs. For more information on handling errors, see "Try Statements" (page 259).

```
set savedTextItemDelimiters to AppleScript's text item delimiters
try
    set AppleScript's text item delimiters to {"**"}
    --other script statements...
    --finally, reset the text item delimiters:
    set AppleScript's text item delimiters to savedTextItemDelimiters
on error m number n from f to t partial result p
    --also reset text item delimiters in case of an error:
    set AppleScript's text item delimiters to savedTextItemDelimiters
    --and resignal the error:
    error m number n from f to t partial result p
end try
```

Reference Expressions

References are compound names that refer to objects in applications, the system, or AppleScript. Because each object has a value, a reference can be used to represent a value in a script. A reference expression is a reference that AppleScript interprets as a value.

A reference can function as a reference to an object or as a reference expression. When a reference is the direct parameter of a command, it usually functions as a reference to an object, indicating to which object the command should be sent. In most other cases, a reference functions as an expression that AppleScript evaluates by getting the reference's value.

For example, the term `word 1 of text body of front document` in the following statement is a reference to an object. It identifies the object to which the Copy command is sent. The statement copies the returned value, the first word of the front document, to the variable `myWord`.

```
tell application "AppleWorks"
    copy word 1 of text body of front document to myWord
end tell
```

On the other hand, the term `(word 1 of text body of front document)` in the following example is a reference expression:

```
tell application "AppleWorks"
    repeat (word 1 of text body of front document) times
        display dialog "Hello"
    end repeat
end tell
```

When AppleScript executes this compound Tell statement, it gets the value of the reference `word 1 of text body of front document`—a string—and then coerces it to an integer, if possible. If the word can't be coerced to an integer, AppleScript reports an error. For information about the Repeat statement, refer to Chapter 7, "Control Statements." For information about coercions, refer to "Coercing Values" (page 97).

Operations

An *operation* is an expression that uses an operator to derive a value from other values, called **operands**. AppleScript includes operators for performing arithmetic operations, comparing values, performing Boolean evaluations, and coercing values.

Each operator can handle operands of specific classes, which are defined in the definition of the operator. For example, the operands for the addition (+) operator must belong to the class Integer or Real, while the operand for the Not operator must belong to class Boolean. Certain operators work with operands from a variety of classes. For example, you can use the concatenation operator (&) to join two strings, two lists, or two records.

The result of each operation is a value of a particular class. For many operators, such as the equality operator (=) and the greater than operator (>), the class of the result is always the same—in these cases, Boolean. For other operators, such as the concatenation operator (&), the class of the result depends on the class of the operands. For example, the result of concatenating two strings is a string, but the result of concatenating two integers is a list of integers.

If you use an operator with operands of the wrong classes, AppleScript attempts to coerce the operands to the correct class, if possible. For example, the concatenation operator (&) works with strings, lists, or records. When AppleScript evaluates the following expression, it coerces the integer `66` to a string before concatenating it with the string `"Route"`.

```
"Route " & 66
--result: "Route 66"
```

For more information, see “Concatenation” (page 229).

When evaluating expressions containing operators, AppleScript checks the leftmost operand first. If the operand does not belong to one of the legal classes for the operator, AppleScript coerces it if possible. After coercing the leftmost operand or verifying that it belongs to a legal class, AppleScript checks the rightmost operand and coerces it (if necessary and possible) to be compatible with the leftmost operand. The exceptions to this rule are expressions with the Is Contained By, Equal, and Is Not Equal operators. AppleScript checks the

rightmost operand first in expressions with the Is Contained By operator. AppleScript never coerces operands of the Equal and Is Not Equal operators.

If AppleScript cannot coerce the operands, it returns an error. For example, the addition operator (+) works with numbers (integers and real numbers) only. If you attempt to evaluate an expression such as `3 + "cat"`, you'll get an error, because AppleScript cannot coerce `"cat"` to a number.

Operations can be performed either by AppleScript or by an application. The rule is that if the leftmost operand is a value, AppleScript performs the operation, and if the leftmost operand is a reference to an application object, the application performs the operation. For example, the comparison

```
tell application "AppleWorks"
    "Hello" contains word 1 of text body of document "Simple"
end tell
```

is performed by AppleScript, because the first operand is a string. Before performing the comparison, AppleScript must get the value of the first word. In contrast, the application specified in the Tell statement, AppleWorks, performs the following comparison

```
tell application "AppleWorks"
    word 1 of text body of document "Simple" contains "Hello"
end tell
```

The Is Contained By operator is an exception to this rule. In expressions with the Is Contained By operator, AppleScript performs the operation if the rightmost operand is a value and the application performs the operation if the rightmost operand is a reference to an application object. For more information, see "Contains, Is Contained By" (page 227).

Table 6-1 summarizes the AppleScript operators. For each operator, it includes a brief description of the operation and lists the class (or classes) of the operands and the class (or classes) of the result. A few of the operators are characters that you type with modifier keys. For these operators, the keystrokes are shown in parentheses. "Operators That Handle Operands of Various Classes" (page 220)

provides more information about how operators treat different classes of operands, along with more detailed explanations and examples of operations.

Table 6-1 AppleScript operators

Operator	Description
`and`	And. Binary logical operator that results in `true` if both the operand to its left and the operand to its right are `true`. Both of the operands must evaluate to Boolean values. When evaluating expressions containing the And operator, AppleScript checks the leftmost operand first. If its value is `false`, AppleScript does not evaluate the rightmost operand, because it already knows the expression is `false`. (This behavior is sometimes called short-circuiting.) *Class of operands*: Boolean *Class of result*: Boolean
`or`	Or. Binary logical operator that results in `true` if either the operand to its left or the operand to its right is `true`. The leftmost operand must evaluate to a Boolean value because it is always checked first when AppleScript evaluates an expression containing the Or operator. If its value is `true`, AppleScript does not evaluate the rightmost operand, because it already knows the expression is `true`. (This behavior is sometimes called short-circuiting.) *Class of operands*: Boolean *Class of result*: Boolean
`&`	Concatenation. Binary operator that joins two values. If the operand to the left of the operator is a string, the result is a string. If the operand to the left of the operator is a record, the result is a record. If the operand to the left of the operator belongs to any other class, the result is a list. For more information, see “Concatenation” (page 229). *Class of operands*: Any *Class of result*: List, Record, String

Table 6-1 AppleScript operators (continued)

Operator	Description

=
`is`
`equal`
`equals`
`[is] equal to`
	Equal. Binary comparison operator that results in `true` if the operand to its left and the operand to its right have the same value. The operands can be of any class. The method AppleScript uses to determine equality depends on the class of the operands—for more information, see "Equal, Is Not Equal To" (page 221).
	Class of operands: Any
	Class of result: Boolean

≠ (Option–equal sign)
`is not`
`isn't`
`isn't equal [to]`
`is not equal [to]`
`doesn't equal`
`does not equal`
	Not equal. Binary comparison operator that results in `true` if the operand to its left and the operand to its right have different values. The operands can be of any class. The method AppleScript uses to determine equality depends on the class of the operands—for more information, see "Equal, Is Not Equal To" (page 221).
	Class of operands: Any
	Class of result: Boolean

>
`[is] greater than`
`comes after`
`is not less than or equal [to]`
`isn't less than or equal [to]`
	Greater than. Binary comparison operator that results in `true` if the value of the operand to its left is greater than the value of the operand to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. The method AppleScript uses to determine which value is greater depends on the class of the operands—for more information, see "Greater Than, Less Than" (page 224).
	Class of operands: Date, Integer, Real, String
	Class of result: Boolean

<
`[is] less than`
`comes before`
`is not greater than or equal [to]`
`isn't greater than or equal [to]`
	Less than. Binary comparison operator that results in `true` if the value of the operand to its left is less than the value of the operand to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. The method AppleScript uses to determine which value is greater depends on the class of the operands—for more information, see "Greater Than, Less Than" (page 224).
	Class of operands: Date, Integer, Real, String
	Class of result: Boolean

Table 6-1 AppleScript operators (continued)

Operator	Description

≥ (Option–greater-than sign)
`>=`
`[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. Binary comparison operator that results in `true` if the value of the operand to its left is greater than or equal to the value of the operand to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. The method AppleScript uses to determine which value is greater depends on the class of the operands.
	Class of operands: Date, Integer, Real, String
	Class of result: Boolean

≤ (Option–less-than sign)
`<=`
`[is] 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. Binary comparison operator that results in `true` if the value of the operand to its left is less than or equal to the value of the operand to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. The method AppleScript uses to determine which value is greater depends on the class of the operands.
	Class of operands: Date, Integer, Real, String
	Class of result: Boolean

`start[s] with` `begin[s] with`	Starts with. Binary containment operator that results in `true` if the list or string to its right matches the beginning of the list or string to its left. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. For more information, see "Starts With, Ends With" (page 226). *Class of operands*: List, String *Class of result*: Boolean
`end[s] with`	Ends with. Binary containment operator that results in `true` if the list or string to its right matches the end of the list or string to its left. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. For more information, see "Starts With, Ends With" (page 226). *Class of operands*: List, String *Class of result*: Boolean

Table 6-1 AppleScript operators (continued)

Operator	Description
`contain[s]`	Contains. Binary containment operator that results in `true` if the list, record, or string to its right matches any part of the list, record, or string to its left. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. For more information, see "Contains, Is Contained By" (page 227). *Class of operands*: List, Record, String *Class of result*: Boolean
`does not contain` `doesn't contain`	Does not contain. Binary containment operator that results in `true` if the list, record, or string to its right does not match any part of the list, record, or string to its left. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. *Class of operands*: List, Record, String *Class of result*: Boolean
`is in` `is contained by`	Is contained by. Binary containment operator that results in `true` if the list, record, or string to its left matches any part of the list, record, or string to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the left of the operator to the class of the operand to the right. For more information, see "Contains, Is Contained By" (page 227). *Class of operands*: List, Record, String *Class of result*: Boolean

`is not in`
`is not contained by`
`isn't contained by`
	Is not contained by. Binary containment operator that results in `true` if the list, record, or string to its left does not match any part of the list, record, or string to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the left of the operator to the class of the operand to the right.
	Class of operands: List, Record, String
	Class of result: Boolean

`*`	Multiply. Binary arithmetic operator that multiplies the number to its left and the number to its right. *Class of operands*: Integer, Real *Class of result*: Integer, Real

Table 6-1 AppleScript operators (continued)

Operator	Description
`+`	Plus. Binary arithmetic operator that adds the number or date to its left and the number or date to its right. Only integers can be added to dates. AppleScript interprets such an integer as a number of seconds. *Class of operands*: Date, Integer, Real *Class of result*: Date, Integer, Real
`-`	Minus. Binary or unary arithmetic operator. The binary operator subtracts the number to its right from the number or date to its left. The unary operator makes the number to its right negative. Only integers can be subtracted from dates. AppleScript interprets such an integer as a number of seconds. *Class of operands*: Date, Integer, Real *Class of result*: Date, Integer, Real
`÷` (Option-slash) `/`	Division. Binary arithmetic operator that divides the number to its left by the number to its right. *Class of operands*: Integer, Real *Class of result*: Real
`div`	Integral division. Binary arithmetic operator that divides the number to its left by the number to its right and returns the integral part of the answer as its result. *Class of operands*: Integer, Real *Class of result*: Integer
`mod`	Remainder. Binary arithmetic operator that divides the number to its left by the number to its right and returns the remainder as its result. *Class of operands*: Integer, Real *Class of result*: Integer, Real
`^`	Exponent. Binary arithmetic operator that raises the number to its left to the power of the number to its right. *Class of operands*: Integer, Real *Class of result*: Real

Table 6-1 AppleScript operators (continued)

Operator	Description
`as`	Coercion. Binary operator that converts the operand to its left to the class listed to its right. Not all values can be coerced to all classes. The coercions that AppleScript can perform are listed in "Coercing Values" (page 97). The additional coercions, if any, that applications can perform are listed in application dictionaries. *Class of operands*: the operand to the right of the operator must be a class identifier; the operand to the left must be a value that can be converted to that class *Class of result*: the class specified by the class identifier to the right of the operator
`not`	Not. Unary logical operator that results in `true` if the operand to its right is `false`, and `false` if the operand to its right is `true`. *Class of operand*: Boolean *Class of result*: Boolean
`[a] ( ref [to] | reference to )`	A Reference To. Unary operator that causes AppleScript to interpret the value to its right as a reference instead of getting its value. For more information about the A Reference To operator, see "The A Reference To Operator" (page 203). *Class of operand*: Reference *Class of result*: Reference

Operators That Handle Operands of Various Classes

Many operators can handle operands of a variety of classes. The following sections describe how the specified operators behave with different classes of operands:

- "Equal, Is Not Equal To" (page 221)
- "Greater Than, Less Than" (page 224)
- "Starts With, Ends With" (page 226)
- "Contains, Is Contained By" (page 227)
- "Concatenation" (page 229)

Equal, Is Not Equal To

The Equal and Is Not Equal To operators can handle operands of any class.

Table 6-1, "AppleScript operators" (page 215) summarizes the Equal and Is Not Equal To operators and other AppleScript operators.

OPERANDS OF DIFFERENT CLASSES

Two expressions of different classes are not equal.

BOOLEAN EXPRESSION

Two Boolean expressions are equal if both of them evaluate to `true` or if both evaluate to `false`. They are not equal if one evaluates to `true` and the other to `false`.

CLASS IDENTIFIER

Two class identifiers are equal if they are the same identifier. They are not equal if they are different identifiers.

CONSTANT

Two constants are equal if they are the same. They are not equal if they are different.

DATA

Two data values are equal if they are the same length in bytes and their bytes are the same (AppleScript does a byte-wise comparison).

DATE

Two dates are equal if they both represent the same date, even if they are expressed in different formats. For example, the following expression is `true`, because `date date "12/31/99"` and `date "December 31st, 1999"` represent the same date.

```
date "12/31/99" = date "December 31st, 1999"
```

When you compile the previous statement, the Script Editor converts it to a form similar to the following (the format may vary depending on the settings of the Date & Time control panel):

```
date "Friday, December 31, 1999 12:00:00 AM" = ¬
    date "Friday, December 31, 1999 12:00:00 AM"
```

INTEGER

Two integers are equal if they are the same. They are not equal if they are different.

LIST

Two lists are equal if each item in the list to the left of the operator is equal to the item in the same position in the list to the right of the operator. They are not equal if items in the same positions in the lists are not equal or if the lists have different numbers of items. For example,

```
{ (1 + 1), (4 > 3) } = {2, true}
```

is `true`, because `(1 + 1)` evaluates to `2`, and `(4 > 3)` evaluates to `true`.

REAL

Two real numbers are equal if they both represent the same real number, even if the formats in which they are expressed are different. For example, the following expression is `true`.

```
0.01 is equal to 1e-2
```

Two real numbers are not equal if they represent different real numbers.

RECORDS

Two records are equal if they both contain the same collection of properties and if the values of properties with the same label are equal. They are not equal if the records contain different collections of properties, or if the values of properties with the same label are not equal. The order in which properties are listed does not affect equality. For example, the following expression is `true`.

```
{ name:"Matt", mileage:"8000" } = { mileage:"8000",¬
    name:"Matt"}
```

REFERENCE

Two references are equal if their classes, reference forms, and containers are identical. They are not equal if their classes, reference forms, and containers are not identical, even if they refer to the same object.

For example, the expression `x = y` in the following Tell statement is always `true`, because the classes (`word`), reference forms (Index), and containers (`paragraph 1 of text body`) of the two references are identical.

```
tell document "Simple" of application "AppleWorks"
    set x to a reference to word 1 of paragraph 1 of text body
    set y to a reference to word 1 of paragraph 1 of text body
    x = y
end tell
--result:true
```

The expression `x = y` in the following statement is `false`, regardless of the text of the document, because the containers are different.

```
tell document "Simple" of application "AppleWorks"
    set x to a reference to word 1 of paragraph 1 of text body
    set y to a reference to word 1 of text body
    x = y
end tell
--result:false
```

When you use references in expressions without the A Reference To operator, the values of the objects specified in the references are used to evaluate the expressions. For example, the result of the following expression is `true` if both documents begin with the same word.

```
tell application "AppleWorks"
        word 1 of text body of document "Report" ¬
    =   word 1 of text body of document "Simple"
end tell
```

STRING

Two strings are equal if they are both the same series of characters. They are not equal if they are different series of characters. AppleScript compares strings character by character. It does not distinguish uppercase from lowercase letters unless you use a Considering statement to consider the `case` attribute. For example, the following expression is `true`.

```
"DUMPtruck" is equal to "dumptruck"
```

AppleScript considers all characters and punctuation, including spaces, tabs, return characters, diacritical marks, hyphens, periods, commas, question marks, semicolons, colons, exclamation points, backslash characters, and single and double quotation marks in string comparisons. AppleScript ignores style in string comparisons.

All string comparisons can be affected by Considering and Ignoring statements, which allow you to selectively consider or ignore the case of characters, as well as specific types of characters. For more information, see "Considering and Ignoring Statements" (page 268).

Greater Than, Less Than

The Greater Than and Less Than operators work with dates, integers, real numbers, and strings.

Table 6-1, "AppleScript operators" (page 215) summarizes the Greater Than and Less Than operators and other AppleScript operators.

DATE

A date is greater than another date if it represents a later time. A date is less than another date if it represents an earlier time.

INTEGER

An integer is greater than a real number or another integer if it represents a larger number. An integer is less than a real number or another integer if it represents a smaller number.

REAL

A real number is greater than an integer or another real number if it represents a larger number. A real number is less than an integer or another real number if it represents a smaller number.

STRING

A string is greater than (comes after) another string if it would appear after the other string in an English-language dictionary. For example,

```
"zebra" comes after "aardvark"
```

and

```
"zebra" > "aardvark"
```

are `true`. A string is less than (comes before) another string if it would appear in a dictionary before the other string. For example,

```
"aardvark" comes before "zebra"
```

and

```
"aardvark" < "zebra"
```

are `true`.

AppleScript uses the language-defined collating sequence specified by the Text control panel to determine a word's position in an English-language dictionary. The order of the collating sequence, in the English language and Roman script, is

space`!"#$%&'()*+,-./`
`0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`{|}~`

AppleScript compares strings character by character. When the corresponding characters in two strings are not the same, the string containing the character closest to the beginning of the collating sequence is less than the other string. If two strings have identical characters but one is shorter than the other, the shorter string is less than the longer string. AppleScript treats all letters as

lowercase letters, unless you use a Considering statement to consider the `case` attribute, in which case letters are ordered as follows, in the English language and Roman script:

```
AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
```

If you use a Considering statement that considers the `diacriticals` attribute, AppleScript uses the following ordering for vowels, in the English language and Roman script:

```
a á à â ä ã å
e é è ê ë
i í ì î ï
o ó ò ô ö õ
u ú ù û ü
```

For more information about Considering statements, refer to "Considering and Ignoring Statements" (page 268).

Starts With, Ends With

The Starts With and Ends With operators work with lists and strings.

Table 6-1, "AppleScript operators" (page 215) summarizes the Starts With and Ends With operators and other AppleScript operators.

LIST

A list starts with another list if the values of the items in the list to the right of the operator are equal to the values of the items at the beginning of the list to the left. A list ends with another list if the values of the items in the list to the right of the operator are equal to the values of the items at the end of the list to the left. In both cases, the items in the two lists must be in the same order. Both Starts With and Ends With work if the operand to the right of the operator is a single value. For example, the following three expressions are all `true`:

```
{ "this", "is", 2, "cool" } ends with "cool"

{ "this", "is", 2, "cool" } starts with "this"

{ "this", "is", 2, "cool" } starts with { "this", "is" }
```

STRING

A string starts with another string if the characters in the string to the right of the operator are the same as the characters at the beginning of the string to the left. For example, the following expression is `true`:

```
"operand" starts with "opera"
```

A string ends with another string if the characters in the string to the right of the operator are the same as the characters at the end of the string to the left. For example, the following expression is `true`:

```
"operand" ends with "and"
```

AppleScript compares strings character by character according to the rules for the Equal operator.

Contains, Is Contained By

The Contains and Is Contained By operators work with lists, records, and strings.

Table 6-1, "AppleScript operators" (page 215) summarizes the Contains and Is Contained By operators and other AppleScript operators.

LIST

A list contains another list if the list to the right of the operator is a sublist of the list to the left of the operator. A sublist is a list whose items appear in the same order and have the same values as any series of items in the other list. For example,

```
{ "this", "is", 1 + 1, "cool" } contains { "is", 2 }
```

is `true`, but

```
{ "this", "is", 2, "cool" } contains { 2, "is" }
```

is `false`.

A list is contained by another list if the list to the left of the operator is a sublist of the list to the right of the operator. For example, the following expression is `true`:

```
{ "is", 2} is contained by { "this", "is", 2, "cool" }
```

Both Contains and Is Contained By work if the sublist is a single value. For example, both of the following expressions are `true`:

```
{ "this", "is", 2, "cool" } contains 2

2 is contained by { "this", "is", 2, "cool" }
```

RECORD

A record contains another record if all the properties in the record to the right of the operator are included in the record to the left, and the values of properties in the record to the right are equal to the values of the corresponding properties in the record to the left. A record is contained by another record if all the properties in the record to the left of the operator are included in the record to the right, and the values of the properties in the record to the left are equal to the values of the corresponding properties in the record to the right. The order in which the properties appear does not matter. For example,

```
{ name:"Matt", mileage:"8000", description:"fast"} ¬
    contains { description:"fast", name:"Matt" }
```

is `true`.

STRING

A string contains another string if the characters in the string to the right of the operator are equal to any contiguous series of characters in the string to the left of the operator. For example,

```
"operand" contains "era"
```

is `true`, but

```
"operand" contains "dna"
```

is `false`.

A string is contained by another string if the characters in the string to the left of the operator are equal to any series of characters in the string to the right of the operator. For example, this statement is `true`:

```
"era" is contained by "operand"
```

Concatenation

The concatenation operator (&) can handle operands of any class. The result type of a concatenation depends on the type of the left hand operand. If the left-hand operand is a string, the result is always a string, and only in this case does AppleScript try to coerce the value of the right-hand operand to match that of the left. If the left-hand operand is a record, the result is always a record. If the left-hand operand is any other type, the result is a list.

Table 6-1, “AppleScript operators” (page 215) summarizes the concatenation operator (&) and other AppleScript operators.

STRING

The concatenation of two strings is a string that begins with the characters in the string to the left of the operator, followed immediately by the characters in the string to the right of the operator. AppleScript does not add spaces or other characters between the two strings. For example,

```
"dump" & "truck"
```

returns the string `"dumptruck"`.

If the operand to the left of the operator is a string, but the operand to the right is not, AppleScript attempts to coerce the operand to the right to a string. For example, when AppleScript evaluates the expression

```
"Route " & 66
```

it coerces the integer `66` to the string `"66"`, and the result is

```
"Route 66"
```

However, you get a different result if you reverse the order of the operands:

```
66 & "Route "
--result: {66, "Route "}
```

RECORD

The concatenation of two records is a record that begins with the properties of the record to the left of the operator, followed by the properties of the record to the right of the operator. If both records contain properties with the same name, the value of the property from the record to the left of the operator appears in the result. For example, the result of the expression

```
{ name:"Matt", mileage:"8000" } & ¬
    { name:"Steve", framesize:58 }
```

is

```
{ name:"Matt", mileage:"8000", frameSize:58 }
```

ALL OTHER CLASSES

The concatenation of two operands that are not strings or records is a list whose first item is the value of the operand to the left of the operator, and whose second item is the value of the operand to the right of the operator. If the operands to be concatenated are lists, then the result is a list containing all the items in the list to the left of the operator, followed by all the items in the list to the right of the operator. For example,

```
{ "This" } & { "and", "that" }
--result: { "This", "and", "that" }

{ "This" } & item 1 of { "and", "that" }
--result: { "This", "and" }
```

In the following example, however, both specified items are strings, so concatenation results in a string:

```
item 1 of { "This" } & item 1 of { "and", "that" }
--result: "Thisand"
```

For large lists, it may be more efficient to use the Copy or Set command, rather than concatenation, to add an item to a list. For information on working efficiently with large lists, see "List" (page 67).

Operator Precedence

AppleScript allows you to combine expressions into larger, more complex expressions. When evaluating expressions, AppleScript uses operator precedence to determine which operations are performed first. Table 6-2 shows the order in which AppleScript performs operations.

To see how operator precedence works, consider the following expression.

```
2 * 5 + 12
--result: 22
```

To evaluate the expression, AppleScript performs the multiplication operation `2 * 5` first, because as shown in Table 6-2, multiplication has higher precedence than addition.

The column labeled "Associativity" in Table 6-2 indicates the order in which AppleScript performs operations if there are two or more operations of the same precedence in an expression. The word "none" in the Associativity column indicates that you cannot have multiple consecutive occurrences of the operation in an expression. For example, the expression `3 = 3 = 3` is not legal because the associativity for the equal operator (=) is "none." The word "unary" indicates that the operator is a unary operator. To evaluate expressions with multiple unary operators of the same order, AppleScript applies the operator

closest to the operand first, then applies the next closest operator, and so on. For example, the expression `not not not true` is evaluated as `not (not (not true))`.

Table 6-2 Operator precedence

Order	Operators	Associativity	Type of operator
1	`( )`	Innermost to outermost	Grouping
2	`+` `-`	Unary	Plus or minus sign for numbers
3	`^`	Right to left	Exponentiation

4	`*`
	`/`
	`÷`
	`div`
	`mod`
		Left to right	Multiplication and division

5	`+` `-`	Left to right	Addition and subtraction
6	`&`	Left to right	Concatenation
7	`as`	Left to right	Coercion

8	`<`
	`≤`
	`>`
	`≥`
		None	Comparison

9	`=` `≠`	None	Equality and inequality
10	`not`	Unary	Logical negation
11	`and`	Left to right	Logical for Boolean values
12	`or`	Left to right	Logical for Boolean values

You can change the order in which AppleScript performs operations by grouping expressions in parentheses. As shown in Table 6-2, AppleScript evaluates expressions in parentheses first. For example, adding parentheses

around `5 + 12` in the following expression causes AppleScript to perform the addition operation first and changes the result.

```
2 * ( 5 + 12 )
--result: 34
```

Date-Time Arithmetic

AppleScript supports these operations with the + and - operators on date and time difference values:

date + *timeDifference*
`--result:` *date*

date - *date*
`--result:` *timeDifference*

date - *timeDifference*
`--result:` *date*

where *date* is a date value and *timeDifference* is an integer value specifying a time difference in seconds.

To simplify the notation of time differences, you can also use one or more of these constants:

`minutes`	60
`hours`	60 * minutes
`days`	24 * hours
`weeks`	7 * days

Here's an example:

```
date "Apr 15, 1998" + 4 * days + 3 * hours + 2 * minutes
```

It is often useful to be able to specify a time difference between two dates; for example:

```
set timeInvestment to current date - date "January 22, 1998"
```

After running this script, the value of the `timeInvestment` variable is an integer that specifies the number of seconds between the two dates. If you then add this time difference to the starting date (January 22, 1998), AppleScript returns a date value equal to the current date when the `timeInvestment` variable was set.

To express a time difference in more convenient form, divide the number of seconds by the appropriate constant:

```
31449600 / weeks
--result: 52.0

62899200 / (weeks * 52)
--result (in years): 2.0

151200 / days
--result: 1.75
```

To get an integral number of hours, days, and so on, use the `div` operator:

```
151200 div days
--result: 1
```

To get the difference, in seconds, between the current time and Greenwich mean time, use the scripting addition command Time to GMT. For example, if you are in Cupertino, California, and your computer is set to Pacific Standard Time, Time to GMT produces this result:

```
time to GMT
--result: -28800
```

For more information about the Time to GMT command, and about the other standard scripting addition commands distributed with AppleScript, see the following website:

<http://www.apple.com/applescript/>

For more information on working with dates, see “Date” (page 62).

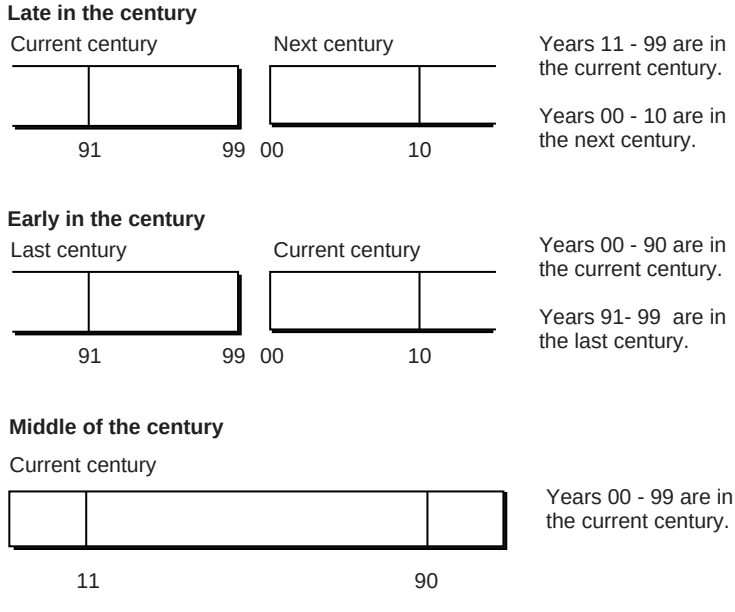
Working With Dates at Century Boundaries

AppleScript's support for dates is based on the same operating system utilities other Macintosh applications use. Mac OS date and time utilities have correctly handled issues related to the year 2000 since the introduction of the Macintosh. The original date and time utilities, introduced with the Macintosh 128K in 1984, use a 32-bit value to store seconds, starting at 12:00:00 a.m., January 1, 1904 and extending to 6:28:15 a.m. on February 6, 2040.

More recent date and time utilities use a 64-bit signed value that can represent dates from 30081 B.C. to 29940 A.D. However, AppleScript currently will not handle dates before 1/1/1000 or after 12/31/9999. For more information on Mac OS date and time utilities, see *Inside Macintosh: Operating System Utilities*, available at the Apple Developer website:

<http://www.apple.com/developer/>

Figure 6-1 Two-digit dates at century boundaries



Two-digit dates near the year 2000, or any century boundary, may still represent a problem if your script accepts two-digit dates as text from applications or users. Whenever your script calls on AppleScript to convert a two-digit date to a Date value, AppleScript converts the text representation to a full date for internal storage, following the rules illustrated in Figure 6-1 and summarized here:

- If the current year is late in any century (years 91 to 99)
  - a date with a year value from 11 to 99 is in the current century (in 1999, for example, 11 = 1911 and 99 = 1999)
  - a date with a year value from 00 to 10 is in the next century (in 1999, for example, 03 = 2003)
- If the current year is early in any century (years 00 to 10)
  - a date with a year value from 00 to 90 is in the current century (in 2000, for example, 00 = 2000, 45 = 2045, and 90 = 2090)
  - a date with a year value from 90 to 99 is in the previous century (in 2000, for example, 99 = 1999)
- If the current year is in the middle of any century (years 11 to 90)
  - a date with any value for year value, from 00 to 99, is in the current century (in 2011, for example, 00 = 2000, 45 = 2045 and 99 = 2099)

So a script that uses nearby dates, such as for office scheduling or other near-term planning, can use two-digit dates with some degree of safety. However, a script that does long-term date calculations, such as for genealogy or mortgages, should require four-digit dates.

Control Statements

A **control statement** is statement that controls when and how other statements are executed. Most control statements are compound statements—that is, statements that contain other statements.

By default, AppleScript executes the statements in a script in sequence, one after the other. Control statements can change the order in which AppleScript executes statements by causing AppleScript to repeat or skip statements or go to a different statement.

The first two sections in this chapter provide general information on working with control statements:

- "Characteristics of Control Statements" (page 238), provides an overview of control statements.
- "Debugging Control Statements" (page 239) describes how to use the Event Log window to help debug control statements.

The rest of the chapter describes the following AppleScript control statements:

- "Tell Statements" (page 240) define the default target to which commands are sent if no direct object is specified.
- "If Statements" (page 245) allow you to execute or skip statements based on the outcome of one or more tests.
- "Repeat Statements" (page 249) allow you to repeat a series of statements.
- "Try Statements" (page 259) allow you to handle error messages.
- "Considering and Ignoring Statements" (page 268) allow you to consider or ignore certain attributes, such as case, punctuation, and white space, in string comparisons.
- "With Timeout Statements" (page 272) allow you to specify how long AppleScript waits for an application command or scripting addition to complete before stopping execution of the script and returning an error.

- "With Transaction Statements" (page 275) allow you to take advantage of applications that support the notion of a transaction—a sequence of related events that should be performed as if they were a single operation.

Table A-4 (page 360) summarizes the syntax for the control statements that are described in this chapter.

Characteristics of Control Statements

Most control statements are compound statements that contain other statements. For example, the following Tell statement is a compound statement that contains two If statements.

```
tell application "Finder"
    if (exists folder "Reports" of disk "Hard Disk") then
        set reportsToPrint to (count every file ¬
                of folder "Reports" of disk "Hard Disk")
    else
            set reportsToPrint to 0
    end if -- checking for any reports to print

    -- If we found any reports, print them.
    if reportsToPrint > 0 then
        tell application "ReportWizard"
            -- Statements to print the reports.
        end tell
    end if -- had some reports to print
end tell
```

Compound statements begin with one or more reserved words, such as `tell` in the example above, that identify the type of compound statement. The last line of a compound statement is always `end`, which can optionally include the word that begins the control statement.

Control statements can contain other control statements. For example, the Tell statement above contains two If statements, one of which in turn contains a Set command. Control statements that are contained within other control statements are sometimes called **nested control statements.**

All control statements can be compound statements. In addition, some control statements can be written as single statements. For example, the statement

```
if (x > y) then return x
```

is equivalent to

```
if (x > y) then
    return x
end if
```

You can use a simple statement only when you're controlling the execution of a single statement (such as `return x` in the previous example).

Debugging Control Statements

One technique that can be especially useful for debugging control statements is to use the Script Editor's Event Log window. You open the Event Log window from the Controls menu or by typing Command-E. The Event Log window displays diagnostic information while a script is running. This information can help you discover and correct errors by showing the results of the script's actions.

In addition to simply opening the Event Log to view the results of a script's actions, you can insert log statements at strategic locations in your script. A log statement reports the value of one or more variables to the Event Log window.

In the following script, the statement `log currentWord` is a debugging statement. It causes the current word to be displayed in the Script Editor's Event Log window each time through the loop. Log statements can be very helpful in testing a Repeat loop or other control statement. Once the loop is working correctly, you can comment out or delete any log statements.

```
set wordList to words in "Where is the hammer?"
repeat with currentWord in wordList
    log currentWord
    if currentWord as text is equal to "hammer" then
        display dialog "I found the hammer!"
    end if
end repeat
```

This script examines a list of words with the *Repeat With (loopVariable) In (list)* form of the Repeat statement, displaying a dialog if it finds the word "hammer" in the list. For more information, see "Repeat With (loopVariable) In (list)" (page 256).

For basic debugging techniques and a detailed description of how to use the Event Log window, see "Debugging Scripts" (page 47). For additional information on the Script Editor's Event Log window, see the AppleScript section of the Mac OS Help Center.

Tell Statements

Tell statements specify the default target, the object to which commands are sent if they do not include a direct parameter. For example, in the following Tell statement, the Close command does not include a direct parameter.

```
tell front window of application "AppleWorks"
    close
end tell
```

As a result, the Close command is sent to the front window, the default target specified in the Tell statement.

When AppleScript encounters a partial reference (a reference that does not specify every container of an object), it uses the default target to complete it. For example, in the following Tell statement, the reference `word 3` does not specify all of the containers of the word object, so AppleScript completes it with the default target.

```
tell front document of application "AppleWorks"
    delete word 3 of text body
end tell
```

The result is that the Delete command is sent to the third word in the text body of the front document of AppleWorks.

A Tell statement also indicates which dictionary AppleScript should use to interpret words contained in the statement. For example, the previous Tell statement tells AppleScript to use the AppleWorks dictionary, which contains the definitions for the Delete command, the Word object, the Text Body object,

and so on. If the Tell statement had not specified the application, AppleScript would not have understood the Delete command.

Nested Tell Statements

You can nest a Tell statement inside another Tell statement as long as every statement in a Tell block can be handled by the target application specified in the Tell statement (or by AppleScript). For example, the following script will compile and run successfully:

```
tell application "Finder"
    tell document 1 of application "AppleWorks"
        get style of text body -- handled by AppleWorks
    end tell
    set fileName to name of first file ¬
        of startup disk -- handled by Finder
    count characters in fileName -- handled by AppleScript
end tell
```

This example works because an AppleWorks document can get the style of its text body, the Finder can get the name of the first file on the startup disk, and AppleScript can count the characters in a string. Notice what happens, however, when we move one statement in the previous script:

```
tell application "Finder"
    tell document 1 of application "AppleWorks"
        get style of text body -- handled by AppleWorks
        set fileName to name of first file of startup disk -- ERROR!
            -- AppleWorks doesn't know how to handle this statement!
    end tell
    count characters in fileName -- handled by AppleScript
end tell
```

Because the script attempts to get the name of a file from within a Tell statement that specifies AppleWorks, running this script generates an error.

Using *it*, *me*, and *my* in Tell Statements

AppleScript defines the variables `it` and `me` for use in Tell statements. It also defines the word `my`, which you can use in place of the phrase `of me`.

The variable `it` is the default target. The value of `it` is a reference, as in

```
tell document "Introduction" of application "AppleWorks"
    get name of it --result: "Introduction"
end tell
```

The value of the variable `it` is `document "Introduction" of application "AppleWorks"`. The result of the Get command is the string `"Introduction"`.

The variable `me` refers to the current script, as in the following example. The line that specifies the property name could be placed at the end of the script as well as at the beginning.

```
property name : "Script"
tell document "Introduction" of application "AppleWorks"
    get name of me --result: "Script"
end tell
```

The reference `name of me` refers to the name property of the current script. The result of the Get command is the string `"Script"`.

The following script, which is equivalent to the previous example, uses the word `my` as an alternative to `of me`.

```
property name : "Script"
tell document "Introduction" of application "AppleWorks"
    get my name --result: "Script"
end tell
```

If you refer to a property in a Tell statement without using either `it`, `me`, or `my`, AppleScript assumes that you want the property of the default target of the Tell statement. For example, the result of the Get command in the following Tell statement is `"Introduction"`.

```
property name : "Script"
tell document "Introduction" of application "AppleWorks"
    get name --result: "Introduction"
end tell
```

If AppleScript cannot find the property in the dictionary of the default target of the Tell statement, then it assumes you want the property of the current script. For example, the result of the Get command in the following Tell statement is `1000000`.

```
property x : 1000000
tell document "Introduction" of application "AppleWorks"
    get x --result: 1000000
end tell
```

In addition to distinguishing script properties from object properties, `me` and `my` are used to distinguish user-defined commands (subroutines) from application commands in Tell statements. For more information, see Chapter 8, "Handlers."

Note
Within tests in Filter references, the direct object is the object being tested, so the variable `it` refers to the object currently being tested. See "Using the Filter Reference Form" (page 187) for information about the use of `it` in tests. ◆

Tell (Simple Statement)

A simple Tell statement specifies the object to which to send a command.

SYNTAX

`tell` *referenceToObject* `to` *statement*

where

referenceToObject is a reference to an application object, system object, or script object.

statement is any AppleScript statement.

EXAMPLE

```
tell front window of application "Finder" to close
```

NOTES

If *referenceToObject* specifies an application on a remote computer, additional conditions must be met. These conditions are described in "References to Applications" (page 194).

If *referenceToObject* specifies an application on the same computer that is not running, AppleScript launches the application.

Tell (Compound Statement)

A compound Tell statement specifies the default target of the commands it contains.

SYNTAX

```
tell referenceToObject
    [ statement ]...
end [ tell ]
```

where

referenceToObject is a reference to an application object, system object, or script object.

statement is any AppleScript statement.

EXAMPLES

The next three examples show how to close a window using, respectively, a compound Tell statement, a simple Tell statement, and a very brief version of a simple Tell statement.

```
tell application "Finder"
    tell front window
        close
    end tell
end tell

tell front window of application "Finder"
    close
end tell

tell app "Finder" to close front window
```

The following example closes a window on a remote computer.

```
tell application "Finder" of machine ¬
    "Steve's PowerBook" of zone "Fourth Floor South"
        tell front window
            close
        end tell
end tell
```

NOTES

If *referenceToObject* specifies an application on a remote computer, additional conditions must be met. These conditions are described in "References to Remote Applications" (page 196).

If *referenceToObject* specifies an application on the same computer that is not running, AppleScript launches the application.

If Statements

If statements allow you to define statements or groups of statements that are executed only in specific circumstances. Each If statement contains one or more **Boolean expressions** whose values can be either `true` or `false`. AppleScript executes the statements contained in the If statement only if the value of the Boolean expression is `true`.

If statements are also called **conditional statements.** Boolean expressions in If statements are also called **tests.**

The following example uses an If statement to determine whether to display a dialog box.

```
if dependents > 2 then
    display dialog "You might need to file an extra form"
end if
```

The If statement contains the Boolean expression `dependents > 2`. If the value of the Boolean expression is `true`, the Display Dialog command is executed. If the value of the Boolean expression is `false`, the Display Dialog command is not executed.

Display Dialog is one of the standard scripting addition commands distributed with AppleScript. For information about these commands, see the AppleScript section of the Mac OS Help Center, or see the AppleScript website at

<http://www.apple.com/applescript/>

If statements can contain multiple tests. For example, the following statement contains two tests to determine if two files exist before using them.

```
tell application "Finder"
    if (the file "Hard Disk:Status Report" exists) then
        if (the file "Hard Disk:Department Status" exists) then
            tell application "AppleWorks"
                -- Statements that open the files, perform operations
                -- on them, and close the files. (Not shown.)
            end tell -- AppleWorks
        end if -- second file exists
    end if -- first file exists
end tell -- tell Finder
```

The two If statements in the previous example can be combined into one If statement with two tests:

```
tell application "Finder"
    if (the file "Hard Disk:Status Report" exists) and ¬
        (the file "Hard Disk:Department Status" exists) then
```

```
        -- Other statements not shown.
    end if -- files exist
end tell -- tell Finder
```

Rather than test whether a file exists before performing operations on it, you can use a Try statement that always attempts to operate on the file, but contains an error branch to deal with the case where the file doesn't exist. Try statements are described in "Try Statements" (page 259).

An If statement can contain any number of Else If clauses; AppleScript looks for the first Boolean expression contained in an If or Else If clause that is `true`, executes the statements contained in its block (the statements between one Else If and the following Else If or Else clause), and then exits the If statement.

An If statement can also include a final Else clause. The statements in its block are executed if no other test in the If statement passes. Consider the following, more elaborate If statement.

```
display dialog "How many dependents?" default answer ""
set dependents to (text returned of result) as integer
--Could test here to be sure user entered a valid number.

display dialog "Have you ever been audited?" buttons {"No", "Yes"}

if button returned of result = "Yes" then
    set audit to true
else
    set audit to false
end if

if dependents < 9 and audit = false then
    display dialog "No extra forms are required."
else if dependents < 9 and audit = true then
    display dialog "You might need to file an extra form."
else --anything greater than 9
    display dialog "You will need to file an extra form."
end if
```

This example uses Boolean operators to create a more complex Boolean expression. For example, the expression

```
dependents < 9 and audit = false
```

uses the Boolean operator And to combine two Boolean operands (`dependents < 9` and `audit = false`). If both expressions are `true`, the value of the entire expression is `true`. You can also use the Boolean operator Or (another binary operator; if either of its operands is `true`, the entire expression is `true`) and the Boolean operator Not (a unary operator; if its operand is `true`, the expression is `false`, and vice versa). For more information about operators, see "Expressions" (page 199).

If (Simple Statement)

A simple If statement contains one Boolean expression and a statement to be executed if the value of the Boolean expression is `true`.

SYNTAX

`if` *Boolean* `then` *statement*

where

Boolean is an expression whose value is `true` or `false`.

statement is any AppleScript statement.

EXAMPLES

In the following If statement, the Display Dialog command is executed only if the value of the Boolean expression `result > 3` is `true`.

```
if result > 3 then display dialog "The result is " & result as string
```

If (Compound Statement)

A compound If statement contains one or more Boolean expressions and groups of statements to be executed if the value of the corresponding Boolean expression is `true`.

SYNTAX

```
if Boolean [ then ]
    [ statement ]...
[ else if Boolean [ then ]
    [ statement ]...]...
[ else
    [ statement ]...]
end [ if ]
```

where

Boolean is an expression whose value is `true` or `false`.

statement is any AppleScript statement.

EXAMPLE

The following example creates a string that specifies whether one value is greater than, less than, or the same as a second value.

```
if ( x > y ) then
    set myMessage to " is greater than "
else if ( x < y ) then
    set myMessage to " is less than "
else
    set myMessage to " is equal to "
end if
set myResult to (x as string) & myMessage & (y as string)
```

Repeat Statements

Repeat statements are used to create **loops,** or groups of repeated statements, in scripts. There are several types of Repeat statements, each differing in the way it terminates the loop. For example, you can loop a specific number of times, loop for each item in a list, loop while a specified condition is true, or loop *until* a condition is true. More elaborate forms of the Repeat statement use a **loop variable,** which you can refer to in the body of the loop, to control the number of iterations.

For information on testing and debugging Repeat statements, see the example in "Repeat With (loopVariable) In (list)" (page 256).

Each available type of Repeat statement is described in one of the following sections:

- "Repeat (forever)" (page 250)
- "Repeat (number) Times" (page 251)
- "Repeat While" (page 252)
- "Repeat Until" (page 253)
- "Repeat With (loopVariable) From (startValue) To (stopValue)" (page 254)
- "Repeat With (loopVariable) In (list)" (page 256)

There may often be no clear advantage in picking one form of Repeat loop over another, so it is a good idea to pick a form that you think will make the most sense to someone reading the script. In some cases, a Filter reference form may be more appropriate than a Repeat statement. For example, the following script closes every AppleWorks window that isn't named `"Old Report (WP)"`.

```
tell application "AppleWorks"
    close every window whose name is not "Old Report (WP)"
end tell
```

You could certainly count the open windows and set up a Repeat loop that checks the name of each window and closes the window if it isn't named `"Old Report (WP)"`. In this case, however, the Filter form does most of the work for you. For more information on filters, see "Filter" (page 173) and "Using the Filter Reference Form" (page 187).

Repeat (forever)

The *Repeat (forever)* form of the Repeat statement is an **infinite loop**—a Repeat statement that does not specify when the repetition stops. The only way to exit the loop is by using an Exit statement. For more information, see "Exit" (page 258).

SYNTAX

```
repeat
    [ statement ]...
end [ repeat ]
```

where

statement is any AppleScript statement.

EXAMPLE

The following example prints each open AppleWorks document, then closes the document window. It uses an Exit statement to exit the loop.

```
tell application "AppleWorks"
    set numberOfDocuments to (count documents)
    set i to 1
    repeat
        if i > numberOfDocuments then
            exit repeat
        end if
        print front document without one copy -- display Print dialog
        close front document saving ask -- ask before saving modified doc
        set i to i + 1
    end repeat
end tell
```

The phrase `without one copy` tells an AppleWorks document to display the Print dialog before printing.

Repeat (number) Times

The *Repeat (number) Times* form of the Repeat statement repeats a group of statements a specified number of times.

SYNTAX

```
repeat integer [ times ]
    [ statement ]...
end [ repeat ]
```

where

integer is an integer that specifies the number of times to repeat the statements in the body of the loop. The word `times` after *integer* is optional.

statement is any AppleScript statement.

EXAMPLE

The following example numbers the paragraphs of a document with the *Repeat (number) Times* form of the Repeat statement.

```
tell document "Simple" of application "AppleWorks"
    set numParagraphs to (count paragraphs of text body)
    set paragraphNum to 1
    repeat numParagraphs times
        set paragraph paragraphNum of text body ¬
            to (paragraphNum as string) ¬
            & " " & paragraph paragraphNum of text body
        set paragraphNum to paragraphNum + 1
    end repeat
end tell
```

Repeat While

The *Repeat While* form of the Repeat statement repeats a group of statements as long as a particular condition, specified in a Boolean expression, is met.

SYNTAX

```
repeat while Boolean
    [ statement ]...
end [ repeat ]
```

where

Boolean is an expression whose value is `true` or `false`. The statements in the loop are repeated until *Boolean* becomes `false`. If *Boolean* is `false` when entering the loop, the statements in the loop are not executed.

statement is any AppleScript statement.

EXAMPLE

The following example numbers the paragraphs of a document with the *Repeat While* form of the Repeat statement. To type the greater than or equal character (≥), press Option–greater-than sign.

```
tell document "Simple" of application "AppleWorks"
    set numParagraphs to (count paragraphs of text body)
    set paragraphNum to 1
    repeat while paragraphNum ≤ numParagraphs
        set paragraph paragraphNum of text body ¬
            to (paragraphNum as string) & " " ¬
            & paragraph paragraphNum of text body
        set paragraphNum to paragraphNum + 1
    end repeat
end tell
```

Repeat Until

The *Repeat Until* form of the Repeat statement repeats a group of statements until a particular condition, specified in a Boolean expression, is met.

SYNTAX

```
repeat until Boolean
    [ statement ]...
end [ repeat ]
```

where

Boolean is an expression whose value is `true` or `false`. The statements in the loop are repeated until *Boolean* becomes `true`. If *Boolean* is `true` when entering the loop, the statements in the loop are not executed.

statement is any AppleScript statement.

EXAMPLE

The following example is based on Figure 2-4 (page 28), which shows a script that copies information from a database to a spreadsheet. The script uses the *Repeat Until* form of the Repeat statement, copying an inventory count from each database record to a spreadsheet cell until a loop variable exceeds the number of records. The script assumes the database and spreadsheet files are already open.

```
tell application "FileMaker Pro"
    set numberRecords to count records of document "Inventory DB"
    set loopCount to 1
    repeat until loopCount > numberRecords
        copy cell "Count" of record loopCount ¬
            of document "Inventory DB" to partCount
        set cellName to "B" & ((loopCount + 1) as string)
        tell application "AppleWorks"
            set cell cellName of spreadsheet of document ¬
                "Spreadsheet" to partCount as string
        end tell
        set loopCount to loopCount + 1
    end repeat
end tell
```

Repeat With (loopVariable) From (startValue) To (stopValue)

In the *Repeat With (loopVariable) From (startValue) To (stopValue)* form of the Repeat statement, the loop variable is an integer that is increased by a specified value after each iteration of the loop. The loop terminates when the value of the variable is greater than a predefined stop value.

SYNTAX

```
repeat with loopVariable from startValue to stopValue [ by stepValue ]
    [ statement ]...
end [ repeat ]
```

where

loopVariable is used to control the number of iterations. It can be any previously defined variable or a new variable you define in the Repeat statement (see "Notes").

startValue (an integer) is the value assigned to *loopVariable* when the loop is entered.

stopValue (an integer) is the value of *loopVariable* at which iteration ends. Iteration continues until the value of *loopVariable* is greater than the value of *stopValue*.

stepValue (an integer) is the value added to *loopVariable* after each iteration of the loop. The default value of *stepValue* is 1.

statement is any AppleScript statement.

EXAMPLE

The following example numbers the paragraphs of a document with the *Repeat With (loopVariable) From (startValue) To (stopValue)* form of the Repeat statement.

```
tell document "Simple" of application "AppleWorks"
    set numParagraphs to (count paragraphs of text body)
    repeat with n from 1 to numParagraphs
        set paragraph n of text body to (n as string) ¬
            & " " & paragraph n of text body
    end repeat
end tell
```

NOTES

You can use an existing variable as the loop variable in a Repeat statement or define a new one in the Repeat statement. In either case, the loop variable is defined outside the loop. You can change the value of the loop variable inside

the loop body but it will get reset to the next loop value the next time through the loop. After the loop completes, the loop variable retains its last value.

AppleScript evaluates *startValue*, *stopValue*, and *stepValue* when it begins executing the loop and stores the values internally. If you change the values in the body of the loop, it has no effect on the execution of the loop.

Repeat With (loopVariable) In (list)

In the *Repeat With (loopVariable) In (list)* form of the Repeat statement, the loop variable is a reference to an item in a list. The number of iterations is equal to the number of items in the list. In the first iteration, the value of the variable is `item 1 of` *list* (where *list* is the list you specified in the first line of the statement), in the second iteration, its value is `item 2 of` *list*, and so on.

SYNTAX

```
repeat with loopVariable in list
    [ statement ]...
end [ repeat ]
```

where

loopVariable is any previously defined variable or a new variable you define in the Repeat statement (see "Notes").

list is a list or a reference (such as `words 1 thru 5`) whose value is a list.

list can also be a record; AppleScript coerces the record to a list (see "Notes").

statement is any AppleScript statement.

EXAMPLE

The following script examines a list of words with the *Repeat With (loopVariable) In (list)* form of the Repeat statement, displaying a dialog if it finds the word "hammer" in the list.

```
set wordList to words in "Where is the hammer?"
repeat with currentWord in wordList
    --log currentWord
    if currentWord as text is equal to "hammer" then
        display dialog "I found the hammer!"
    end if
end repeat
```

For a description of the commented-out statement `--log currentWord`, see "Debugging Control Statements" (page 239).

NOTES

You can use an existing variable as the loop variable in a Repeat statement or define a new one in the Repeat statement. In either case, the loop variable is defined outside the loop. You can change the value of the loop variable inside the loop body but it will get reset to the next loop value the next time through the loop. After the loop completes, the loop variable retains its last value.

AppleScript evaluates *loopVariable* `in` *list* as `item 1 of` *list*, `item 2 of` *list*, `item 3 of` *list*, and so on until it reaches the last item in the list, as shown in the following example.

```
repeat with i in {1, 2, 3, 4}
    set x to i
end repeat
--result: item 4 of {1, 2, 3, 4}
```

To set a variable to the value of an item in the list, rather than a reference to the item, you can use the `contents of` operator:

```
repeat with i in {1, 2, 3, 4}
    set x to contents of i
end repeat
--result: 4
```

You can also use the list items directly in expressions:

```
set x to 0
repeat with i in {1, 2, 3, 4}
    set x to x + i
end repeat
--result: 10
```

If the value of *list* is a record, AppleScript coerces the record to a list by stripping the property labels. For example, `{a:1, b:2, c:3}` becomes `{1, 2, 3}`.

Exit

An **Exit statement** is used in a Repeat statement to exit the Repeat statement. When AppleScript executes an Exit statement, it terminates loop execution and resumes execution with the next statement following the Repeat statement. You cannot use Exit statements outside of Repeat statements.

SYNTAX

```
exit
```

EXAMPLE

For more information on the following script, see the example in "Repeat (forever)" (page 250).

```
tell application "AppleWorks"
    set numberOfDocuments to (count documents)
    set i to 1
    repeat
        if i > numberOfDocuments then
            exit repeat
        end if
        print document i one copy false -- display Print dialog
        close document i saving ask -- ask before saving modified doc
```

```
        set i to i + 1
    end repeat
end tell
```

Try Statements

Scripts don't always work perfectly. When a script is executed, errors can occur in the operating system (for example, when a specified file isn't found), in an application (for example, when you specify an object that doesn't exist), and in the script itself. When an error occurs, AppleScript sends a special message known as an error message. An **error message** is a message that is returned by an application, AppleScript, or the Mac OS when an error occurs during the handling of a command. An error message can include an **error number,** which is an integer that identifies the error, an **error expression,** which is an expression, usually a string, that describes the error, and other information.

A script can include one or more collections of statements called **error handlers** to handle error messages. Error handlers are contained in compound statements, called Try statements, that define the scope of the error handlers they contain. A **Try statement** is a two-part compound statement that contains a series of AppleScript statements, followed by an error handler to be invoked if any of those statements causes an error. If an error message occurs and there is no handler for it, script execution stops.

For information on other types of handlers, see "Handlers" (page 279).

Kinds of Errors

Every script error falls into one of the following categories:

- Operating system errors are errors that occur when AppleScript or an application requests services from the Mac OS. They are rare, and, more importantly, there's usually nothing you can do about them in a script. A few, such as `"File <name> wasn't found"` and `"Application isn't running"`, make sense for scripts to handle. These errors are listed in "Operating System Errors" (page 384).
- Apple event errors are operating system errors that occur when the underlying message system for AppleScript—known as Apple events—fails.

Many of these errors, such as `"No user interaction allowed"`, are of interest to scripters. Also of interest to users are errors that have to do with reference forms, as well as errors such as `"No such object"`. These errors are listed in "Apple Event Errors" (page 385).

- An application scripting error is an error returned by an application when handling standard AppleScript commands (commands that apply to all applications). Many of these, such as `"The specified object is a property, not an element"`, are of interest to users and should be handled. These errors are listed in "Application Scripting Errors" (page 387). Applications can define additional error messages for the AppleScript commands they handle. An application that defines such errors should list them in its documentation.
- AppleScript errors are errors that occur when AppleScript processes script statements. Nearly all of these are of interest to scripters. These errors are listed in "AppleScript Errors" (page 388).
- Script errors are error messages sent by a script using the "Error" (page 264) command. Scripts that define additional errors will often include descriptions of the errors in their documentation.

Note
Some "errors" are the result of the normal operation of a command. For example, the Choose File command returns error –128, User canceled, if the user presses the Cancel button in the resulting dialog box. Scripts must routinely handle such errors to ensure normal operation. For a script that handles this error, see the Examples section in "Try" (page 261). ◆

How Errors Are Handled

When an error occurs, AppleScript checks to see if the statement that caused the error is contained in a Try statement. A Try statement is a two-part compound statement that contains a series of AppleScript statements, followed by an error handler to be invoked if any of those statements causes an error. If the statement that caused the error is included in a Try statement, then AppleScript passes control to the error handler in the Try statement. After the error handler completes, control passes to the statement immediately following the end of the Try statement.

If the error occurs within a subroutine and AppleScript does not find a Try statement in that subroutine, AppleScript checks to see if the statement that invoked the current subroutine is contained in a Try statement. If that statement is not contained in a Try statement, AppleScript continues up the call chain, going to the statement that invoked that subroutine, if any, and so on. If none of the calls in the call chain is contained in a Try statement, AppleScript stops execution of the script.

Writing a Try Statement

A Try statement is two-part compound statement. The first part, which begins with the word `try`, is a collection of AppleScript statements. The second part, which begins with the words `on error`, is an error handler—a series of statements that is executed if any of the statements in the first part causes an error message. The Try statement ends with the word `end` (followed optionally by `error` or `try`).

The error handler can include up to five **parameter variables** (also called **formal parameters**) that represent the actual information sent in the error message when the error occurs. When the error handler is called, the parameter variables become local variables in the error handler.

Try

A Try statement is a compound statement consisting of a list of AppleScript statements followed by an error handler to be executed if any of the statements cause an error message.

SYNTAX

```
try
    [ statement ]...
on error                                                ¬
        [ errorMessageVariable ]                        ¬
        [ number errorNumberVariable ]                  ¬
        [ from offendingObjectVariable ]                ¬
        [ partial result resultListVariable ]           ¬
```

```
        [ to expectedTypeVariable ]
    [ global variable [, variable ]...]
    [ local variable [, variable ]...]
    [ statement ]...
end [ error | try ]
```

where

statement is any AppleScript statement.

errorMessageVariable (an identifier) You use this parameter variable within the error handler to refer to the error expression, usually a string, that describes the error.

errorNumberVariable (an identifier) You use this parameter variable within the error handler to refer to the error number, an integer.

offendingObjectVariable (an identifier) You use this parameter variable within the error handler as a reference to the application object that caused the error.

resultListVariable (an identifier) You use this parameter variable within the error handler to refer to partial results for objects that were handled before the error occurred. Its value is a list that can contain values of any class. This parameter applies only to commands that return results for multiple objects. For example, if an application handles the command `get words 1 thru 5` and an error occurs when handling word 4, the `partial result` parameter contains the results for the first three words.

expectedTypeVariable (an identifier) This parameter variable identifies the expected value class (a class identifier)—that is, the value class to which AppleScript was attempting to coerce the value of *offendingObjectVariable*. If an application receives data of the wrong class and cannot coerce it to the correct class, the value of this parameter variable is the class of the coercion that failed. (The example at the end of this definition demonstrates how this works.)

variable (an identifier) This parameter variable is either a global variable or a local variable that can be used in the handler. The scope of a local variable is the handler. You cannot refer to a local variable outside the handler. The scope of a global variable can extend to any other part of the script, including other handlers and script objects. For detailed information about the scope of local and global variables, see “Scope of Script Variables and Properties” (page 311).

EXAMPLES

The following Try statement provides an error handler for the Choose File command, one of the scripting addition commands distributed with AppleScript. The Choose File command returns an error if the user clicks the Cancel button in the Choose File dialog box. If the user doesn't choose a file, the error handler asks whether to continue, using a default file. If the user chooses to continue, a second dialog confirms the choice and displays the name of the default file.

```
set fileName to "Generic Prefs" -- Use if no filename chosen.
try
    choose file -- Ask user to choose a file.
    -- If the user cancels, the next statement won't be executed.
    set fileName to result
on error errText number errNum
    if (errNum is equal to -128) then -- User cancelled.
        display dialog "No file chosen. Use the default file?"
        if button returned of result is equal to "OK" then
            display dialog "The script will continue " & ¬
                "using the default file \"" & fileName & "\"."
        end if
    else
        -- If any other error, do nothing.
    end if
end try
```

The next example demonstrates the use of the To keyword to capture additional information about an error that occurs during a coercion failure.

```
try
    repeat with i from 1 to "Toronto"
        i
    end repeat
on error from obj to newClass
    log {obj, newClass} -- Display from and to info in log window.
end try
```

The Repeat statement fails because the string `"Toronto"` is the wrong class—it's a string, not an integer. The error handler simply writes the values of `obj` (the offending value, `"Toronto"`) and `newClass` (the class of the coercion that failed, `integer`) to the Script Editor's Event Log window. The result is "(*Toronto,

integer*)", indicating the error occurred while trying to coerce "Toronto" to an integer. You can open the Script Editor's Event Log window from the Controls menu or by typing Command-E. For more information on using the Event Log window, see "Debugging Scripts" (page 47) and the AppleScript section of the Mac OS Help Center.

Signaling Errors in Scripts

A script can signal an error—which can then be handled by an error handler—with the Error command. This allows scripts to define their own messages for errors that occur within the script.

Error

The Error command signals an error in a script.

SYNTAX

```
error                                   ¬
    [ errorMessage ]                    ¬
    [ number errorNumber ]              ¬
    [ from offendingObject ]            ¬
    [ partial result resultList ]       ¬
    [ to expectedType ]
```

where

errorMessage is a string describing the error. Although this parameter is not required, you should provide descriptive expressions for errors wherever possible, and you should always provide an expression if you do not include a `number` parameter. If you do not include an error expression, an empty string (`""`) is passed to the error handler.

errorNumber is the error number for the error. You do not have to include an error number, but if you do, the number must not be any of the error numbers listed in "Error Numbers and Error Messages" (page 384). In general, positive numbers from 500 to 10,000 do not conflict with error numbers for AppleScript,

the Mac OS, or Apple events. If you do not include a `number` parameter, the value `-2700` (unknown error) is passed to the error handler.

offendingObject is a reference to the object, if any, that caused the error. If you provide a partial reference, AppleScript completes it using the value of the default object (for example, the target of a Tell statement).

resultList applies only to commands that return results for multiple objects. If results for some, but not all, of the objects specified in the command are available, you can include them in the `partial result` parameter. If you do not include a `partial result` parameter, an empty list `({})` is passed to the error handler.

expectedType is a class identifier. If a parameter specified in the command was not of the expected class, and AppleScript was unable to coerce it to the expected class, then you can include the expected class in the `to` parameter.

EXAMPLES

The following example uses the Error command to resignal an error. The error handler resignals the error exactly as it was received, causing AppleScript to display an error dialog.

```
try
    word 5 of "one two three"
on error number errNum from badObj
    -- statements that handle the error
    error number errNum from badObj
end try
```

In the next example, an Error command in an error handler resignals the error, but changes the message that appears in the error dialog. It also changes the error number to `600`.

```
try
    word 5 of "one two three"
on error
    -- statements that determine the cause of the error
    error "There are not enough words." number 600
end try
```

The following example shows how to signal an error to handle bad data, and how to provide a handler that deals with other errors. The `SumIntegersInList` subroutine expects a list of integers. If any item in the passed list is not an integer, `SumIntegersInList` signals error number 750 and returns 0. The subroutine includes an error handler that displays a dialog if the error number is equal to 750; otherwise it uses the Error command to resignal the error. That allows other statements in the call chain to handle the unknown error number. If no statement handles the error, AppleScript displays an error dialog and execution stops.

```
on SumIntegerList from itemList
    try
        -- Initialize return value.
        set integerSum to 0
        -- Before doing sum, check that all items in list are integers.
        if ((count items in itemList) is not equal to ¬
            (count integers in itemList)) then
            -- If all items aren't integers, signal an error.
            error number 750
        end if
        -- Use a Repeat statement to sum the integers in the list.
        repeat with currentItem in itemList
            set integerSum to integerSum + currentItem as integer
        end repeat
        return integerSum -- Successful completion of subroutine.
    on error errStr number errorNumber
        -- If our own error number, warn about bad data.
        if the errorNumber is equal to 750 then
            display dialog "All items in the list must be integers."
            return integerSum -- Return the default value (0).
        else
            -- An unknown error occurred. Resignal, so the caller
            -- can handle it, or AppleScript can display the number.
            error number errorNumber
        end if
    end try
end SumIntegerList
```

Let's look at how the `SumIntegersInList` subroutine handles various error conditions.

A Successful Call

This first call completes without error.

```
set sumList to {1, 3, 5}
set listTotal to SumIntegerList from sumList --result: 9
```

A Call With Bad Data

The following call passes bad data—the list contains an item that isn't an integer.

```
set sumList to {1, 3, 5, "A"}
set listTotal to SumIntegerList from sumList
if listTotal is equal to 0 then
-- The subroutine didn't total the list--do something
-- to handle the error. (Not shown.)
```

The `SumIntegerList` routine checks the list and signals an error 750 because the list contains at least one non-integer item. The routine's error handler recognizes error number 750 and puts up a dialog to describe the problem. The `SumIntegerList` routine returns 0. The script checks the return value and, if it is equal to 0, does something to handle the error.

An Unknown Error Occurs and Is Not Handled By the Caller

Suppose some unknown error occurs while `SumIntegerList` is processing the integer list in the previous call. When the unknown error occurs, the `SumIntegerList` error handler calls the Error command to resignal the error. Since the caller doesn't handle it, AppleScript displays an error dialog and execution halts. The `SumIntegerList` routine does not return a value.

An Unknown Error Is Handled By the Caller

Finally, suppose the caller has its own error handler, so that if the subroutine passes on an error, the caller can handle it. Assume again that an unknown error occurs while `SumIntegerList` is processing the integer list.

```
try
    set sumList to {1, 3, 5}
    set listTotal to SumIntegerList from sumList
on error errMsg number errorNumber
```

```
    display dialog "An unknown error occurred:  " ¬
        & errorNumber as string
end try
```

In this case, when the unknown error occurs, the `SumIntegerList` error handler calls the Error command to resignal the error. Because the caller has an error handler, it is able to handle the error by displaying a dialog that includes the error number. Execution can continue if it is meaningful to do so.

Considering and Ignoring Statements

Considering statements allow you to control the way AppleScript executes operations and commands by listing specific characteristics, called *attributes*, to be taken into account as the operations and commands are executed. **Ignoring statements** work the same way, except that you list specific attributes to be ignored.

The attributes you can use include

- case, white space, and others that affect string comparisons
- an attribute called `application responses` that controls whether or not AppleScript waits for responses from commands sent to applications

The following is an example of a string comparison without a Considering statement:.

```
"This" = "this"
--result: true
```

The value of the string comparison is `true`, because by default, AppleScript does not distinguish uppercase from lowercase letters.

Here's the same comparison within a Considering statement:

```
considering case
    "This" = "this"
end considering
--result: false
```

The Considering statement specifies that a particular attribute of strings—their case—is to be used in comparisons. As a result the comparison `"This" = "this"` is now `false`, because the uppercase "T" in `"This"` does not match the lowercase "t" in `"this"`.

Considering/Ignoring

Considering and Ignoring statements cause AppleScript to consider or ignore specific characteristics, called attributes, as it executes groups of statements.

SYNTAX

```
considering attribute [, attribute ... and attribute ]                    ¬
        [ but ignoring attribute [, attribute ... and attribute ] ]
    [ statement ]...
end considering

ignoring attribute [, attribute ... and attribute ]                       ¬
        [ but considering attribute [, attribute ... and attribute ] ]
    [ statement ]...
end ignoring
```

where

statement is any AppleScript statement.

attribute is an attribute to be considered or ignored. Attributes are listed next under "Attributes".

ATTRIBUTES

An **attribute** is a characteristic that can be considered or ignored in a Considering or Ignoring statement. A Considering or Ignoring statement can include any of the following attributes:

`application responses`: AppleScript waits for a response from each application command before proceeding to the next statement or operation. The response indicates if the command completed successfully, and also returns results and error messages, if there are any. If this attribute is ignored,

AppleScript does not wait for responses from application commands before proceeding to the next statement, and ignores any results or error messages that are returned. Results and error messages from AppleScript commands, scripting additions, and expressions are not affected by the `application responses` attribute.

`case`: In string comparisons, uppercase letters are not distinguished from lowercase letters. If this attribute is considered, uppercase letters are distinguished from lowercase letters. See "Greater Than, Less Than" (page 224) for a description of how AppleScript sorts letters, punctuation, and other symbols.

`diacriticals`: Diacritical marks (such as ´, `, ^, ¨, and ˜) are considered in string comparisons. If this attribute is ignored, `"résumé"` is considered equal to `"resume"`, and so on. See "Greater Than, Less Than" (page 224) for a description of how AppleScript sorts letters with diacritical marks.

`expansion`: In string comparisons, AppleScript treats the characters æ, Æ, œ, and Œ as identical to the character pairs ae, AE, oe, and OE, respectively. If this attribute is ignored, AppleScript treats these characters like single characters; for example æ would be considered not equal to the character pair ae.

`hyphens`: In string comparisons, hyphenated words are considered different from their nonhyphenated counterparts. If this attribute is ignored, the strings are compared as if any hyphens were not present; for example `"anti-war"` would be considered equal to `"antiwar"`.

`punctuation`: The punctuation marks (. , ? : ; ! \ ' " `) are considered in string comparisons. If this attribute is ignored, the strings are compared as if these punctuation marks were not present; for example `"This!"` would be considered equal to `"This"`.

`white space`: Spaces, tab characters, and return characters are considered in string comparisons. If this attribute is ignored, the strings are compared as if these characters were not present; for example `"Brick house"` would be considered equal to `"Brickhouse"`.

EXAMPLES

```
"BOB" comes before "bob" --result: false
considering case
    "BOB" comes before "bob" --result: true
end considering
```

```
"a" comes before "á" --result: true
ignoring diacriticals
    "a" comes before "á" --result: false
end considering

"Babs" comes before "bábs" --result: true
ignoring case
    "Babs" comes before "bábs" --result: true
end ignoring
ignoring case and diacriticals
    "Babs" comes before "bábs" --result: false
end ignoring

ignoring punctuation
    "this !,:book" = "this book" --result: true
end ignoring
--result: Dialog is displayed.
```

NOTES

The `case`, `white space`, `diacriticals`, `hyphens`, `expansion`, and `punctuation` considerations apply only to comparisons performed by AppleScript. Comparisons are performed by AppleScript if the first operand in the comparison is a value in a script; if the first operand is a reference to an application or system object, the comparison is performed by the application or operating system.

In contrast, the `application responses` consideration applies only to application commands. AppleScript commands, scripting additions, and AppleScript expressions are not affected.

As with all other control statements, you can nest Considering and Ignoring statements. If the same attribute appears in both an outer and inner statement, the attribute specified in the inner statement takes precedence. For example, in the following statement, the first comparison is `true`, because case attribute is ignored (as specified in the Ignoring statement), while the second comparison is `false`, because the case attribute is once again considered (as specified in the inner Considering statement).

```
ignoring case and punctuation
    if "This" = "this" then beep 1 --true
    considering case
        if "This" = "this" then beep 2 --false
    end considering
end considering
--result: Beeps once.
```

When attributes in an inner Considering or Ignoring statement are different from those in outer statements, they are added to the attributes to be considered and ignored. For example, in the following statement, the first comparison is `false`, because only case is ignored, while the second comparison is `true`, because both case and white space are ignored.

```
ignoring case
    if "This or that" = "thisorthat" then beep 2 --false
    ignoring white space
        if "This or that" = "thisorthat" then beep 1 --true
    end ignoring
end ignoring
--result: Beeps once.
```

With Timeout Statements

When AppleScript sends a command to an application, it normally waits for the command to complete execution before continuing with the rest of the script. If the command takes longer than one minute to complete, AppleScript stops running the script and returns the error `"event timed out"`. AppleScript does not cancel the operation—it merely stops execution of the script.

A **With Timeout statement** lets you change how long AppleScript waits before stopping execution of a script. The amount of time you specify in a With Timeout statement applies to some types of commands within the statement that are sent to other applications, but not to any commands sent to the application that's running the script.

As with AppleScript's default timeout of one minute, when a With Timeout statement times out, AppleScript does not cancel the operation—it merely stops execution of the script. In addition, AppleScript can only check for a timeout if

the application that gets the command yields time to the script. If an application is nonresponsive (clicking on the screen produces no result), the timeout may not be checked. For example, the following statement will not time out if the user fails to dismiss the modal Save dialog:

```
with timeout of 5 seconds
    tell application "AppleWorks"
        close front document saving ask
    end tell
end timeout
```

Your script can use a With Timeout statement in conjunction with a Try statement so that it has the opportunity to deal with a timeout. However, whether your script can send a command to cancel the offending lengthy operation after a timeout is dependent on the application that is performing the command.

The time specified by a With Timeout statement applies to all application commands and to any scripting addition commands whose targets are application objects, which includes scripting addition commands whose direct parameters are application objects and scripting addition commands within Tell statements to application objects. The time specified by a With Timeout statement does not apply to AppleScript commands, AppleScript operations, or scripting addition commands whose targets are not application objects.

Note
If you want AppleScript to proceed to the next statement without waiting for application commands to complete, use an Ignoring statement to ignore the `application responses` attribute. For more information, see "Considering and Ignoring Statements" (page 268). ◆

With Timeout

By default, AppleScript waits one minute for a response before stopping execution of application and scripting addition commands that are sent to other applications. A With Timeout statement lets you change how long AppleScript waits.

SYNTAX

```
with timeout [ of ] integer second[s]
    [ statement ]...
end [ timeout ]
```

where

integer is an integer that specifies the amount of time, in seconds, AppleScript allows for each application command or command addition contained in the With Timeout statement that is sent to any application other than the current one.

statement is any AppleScript statement.

EXAMPLE

The following script starts the Finder on a task that may take a long time to complete. First it creates 40 folders, then it opens them, then it starts to close them, using a With Timeout statement to interrupt script execution one second after starting the Close operation. If the With Timeout statement generates an error, the error section of the Try statement calls the Beep scripting addition command and also writes "beep" to the Script Editor's Event Log window.

```
tell application "Finder"
    repeat 40 times
        make new folder at startup disk
    end repeat
    open (every folder of startup disk whose name contains "untitled")
    try
        with timeout of 1 second
            close (every folder of startup disk ¬
                whose name contains "untitled")
        end timeout
    on error
        --Just beep and notify Script Editor's Event Log window.
        beep
        log ("beep")
    end try
end tell
```

With Transaction Statements

Some applications, such as databases, support the notion of a transaction—that is, a sequence of related events that should be performed as if they were a single operation. The **With Transaction statement** allows you to specify transactions for such applications.

At the beginning of a With Transaction statement, AppleScript requests a transaction ID from the target application (established by an enclosing Tell statement) and attaches that transaction ID to every Apple event it sends to the target application as a result of executing commands in the body of the With Transaction statement.

Whenever AppleScript exits a With Transaction statement, it informs the application that the transaction is over, even if the exit occurs before the end of the statement because of an error. Thus, if an error occurs within the body of the With Transaction statement but is not handled within the statement, AppleScript exits the statement, the application is informed that the transaction is over, and the error continues through subsequent statements until it is handled.

With Transaction

A With Transaction statement causes AppleScript to associate a single transaction ID with any events it sends to a target application as a result of executing commands in the body of the With Transaction statement.

SYNTAX

```
with transaction [ session ]
    [ statement ]...
end [ transaction ]
```

where

session is an object that specifies a specific session.

statement is any AppleScript statement.

EXAMPLES

This example uses a With Transaction statement to ensure that a record can be modified by one user without being modified by another user at the same time.

```
tell application "Small DB"
    with transaction
        set oldName to Field "Name"
        set oldAddress to Field "Address"
        set newName to display dialog ¬
            "Please type a new name" ¬
            default answer oldName
        set newAddress to display dialog ¬
            "Please type the new address" ¬
            default answer oldAddress
        set Field "Name" to newName
        set Field "Address" to newAddress
    end transaction
end tell
```

The Set statements obtain the current values of the Name and Address fields and invite the user to change them. Enclosing these Set statements in a single With Transaction statement informs the application that other users should not be allowed to access the same record at the same time.

With Transaction statements only work with applications that explicitly support them. Some applications only support With Transaction statements (like the one in the previous example) that do not take a session object as a parameter. Other applications support both With Transaction statements that have no parameter and With Transaction statements that take a session parameter.

The following example demonstrates how to specify a session for a With Transaction statement.

```
tell application "Super DB"
    set mySession to make session with ¬
        data {user: "Bob", password: "Secret"}
    with transaction mySession
```

```
        ...
    end transaction
end tell
```

Handlers

A **handler** is a collection of statements that AppleScript executes in response to a command or error message.

This chapter describes handlers in the following sections:

- "Script Applications" (page 279) describes script applications and how to create them. Sections throughout this chapter refer to script applications.
- "About Subroutines" (page 280) describes subroutines, which are handlers for user-defined commands, and provides a general overview of how they work.
- "Defining and Calling Subroutines" (page 289) provides a detailed description of how you work with subroutines, including examples that demonstrate labeled and positional parameters.
- "Command Handlers" (page 300) describes routines you write to handle application or system commands.
- "Scope of Script Variables and Properties" (page 311) describes the scope, or range, over which AppleScript recognizes a declared identifier within a script.

For information on writing error handlers, see "Try Statements" (page 259).

Script Applications

Several sections in this chapter refer to script applications. A **script application** is an application whose only function is to run the script associated with it. You can run a script application from the Finder much like any other application.

You can use the Script Editor's Save As command to save a script as a script application. To do so, you specify Application in the pop-up menu on the Script

Editor's Save As dialog box. By default, a startup screen appears before the script runs. The startup screen displays the description of the script you write in the top part of the Script Editor window. The user must click the startup screen's Run button or press the Return key before the Finder actually sends the Run command. This allows the user to read the description of the script before running it. If you check the Never Show Startup Screen checkbox in the Save As dialog box, the script runs without displaying the startup screen.

You can also use a checkbox on the Script Editor's Save As dialog box to specify that the script application should stay open after running. The default is for the script to quit right after you run it. For more information on working with the Script Editor, see the AppleScript section of the Mac OS Help Center. If you are using a different script editor, see the documentation that came with it.

For related information, see "Handlers for Stay-Open Script Applications" (page 306).

About Subroutines

A **subroutine** is a collection of statements that AppleScript runs in response to a user-defined command. Subroutines are similar to functions, methods, and procedures in other programming languages.

Subroutines are useful in scripts that perform the same action in more than one place. For example, if you have a series of statements for comparing values and you need to use those statements at several places in a script, you can package the statements as a subroutine and call it from anywhere in the script. Your script becomes shorter and easier to maintain. In addition, you can give subroutines descriptive names that make their purposes clear and make scripts easy to read.

The following sections describe how to write and call subroutines:

- "Recursive Subroutines" (page 286)
- "Saving and Loading Libraries of Subroutines" (page 287)

The Return Statement

A Return statement allows you to stop execution of a handler before all its statements are executed and to return a value. Many of the examples in this chapter use Return statements.

A Return statement exits a handler and returns a value. When AppleScript executes a Return statement, it stops handler execution and resumes execution at the place in the script where the handler was called, using the value returned as the value of the handler.

SYNTAX

```
return expression
```

where

expression is an AppleScript expression. When AppleScript executes a Return statement, it returns the value of the expression. For related information, see "Expressions" (page 199).

EXAMPLE

To return a value and exit a subroutine, include a Return statement in the body of the subroutine. For example, the following statement returns the integer `2`:

```
return 2
```

If you include a Return statement without an expression, AppleScript exits the subroutine immediately and no value is returned.

NOTES

If a subroutine does not include a Return statement, AppleScript executes the statements in the subroutine and, after handling the last statement, returns the

value of the last statement in the subroutine. If the last statement does not return a value, then no value is returned.

When AppleScript has finished executing a subroutine (that is, when it executes a Return statement or the last statement in the subroutine), it passes control to the place in the script immediately after the place where the subroutine was called.

In general, it is best to have just one Return statement and locate it at the end of a subroutine or handler. For most scripts, doing so provides the following benefits:

- The script is easier to understand.
- The script is easier to debug.
- You can place cleanup code in one place and make sure it is executed.

In some cases, however, it may make more sense to use multiple Return statements. For example, the `minimumValue` subroutine in "A Sample Subroutine" (page 282) is a simple script that uses two return statements.

A Sample Subroutine

Here's a subroutine, called `minimumValue`, that returns the smaller of two values:

```
-- minimumValue subroutine:
on minimumValue(x, y)
    if x < y then
        return x
    else
        return y
    end if
end minimumValue

-- To call minimumValue:
minimumValue(5, 105)
```

The first line of the `minimumValue` subroutine specifies the parameters of the subroutine. These can be positional parameters—like `x` and `y` in the example—where the order of the parameters is significant, or labeled parameters—like those for many of the AppleScript and application commands described in

"Commands" (page 109)—where the order of parameters other than the direct parameter doesn't matter.

The `minimumValue` subroutine includes two Return statements. A Return statement is one of the ways a subroutine can return a result. When AppleScript executes a Return statement, it returns the value (if any) listed in the statement and immediately exits the subroutine. If AppleScript executes a Return statement without a value, it exits the subroutine immediately and does not return a value.

If a subroutine does not include any Return statement, AppleScript executes the statements in the subroutine and, after handling the last statement, returns the value of the last statement in the subroutine. If the last statement does not return a value, then the subroutine does not return a value.

When AppleScript has finished executing a subroutine, it passes control to the place in the script immediately after the place where the subroutine was called. If a subroutine call is part of an expression, AppleScript uses the value returned by the subroutine to evaluate the expression. For example, to evaluate the following expression, AppleScript calls the subroutine for `minimumValue`, then evaluates the rest of the expression.

```
minimumValue(5, 105) + 50 --result: 55
```

For related information, see "Using Results" (page 121) and "The Return Statement" (page 281).

Types of Subroutines

There are two types of subroutines: those with labeled parameters and those with positional parameters.

- **Labeled parameters** are identified by their labels and can be listed in any order. Subroutines with labeled parameters can also have a direct parameter. The direct parameter, if present, must be listed first.
- **Positional parameters** must be listed in a specific order, which is defined in the subroutine definition.

For example, the following statement calls a subroutine with positional parameters.

```
minimumValue(150, 4000)
```

The following statement calls a subroutine with a labeled parameter. The `searchFiles` routine is defined in "Examples of Subroutines With Labeled Parameters" (page 293). The direct parameter is the list of filenames. The string to search for, `"LeChateau"`, is the labeled parameter.

```
searchFiles of {"March Expenses", "April Expenses", ¬
    "May Expenses", "June Expenses"} for "LeChateau"
```

The definition for a subroutine determines what kind of parameters the subroutine requires. When you call a subroutine, you must list its parameters in the same way they are specified in the subroutine definition.

You can also have subroutines with no parameters. To indicate that a subroutine has no parameters, you must include a pair of empty parentheses after the subroutine name in both the subroutine definition and the subroutine call. For example, the following script shows the definition and subroutine call for a subroutine called `helloWorld` that has no parameters.

```
on helloWorld()
    display dialog "Hello World"
end

helloWorld()
```

Scope of Subroutine Calls in Tell Statements

If you need to call a subroutine from within a Tell statement, you must use the reserved words `of me` or `my` to indicate that the subroutine is part of the script (not a command that should be sent to the object of the Tell statement).

For example, the `minimumValue` subroutine call in the following Tell statement is unsuccessful, even if the script contains the `minimumValue` routine defined in "A Sample Subroutine" (page 282), because AppleScript sends the `minimumValue` command to AppleWorks. If you run this script, you get an error message saying that AppleWorks does not understand the `minimumValue` message.

```
tell front document of application "AppleWorks"
    minimumValue(12, 400)
    copy result as string to word 10 of text body
```

```
end tell
--result: An error, because AppleWorks doesn't
--          understand the minimumValue message.
```

If you use the words `of me` in the subroutine call, as shown in the following Tell statement, the subroutine call is successful, because AppleScript knows that the subroutine is part of the script.

```
tell front document of application "AppleWorks"
    minimumValue(12, 400) of me
    copy result as string to word 10 of text body
end tell
--result: The subroutine call is successful.
```

You can use the word `my` before the subroutine call as a synonym for the words `of me` after the subroutine call. For example, the following two subroutine calls are equivalent:

```
minimumValue(12, 400) of me
my minimumValue(12, 400)
```

Checking the Classes of Subroutine Parameters

You cannot specify the class of a parameter in a subroutine definition. You can, however, get the value of the Class property of a parameter and check it to see if the parameter belongs to the correct class. If it doesn't, you may be able to coerce it with the As operator, or failing that, you can return an error. For information about coercing values, see "Expressions" (page 199). For information about returning errors, see "Try Statements" (page 259).

Here's an example of a subroutine that checks to see if its parameter is a real number or an integer. If not, the routine uses the "Error" (page 264) command to signal an error.

```
on areaOfCircle from radius
    -- Make sure the parameter is a real number or an integer.
    if class of radius is contained by {integer, real}
        return radius * pi  -- pi is predefined by AppleScript.
    else
        error "The parameter must be a real number or an integer"
```

```
    end if
end areaOfCircle

-- To call areaOfCircle:
areaOfCircle from 7 --result: 21.991148575129
```

Recursive Subroutines

A **recursive subroutine** is a subroutine that calls itself. Recursive subroutines are legal in AppleScript. You can use them to perform repetitive actions. For example, this recursive subroutine generates a factorial. (The factorial of a number is the product of all the positive integers from 1 to that number. For example, 4 factorial is equal to 1 * 2 * 3 * 4, or 24.)

```
on factorial(x)
    if x > 0 then
        return x * (factorial(x - 1))
    else
        return 1
    end if
end factorial

-- To call factorial:
factorial(10)   --result: 3628800
```

In the example above, the subroutine `factorial` is called once from the top level of the script, passing the value `10`. The subroutine then calls itself recursively with a value of `x - 1`, or `9`. Each time the subroutine calls itself, it makes another recursive call, until the value of `x` is `0`. When `x` is equal to `0`, AppleScript skips to the Else clause and finishes executing all the partially executed subroutines, including the original `factorial` subroutine call.

When you call a recursive subroutine, AppleScript keeps track of the variables and pending statements in the original (partially executed) subroutine until the recursive subroutine has completed. Because each call uses some memory, the maximum number of pending subroutines is limited by the available memory. As a result, a recursive subroutine may generate an error before the recursive calls complete.

In addition, a recursive subroutine may not be the most efficient solution to a problem. For example, the factorial subroutine shown above can be rewritten to use a Repeat loop instead of a recursive call:

```
on factorial(x)
    set returnVal to 1
    if x > 1 then
        repeat with n from 2 to x
            set returnVal to returnVal * n
        end repeat
    end if
    return returnVal
end factorial
```

Saving and Loading Libraries of Subroutines

So far, you've seen examples of defining and calling subroutines in the same script. This is useful for functions that are repeated more than once in the same script. But you can also write subroutines for generic functions, such as numeric operations, that are useful in many different scripts. To make a subroutine available in any script, save it as a compiled script, and then use the scripting addition command Load Script to make it available in a particular script. You can use this technique to create libraries of subroutines for use in many scripts.

For example, the following script contains three subroutines: `areaOfCircle`, which returns the area of a circle based on its radius; `factorial`, which returns the factorial of a number; and `min`, which returns the smallest number in a list of numbers.

```
-- This subroutine computes the area of a circle from its radius.
on areaOfCircle from radius
    -- Make sure the parameter is a real number or an integer.
    if class of radius is contained by {integer, real}
        return radius * pi  -- pi is predefined by AppleScript.
    else
        error "The parameter must be a real number or an integer"
    end if
end areaOfCircle
```

```
-- This subroutine returns the factorial of a number.
on factorial(x)
    set returnVal to 1
    if x > 1 then
        repeat with n from 2 to x
            set returnVal to returnVal * n
        end repeat
    end if
    return returnVal
end factorial


-- This subroutine returns the smallest number in a list
on min(numberList)
    -- Check for a valid list.
    if class of numberList is not equal to list ¬
            or numberList is equal to {} then ¬
        return numberList
    set minNum to first item in numberList
    -- If more than one item, find the smallest.
    if length of numberList > 1 then
        repeat with curNum in numberList
            if curNum < minNum then set minNum to curNum
        end repeat
    end if
    return minNum as number
end min
```

To save this script as a compiled script, choose Save As from the Script Editor's File menu and choose Compiled Script from the Kind pop-up menu. Then save the script as a file called Numeric Operations.

After you save the script as a compiled script, use the Load Script scripting addition command to make the subroutines it contains available in the current script. For example, the Load Script command in the following script assigns the compiled script Numeric Operations to the variable `numberLib`. To call the subroutines in Numeric Operations, use a Tell statement. The Tell statement in the example calls the `factorial` subroutine. (You must have a compiled script called Numeric Operations in the specified location for this script to work correctly.)

```
set numberLib to (load script file ¬
    "Hard Disk:Scripts:Numeric Operations")

tell numberLib
    min({77, 905, 50, 6, 3, 111})   --result: 3
    areaOfCircle from 12            --result: 37.699111843078
    factorial(10)                   --result: 3628800
end tell
```

The Load Script scripting addition command loads the compiled script as a script object. Script objects are user-defined objects that are treated as values by AppleScript; for more information about them, see "Script Objects" (page 325). For more information about the Load Script command, and about the other standard scripting addition commands distributed with AppleScript, see the following website:

<http://www.apple.com/applescript/>

Defining and Calling Subroutines

A subroutine definition contains

- a template for calls to the subroutine
- optional variable declarations
- statements; among these can be a Return statement that when executed returns a value and exits the subroutine

You cannot nest subroutine definitions; that is, you cannot define a subroutine within a subroutine definition.

The way you call a subroutine is determined by the way the subroutine was defined:

- You must provide all the parameters specified in the definition.
- You must provide either labeled parameters or positional parameters, as specified in the definition.

The following sections describe how to define and call subroutines:

- "Subroutines With Labeled Parameters" (page 290)

- "Subroutines With Positional Parameters" (page 296)

Subroutines With Labeled Parameters

The following sections describe the syntax for defining and calling subroutines with labeled parameters and provide examples of subroutines that use this syntax.

- "Defining a Subroutine With Labeled Parameters" (page 290)
- "Calling a Subroutine With Labeled Parameters" (page 291)
- "Examples of Subroutines With Labeled Parameters" (page 293)

Defining a Subroutine With Labeled Parameters

The definition for a subroutine with labeled parameters lists the labels to use when calling the subroutine and the statements to be executed when it is called.

SYNTAX

```
( on | to ) subroutineName                                          ¬
            [ [ of | in ] directParameterVariable ]                 ¬
            [ subroutineParamLabel paramVariable ]...               ¬
            [ given label:paramVariable [, label:paramVariable ]...]
    [ global variable [, variable ]...]
    [ local variable [, variable ]...]
    [ statement ]...
end [ subroutineName ]
```

where

subroutineName (an identifier) is the subroutine name.

directParameterVariable (an identifier) is a parameter variable (also called a formal parameter) that represents the actual value of the direct parameter. You use this identifier to refer to the direct parameter in the body of the subroutine definition. As with application commands, the direct parameter must be first.

Note
If a subroutine includes a direct parameter, the subroutine must also include either the *subroutineParamLabel* parameter or the `given` *label*:*paramVariable* parameter. ◆

subroutineParamLabel is one of the following labels: `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`, `thru` (or `through`), `under`. These labels are the only labels that can be used without the special label `given`. As in other commands, each label must be unique among the labels for the subroutine (that is, you cannot use the same label for more than one parameter).

paramVariable (an identifier) is a parameter variable for the actual value of a parameter. You use this identifier to refer to the parameter in the body of the subroutine.

label is any parameter label. This can be any valid AppleScript identifier. You must use the special label `given` to specify parameters whose labels are not among the labels for *subroutineParamLabel*.

variable is an identifier for either a global or local variable that can be used in the handler. The scope of a local variable is the handler. The scope of a global variable can extend to any other part of the script, including other handlers and script objects. For detailed information about the scope of local and global variables, see "Scope of Script Variables and Properties" (page 311).

statement is any AppleScript statement.

NOTES

For more information, see "Examples of Subroutines With Labeled Parameters" (page 293).

Calling a Subroutine With Labeled Parameters

A subroutine call for a subroutine with labeled parameters lists parameters other than the direct parameter in any order, using the labels in the subroutine definition to identify the parameter values.

SYNTAX

```
subroutineName                                                    ¬
    [ [ of | in ] directParameter ]                               ¬
    [ [ subroutineParamLabel parameterValue ]                     ¬
        | [ with labelForTrueParam [, labelForTrueParam ]...      ¬
            [ ( and | or | , ) labelForTrueParam ] ]              ¬
        | [ without labelForFalseParam [, labelForFalseParam ]...]¬
            [ ( and | or | , ) labelForFalseParam ] ]             ¬
        | [ given label:parameterValue                            ¬
            [, label:parameterValue ]...]...
```

where

subroutineName (an identifier) is the name of the subroutine.

directParameter is the direct parameter, if one is included in the subroutine definition. It can be any valid expression. As with application commands, the direct parameter must be first if it is included at all.

subroutineParamLabel is one of the following labels used in the definition of the subroutine: `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`, `thru` (or `through`), `under`.

parameterValue is the value of a parameter, which can be any valid expression.

labelForTrueParam is the label for a Boolean parameter whose value is `true`. You use this form in With clauses; because the value `true` is implied by the word With, you provide only the label, not the value. For an example of how to use a With clause, see the information on calling the subroutine `findNumbers` in "Examples of Subroutines With Labeled Parameters" (page 293). If you use `or` or a comma instead of `and` with the last parameter of a `with` clause, AppleScript changes the `of` or the comma to `and` during compilation.

labelForFalseParam is the label for a Boolean parameter whose value is `false`. You use this form in Without clauses; because the value `false` is implied by the word Without, you provide only the label, not the value. If you use `or` or a comma instead of `and` with the last parameter of a `without` clause, AppleScript changes the `or` or the comma to `and` during compilation.

label is any parameter label used in the definition of the subroutine that is not among the labels for *subroutineParamLabel*. You must use the special label `given` to specify these parameters. For an example, see "Examples of Subroutines With Labeled Parameters" (page 293).

If you use `or` or a comma instead of `and` with the last parameter of a `with` clause, AppleScript changes the `or` or the comma to `and` during compiling.

NOTES

A subroutine call must include all the parameters specified in the subroutine definition. There is no way to specify optional parameters.

With the exception of the direct parameter, which must directly follow the subroutine name, labeled parameters can appear in any order. This includes parameters listed in Given, With, and Without clauses. Furthermore, you can include any number of Given, With, and Without clauses in a subroutine call. The examples for the `findNumbers` subroutine in "Examples of Subroutines With Labeled Parameters" (page 293) demonstrate how a Given clause can be replaced by equivalent With and Without clauses.

Examples of Subroutines With Labeled Parameters

This section provides examples of subroutine definitions with labeled parameters and of calls to those subroutines.

The following subroutine returns the area of a circle based on its radius:

```
on areaOfCircle from radius
    -- Make sure the parameter is a real number or an integer.
    if class of radius is contained by {integer, real}
        return radius * pi  -- pi is predefined by AppleScript.
    else
        error "The parameter must be a real number or an integer."
    end if
end areaOfCircle

-- To call areaOfCircle:
areaOfCircle from 7 --result: 21.991148575129
```

The following subroutine searches for a specific string in a list of files. It returns a list containing the names of any files that contained the specified string. For

the `searchFiles` handler to work, the specified files must be on the startup disk (the current system disk, obtained from the Finder).

```
to searchFiles of filesToSearch for theString
    -- filesToSearch: list of AppleWorks files.
    -- theString: the string to be searched for.
    -- Note: always searches on startup disk.
    set hits to {}
    tell application "Finder" to set theDisk to (startup disk as string)
    tell application "AppleWorks"
        repeat with i from 1 to (count items of filesToSearch)
            set currentWindow to item i of filesToSearch
            set currentFile to theDisk & currentWindow
            open currentFile
            set docText to text body of front document
            if docText contains theString then
                -- Append currentWindow to list of hits.
                set hits to hits & currentWindow
            end if
            close front document saving no
        end repeat
        return hits
    end tell -- application "AppleWorks"
end searchFiles

-- To call searchFiles:
searchFiles of {"March Expenses", "April Expenses", ¬
    "May Expenses", "June Expenses"} for "LeChateau"
--result: {"March Expenses", "May Expenses"} (if those two
--          documents contain the phrase "LeChateau")
```

The following subroutine uses the special label `given` to define a parameter with the label `rounding`. By using verb forms ending with “ing” as labels, you can often make subroutine calls easier to read.

```
to findNumbers of numberList above minLimit ¬
    given rounding:roundBoolean
        set resultList to {}
        repeat with i from 1 to (count items of numberList)
            set x to item i of numberList
            if roundBoolean = true then
```

```
                -- Use copy so original list isn't modified.
                copy (x + 0.5) div 1 to x
            end if
            if x > minLimit then
                copy resultList & x to resultList
            end if
        end repeat
        return resultList
end findNumbers

-- To call findNumbers:

--  where myList is thelist of numbers {2, 5, 19.75, 99, 1}:
findNumbers of myList above 19 given rounding:true
    --result: {20, 99}
findNumbers of myList above 19 given rounding:false
    --result: {19.75, 99}

findNumbers of {5.1, 20.1, 20.5, 33} above 20 given rounding:true
    --result: {21, 33}

findNumbers of {5.1, 20.1, 20.5, 33.7} above 20 given rounding:false
    --result: {20.1, 20.5, 33.7}
```

Another way to call the `findNumbers` subroutine is to use a With or Without clause to specify the value of the `rounding` parameter. A With or Without clause specifies whether a parameter's value is `true` or `false`. (In fact, when you compile the previous examples, AppleScript automatically converts `given rounding:true` to `with rounding` and `given rounding:false` to `without rounding`.) So the following statements are equivalent to the last two statements in the previous example and generate the same results.

```
findNumbers of {5.1, 20.1, 20.5, 33} above 20 with rounding
    --result: {21, 33}

findNumbers of {5.1, 20.1, 20.5, 33.7} above 20 without rounding
    --result: {20.1, 20.5, 33.7}
```

The subroutine parameter labels that can be used without the special label `given` allow you considerable flexibility in defining handlers that sound English-like.

For example, here's a routine that takes any parameter that can be displayed as a string and displays it in a dialog box:

```
on rock around the clock
    display dialog (clock as string)
end rock
```

The statement

```
rock around the current date
```

later in the same script displays the current date in a dialog box.

Here's another example of the use of subroutine parameter labels:

```
to check for yourNumber from bottom thru top
    if bottom ≤ yourNumber and yourNumber ≤ top then
        display dialog "Congratulations! You scored."
    end if
end check
```

The statement

```
check for 8 from 7 thru 10
```

later in the same script displays the specified dialog box.

Subroutines With Positional Parameters

The following sections describe the syntax for defining and calling subroutines with positional parameters and provide examples of subroutines that use this syntax.

- "Defining a Subroutine With Positional Parameters" (page 297)
- "Calling a Subroutine With Positional Parameters" (page 297)
- "Examples of Subroutines With Positional Parameters" (page 298)

Defining a Subroutine With Positional Parameters

The definition for a subroutine with positional parameters lists the order in which to list parameters when calling the subroutine and the statements to be executed when the subroutine is called.

SYNTAX

```
( on | to ) subroutineName ( [ paramVariable [, paramVariable ]...] )
    [ global variable [, variable ]...]
    [ local variable [, variable ]...]
    [ statement ]...
end [ subroutineName ]
```

where

subroutineName (an identifier) is the name of the subroutine.

paramVariable (an identifier) is a parameter variable for the actual value of the parameter. You use this identifier to specify the parameter in the body of the subroutine.

variable is an identifier for either a global or local variable that can be used in the handler. The scope of a local variable is the handler. The scope of a global variable can extend to any other part of the script, including other handlers and script objects. For detailed information about the scope of local and global variables, see "Scope of Script Variables and Properties" (page 311).

statement is any AppleScript statement.

The parentheses that enclose the series of positional parameters in the syntax definition are a required part of the language. They are shown in bold to distinguish them from parentheses that show grouping but are not part of the language. The parentheses must be included even if the subroutine definition doesn't include any parameters.

For more information, see "Examples of Subroutines With Positional Parameters" (page 298).

Calling a Subroutine With Positional Parameters

A subroutine call for a subroutine with positional parameters lists the parameters in the same order as they are specified in the subroutine definition.

SYNTAX

subroutineName ([*parameterValue* [, *parameterValue*]...])

where

subroutineName (an identifier) is the name of the subroutine.

parameterValue is the value of a parameter, which can be any valid expression. If there are two or more parameters, they must be listed in the same order in which they were specified in the subroutine definition.

The parentheses that enclose the series of positional parameters are a required part of the language. They are shown in bold to distinguish them from parentheses that show grouping but are not part of the language. The parentheses must be included even if the subroutine definition doesn't include any parameters.

NOTES

A subroutine call must include all the parameters specified in the subroutine definition. There is no way to specify optional parameters.

You can use a subroutine call as a parameter of another subroutine call. Here's an example.

```
minimumValue(2, maximumValue(x, y))
```

The second parameter of the call to `minimumValue` is the value from the subroutine call to `maximumValue`. The `minimumValue` subroutine is defined in "Examples of Subroutines With Positional Parameters" (page 298).

A call to a subroutine with positional parameters can include parameters that aren't literals as long as they evaluate to a pattern defined for the subroutine. Similarly, the properties of a record passed to a subroutine don't have to be given in the same order they are given in the subroutine's declaration, as long as all the properties required to fit the defined pattern are present. The examples that follow include subroutines with positional parameters that define a pattern.

Examples of Subroutines With Positional Parameters

Here is a subroutine that returns the minimum value of a pair of values.

```
on minimumValue(x, y)
    if x ≤ y then
        return x
    else
        return y
    end if
end minimumValue

-- To call minimum value:
minimumValue(21, 40000)
```

You can also define a subroutine whose positional parameters define a pattern to match when calling the subroutine. For example, the subroutine that follows takes a single parameter whose pattern consists of two items in a list.

```
on point({x, y})
    display dialog ("x = " & x & ", y = " & y)
end point

set mypoint to {3, 8}
point(mypoint)
```

A parameter pattern can be much more complex than a single list. The handler in the next example takes two numbers and a record whose properties include a list of bounds. The handler displays a dialog box summarizing some of the passed information.

```
on hello(a, b, {length:l, bounds:{x, y, w, h}, name:n})
    set q to a ÷ b

    set response to "Hello " & n & ", you are " & l & ¬
        " inches tall and occupy position (" & x & ", " & y & ")."

    display dialog response

end hello

set thing to {bounds:{1, 2, 4, 5}, name:"George", length:72}
hello (2, 3, thing)
--result: A dialog displaying “Hello George, you are 72 inches tall
--          and occupy position (1,2).”
```

As you can see from this example, a call to a subroutine with patterned parameters can include parameters that aren't literals, as long as they evaluate to the appropriate pattern. Similarly, the properties of a record passed to a subroutine with patterned parameters don't have to be given in the same order in which they are given in the subroutine's definition, as long as all the properties required to fit the pattern are present.

Command Handlers

Command handlers are handlers for application or system commands. They are similar to subroutine handlers, but instead of defining responses to user-defined commands, they define responses to commands, such as Open, Print, or Move, that are sent to applications. They may also define responses to commands that are sent to scriptable items in the Mac OS, such as desktop printers or the Appearance control panel.

Command handlers are described in the following sections:

- "Command Handler Syntax" (page 300)
- "Command Handlers for Application Objects" (page 302)
- "Command Handlers for Script Applications" (page 302)

For information about recursion in command handlers, see "Recursive Subroutines" (page 286). For information about the scope of variables and properties in handlers, see "Scope of Script Variables and Properties" (page 311).

Command Handler Syntax

A command handler definition is a set of statements that is executed in response to an application command. Command handler definitions need not include all of the possible parameters of the commands they respond to. If a command handler receives more parameters than are specified in the command handler definition, it ignores the extra parameters.

SYNTAX

The syntax for a command handler definition is

```
( on | to ) commandName                                          ¬
        [ [ of ] directParameterVariable ]                       ¬
        [ given label:paramVariable [, label:paramVariable ]...]
    [ global variable [, variable ]...]
    [ local variable [, variable ]...]
    [ statement ]...
end [ commandName ]
```

where

commandName (an identifier) is a command name.

directParameterVariable (an identifier) is a parameter variable for the actual value of the direct parameter. You use this parameter variable to refer to the direct parameter in the body of the subroutine. If it is included, *directParameter* must be listed immediately after the command name. The word `of` before *directParameter* is optional.

label is the parameter label for one of the parameters of the command being handled. The label `given` is optional.

paramVariable (an identifier) is a parameter variable for the actual value of the parameter. You use this identifier to refer to the parameter in the body of the handler.

variable is an identifier for either a global or local variable that can be used in the handler. The scope of a local variable is the handler. The scope of a global variable can extend to any other part of the script, including other handlers and script objects. For detailed information about the scope of local and global variables, see "Scope of Script Variables and Properties" (page 311).

statement is any AppleScript statement.

EXAMPLES

For examples of command handler definitions, see "Run Handlers" (page 303).

NOTES

The statements in a command handler can include a Continue statement, which passes the command to the application's default handler for that command. This allows you to invoke an application's default behavior for a command from within a command handler. For more information, see "The Continue Statement" (page 336).

Command Handlers for Application Objects

You can use a command handler in a script to handle a command that is sent to an application object. Associating a script that contains a command handler with an application object that receives a command is called **attaching a script to an application object.** An application that supports this capability is called an attachable application. Such an application might allow you to attach a script to a button or menu object, for example, to handle commands sent to those objects.

Scripts that are attached to objects can change the way those objects respond to particular commands. Each application determines which of its objects, if any, can have attached scripts and how you attach the scripts. To determine whether you can attach a script to an application's objects, see the documentation for the application.

Command Handlers for Script Applications

A script application is an application whose only function is to run the script associated with it. You can run a script application from the Finder much like any other application. For a description of how to create a script application, see "Script Applications" (page 279).

Every script application is a command handler and can respond to at least two commands: the Run command and the Open command. A script application receives a Run command whenever it is launched, and an Open command whenever another icon is dropped on its icon in the Finder.

A stay-open script application can receive and handle any commands for which it has a handler. All stay-open applications receive periodic Idle commands whenever they're not responding to other events. They also receive a Quit command when the user quits the application.

For descriptions of common command handlers, see the following sections:

- "Run Handlers" (page 303)
- "Open Handlers" (page 305)
- "Handlers for Stay-Open Script Applications" (page 306)

Run Handlers

All Mac OS applications can respond to the Run command, even if they aren't scriptable. The Finder sends a Run command to an application whenever one of the following actions occurs while that application is not already running:

- The user double-clicks the application's icon.
- The user selects the application's icon and chooses Open from the File menu.
- The application's icon is in the Apple Menu Items folder and the user chooses it from the Apple menu.
- The application's icon is in the Startup Items folder and the user restarts the computer.

If the application is already running when one of these actions occurs, the application is activated but no commands are sent to it. If the application isn't running, the Finder launches the application and sends it a Run command. The application responds by performing the actions the user expects when the application first opens, such as opening an untitled document.

Like any other application, a script application, described in "Script Applications" (page 279), receives a Run command whenever one of the actions just listed occurs. You can provide a handler for the Run command in one of two ways. An **implicit Run handler** consists of all statements at the top level of a script except for property declarations, script object definitions, and other command handlers. An **explicit Run handler** is enclosed within an `on...end` or an `on run...end` statement, like other handlers.

For example, the script that follows consists of a property declaration, an `increment` command, a Tell statement, and a handler for the `increment` command. For the Tell statement to work, you must have an AppleWorks document named Count Log open before you run the script. Each time you run the script, the value of the property x increases by 1 and the increase is recorded in the Count Log file (by replacing the first paragraph with the count).

```
property x : 0

increment()

tell document "Count Log" of application "AppleWorks"
    select first paragraph of text body
    set selection to "Count is now " & x & "."
end tell

on increment()
    set x to x + 1
    display dialog "Count is now " & x & "."
end increment
```

The implicit Run handler for this script consists of the statement `increment()` and the Tell statement—that is, the statements outside the handler, but not including property declarations. If you store this script in a script application and then double-click the script application's icon, the Finder sends a Run command to the script, and the Run command invokes the two statements in the implicit Run handler.

If you rewrite the previous script to use an explicit Run handler, it provides the exact same behavior:

```
property x : 0

on run
    increment()
    tell document "Count Log" of application "AppleWorks"
        select first paragraph of text body
        set selection to "Count is now " & x & "."
    end tell
end run

on increment()
    set x to x + 1
    display dialog "Count is now " & x & "."
end increment
```

A script can't include both an implicit and an explicit Run handler. If a script includes both an explicit `on run` handler and top level commands that constitute

an implicit Run handler, AppleScript returns an error when you try to compile the script—that is, when you try to run it, check its syntax, or save it.

The Run handlers in the preceding examples respond the same way to a Run command whether the script is saved as a script application or as a compiled script. If the script is saved as a compiled script, you can invoke its Run handler by clicking the Run button in the Script Editor.

You can also send a Run command to a script application from within another script. For information about how to do this, see "Calling a Script Application From a Script" (page 310).

Open Handlers

All Mac OS applications can respond to the Open command, even if they aren't scriptable. The Finder sends an Open command to an application whenever the user drags file, folder, or disk icons over the application's icon and releases the mouse button. The Open command is sent even if the application is already running.

Like any other application, a script application receives an Open command whenever the user drags file, folder, or disk icons over the application's icon. If the script in the script application includes an Open handler, the statements within the handler run when the application receives the Open command. The Open handler takes a single parameter; when the handler is called, the value of that parameter is a list of all the items whose icons were dropped on the script application's icon. (Each item in the list is an alias; you can convert it to a pathname by using `as string`.)

For example, this Open handler makes a list of the pathnames of all items dropped on the script application's icon and saves them in a specified AppleWorks document:

```
on open names
    set listOfPaths to "" -- Start with empty string.
    repeat with i in names
        -- Get the name and append a return character so that
        --  each name starts on a separate line.
        set iPath to (i as string)
        set listOfPaths to listOfPaths & iPath & return
    end repeat
    -- Open document and replace current first paragraph with list.
    tell application "AppleWorks"
```

```
        open file "Hard Disk:File List"
        tell front document
            select first paragraph of text body
            set selection to listOfPaths
        end tell
        close front document saving ask
    end tell
    return
end open
```

Files, folders, or disks are not moved, copied, or affected in any way when their icons are dragged and dropped over a script application's icon. The Finder just gets a list of their identities and sends that list to the script application as the direct parameter of the Open event. Of course, the script in the script application could easily tell the Finder to move, copy, or otherwise manipulate the items. The script could also ask the user to specify a filename for saving the list of pathnames.

Note
Due to a known limitation of system software, you can't drop icons on an icon for a script application that's stored on a floppy disk. ◆

You can also run an Open handler by sending a script application the Open command. For details, see "Calling a Script Application From a Script" (page 310).

Handlers for Stay-Open Script Applications

By default, a script application that receives a Run or Open command handles that single command and then quits. This allows it to perform a single task and get out of your way. In contrast, a stay-open script application (one saved with the Stay Open checkbox selected in the Script Editor's Save As dialog box) stays open after it's launched.

A stay-open script application can be useful for any of the following reasons:

- If you run a script frequently, it runs faster as a stay-open application than it does if it has to be launched each time.
- Stay-open script applications can receive and handle other commands in addition to Run and Open. This allows you to use a script application as a

script server that, when its running, provides a collection of handlers that can be invoked by any other script.

- Stay-open script applications can perform periodic actions, even in the background, as long as the script application is running.

All stay-open applications receive periodic Idle events. If a stay-open script application includes a handler for the Idle event, it can perform periodic actions whenever it is not responding to other events. If a stay-open script application includes a handler for the Quit event, it can perform some action, such as checking with the user, before quitting.

Idle Handlers

If a stay-open script application includes an Idle handler, AppleScript sends the script application periodic Idle commands whenever it's not responding to incoming events. The statements in the handler run periodically (every 30 seconds, by default).

For example, the following handler causes a stay-open script application to beep every 30 seconds after it has been launched.

```
on idle
    beep
end idle
```

To change the rate, return the number of seconds to wait as the result of the idle handler. For example, the following script beeps every 5 seconds.

```
on idle
    beep
    return 5
end idle
```

If an Idle handler returns a positive number, that number becomes the rate (in seconds) at which the handler is called. If the handler returns a non-numeric value, the rate is not changed.

Remember that the result returned from a handler is just the result of the last statement, even if it doesn't include the word `return` explicitly. For example, this handler only gets called every 15 minutes.:

```
on idle
    set x to 30
    beep
    set x to x * x  -- The handler returns the result (900).
end idle
```

To make sure you're not changing the idle rate, return 0 at the end of the handler.

Quit Handlers

AppleScript sends a stay-open script application a Quit command whenever the user chooses the Quit menu command or presses Command-Q while the application is active. If the script includes a Quit handler, the statements in the handler are run before the application quits.

A Quit handler can be used to set script properties, tell another application to do something, display a dialog box, or perform almost any other task. If the handler includes a `continue quit` statement, the script application's default quit behavior is invoked and it quits. If the Quit handler returns before it encounters a `continue quit` statement, the application doesn't quit.

For example, this handler checks with the user before allowing the application to quit:

```
on quit
    display dialog "Really quit?" ¬
        buttons {"No", "Quit"} default button "Quit"
    if the button returned of the result is "Quit" then
        continue quit
    end if
    -- Without the continue statement, the
    -- script application doesn't quit.
end quit
```

▲ **WARNING**
If AppleScript doesn't encounter a `continue quit` statement while executing an `on quit` handler, it may seem impossible to quit the application. For example, if the handler gets an error before the `continue quit` statement, attempting to quit the application just produces an error alert. As a last resort, use the emergency Quit command: press Command-Shift-Q or hold down the Shift key and choose Quit from the File menu. This saves changes to script properties and quits immediately, bypassing the Quit handler. ▲

Interrupting a Script Application's Handlers

A stay-open script application handles incoming commands even if it is already running a handler in response to a previous command. This means that execution of a handler can be interrupted while another handler is run. Because script applications are not multithreaded, execution of the first handler pauses until the second one finishes. This is known as "last-in, first-out" event handling.

This can cause problems if both handlers modify the same script property or global variable or if both attempt to modify an application's data. For example, suppose that running a script application named `increment` causes it to increment the property `p` for several minutes:

```
property p : 0

on close
    set temp to p
    set p to 0
    return temp
end close

set p to 0
repeat 1000000 times
    set p to p + 1
end repeat
```

If this script application receives a Close command while it is running:

```
tell application "Increment" to close
```

AppleScript can't deal with such interruptions automatically.

Calling a Script Application From a Script

Any script can send commands to a script application just as it can to any other application. However, script applications, like other applications, sometimes respond to the Run command in ways that you might not expect.

As explained in the description of the command "Launch" (page 143), AppleScript sends an implicit Run command whenever it begins to execute a Tell statement whose target is an application that is not already open. This creates problems for a script application that doesn't stay open.

For example, a script like this won't run correctly if the target application is a script application that doesn't stay open:

```
tell application "NonStayOpen" to run
```

Instead, the Tell statement launches the script application and sends it an implicit Run command. The application handles that Run command. AppleScript then gets to the explicit Run command in the calling script and tries to send another run event to the script application. Unfortunately, the application has already handled its one event and quits without responding to the second Run command. The calling script waits in vain until it times out, and then receives an error.

The culprit is the implicit Run command sent by the Tell statement when it launches the application. To launch a non-stay-open application and run its script, use a Launch command followed by a Run command, like this:

```
launch application "NonStayOpen"
run application "NonStayOpen"
```

The Launch command launches the script application without sending it an implicit Run command. When the Run command is sent to the script application, it processes the event, sends back a reply if necessary, and quits.

Similarly, to launch a non-stay-open application and run its Open Handler, use a Launch command followed by an Open command, like this:

```
tell application "NonStayOpen"
    launch
    open {alias "HardDisk:MyFile", ¬
        alias "HardDisk:MyOtherFile"}
end tell
```

For example, if the Open handler `on open names` in "Open Handlers" (page 305) were saved as a script application called "NonStayOpen," the script in the preceding example would cause the handler to create a list of the two specified pathnames.

For information on how to create script applications, see "Script Applications" (page 279).

Scope of Script Variables and Properties

Before reading this section, you should be familiar with the information provided in "Run Handlers" (page 303).

The **declaration** of a variable or property identifier is the first valid occurrence of the identifier in a script. The form and location of the declaration determine how AppleScript treats the identifier in that script.

The **scope** of a variable or property declaration is the range over which AppleScript recognizes the declared identifier within a script. The scope of a property declaration is the entire script or script object in which it is defined.

It is often convenient to limit the scope of a particular identifier to a single handler—that is, to treat the identifier as a **local variable** within a handler. After a local variable has served its purpose, its identifier no longer has any value associated with it and can be used again for other purposes elsewhere in the script.

If you want the value of a script variable to persist after a script is run, or if you wish to use the same identifier in several different places in a script, you can declare it as either a script property or a **global variable.** AppleScript keeps track of properties and global variables across multiple handlers and script objects within a single script.

The following sections provide additional information about the scope of AppleScript variables and properties:

- "Declaring Variables and Properties" (page 312)
- "Scope of Properties and Variables Declared at the Top Level of a Script" (page 313)
- "Scope of Properties and Variables Declared in a Script Object" (page 316)
- "Scope of Variables Declared in a Handler" (page 321)

Declaring Variables and Properties

The following examples show the four basic forms for declaring variables and properties in AppleScript:

- `property x: 3`

  This statement declares a property and sets its initial value. The scope of a property declaration can be either a script object or an entire script. The value set by a property declaration is not reset each time the script is run; instead, it persists until the script is recompiled.

- `global x`

  This global declaration is similar to a property declaration except that it doesn't set an initial value: The scope of a global variable declaration can be limited to specific handlers or script objects or can extend throughout an entire script. Like the value of a property, the value of a global variable is not reset each time a script is run. However, the value of a global variable must be set by other statements in the script.

- `set x to 3`

  You can use a Set command or the Copy command to set the value of any property or variable.If the variable has not been declared previously, the Set or Copy command declares it as a local variable.

- `local x`

  This statement declares a local variable explicitly. Like a global declaration, an explicit local declaration doesn't set an initial value.

Scope of Properties and Variables Declared at the Top Level of a Script

Figure 8-1 summarizes the scope of properties and variables declared at the top level of a script. Sample scripts using each form of declaration follow.

Figure 8-1 Scope of property and variable declarations at the top level of a script

Form of declaration	Scope of declaration	Where AppleScript looks for x
`property x: 3`	Everywhere in script	To top level of script
`global x`		
`set x to 3`	Within Run handler only	
`local x`		Within Run handler only

The scope of a property declaration at the top level of a script extends to any subsequent statements anywhere in the script. Here's an example:

```
property currentCount : 0
increment()

on increment()
    set currentCount to currentCount + 1
    display dialog "Count is now " & currentCount & "."
end increment
```

When it encounters the identifier `currentCount` at any level of this script, AppleScript associates it with the `currentCount` property declared at the top level.

The value of a property persists after the script in which the property is defined has been run. Thus, the value of `currentCount` in the previous example is 0 the first time the script is run, 1 the next time, and so on. The property's current value is saved with the script and is not reset to 0 until the script is recompiled—that is, modified and then run again, saved, or checked for syntax.

Similarly, the scope of a global variable declaration at the top level of a script extends to any subsequent statements anywhere in the script. The next example accomplishes the same thing as the previous example, except that it uses a

global variable instead of a property to keep track of the count. Note that the first time the script is run, the statement

```
set currentCount to currentCount + 1
```

generates an error because the variable `currentCount` has not been initialized yet. When the error happens, the `on error` block initializes `currentCount`.

```
global currentCount
increment()

on increment()
    try
        set currentCount to currentCount + 1
        display dialog "Count is now " & currentCount & "."
    on error
        set currentCount to 1
        display dialog "Count is now 1."
    end try
end increment
```

When it encounters the identifier `currentCount` at any level of this script, AppleScript associates it with the `currentCount` variable declared as a global at the top level of the script. However, because a global variable declaration doesn't set the initial value of a property, the script must use a Try statement to determine whether the value has been previously set. Thus, if you want the value associated with an identifier to persist, it is often easier to declare it as a property so that you can declare its initial value at the same time.

If you don't want the value associated with an identifier to persist after a script is run but you want to use the same identifier throughout a script, declare a global variable and use the Set command to set its value each time the script is run. Here's an example:

```
global currentCount
set currentCount to 0
on increment()
    set currentCount to currentCount + 1
end increment

increment() --result: 1
increment() --result: 2
```

Each time the `on increment` handler is called within the script, the global variable `currentCount` increases by 1. However, when you run the entire script again, `currentCount` is reset to 1.

In the absence of a global variable declaration at the top level of a script, the scope of a variable declaration using the Set command at the top level of a script is normally restricted to the Run handler for the script. For example, this script declares two separate `currentCount` variables:

```
set currentCount to 10
on increment()
    set currentCount to 5
end increment

increment() --result: 5
currentCount --result: 10
```

The scope of the first `currentCount` variable's declaration, at the top level of the script, is limited to the Run handler for the script. Because this script has no explicit Run handler, statements at the top level are part of its implicit Run handler, as described in "Run Handlers" (page 303). The scope of the second `currentCount` declaration, within the `on increment` handler, is limited to that handler. AppleScript keeps track of each variable independently.

To associate a variable in a handler or a script object with the same variable declared at the top level of a script with the Set command, you can use a global declaration in the handler, as shown in the next example.

```
set currentCount to 0
on increment()
    global currentCount
    set currentCount to currentCount + 1
end increment

increment() --result: 1
currentCount --result: 1
```

In this case, when AppleScript encounters the `currentCount` variable within the `on increment` handler, it looks for a previous mention of `currentCount` not only within the handler, but also at the top level of the script. However, references to `currentCount` in any other handler in the script are local to that handler unless the handler also explicitly declares `currentCount` as a global. This kind of global

declaration is discussed in more detail in "Scope of Variables Declared in a Handler" (page 321).

To restrict the context of a variable to a script's Run handler regardless of subsequent global declarations, you must declare it explicitly as a local variable, as shown in this example:

```
local currentCount
set currentCount to 10
on increment()
    global currentCount
    set currentCount to currentCount + 2
end increment

increment() --error: "The variable currentCount is not defined"
```

Because the `currentCount` variable in this example is declared as local to the script, and hence to its implicit Run handler, any subsequent attempt to use the same variable as a global results in an error.

Note
If you declare a variable with the Set command at the top level of a script or script object and then declare the same identifier as a property, the declaration with the Set command overrides the property declaration. For example, the script

```
set x to 10
property x: 5
return x
```

returns 10, not 5. This occurs because AppleScript always evaluates property declarations at the top level of a script before it evaluates Set command declarations. ◆

Scope of Properties and Variables Declared in a Script Object

You should be familiar with the information in "Script Objects" (page 325) before you read this section.

Figure 8-2 summarizes the scope of properties and variables declared at the top level of a script object. Sample scripts using each form of declaration follow.

Figure 8-2 Scope of property and variable declarations at the top level of a script object

Form of declaration	Scope of declaration	Where AppleScript looks for x
`property x: 3`	Everywhere in script object	To top level of script object
`global x`		To top level of script
`set x to 3`	Within script object's Run handler only	Within script object's Run handler only
`local x`		

The scope of a property declaration at the top level of a script object extends to any subsequent statements in that script object. Here's an example.

```
script Joe
property currentCount : 0
    on increment()
        set currentCount to currentCount + 1
        return currentCount
    end increment
end script

tell Joe to increment() --result: 1
tell Joe to increment() --result: 2
```

When it encounters the identifier `currentCount` at any level of the script object `Joe`, AppleScript associates it with the same identifier declared at the top level of the script object. The value of the property `currentCount` persists until you reinitialize the script object by running the script again.

The scope of a property declaration at the top level of a script object doesn't extend beyond the script object. Thus, it is possible to use the same identifier in different parts of a script to refer to different properties, as this example demonstrates:.

```
property currentCount : 0
    script Joe
        property currentCount : 0
        on increment()
            set currentCount to currentCount + 1
            return currentCount
        end increment
    end script

tell Joe to increment() --result: 1
tell Joe to increment() --result: 2
currentCount --result: 0
```

AppleScript keeps track of the property `currentCount` declared at the top level of the script separately from the property `currentCount` declared within the script object `Joe`. Thus, the `currentCount` property declared at the top level of the script `Joe` is increased by 1 each time `Joe` is told to increment, but the `currentCount` property declared at the top level of the script is not affected.

Like the scope of a property declaration, the scope of a global variable declaration at the top level of a script object extends to any subsequent statements in that script object. However, as the next example demonstrates, AppleScript also associates a global variable with the same variable declared at the top level of the entire script.

```
set currentCount to 0
script Joe
global currentCount
    on increment()
        set currentCount to currentCount + 1
        return currentCount
    end increment
end script

tell Joe to increment() --result: 1
tell Joe to increment() --result: 2
```

The preceding example first sets the value of `currentCount` at the top level of the script. When AppleScript encounters the `currentCount` variable within the `on increment` handler, it first looks for an earlier occurrence within the handler, then at the top level of the script `Joe`. When AppleScript encounters the global

declaration for currentCount at the top level of script object Joe, it continues looking at the top level of the script until it finds the original declaration for currentCount. This can't be done with a property of a script object, because AppleScript looks no further than the top level of a script object for that script object's properties.

Like the value of a script object's property, the value of a script object's global variable persists after the script object has been run, but not after the script itself has been run. Thus, telling Joe to increment repeatedly in the preceding example continues to increment the value of currentCount, but running the whole script again sets currentCount to 0 again before incrementing it.

The next example demonstrates how you can use a global variable declaration in a script object to associate a global variable with a property declared at the top level of a script.

```
property currentCount : 0
script Donna
    property currentCount : 20
    script Joe
        global currentCount
        on increment()
            set currentCount to currentCount + 1
            return currentCount
        end increment
    end script
    tell Joe to increment()
end script

run Donna --result: 1
run Donna --result: 2
currentCount --result: 2
currentCount of Donna --result: 20
```

This script declares two separate currentCount properties: one at the top level of the script and one at the top level of the script object Donna. Because the script Joe declares the global variable currentCount, AppleScript looks for currentCount at the top level of the script, thus treating Joe's currentCount and currentCount at the top level of the script as the same variable.

If the script object Joe in the preceding example doesn't declare currentCount as a global variable, AppleScript treats Joe's currentCount and the currentCount at

the top level of the script object `Donna` as the same variable. This leads to quite different results, as shown in the next example.

```
property currentCount : 0
script Donna
    property currentCount : 20
    script Joe
        on increment()
            set currentCount to currentCount + 1
            return currentCount
        end increment
    end script
    tell Joe to increment()
end script

run Donna --result: 21
run Donna --result: 22
currentCount --result: 0
currentCount of Donna --result:22
```

The scope of a variable declaration using the Set command at the top level of a script object is limited to the Run handler:

```
script Joe
    set currentCount to 10
    on increment()
        global currentCount
        set currentCount to currentCount + 2
    end increment
    return currentCount
end script

tell Joe to increment()
--error: "The variable currentCount is not defined."

run Joe--result: 10
```

In contrast to the way it treats such a declaration at the top level of a script, AppleScript treats the `currentCount` variable declared at the top level of the script object `Joe` in the preceding example as local to the script object's Run

handler. Any subsequent attempt to use the same variable as a global results in an error.

Similarly, the scope of an explicit local variable declaration at the top level of a script object is limited to that script object's Run handler, even if the same identifier has been declared as a property at a higher level in the script:

```
property currentCount : 0
script Joe
    local currentCount
    set currentCount to 5
    on increment()
        set currentCount to currentCount + 1
    end increment
end script

run Joe --result: 5
tell Joe to increment() --result: 1
```

Scope of Variables Declared in a Handler

You can't declare a property in a handler, although you can refer to a property declared at the top level of the script or script object to which the handler belongs.

Figure 8-3 summarizes the scope of variables declared in a handler. Examples of each form of declaration follow.

Figure 8-3 Scope of variable declarations within a handler

Form of declaration	Scope of declaration	Where AppleScript looks for x
`global x`	Within handler only	To top level of script
`set x to 3`		Within handler only
`local x`		

The scope of a global variable declared in a handler is limited to that handler, although AppleScript looks beyond the handler when it tries to locate an earlier occurrence of the same variable. Here's an example.

```
set currentCount to 10
on increment()
    global currentCount
    set currentCount to currentCount + 2
end increment

increment() --result: 12
currentCount --result: 12
```

When AppleScript encounters the `currentCount` variable within the `on increment` handler, it doesn't restrict its search for a previous occurrence to that handler but keeps looking until it finds the declaration at the top level of the script. However, references to `currentCount` in any subsequent handler in the script are local to that handler unless the handler also explicitly declares `currentCount` as a global variable.

The scope of a variable declaration using the Set command within a handler is limited to that handler:

```
script Henry
    set currentCount to 10
    on increment()
        set currentCount to 5
    end increment
    return currentCount
end script

tell Henry to increment() --result: 5
run Henry --result: 10
```

The scope of the first declaration of the first `currentCount` variable, at the top level of the script object `Henry`, is limited to the Run handler for the script object. The scope of the second `currentCount` declaration, within the `on increment` handler, is limited to that handler. AppleScript keeps track of each variable independently.

The scope of a local variable declaration in a handler is limited to that handler, even if the same identifier has been declared as a property at a higher level in the script:

```
property currentCount : 10
on increment()
    local currentCount
    set currentCount to 5
end increment

increment() --result: 5
currentCount --result: 10
```

Script Objects

Script objects are objects that you define and use in scripts. Like the application and system objects described earlier in this guide, script objects have properties and can respond to commands. Unlike application or system objects, script objects are defined within scripts.

AppleScript's script objects have capabilities common to object-oriented programming languages. For example, you can define groups of script objects that share properties and handlers, and you can extend or modify the behavior of a handler in one script object when calling it from another script object.

This chapter describes handlers in the following sections:

- "About Script Objects" (page 326) provides a brief overview of script objects and includes a simple example.
- "Defining Script Objects" (page 327) provides the syntax for defining script objects.
- "Sending Commands to Script Objects" (page 328) describes how you use Tell statements to send commands to script objects.
- "Initializing Script Objects" (page 329) describes how AppleScript creates a script object with the properties and handlers you have defined.
- "Inheritance and Delegation" (page 331) describes how you can share property and handler definitions among script objects without repeating the shared definitions.
- "Using the Copy and Set Commands With Script Objects" (page 342) describes differences in the way the Copy and Set commands work with script objects. It also describes how to write a handler that creates copies of script objects.

About Script Objects

A **script object** is a user-defined object that combines data (in the form of properties) and potential actions (in the form of handlers). A **script object definition** is a compound statement that can contain collections of properties, handlers, and other AppleScript statements.

Here is a simple script object definition:

```
script John
    property HowManyTimes : 0
    to sayHello to someone
        set HowManyTimes to HowManyTimes + 1
        return "Hello " & someone
    end sayHello
end script
```

It defines a script object that can handle the `sayHello` command. It assigns the script object to the variable `John`. The definition includes a handler for the `sayHello` command. It also includes a property, called HowManyTimes, that indicates how many times the `sayHello` command has been called.

A handler within a script object definition follows the same syntax rules as a subroutine definition. Unlike a subroutine definition, however, you can group a handler within a script object definition with properties whose values are related to the handler's actions.

After you define a script object, you initialize it by running the script that contains the script object definition. You can then use a Tell statement to send commands to the script object. For example, the following statement sends the `sayHello` command the script object defined above.

```
tell John to sayHello to "Herb"
```

The result is `"Hello Herb"`.

You can manipulate the properties of script objects in the same way you manipulate the properties of system and application objects. Use the Get command to get the value of a property and the Set or Copy command to change the value of a property.

The following statement uses a Get command to get the value of the HowManyTimes property of script object `John`.

```
get HowManyTimes of John
if the result > 10
    return "John, aren't you tired of saying hello?"
end if
```

Defining Script Objects

Each script object definition begins with the keyword `script`, followed by an optional variable name, and ends with the keyword `end` (or `end script`). The statements in between can be any combination of property definitions, handler definitions, and other AppleScript statements.

The syntax of a script object definition is

```
script [ scriptObjectVariable ]
    [( property | prop ) propertyLabel : initialValue ]...
    [ handlerDefinition ]...
    [ statement ]...
end [script]
```

where

scriptObjectVariable is a variable identifier. If you include *scriptObjectVariable*, AppleScript stores the script object in a variable. You can use the variable identifier to refer to the script object elsewhere in the script.

propertyLabel is an identifier for a property. Properties are characteristics that are identified by unique labels. They are similar to instance variables in object-oriented programming.

initialValue is the value that is assigned to the property each time the script object is initialized. Script objects are initialized when the scripts or handlers that contain them are run. *initialValue* is required in property definitions.

handlerDefinition is a handler for a user-defined or system command. The handlers within a script object definition determine which commands the script object can respond to. Script object definitions can include handlers for

user-defined commands (subroutines) or for system or application commands. Handlers in script objects are similar to methods in object-oriented programming. For a detailed description of the syntax of handler definitions, refer to "Handlers" (page 279).

statement is any AppleScript statement. Statements other than handler and property definitions are treated as if they were part of a handler definition for the Run command; they are executed when a script object receives the Run command.

Sending Commands to Script Objects

You use Tell statements to send commands to script objects. A Tell statement sent to a script object is similar to a Tell statement sent to an application, except that it uses a variable name, instead of a reference, to identify the script object. For example,

```
tell John
    sayHello to "Herb"
    sayHello to "Grace"
end tell
```

sends two `sayHello` commands to the script object `John`. The parameters of the commands in the Tell statement, if any, must match the parameters defined in the handler definitions in the script object definition. For example, the following statement results in an error message because the handler definition for the `sayHello` command, shown in "About Script Objects" (page 326), defines a labeled parameter, not a positional parameter.

```
tell John
    sayHello ("Herb")
end tell
--results in an error
```

For a script object to respond to a command within a Tell statement, either the script object or its parent script object must have a handler for the command. A parent script object is a script object from which a script object inherits handlers and properties. For more information about parent script objects, see "Inheritance and Delegation" (page 331).

The one command that any script object can handle, even without an explicitly defined handler, is the Run command. A handler for the Run command can consist of all statements at the top level of a script object definition other than property and handler definitions. If the script object definition contains only handler and property definitions, and does not include any additional top-level statements, the definition may include an explicit Run handler that begins with `on run`. If a script object definition includes neither an implicit Run handler (in the form of top-level statements) nor an explicit Run handler, the Run command doesn't do anything. For more information, see "Run Handlers" (page 303).

For example, the Display Dialog scripting addition command in the following script object definition is executed only if you send a Run command to script object `John`.

```
script John
    property HowManyTimes : 0
    to sayHello to someone
        set HowManyTimes to HowManyTimes + 1
        return "Hello " & someone
    end sayHello
    display dialog "John received the Run command"
end script
```

Initializing Script Objects

When you define a script object, you define a collection of handlers and properties. When you run a script containing a script object definition, AppleScript creates a script object with the properties and handlers listed in the definition. This is called **initializing a script object.** A script object must be initialized before it can respond to commands.

If you include a script object definition at the top level of a script—that is, as part of the script's Run handler—AppleScript initializes the script object each time the script's Run handler is executed. For more information, see "Run Handlers" (page 303).

Similarly, if you include a script definition in another handler within a script, AppleScript initializes a script object each time the handler is called. The

parameter variables in the handler definition become local variables of the script object. For example, the `makePoint` handler in the following script contains a script object definition for the script object `point`:

```
on makePoint(x, y)
    script point
        property xCoordinate:x
        property yCoordinate:y
    end script
    return point
end makePoint

set myPoint to makePoint(10,20)
get xCoordinate of myPoint  --result: 10
get yCoordinate of myPoint  --result: 20
```

AppleScript initializes the script object `point` when it executes the `makePoint` command. The parameter variables in the `makePoint` handler, in this case, `x` and `y`, become local variables of the script object `point`. The initial value of `x` is 10, and the initial value of `y` is 20, because those are the parameters of the `makePoint` command that initialized the script object.

One way to use script object definitions in handlers is to define constructor functions, that is, handlers that create script objects. The following script uses a constructor function to create three script objects.

```
on makePoint(x, y)
    script
        property xCoordinate:x
        property yCoordinate:y
    end script
end makePoint

set PointA to makePoint(10,20)
set PointB to makePoint(100,200)
set PointC to makePoint(1,1)
```

As in the previous example, you can retrieve the coordinates of the three script objects using the Get command.

Note
The distinction between defining a script object and initializing a script object is similar to the distinction between a class and an instance in object-oriented design. When you define a script object, you define a class of objects. When AppleScript initializes a script object, it creates an instance of the class. The script object gets its initial context (property values and handlers) from the script object definition, but its context can change as it responds to commands. ◆

Inheritance and Delegation

You can use AppleScript's inheritance mechanism to define related script objects in terms of one another. This allows you to share property and handler definitions among many script objects without repeating the shared definitions. Inheritance and delegation are described in the following sections.

- "Defining Inheritance" (page 331) describes how to a define a script object that inherits properties and handlers from another script object.
- "How Inheritance Works" (page 332) demonstrates inheritance in the relationships between several parent-child scripts.
- "The Continue Statement" (page 336) describes how to extend the behavior of an inherited handler without completely replacing it.

Defining Inheritance

Inheritance is the ability of a child script object to take on the properties and handlers of a parent script object. You specify inheritance with the Parent property. A script object that includes a Parent property inherits the properties and handlers of the script object listed in the Parent property.

The script object listed in a Parent property definition is called the **parent script object,** or parent. A script object that includes a Parent property is referred to as a **child script object,** or child. The Parent property is not required. A script object can have many children, but a child script object can have only one parent.

The syntax for defining a parent script object is

```
( property | prop ) parent : variable
```

where

variable is a variable that contains the parent script object.

A script object must be initialized before it can be assigned as a parent of another script object. This means that the definition of the parent script object (or a command that calls a function that creates the parent script object) must come before the definition of the child in the same script.

How Inheritance Works

The inheritance relationship between script objects should be familiar to those who are acquainted with C++ or other object-oriented programming languages. A child script object that inherits the handlers and properties defined in its parent is like a C++ class that inherits methods and instance variables from its parent class. If the child does not have its own definition of a property or handler, it uses the inherited (hidden) property or handler. If the child has its own definition of a particular property or handler, then it ignores (or overrides) the inherited property or handler.

Figure 9-1 shows the relationship between a parent script object called `John` and a simple child script object called `Simple`. The figure includes two versions of the child script object. The version on the left shows the actual script object definition for the child script `Simple`. The version on the right shows how the script object definition would look with the inherited properties and handlers copied in. The inherited properties and handlers are shown between dotted lines, to indicate that they aren't actually a part of the script object definition for `Simple`. As you can see, `Simple` inherits the HowManyTimes property and the `sayHello` handler from its parent.

Figure 9-2 shows another parent-child relationship. As in the previous example, the child script object inherits the HowManyTimes property and the `sayHello` handler from its parent, `John`. But this time, the child script object, called `Rebel`, has its own HowManyTimes property, so it doesn't use the one inherited from the parent. In the figure, the inherited property that is not used is crossed out.

Figure 9-1 Relationship between a simple child script and its parent

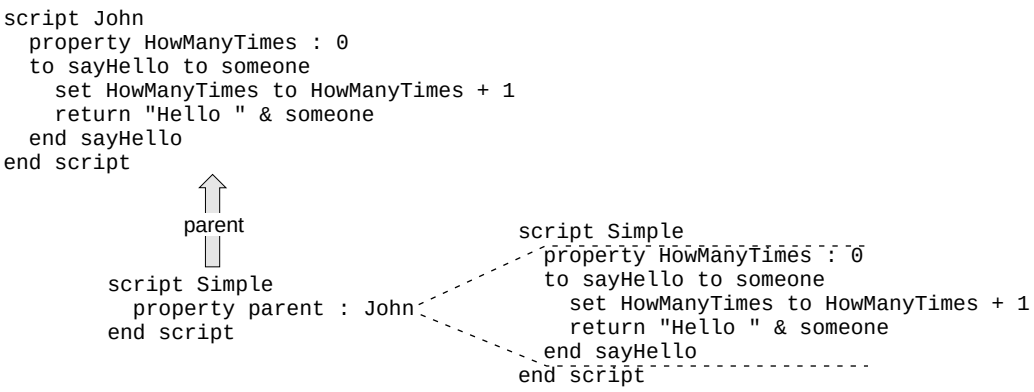
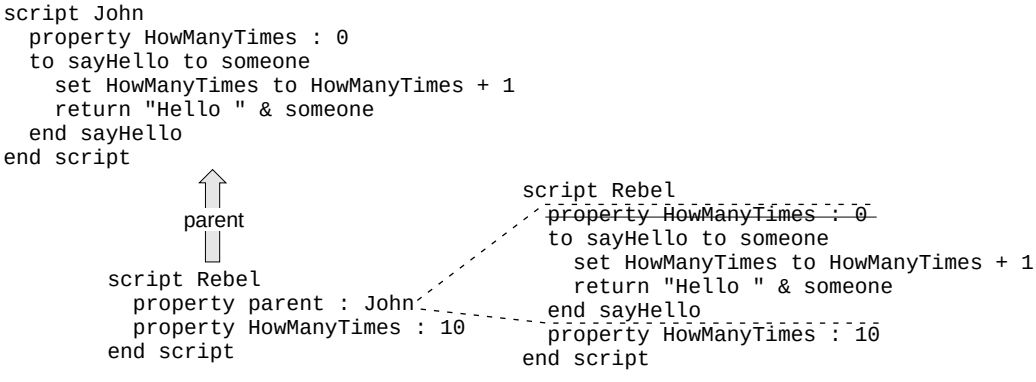


Figure 9-2 Another child-parent relationship



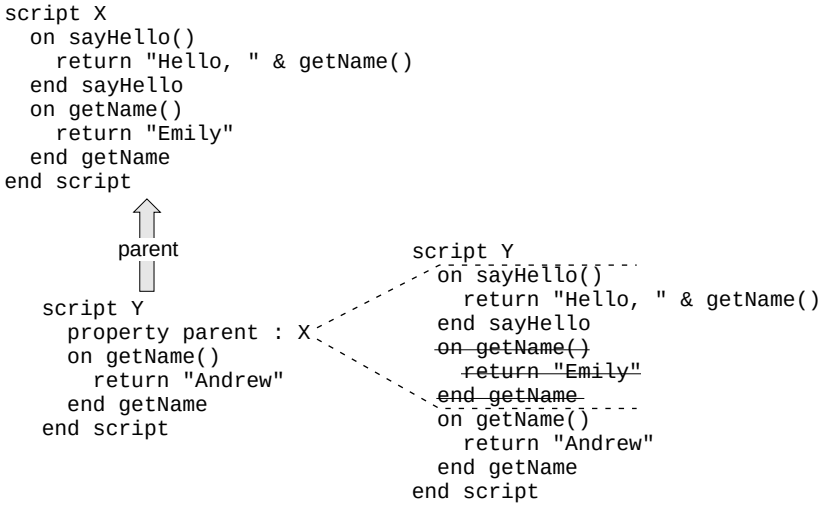
Consider the parent and child script objects in the following script. At first glance, it might appear that the result of the `sayHello` command is `"Hello Emily"`. However, because script object `Y` has its own `getName` handler, the actual result is `"Hello Andrew"`. The inheritance relationships for the script are shown in Figure 9-3.

```
script X
    on sayHello()
        return "Hello, " & getName()
    end sayHello
    on getName()
        return "Emily"
    end getName
end script

script Y
    property parent : X
    on getName()
        return "Andrew"
    end getName
end script

tell Y to sayHello()
```

Figure 9-3 A more complicated child-parent relationship



Even though script `X` in Figure 9-3 sends itself the `getName` command, the command is intercepted by the child script, which substitutes its own version of

the `getName` handler. AppleScript always maintains the first target of a command as the "self" to which inherited commands are sent, redirecting to the child any inherited commands the parent sends to itself.

The relationship between a parent script object and its child script objects is dynamic. If the properties of the parent change, so do the inherited properties of the children. For example, the script object `Simple` in the following script inherits its Vegetable property from script object `John`.

```
script John
    property Vegetable : "Spinach"
end script

script Simple
    property parent : John
end script

set Vegetable of John to "Swiss chard"
Vegetable of Simple
--result: "Swiss chard"
```

When you change the Vegetable property of script object `John` with the Set command, you also change the Vegetable property of the child script object `Simple`. The result of the last line of the script is `"Swiss chard"`.

Similarly, if a child changes one of its inherited properties, the value of the parent property changes. For example, the script object `JohnSon` in the following script inherits the Vegetable property from script object `John`.

```
script John
    property Vegetable : "Spinach"
end script

script JohnSon
    property parent : John
    on changeVegetable()
        set my Vegetable to "Zucchini"
    end changeVegetable
end script
tell JohnSon to changeVegetable()
Vegetable of John
--result: "Zucchini"
```

When you change the Vegetable property of script object `JohnSon` to `"Zucchini"` with the `changeVegetable` command, the Vegetable property of script object `John` also changes.

The previous example demonstrates an important point about inherited properties: to refer to an inherited property from within a child script object, you must use the reserved word `my` or `of me` to indicate that the value to which you're referring is a property of the current script object. (You can also use the words `of parent` to indicate that the value is a property of the parent script object.) If you don't, AppleScript assumes the value is a local variable.

For example, if you refer to `Vegetable` instead of `my Vegetable` in the `changeVegetable` handler in the previous example, the result is `"Spinach"`.

```
script John
    property Vegetable : "Spinach"
end script

script JohnSon
    property parent : John
    on changeVegetable()
        set Vegetable to "Zucchini"
        -- Creates a local variable called Vegetable;
        -- doesn't change value of the parent's Vegetable property.
    end changeVegetable
end script

tell JohnSon to changeVegetable()
Vegetable of John
--result: "Spinach"
```

The Continue Statement

Normally, if a child script object and its parent both have handlers for the same command, the child uses its own handler. However, the handler in a child script object can handle a command first, and then use a Continue statement to call the handler for the same command in the parent.

The use of a Continue statement to call a handler in a parent script object is called **delegation.** By delegating commands to a parent script object, a child can extend the behavior of a handler contained in the parent without having to

repeat the entire handler definition. After the parent handles the command, AppleScript continues at the place in the child where the Continue statement was called. Handlers in child script objects that contain Continue statements are similar to wrapper methods in object-oriented programming.

The syntax of a Continue statement is

```
continue commandName parameterList
```

where

commandName is the name of the current command.

parameterList is the list of parameters to be passed with the command. The list must follow the same format as the parameter definitions in the handler definition for the command. For handlers with labeled parameters, this means that the parameter labels must match those in the handler definition. For handlers with positional parameters, the parameters must appear in the correct order. You can list actual values or parameter variables. If you list actual values, those values replace the parameter values that were specified in the original command. If you list parameter variables, the Continue statement passes the parameter values that were specified in the original command.

The following script includes two script object definitions similar to those shown in Figure 9-1 (page 333). The first, `Elizabeth`, works just like the script `John` in the figure. The second, `ChildOfElizabeth`, includes a handler with a Continue statement that is not included in the child script object (`Simple`) shown in the figure.

```
script Elizabeth
    property HowManyTimes : 0
    to sayHello to someone
        set HowManyTimes to HowManyTimes + 1
        return "Hello " & someone
    end sayHello
end script

script ChildOfElizabeth
    property parent : Elizabeth
    on sayHello to someone
        if my HowManyTimes > 3 then
            return "No, I'm tired of saying hello."
        else
```

```
            continue sayHello to someone
        end if
    end sayHello
end script

tell Elizabeth to sayHello to "Matt"
--result: "Hello Matt", no matter how often the tell is executed

tell ChildOfElizabeth to sayHello to "Bob"
--result: "Hello Bob", the first four times the tell is executed;
--  after the fourth time: "No, I'm tired of saying hello."
```

In the preceding example, the handler defined by `ChildOfElizabeth` for the `sayHello` command checks the value of the HowManyTimes property each time the handler is run. If the value is greater than 3, `ChildOfElizabeth` returns a message refusing to say hello. Otherwise, `ChildOfElizabeth` calls the `sayHello` handler in the parent script object (`Elizabeth`), which returns the standard hello message. The word `someone` in the Continue statement is a parameter variable. It indicates that the parameter received with the original `sayHello` command will be passed to the handler in the parent script.

Note
The reserved word `my` in the statement `if my HowManyTimes > 10` in the previous example is required to indicate that `HowManyTimes` is a property of the script object. Without the word `my`, AppleScript assumes that `HowManyTimes` is an undefined local variable. ◆

A Continue statement can change the parameters of a command before delegating it. For example, suppose the following script object is defined in the same script as the preceding example. The first Continue statement changes the direct parameter of the `sayHello` command from `"Bill"` to `"William"`. It does this by specifying the value `"William"` instead of the parameter variable `someone`.

```
script AnotherChildOfElizabeth
    property parent : Elizabeth
    on sayHello to someone
        if someone = "Bill" then
            continue sayHello to "William"
        else
            continue sayHello to someone
```

```
        end if
    end sayHello
end script

tell AnotherChildOfElizabeth to sayHello to "Matt"
--result: "Hello Matt"

tell AnotherChildOfElizabeth to sayHello to "Bill"
--result: "Hello William"
```

If you override a parent's handler in this manner, the reserved words `me` and `my` in the parent's handler no longer refer to the parent, as demonstrated in the example that follows.

```
script Hugh
    on identify()
        me
    end identify
end script

script Andrea
    property parent : Hugh
    on identify()
        continue identify()
    end identify
end script

tell Hugh to identify()
--result: «script Hugh»

tell Andrea to identify()
--result: «script Andrea»
```

Using Continue Statements to Pass Commands to Applications

A scripting addition command or application command sent to a script object doesn't trigger any action until it is passed on to the default target application. As a result, you can use a command handler in script object to modify the way a command works.

For example, the handler for the Beep command in the example that follows modifies the scripting addition version of the Beep command by displaying a dialog box and allowing the user to decide whether to continue or not:

```
script Joe
    on beep
        set x to display dialog ¬
            "Do you really want to hear this awful noise?" ¬
            buttons {"Yes", "No"}
        if the button returned of x is "Yes" then ¬
            continue beep -- Let scripting addition handle the beep.
    end beep
end script

tell Joe to beep --result: dialog to confirm whether to beep
```

When AppleScript encounters the Tell statement, it sends a Beep command to script `Joe`. The Beep handler causes the default target application (for example, the Script Editor) to display a dialog box that gives the user a choice about hearing the alert sound. If the user clicks Yes, the handler uses a Continue statement to pass the Beep command to the default target application. If the user clicks No, the target application never receives the Beep command and no alert sound is heard.

In applications that allow you to attach script objects to application objects, you can use a handler for an application command in a script object to modify the way the application responds to the command.

For example, if a drawing application allows you to associate script objects with geometric shapes such as circles or squares, you could include a handler like this in a script object associated with a shape in a document:

```
on move to {x, y}
    continue move to {x, item 2 of my position}
end move
```

Whenever the shape the script object is associated with is named as the target of a Move command, the `on move` handler handles the command by modifying one of the parameters and using the continue statement to pass the command on to the default parent—that is, the drawing application. The location specified by `{x, item 2 of my position}` has the same horizontal coordinate as the location specified by the original Move command, but specifies the shape's original

vertical coordinate (item 2 of the circle's original position), thus constraining the shape's motion to a horizontal direction.

The documentation for attachable applications that allow you to associate script objects with application objects in this manner should provide more information about how to write handlers for application commands.

The Parent Property and the Current Application

The **current application** is either the default target application or whatever application is currently set as a script's parent property. The default parent property for any script that doesn't explicitly declare one is the default target application—usually, the application that is running the script, such as the Script Editor. You can use the predefined variable `current application` to refer to the current application.

You can make any application the current application for a script or script object simply by declaring it as a parent property. Any subsequent command in the script for which the script doesn't have a handler is passed to the application you declare as the parent, and subsequent occurrences of the constant `current application` refer to that application.

For example, this script declares the Finder as its parent property, then sends commands that close the Finder's frontmost window and return the application's name:

```
property parent: application "Finder"
close front window
tell current application to return my name --result: "Finder"
```

In this case, `my` refers to the current application (Finder). The Tell statement is optional; using `return the name of me` would produce the same result, because AppleScript sends the command to the Finder. If you remove the property declaration from the script, the Script Editor becomes the current application. When sent to the Script Editor, the Close command and the Return statement produce errors because the Script Editor doesn't understand them.

In the next example, the script `Gertrude` declares the Finder as its parent property and includes a handler that modifies the behavior of the scripting addition command Display Dialog.

```
script Gertrude
    property parent : application "Finder"
    on display dialog x
        tell application "Script Editor" to display dialog ¬
            "Finder has something to say"
        continue display dialog x
    end display dialog
end script

tell Gertrude to display dialog "Hello"
```

Because the script object `Gertrude` declares the Finder as its parent property, the `on display dialog` handler must use a Tell statement to send a separate Display Dialog command to the Script Editor. The handler then uses a Continue statement to pass the original Display Dialog command to the Finder, which becomes the frontmost application and uses the Display Dialog scripting addition to display “Hello”.

Using the Copy and Set Commands With Script Objects

The Copy and Set commands both assign values to variables, but they produce different results when the value assigned is a script object. The Copy command makes a new copy of the script object, while the Set command creates a variable that shares data with the original script object. Note that this behavior (Copy creates a new copy, Set shares the original data) is the same when you work with lists and records, as described in “Data Sharing” (page 206).

To examine how Copy and Set work with script objects, consider the following example, which defines a script object, called `John`, with a property called Vegetable.

```
script John
    property Vegetable: "Spinach"
end script

set myScriptObject to John
set Vegetable of John to "Swiss chard"
get Vegetable of myScriptObject
--result: "Swiss chard"
```

The first Set command defines a variable, called `myScriptObject`, that shares data with the original script object `John`. The second Set command changes the value of the Vegetable property of script object `John` from `"Spinach"` to `"Swiss chard"`. Because `myScriptObject` shares data with `John`, it shares the change to the Vegetable property of `John`. When you get the Vegetable property of `myScriptObject`, the result is `"Swiss chard"`.

Now consider the following example, which uses the Copy command to define the variable `myScriptObject`.

```
script John
    property Vegetable: "Spinach"
end script

copy John to myScriptObject
set Vegetable of John to "Swiss chard"
get Vegetable of myScriptObject
--result: "Spinach"
```

In this case, the Copy command creates a new script object. Setting the Vegetable property of the original script object has no effect on the new script object. The result of the Get command is `"Spinach"`.

When you copy a child script object to a variable, the variable contains a complete copy of both the child and its parent, including all the parent's properties and handlers. Each new copy, including its inherited properties and handlers, is completely independent of both the original and any other copies.

For example, if you copy a modified version of the `JohnSon` script in this example to two different variables, you can set each variable's Vegetable property independently:

```
script John
    property Vegetable : "Spinach"
end script

script JohnSon
    property parent : John
    on changeVegetable(x)
        set my Vegetable to x
    end changeVegetable
end script
```

```
copy JohnSon to J1
copy JohnSon to J2

tell J1 to changeVegetable("Zucchini")
tell J2 to changeVegetable("Swiss chard")

Vegetable of J1
--result: "Zucchini"

Vegetable of J2
--result: "Swiss chard"

Vegetable of John
--result: "Spinach"
```

You can create handlers that construct copies of script objects for use elsewhere in a script. For example, the script that follows includes a handler that takes an initial balance as a parameter and creates a copy of a script object that acts as an independent account. Each copy includes several properties and an `on deposit` handler that enables the script object to increment its own balance when it receives a Deposit command.

```
on makeAccount(initialBalance)
    script account
        property StartDate : current date
        property Balance : initialBalance
        on deposit(amount)
            set Balance to Balance + amount
        end deposit
    end script
end makeaccount

set a to makeAccount(3300)
set b to makeAccount(33)


tell a
    deposit(30)
    deposit(60)
end tell
```

```
{Balance of a, StartDate of a}
--result: {3390, date "Saturday, February 13, 1999 12:12:57 PM"}

{Balance of b, StartDate of b}
--result: {33, date "Saturday, February 13, 1999 12:13:38 PM"}
```

Appendixes

APPENDIX A

The Language at a Glance

This appendix summarizes the AppleScript language in the following sections:

Common Scripting Tasks

Table A-1 lists common scripting tasks and provides links to sections in this book with related sample scripts or overview material.

Table A-1 Links to sample scripts and other useful information

Working With Applications	
Launching an application	"Launch" (page 143)
Moving data from a database to a spreadsheet	"Repeat Until" (page 253)
Quitting an application	"Quit" (page 151)
Reopening an application	"Reopen" (page 152)
Running an application	"Run" (page 154)
Specifying an application on a remote machine	"References to Remote Applications" (page 196)
Working With Constants	
Arithmetic, Boolean, Consider and Ignoring, Date and Time, Miscellaneous, Save Option, String, Text Style, and Version	"Constants" (page 100)
Working With Error Handling	
Handling a User Cancelled error returned from the Display Dialog scripting addition command	"Try" (page 261)
Setting a timeout interval	"With Timeout" (page 273)
Using a Try statement to handle errors	"Error" (page 264)
(See also **Miscellaneous Tasks** in this table.)	

Table A-1 Links to sample scripts and other useful information (continued)

Working With Files and Documents	
Creating a new application document	"Make" (page 146)
Opening a file	"Open" (page 149)
Printing a document	"Print" (page 150)
Printing all open documents with a Repeat loop and Exit statement	"Exit" (page 258)
Saving a document	"Save" (page 156)
Searching for a string in a list of documents	"Examples of Subroutines With Labeled Parameters" (page 293)
Specifying a file by alias, by file specification, or by reference	"References to Files and Applications" (page 190)
Working With the Finder	
Cleaning up a Finder window	"Dictionaries" (page 34)
Counting files in a folder	"Using Results" (page 121)
Counting files on a disk	"Count" (page 134)
Deleting a file on the startup disk	"Delete" (page 137)
Determining if a file exists	"Characteristics of Control Statements" (page 238); "If Statements" (page 245)
Determining if a folder exists	"Exists" (page 139)
Duplicating a file on the startup disk	"Duplicate" (page 138)
Examining files by file type	"Filter" (page 173)
Moving a file to a different folder	"Move" (page 148)
Opening all the aliases in a folder	Figure 2-3 (page 27)
Opening files and folders by ID	"ID" (page 174)
Special disk and folder names the Finder understands (`control panels folder`, `startup disk`, `system folder`, and others)	"Viewing a Result in the Script Editor's Result Window" (page 121)

Table A-1 Links to sample scripts and other useful information (continued)

Working With Handlers and Subroutines	
Handling pathnames of files dropped on a script application	"Open Handlers" (page 305)
Working with libraries of subroutines	"Saving and Loading Libraries of Subroutines" (page 287)
Writing an idle handler	"Idle Handlers" (page 307)
Writing a minimum value subroutine	"A Sample Subroutine" (page 282)
Writing a quit handler	"Quit Handlers" (page 308)
Writing a recursive subroutine	"Recursive Subroutines" (page 286)
Working with references	
Overview of references and containers	"References" (page 165)
Summary of reference forms	"Reference Forms" (page 169)
Working with reference filters	"Using the Filter Reference Form" (page 187)
Working with references to files and applications	"References to Files and Applications" (page 190)
Working with the Reference value class	"Reference" (page 77)
Working With the Script Editor	
Compiling a script with the Script Editor	"Compiling Scripts With the Script Editor" (page 47)
Entering raw data in a script	"Entering Script Information in Raw Format" (page 125)
Examining an application's dictionary	"Dictionaries" (page 34)
Using comments in a script	"Comments" (page 43)
Using the Script Editor's Event Log window	"Debugging Scripts" (page 47)

Table A-1 Links to sample scripts and other useful information (continued)

Working With Text	
Getting a paragraph of text	"Get" (page 141)
Moving text within a document	"Relative" (page 185)
Replacing text in a word processing document	"Styled Text" (page 84)
Styled Clipboard Text	"Styled Clipboard Text" (page 96)
Unicode and International Text	"Unicode Text and International Text" (page 87)
Working With Value Classes	
Common value classes	Table 3-1 (page 56)
Coercing values	"Coercing Values" (page 97)
Dates	"Date" (page 62); "Date-Time Arithmetic" (page 233); "Working With Dates at Century Boundaries" (page 235)
Measurements of length, area, cubic and liquid volume, mass, and temperature	"Unit Type Value Classes" (page 91)
RGB Colors	"RGB Color" (page 96)
Working With Windows	
Closing a window	"Close" (page 130)
Closing a window on a remote computer	"Tell (Compound Statement)" (page 244)
Getting the screen position of a window	"Copy" (page 132)
Miscellaneous Tasks	
Changing Text Item Delimiters	"AppleScript Properties" (page 210)

Table A-1 Links to sample scripts and other useful information (continued)

Comparing strings	"Greater Than, Less Than" (page 224); "Considering/Ignoring" (page 269)
Creating variables	"Creating Variables" (page 201);
Debugging a script	"Debugging Scripts" (page 47)
Getting and setting Clipboard data	"Styled Clipboard Text" (page 96)
Inserting and accessing items in large lists	"List" (page 67); "The A Reference To Operator" (page 203)
Rules for naming identifiers	"Identifiers" (page 44)
Scope of properties and variables	"Scope of Script Variables and Properties" (page 311)
Speaking text from a script	"Debugging Scripts" (page 47)
Working with script objects	"Script Objects" (page 325)

Constants

Table A-2 lists constants defined by AppleScript. See "Constants" (page 100) for additional information on constants.

As with all other identifiers, constants are not case sensitive. For example, `false`, `False`, and `FALSE` are all treated as the same constant.

Table A-2 Constants defined by AppleScript

Identifier	Meaning
Attributes specified in Considering and Ignoring statements	
`application responses`	If ignored, AppleScript doesn't wait for responses from application commands before proceeding to the next statement in a script and ignores any results or errors returned. By default, AppleScript waits for `application responses`.
`case`	If considered, AppleScript distinguishes uppercase letters from lowercase. By default, `case` is ignored.
`current application`	Either the default target application or whatever application is currently set as a script's parent property.
`diacriticals`	If ignored, AppleScript ignores diacritical marks in string comparisons. By default, `diacriticals` are considered.
`expansion`	If ignored, AppleScript treats the characters æ, Æ, œ, and Œ as single characters and thus not equal to the character pairs ae, AE, oe, and OE. By default, `expansion` is considered.
`hyphens`	If ignored, AppleScript ignores hyphens in string comparisons By default, `hyphens` are considered.
`punctuation`	If ignored, AppleScript ignores punctuation marks in string comparisons. By default, `punctuation` is considered.
`white space`	If ignored, AppleScript ignores spaces, tab characters, and return characters in string comparisons. By default, `white space` is considered.
Text styles	
`all caps`	All caps
`all lowercase`	All lowercase
`bold`	Boldface
`condensed`	Condensed

Table A-2 Constants defined by AppleScript (continued)

Identifier	Meaning
`expanded`	Expanded
`hidden`	Hidden
`italic`	Italic
`outline`	Outline
`plain`	Plain text
`shadow`	Shadow
`small caps`	Small caps
`strikethrough`	Strikethrough
`subscript`	Subscript
`superscript`	Superscript
`underline`	Underline
Save options	
`ask`	Ask user whether to save modified object or objects.
`no`	Don't save modified object or objects.
`yes`	Save modified object or objects.
Boolean constants	
`false`	The Boolean `false` value.
`true`	The Boolean `true` value.
Miscellaneous	
`current application`	Either the default target application or whatever application is currently set as a script's parent property.
Days	
`weekday`	Property of date (Monday through Sunday)
`Monday, Mon`	Monday
`Tuesday, Tue`	Tuesday

Table A-2 Constants defined by AppleScript (continued)

Identifier	Meaning
`Wednesday, Wed`	Wednesday
`Thursday, Thu`	Thursday
`Friday, Fri`	Friday
`Saturday, Sat`	Saturday
`Sunday, Sun`	Sunday
Months	
`month`	Property of date (January through December)
`January, Jan`	January
`February, Feb`	February
`March, Mar`	March
`April, Apr`	April
`May, May`	May
`June, Jun`	June
`July, Jul`	July
`August, Aug`	August
`September, Sep`	September
`October, Oct`	October
`November, Nov`	November
`December, Dec`	December
Time	
`minutes`	60 (seconds in a minute)
`hours`	60 * `minutes` (or 3600)
`days`	24 * `hours` (or 86,400)
`weeks`	7 * `days` (or 604,800)

Predefined Variables

Table A-3 lists special variables that are defined by AppleScript. These variables are global, that is, they are available anywhere in a script.

As with all other identifiers, predefined variables are not case sensitive. For example, `result`, `Result`, and `RESULT` are all treated as the same variable.

See "Constants" (page 100) for additional information on predefined variables.

Table A-3 Predefined variables

Identifier	Class	Description
`it`	Reference	The default target. For more information, see Chapter 7, "Control Statements."
`me`	Reference	The current script (used within Tell statements to refer to handlers or properties of the current script). For more information, see Chapter 7, "Control Statements," and Chapter 8, "Handlers."

`version`	version	The current AppleScript version. Property of AppleScript; also property of other applications.
		`if (version as string) ≥ "1.3.4" then`
		`-- do task that requires a recent version`

`pi`	Real	The value π (roughly 3.14159).
`result`	Any class	The result returned by the most recently executed command or the most recently evaluated expression. The value of `result` is undefined if the most recently executed command did not return a result.
`return`	String	A return character.
`space`	String	A space character.

Table A-3 Predefined variables (continued) (continued)

Identifier	Class	Description
`tab`	String	A tab character.
`anything`	Class	Accept any kind of value.
`missing value`	Class	One or more items in a group of elements is missing a checked-for value.

Commands

Commands are described in detail in “Commands” (page 109).

A command is a request for action. In AppleScript, you can use application commands, which are defined in each application’s dictionary; AppleScript commands, which are defined and handled by AppleScript; or scripting addition commands, which are defined and handled by AppleScript extensions called scripting additions.

Table A-4 lists standard application commands and AppleScript commands. For information about scripting addition commands, see “Scripting Additions” (page 40) and “Scripting Addition Commands” (page 112). You can also refer to the AppleScript section of the Mac OS Help Center, or visit the AppleScript website at

<http://www.apple.com/applescript/>

The syntax shown for standard application commands is the syntax supported by most applications. Individual applications can extend or change the way the standard application commands work.

For information about how a specific application handles a particular application command, see the application’s dictionary. For more detailed descriptions of the commands listed here, see Chapter 4, “Commands.”

IMPORTANT

Some syntax statements use the continuation character (¬) to identify text that, if included, must appear on the same line. The ¬ character is not a required part of the syntax, although you can use it to extend a statement beyond one line. ▲

Table A-4 Command syntax

Command	Syntax	Result
`close` (application command)	close *referenceToObject*	None
	close *referenceToObject* saving in *referenceToFile*	
	close *referenceToObject* saving *saveOption*	
	close *referenceToObject* saving in *referenceToFile*¬ saving *saveOption*	
`copy` (AppleScript command)	(copy ∣ put) *expression* (to ∣ into) *variablePattern*	Value copied
`count` (AppleScript command)	count *compositeValue*	Integer
	count [each ∣ every] *className* (in ∣ of)¬ *compositeValue*	
	number of *compositeValue*	
	number of *pluralClassName* (in ∣ of) *compositeValue*	
`count` (application command)	count [each ∣ every] *className*	Integer or list of integers
	count [each ∣ every] *className* (in ∣ of) *referenceToObject*]	
	number of *className*	
	number of *className* [(in ∣ of) *referenceToObject*]	

Table A-4 Command syntax (continued)

Command	Syntax	Result
`delete` (application command)	delete *referenceToObject*	None
`duplicate` (application command)	duplicate *referenceToObject* duplicate *referenceToObject* to *referenceToLocation*	Reference
`error` (AppleScript command)	error¬ [*errorMessage*]¬ [number *errorNumber*]¬ [from *offendingObject*]¬ [partial result *resultList*]¬ [to *expectedType*]	
`exists` (application command)	exists *referenceToObject* *referenceToObject* exists	Boolean
`get` (AppleScript command)	get *expression* get *expression* as *className*	Value of expression
`get` (application command)	get *referenceToObject* get *referenceToObject* as *className*	Value of reference
`launch` (application command)	launch launch *referenceToApplication*	None
`make` (application command)	make [new] *className* at *referenceToLocation*¬ [with properties {*propertyLabel*:*propertyValue*¬ [, *propertyLabel*:*propertyValue*]...}]¬ [with data *dataValue*]	Reference to the new object
`move` (application command)	move *referenceToObject* to *referenceToLocation*	Reference to the moved object

Table A-4 Command syntax (continued)

Command	Syntax	Result
`open` (application command)	open *referenceToFile* open *listOfFiles*	None
`print` (application command)	print *referenceToObject*	None
`quit` (application command)	quit *referenceToApplication* quit *referenceToApplication* saving *saveOption*	None
`reopen` (application command)	`reopen` `reopen` *referenceToApplication*	None
`run` (AppleScript command)	run run *scriptObjectVariable*	The value, if any, returned by the script object
`run` (application command)	run run *referenceToApplication*	None
`save` (application command)	save *referenceToObject* save *referenceToObject* in *referenceToFile*	None
`set` (AppleScript command)	set *variablePattern* to *expression* *expression* returning *variablePattern*	Value assigned
`set` (application command)	set *referencePattern* to *expression* *expression* returning *referencePattern*	Value assigned

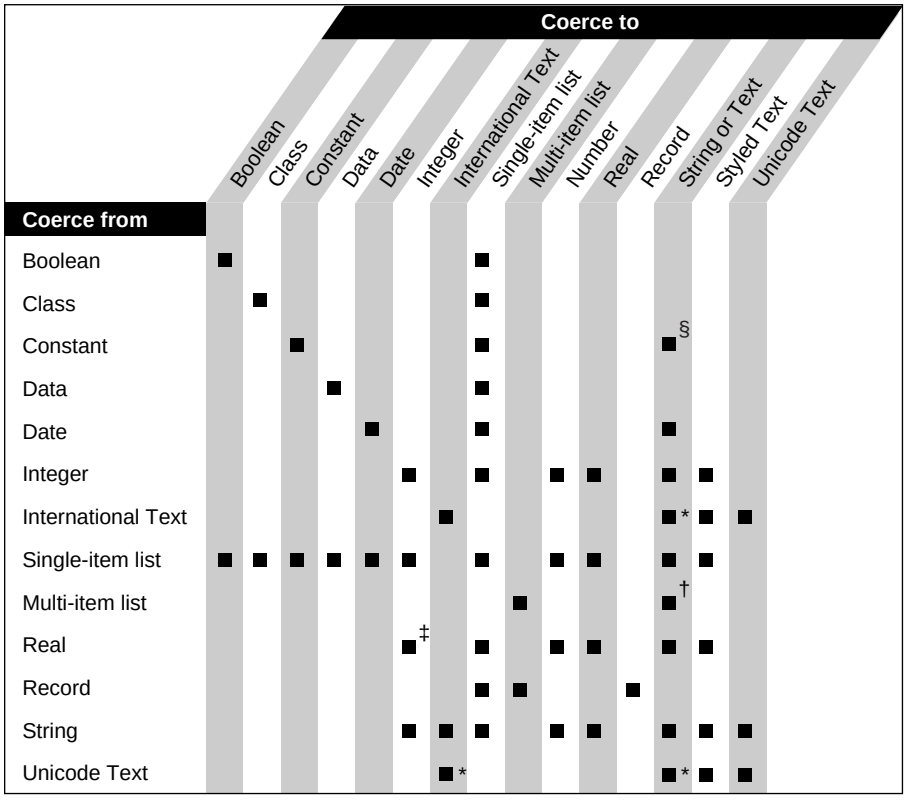
Coercions

Coercions are described in detail in “Coercing Values” (page 97).

Figure A-1 summarizes the coercions that AppleScript supports for commonly-used value classes. To use the figure, find the class of the value to be coerced in the column at the left. Search across the table to the column labeled with the class to which you want to coerce the value. If there is a square at the intersection, then AppleScript supports the coercion.

Three of the identifiers mentioned at the top of Figure A-1 act only as synonyms for other value classes: “Number” is a synonym for either “Integer” or “Real,” “Text” is a synonym for “String,” and “Styled Text” is a synonym for a string that contains style and font information. You can coerce values using these synonyms, but the class of the resulting value is always the appropriate value class, not the synonym.

Figure A-1 Coercions supported by AppleScript



* Some information may be lost in performing these coercions.
† Only a list whose items can all be coerced to strings can be coerced to a string.
‡ Only a real value that has no fractional part can be coerced to an integer.
§ Available with AppleScript version 1.3.7.

References

References are described in detail in "References" (page 165).

A reference is a phrase that specifies an object. Table A-5 summarizes the reference forms you can use to specify objects in AppleScript. The first column lists the name of the reference form. The second column lists the syntax for that form.

The following forms specify a reference to an item in a container: Arbitrary Element, Every Element, Filter, Index, Middle, and Relative.

The following forms are rarely used: Arbitrary Element and Middle.

When you use references to specify objects, you can specify a series of containers, each of which is itself a reference, to identify the object uniquely. Table A-6 lists the ways to specify containers.

For examples and more detailed descriptions of the AppleScript reference forms, see Chapter 5, "Objects and References."

Table A-5 Reference form syntax

Reference form	Syntax
Arbitrary Element	some *className*
Every Element	every *className*
	pluralClassName
Filter	*referenceToObject* whose *Boolean*
	referenceToObject where *Boolean*
ID	*className* ID *IDvalue*
Index	*className integer*
	className index *integer* (positive *integer* indicates position relative to beginning of container; negative indicates position relative to end of container)

Table A-5 Reference form syntax (continued)

Reference form	Syntax
	first *className*
	second *className*
	third *className*
	fourth *className*
	fifth *className*
	sixth *className*
	seventh *className*
	eighth *className*
	ninth *className*
	tenth *className*
	integer st *className*
	integer nd *className*
	integer rd *className*
	integer th *className*
	last *className*
	front *className*
	back *className*
Middle Element	middle *className*
Name	*className string*
	className named *string*
Property	*propertyLabel*
Range	every *className* from *reference* to *reference*
	pluralClassName from *reference* to *reference*

Table A-5 Reference form syntax (continued)

Reference form	Syntax
	className integer through *integer*
	className integer thru *integer*
	pluralClassName integer through *integer*
	pluralClassName integer thru *integer*
Relative	*className* before *reference*
	className front of *reference*
	className in front of *reference*
	className after *reference*
	className back of *reference*
	className in back of *reference*
(insertion point)	beginning of *reference*
	end of *reference*

Table A-6 Container notation in references

Container notation	Syntax
in	*reference* in *containerReference*
of	*reference* of *containerReference*
's	*containerReference's reference*

Operators

Operators are described in detail in "Operations" (page 213).

Table A-7 summarizes the operators AppleScript supports. The first column lists the operators. The second column shows the syntax for using the operators in expressions. The placeholders in the syntax descriptions correspond to AppleScript value classes, which are described briefly in "Placeholders" (page 380), and in more detail in Chapter 3, "Values and Constants."

Synonyms are listed in groups. The table shows the syntax for the first operator, but operators that are synonyms follow the same syntax rules.

Table A-7 Operators

Operator	Syntax
Arithmetic operators	
*	*number* * *number*
+	*number* + *number* *date* + *number*
-	*number* - *number* *date* - *number* *date* – *date*
÷ /	*number* (÷ | /) *number*
^	*number* ^ *number*
div	*number* div *number*
mod	*number* mod *number*
Logical operators	
and	*Boolean* and *Boolean*
not	not *Boolean*
or	*Boolean* or *Boolean*
Containment operators	

Table A-7 Operators (continued)

Operator	Syntax
start[s] with begin[s] with	*list* starts with *list* *string* starts with *string*
end[s] with	*list* ends with *list* *string* ends with *string*
contains	*list* contains *list* *record* contains *record* *string* contains *string*
does not contain doesn't contain	*list* does not contain *list* *record* does not contain *record* *string* does not contain *string*
is in is contained by	*list* is in *list* *record* is in *record* *string* is in *string*
is not in is not contained by isn't contained by	*list* is not in *list* *record* is not in *record* *string* is not in *string*
Comparison operators (equality and inequality)	
= equal equals equal to is is equal to	*expression* = *expression*

Table A-7 Operators (continued)

Operator	Syntax
≠ does not equal doesn't equal is not is not equal [to] isn't isn't equal [to]	*expression* ≠ *expression*
Comparison operators (precedence)	
< comes before is less than is not greater than or equal [to] isn't greater than or equal [to] less than	*date* < *date* *integer* < *integer* *real* < *real* *string* < *string*
> comes after greater than is greater than is not less than or equal [to] isn't less than or equal [to]	*date* > *date* *integer* > *integer* *real* > *real* *string* > *string*
≤ <= does not come after doesn't come after is less than or equal [to] is not greater than isn't greater than less than or equal [to]	*date* ≤ *date* *integer* ≤ *integer* *real* ≤ *real* *string* ≤ *string*

Table A-7 Operators (continued)

Operator	Syntax
≥	*date* ≥ *date*
>=	*integer* ≥ *integer*
does not come before	*real* ≥ *real*
doesn't come before	*string* ≥ *string*
greater than or equal [to]	
is greater than or equal [to]	
is not less than	
isn't less than	
Miscellaneous operators	
& (concatenation)	*expression* & *expression*
as	*expression* as *className*
a reference to	[a] (ref [to] | reference to) ¬ *reference*

Table A-8 shows the order in which AppleScript performs operations. The column labeled “Associativity” indicates the order in which AppleScript performs operations if there are two or more operations of the same precedence in an expression. The word “none” in the Associativity column indicates that you cannot have multiple consecutive occurrences of the operation in an expression. For example, the expression `3 = 3 = 3` is not legal because the associativity for the equal operator (=) is “none.” The word “unary” indicates that the operator is a unary operator. To evaluate expressions with multiple unary operators of the same order, AppleScript applies the operator closest to

the operand first, then applies the next closest operator, and so on. For example, the expression `not not not true` is evaluated as `not (not (not true))`.

Table A-8 Operator precedence

Order	Operators	Associativity	Type of operator
1	( )	Innermost to outermost	Grouping
2	+ -	Unary	Plus or minus sign for numbers
3	^	Right to left	Exponentiation
4	* / ÷ div mod	Left to right	Multiplication and division
5	+ -	Left to right	Addition and subtraction
6	&	Left to right	Concatenation
7	as	Left to right	Coercion
8	< ≤ > ≥	None	Comparison
9	= ≠	None	Equality and inequality
10	not	Unary	Logical negation
11	and	Left to right	Logical for Boolean values
12	or	Left to right	Logical for Boolean values

Control Statements

Control statements are described in detail in “Control Statements” (page 237).

Control statements are statements that control when and how other statements are executed. Table A-9 summarizes the control statements in the AppleScript English dialect. For more information about control statements, see Chapter 7, “Control Statements.”

Table A-9 Control statements

Control statement	Syntax
tell	tell *referenceToObject* to *statement* tell *referenceToObject* [*statement*]... end [tell]
if	if *Boolean* then *statement* if *Boolean* [then] [*statement*]... [else if *Boolean* [then] [*statement*]...]... [else [*statement*]...] end [if]
repeat	repeat [*statement*]... end [repeat]

Table A-9 Control statements (continued)

Control statement	Syntax
	repeat *integer* [times] [*statement*]... end [repeat]
	repeat while *Boolean* [*statement*]... end [repeat]
	repeat until *Boolean* [*statement*]... end [repeat]
	repeat with *variable* from *integer* to *integer* [by *integer*] [*statement*]... end [repeat]
	repeat with *variable* in *list* [*statement*]... end [repeat]
exit	exit

Table A-9 Control statements (continued)

Control statement	Syntax
try	try [*statement*]... on error¬ [*errorMessageVariable*]¬ [number *errorNumberVariable*]¬ [from *offendingObjectVariable*]¬ [partial result *resultListVariable*]¬ [to *expectedTypeVariable*] [global *variable* [, *variable*]...] [local *variable* [, *variable*]...] [*statement*]... end [error | try]
considering	considering *attribute* [, *attribute* ... and *attribute*] ¬ [but ignoring *attribute* [, *attribute* ... and *attribute*]] [*statement*]... end considering
ignoring	ignoring *attribute* [, *attribute* ... and *attribute*] ¬ [but considering *attribute* [, *attribute* ... and *attribute*]] [*statement*]... end ignoring
with timeout	with timeout [of] *integer* second[s] [*statement*]... end [timeout]
with transaction	with transaction [*session*] [*statement*]... end [transaction]

Handlers

Handlers are collections of statements that are executed in response to commands or error messages. Table A-10 summarizes handler definitions and subroutine calls. Handlers are described in detail in "Handlers" (page 279).

Table A-10 Handler definitions and calls

Handler	Syntax
Subroutine definition (labeled parameters)	(on | to) *subroutineName*¬ [of | in *directParameterVariable*]¬ [*subroutineParamLabel paramVariable*] ...¬ [given *label*:*paramVariable* [, label:*paramVariable*]...] [global *variable* [, *variable*]...] [local *variable* [, *variable*]...] [*statement*]... end [*subroutineName*]
Subroutine call (labeled parameters)	*subroutineName*¬ [(of | in) *directParameter*]¬ [*subroutineParamLabel parameterValue*]¬ | [with *labelForTrueParam* [, *labelForTrueParam*]...¬ [(and | or | ,) *labelForTrueParam*]]¬ | [without *labelForFalseParam* [, *labelForFalseParam*]...¬ [(and | or | ,) *labelForFalseParam*]]¬ | [given *label*:*parameterValue*¬ [, *label*:*parameterValue*]...] ...
Subroutine definition (positional parameters)	(on | to) *subroutineName* **(** [*paramVariable* [, *paramVariable*]...] **)** [global *variable* [, *variable*]...] [local *variable* [, *variable*]...] [*statement*]... end [*subroutineName*]

Table A-10 Handler definitions and calls (continued)

Handler	Syntax
Subroutine call (positional parameters)	*subroutineName* **(** [*parameterValue* [, *parameterValue*]...] **)**
Return statement	return *expression*
Command handler definition	(on | to) *commandName*¬ [[of] *directParameterVariable*]¬ [[given] *label*:*paramVariable* [, *label*:*paramVariable*]...] [global *variable* [, *variable*]...] [local *variable* [, *variable*]...] [*statement*]... end [*commandName*]

Script Objects

Script objects are user-defined objects. Table A-11 summarizes the syntax for defining script objects in AppleScript. For more information about script objects, see Chapter 9, "Script Objects."

Table A-11 Script objects

Script object element	Syntax
Script object definition	script [*scriptObjectVariable*] [(property | prop) *propertyLabel* : *expression*]... [*handlerDefinition*]... [*statement*]... end [script]
Continue statement (to pass a command to a handler in the parent script object)	continue *commandStatement*

Variable and Property Assignments and Declarations

Table A-12 summarizes the syntax for assigning values to variables and script properties and declaring local and global variables. For information about variables and script properties, see Chapter 3, "Values and Constants." For

detailed information about the scope of script variables and properties, see "Scope of Script Variables and Properties" (page 311).

Table A-12 Assignments and declarations

Assignment or declaration	Syntax
Variable assignment (and declaration if variable has not previously been declared)	copy *expression* to *variable* copy *reference* to *variable* set *variable* to *expression* set *variable* to *reference*
Global variable declaration	global *variable* [, *variable*]...
Local variable declaration	local *variable* [, *variable*]...
Script property declaration and assignment	property *propertyLabel* : *expression* prop *propertyLabel* : *expression*

The Text Item Delimiters property, which is the only property you can get and set using the global variable `AppleScript`, consists of a list of the delimiters used by AppleScript when coercing lists to strings and when getting text items from strings. This property is declared by AppleScript and is available from any script. You can get and set this property with statements such as the following:

```
set savedDelimiters to AppleScript's text item delimiters
set AppleScript's text item delimiters to {":"}
get last text item of "Hard Disk:CD Contents:Release Notes"
--result: "Release Notes"
set AppleScript's text item delimiters to savedDelimiters
```

Currently, only the first delimiter in the list is used by AppleScript. For more information, see "AppleScript Properties" (page 210).

Placeholders

Table A-13 explains the placeholders used in the syntax descriptions in this book.

Table A-13 Placeholders used in syntax descriptions

Placeholder	Explanation
applicationName	A string containing the name of the application as it would be listed in the Application menu, or a string of the form "*Disk*:*Folder1*:*Folder2*:. . . .:*ApplicationName*" that specifies where the application is stored. For more information, see "References to Applications" (page 194).
attribute	An attribute, identified by a constant, that can be considered or ignored in a Considering or Ignoring control statement. The constants for attributes are `case`, `white space`, `diacriticals`, `hyphens`, `expansion`, `punctuation`, and `application responses`.
Boolean	An expression that evaluates to `true` or `false`. Boolean is an AppleScript value class. For more information, see "Boolean" (page 58).
className	A class identifier or an expression that evaluates to an object class identifier.
commandName	An identifier (name) for a command.
commandStatement	A statement, consisting of a command with either parameter values or formal parameters, to be passed to a parent script object.
compositeValue	A value that contains other values. AppleScript has three types of composite values: lists, records, and strings.
containerReference	A reference that specifies a container for another object.
dataValue	An expression that evaluates to a value of the appropriate class for the object being created.
date	An expression that evaluates to a date. Date is an AppleScript value class. For more information, see "Date" (page 62).

Table A-13 Placeholders used in syntax descriptions (continued)

Placeholder	Explanation
directParameter	The direct parameter of a subroutine definition.
directParameterVariable	A parameter variable used as a placeholder for the value of the direct parameter in a subroutine definition.
errorMessage	An expression, usually a string, that describes an error.
errorMessageVariable	A parameter variable for the expression that describes the error.
errorNumber	The error number for the error.
errorNumberVariable	A parameter variable for the error number.
expectedType	A class identifier for the value class to which AppleScript was attempting to coerce a value when an error occurred.
expectedTypeVariable	A parameter variable for the value class to which AppleScript was attempting to coerce a value when an error occurred.
expression	A series of AppleScript words whose value is a Boolean, class identifier, constant, data, date, integer, list, real, record, reference, script object, or string.
handlerDefinition	A command or subroutine handler definition.
IDvalue	An expression that evaluates to an object's ID property. For most objects, the ID property is an integer.
integer	An expression that evaluates to an integer. Integer is an AppleScript value class. For more information, see "Integer" (page 66).
label	An identifier for a parameter.
labelForFalseParam	An identifier for a Boolean parameter whose value is `false`.
labelForTrueParam	An identifier for a Boolean parameter whose value is `true`.
list	An expression that evaluates to a list.
listOfFiles	A list of references, each of which has the form `file` "*Disk*:*Folder1*:*Folder2*:. . . .:*Filename*" or `alias` "*Disk*:*Folder1*:*Folder2*:. . . .:*Filename*" and specifies a file. For more information, see "References to Files" (page 191).
nameString	A string of the form "*Disk*:*Folder1*:*Folder2*:. . . .:*FileName*" that specifies where a file is stored. For more information, see "References to Files" (page 191).

Table A-13 Placeholders used in syntax descriptions (continued)

Placeholder	Explanation
number	An expression that evaluates to an integer or real number.
offendingObject	A reference to an object that caused an error.
offendingObjectVariable	A parameter variable for the reference to the object that caused an error.
parameterValue	An expression that evaluates to a value of a parameter.
paramVariable	A parameter variable (also known as a formal parameter) used as a placeholder for the value of a parameter in a handler definition.
pluralClassName	A plural class identifier or an expression that evaluates to a plural class identifier.
propertyLabel	The identifier for a property of an object, or an expression that evaluates to the identifier for a property of an object.
propertyValue	An expression that evaluates to a value of the appropriate class for the property being defined.
real	An expression that evaluates to a real number. Real is an AppleScript value class. For more information about real numbers, see "Real" (page 72).
record	An expression that evaluates to a record. Record is an AppleScript value class. For more information about records, see "Record" (page 74).
reference	A reference that specifies an object or location. For more information about references, see Chapter 5, "Objects and References."
referencePattern	A reference, a list of reference patterns, or a record of reference patterns.
referenceToApplication	A reference of the form `application "`*Disk*`:`*Folder1*`:`*Folder2*`:. . . :`*ApplicationName*`"` that specifies an application. For more information, see "References to Applications" (page 194).
referenceToFile	A reference of the form `file "`*Disk*`:`*Folder1*`:`*Folder2*`:. . . :`*Filename*`"` or `alias "`*Disk*`:`*Folder1*`:`*Folder2*`:. . . :`*Filename*`"` that specifies a file. For more information, see "References to Files" (page 191).

Table A-13 Placeholders used in syntax descriptions (continued)

Placeholder	Explanation
referenceToLocation	A reference that specifies a location. For more information about locations, see "Parameters That Specify Locations" (page 119).
referenceToObject	A reference that specifies an object or objects. For more information about references, see Chapter 5, "Objects and References."
resultList	List of results for objects that were handled before an error occurred.
*resultList*Variable	A parameter variable for a list of results for objects that were handled before an error occurred.
saveOption	A constant (`yes`, `no`, or `ask`) that specifies whether to save an object that has been modified before closing it.
scriptObjectVariable	A variable whose value is a script object. For more information about script objects, see Chapter 9, "Script Objects."
session	An object that specifies a specific session.
statement	An AppleScript statement.
string	An expression that evaluates to a string. String is an AppleScript value class. For more information, see "String" (page 80).
subroutineName	An identifier (name) for a subroutine.
subroutineParamLabel	Any of the following labels: `above`, `against`, `apart from`, `around`, `aside from`, `at`, `below`, `beneath`, `beside`, `between`, `by`, `for`, `from`, `instead of`, `into`, `on`, `onto`, `out of`, `over`, `thru` (or `through`), `under`.
timeDifference	An integer specifying a time difference in seconds.
variable	A variable (a user-defined identifier that represents a value).
variablePattern	A variable, a list of variable patterns, or a record of variable patterns.

Error Numbers and Error Messages

See the following sections for error numbers and error messages for the specified types of errors:

- “Operating System Errors” (page 384)
- “Apple Event Errors” (page 385)
- “Application Scripting Errors” (page 387)
- “AppleScript Errors” (page 388)

Operating System Errors

An operating system error is an error that occurs when AppleScript or an application requests services from the Mac OS. They are rare, and more important, there’s usually nothing you can do about them in a script. A few, such as `"File <name> wasn't found"` and `"Application isn't running"`, make sense for scripts to handle.

Error number	Error message
0	No error.
–34	Disk <name> is full.
–35	Disk <name> wasn't found.
–37	Bad name for file.
–38	File <name> wasn't open.
–39	End of file error.
–42	Too many files open.
–43	File <name> wasn't found.
–44	Disk <name> is write protected.
–45	File <name> is locked.
–46	Disk <name> is locked.
–47	File <name> is busy.

Error number	Error message
–48	Duplicate file name.
–49	File <name> is already open.
–50	Parameter error.
–51	File reference number error.
–61	File not open with write permission.
–108	Out of memory.
–120	Folder <name> wasn’t found.
–124	Disk <name> is disconnected.
–128	User canceled.
–192	A resource wasn’t found.
–600	Application isn't running.
–601	Not enough room to launch application with special requirements.
–602	Application is not 32-bit clean.
–605	More memory is needed than is specified in the size resource.
–606	Application is background-only.
–607	Buffer is too small.
–608	No outstanding high-level event.
–609	Connection is invalid.
–904	Not enough system memory to connect to remote application.
–905	Remote access is not allowed.
–906	<name> isn't running or program linking isn't enabled.
–915	Can't find remote machine.
–30720	Invalid date and time <date string>.

Apple Event Errors

An Apple event error is an error that occurs when Apple events sent by AppleScript fail. Many of these errors, such as `"No user interaction allowed"`,

are of interest to users. Also of interest to users are errors that have to do with reference forms, as well as errors such as `"No such object"`.

Error number	Error message
–1700	Can't make some data into the expected type.
–1701	Some parameter is missing for <commandName>.
–1702	Some data could not be read.
–1703	Some data was the wrong type.
–1704	Some parameter was invalid.
–1705	Operation involving a list item failed.
–1706	Need a newer version of the Apple Event Manager.
–1707	Event isn't an Apple event.
–1708	<reference> doesn't understand the <commandName> message.
–1709	AEResetTimer was passed an invalid reply.
–1710	Invalid sending mode was passed.
–1711	User canceled out of wait loop for reply or receipt.
–1712	Apple event timed out.
–1713	No user interaction allowed.
–1714	Wrong keyword for a special function.
–1715	Some parameter wasn't understood.
–1716	Unknown Apple event address type.
–1717	The handler <identifier> is not defined.
–1718	Reply has not yet arrived.
–1719	Can't get <reference>. Invalid index.
–1720	Invalid range.
–1721	<expression> doesn't match the parameters <parameterNames> for <commandName>.
–1723	Can't get <expression>. Access not allowed.
–1725	Illegal logical operator called.
–1726	Illegal comparison or logical.
–1727	Expected a reference.

Error number	Error message
–1728	Can't get <reference>.
–1729	Object counting procedure returned a negative count.
–1730	Container specified was an empty list.
–1731	Unknown object type.
–1750	Scripting component error.
–1751	Invalid script id.
–1752	Script doesn't seem to belong to AppleScript.
–1753	Script error.
–1754	Invalid selector given.
–1755	Invalid access.
–1756	Source not available.
–1757	No such dialect.
–1758	Data couldn't be read because its format is obsolete.
–1759	Data couldn't be read because its format is too new.
–1760	Recording is already on.

Application Scripting Errors

An application scripting error is an error returned by an application when handling standard AppleScript commands (commands that apply to all applications). Many of these errors, such as `"The specified object is a property, not an element"`, are of interest to users and should be handled. Developers can define new application errors in the -10,000 range for their applications.

Error number	Error message
–10000	Apple event handler failed.
–10001	A descriptor type mismatch occurred.

–10002	Invalid key form.
–10003	Can't set <object or data> to <object or data>. Access not allowed.
–10004	A privilege violation occurred.
–10005	The read operation wasn't allowed.
–10006	Can't set <object or data> to <object or data>.
–10007	The index of the event is too large to be valid.
–10008	The specified object is a property, not an element.
–10009	Can’t supply the requested descriptor type for the data.
–10010	The Apple event handler can’t handle objects of this class.
–10011	Couldn’t handle this command because it wasn’t part of the current transaction.
–10012	The transaction to which this command belonged isn’t a valid transaction.
–10013	There is no user selection.
–10014	Handler only handles single objects.
–10015	Can’t undo the previous Apple event or user action.

AppleScript Errors

An AppleScript error is an error that occurs when AppleScript processes script statements. Nearly all of these are of interest to users. For errors returned by an application, see the documentation for that application.

Error number	Error message
-2700	Unknown error.
–2701	Can’t divide <number> by zero.
–2702	The result of a numeric operation was too large.
–2703	<reference> can’t be launched because it is not an application.
–2704	<reference> isn’t scriptable.
–2705	The application has a corrupted dictionary.
–2706	Stack overflow.
–2707	Internal table overflow.

–2708	Attempt to create a value larger than the allowable size.
–2709	Can't get the event dictionary.
–2720	Can't both consider and ignore <attribute>.
–2721	Can't perform operation on text longer than 32K bytes.
–2729	Message size too large for the 7.0 Finder.
–2740	A <language element> can't go after this <language element>.
–2741	Expected <language element> but found <language element>.
–2750	The <name> parameter is specified more than once.
–2751	The <name> property is specified more than once.
–2752	The <name> handler is specified more than once.
–2753	The variable <name> is not defined.
–2754	Can't declare <name> as both a local and global variable.
–2755	Exit statement was not in a repeat loop.
–2760	Tell statements are nested too deeply.
–2761	<name> is illegal as a formal parameter.
–2762	<name> is not a parameter name for the event <event>.
–2763	No result was returned for some argument of this expression.

Document Revision History

This document has had the following releases.

Table B-1 *AppleScript Language Guide* document revision history

May 5, 1999	This release covers AppleScript features through version 1.3.7. Added new sample scripts and revised existing scripts. Examples now use the Mac OS, the Finder, and scriptable applications such as AppleWorks (available from Apple Computer, Inc.) Removed all reference to the Scriptable Text Editor (a demo application that is no longer supported). Revised and updated illustrations and added new illustrations. Corrected errors and deleted outdated material. Reformatted to provide improved online viewing. **Chapter 1, "Introduction."** Chapter 1, "Introduction," replaces the Preface and provides a brief overview of the target audience and the material covered in the Language Guide. **Chapter 2, "Overview of AppleScript."** Chapter 2, "Overview of AppleScript," combines the former Chapters 1 and 2, which together provide an introduction to the AppleScript Language. Added section "Debugging Scripts" (page 47) that describes simple techniques for debugging scripts and points to a website with information on commercial script debugger applications. Revised existing illustrations and added a new illustration,Figure 2-6 (page 35), that shows a portion of the Finder application's dictionary. **Chapter 3, "Values and Constants."** Date class description, "Date" (page 62), now includes Day property; Date String and Time String properties replace Date property.

Table B-1 *AppleScript Language Guide* document revision history (continued)

	Added a section, "Unicode Text and International Text" (page 87), that describes new value classes for working with Unicode Text and International Text
	Added illustrations that show the format for the new text value classes, Figure 3-1 (page 88), and how the Script Editor displays text data for these classes, Figure 3-2 (page 89).
	Added section, "Unit Type Value Classes" (page 91), that describes the new value classes for working with units of length, area, cubic and liquid volume, mass, and temperature.
	Added a section, "Other Value Classes" (page 94), that describes value classes for working with file specifications, RGB colors, and styled Clipboard text.
	Updated the table in Figure 3-3 (page 99) showing the coercions AppleScript supports.
	Added a section, "Constants" (page 100), that categorizes AppleScript constants, shows how to use them in scripts, and provides links to related information.
Chapter 4, "Commands."	
	Added a section on name conflicts, "Name Conflicts" (page 114), to describe conflicts in the AppleScript name space.
	Expanded section "Double Angle Brackets in Results and Scripts" (page 123); double angle brackets are used to display raw data, and in certain other situations.
	Expanded section, "Using Results" (page 121), on using results of commands.
	Deleted description of Data Size command, which is rarely used.
	Deleted description of the application version of the Copy command. (To copy within an application, use the command "Duplicate" (page 138). To copy between applications, use the Clipboard-related scripting addition commands.)
	Added a description of the "Reopen" (page 152) command.

Table B-1 *AppleScript Language Guide* document revision history (continued)

Chapter 5, "Objects and References."

Added tables showing dictionary definitions from the Finder, Table 5-1 (page 163), and from AppleWorks, Table 5-2 (page 164).

Expanded section, "References to Files" (page 191), that describes how to specify files in scripts. Added information on using file specifications.

Chapter 6, "Expressions."

Added information on the sorting order for diacritical marks in the section "Greater Than, Less Than" (page 224).

Added concatenation operator to operator precedence table, Table 6-2 (page 232).

Added section, "Working With Dates at Century Boundaries" (page 235), describing how AppleScript interprets two-digit dates near century boundaries (which encompasses the Y2K issue).

Chapter 7, "Control Statements."

Added section on "Debugging Control Statements" (page 239).

Revised description of possible errors in "Kinds of Errors" (page 259).

Expanded coverage of using the "Error" (page 264) command to signal an error.

Expanded coverage of "With Timeout" (page 273) statements.

Chapter 8, "Handlers."

Reorganized Chapter 8, "Handlers," and renamed chapter sections for consistency.

Added a section on creating "Script Applications" (page 279).

Expanded the section on "Recursive Subroutines" (page 286).

Moved material on declaring variables and properties into a separate section, "Declaring Variables and Properties" (page 312).

Chapter 9, "Script Objects."

No major changes.

Table B-1 *AppleScript Language Guide* document revision history (continued)

Appendix A, "The Language at a Glance" (page 349)

Reordered sections and tables.

Added a table, Table A-1 (page 350), that provides links to examples for common scripting tasks.

In Table A-2 (page 355), added constants for working with days, months, and time, and removed constants for alignment, from table of constants.

In Table A-3 (page 358), added `anything`, `missing value`, and `version` to table of predefined variables.

In Table A-4 (page 360), deleted the application Copy and Data Size commands, added the Reopen command, and simplified the syntax for the Make command in the command syntax table.

Added a section "Coercions" (page 363), with a table of AppleScript-supported coercions.

In Table A-5 (page 365), added insertion point reference forms to reference form syntax.

In Table A-8 (page 372), added table of operator precedence.

Moved section "Error Numbers and Error Messages" (page 384) from former Appendix B. Reorganized errors and added some that were not in the previous version.

Appendix B, "Document Revision History"

Added this revision history

1993 This was the first release of the AppleScript Language Guide.

Glossary

Apple event A high-level message that adheres to the interprocess messaging protocol on which AppleScript is based.

AppleScript A scripting language defined by Apple Computer, Inc., that allows you to control Macintosh computers without using the keyboard or mouse.

AppleScript command A command handled by AppleScript. AppleScript commands do not have to be included in Tell statements.

application command A command handled by an application or its objects. An application command must either be included in a Tell statement or include the name of the application in its direct parameter.

application object An object stored in an application or its documents and managed by the application.

Arbitrary Element reference form A reference form that specifies an arbitrary object in a container. If the container is a value, AppleScript uses a random-number generator to choose the object. If the container is an application object, the application chooses the object.

assignment statement A statement that assigns a value to a variable. Assignment statements begin with Set or Copy.

attachable application An application that can be customized by attaching scripts to specific objects in the application, such as buttons and menu items.

attaching a script to an application object The process of associating a script with a specific application object. Each application determines which, if any, of its objects can have scripts attached.

attribute A characteristic that can be considered or ignored in a Considering or Ignoring statement.

binary operator An operator that derives a new value from a pair of values.

Boolean A logical truth value. The two possible Boolean values are `true` and `false`. Boolean is an AppleScript value class.

Boolean expression An expression whose value can be either `true` or `false`.

chevrons See **double angle brackets.**

child script object A script object that inherits properties and handlers from another script object, called the parent.

Class The name of the AppleScript value class for a class identifier, a reserved word that specifies the class to which an object or value belongs. See also **object class, value class.**

coercion The process of converting a value from one class to another. For example, an integer value can be coerced into a real value. Also, the software that performs such a conversion.

command A word or phrase that requests an action. In AppleScript, there are four types of commands: AppleScript commands, application commands, scripting additions, and user-defined commands.

command handler A handler for an application or system command. Command handlers are similar to subroutines, but instead of defining responses to user-defined commands, they define responses to commands, such as Open, Print, or Move, that are sent to applications.

comment Descriptive text that is ignored by AppleScript when a script is executed.

compile In AppleScript, to convert a script from the form typed into a script editor to a form that can be used by AppleScript. The process of compiling a script includes syntax and vocabulary checks. A script is compiled when you first run it and again when you modify it and then run it again, save it, or check its syntax.

compiled script The form to which a script is converted when you compile it. The form of a compiled script is independent of the dialect in which a script is written.

complete reference A reference that has enough information to identify an object or objects uniquely. For a reference to an application object to be complete, its outermost container must be the application itself.

composite value A value that contains other values. AppleScript has three types of composite values: lists, records, and strings.

compound statement A statement that occupies more than one line and contains other statements. A compound statement begins with a reserved word indicating its function and ends with the word `end`.

conditional statement See **If statement.**

Considering statement A control statement that lists a specific set of attributes to be considered when AppleScript performs operations on strings or sends commands to applications.

constant A reserved word with a predefined value. A constant may be defined by AppleScript or by an application in its dictionary. Constant is an AppleScript value class.

container An object that contains one or more other objects, known as elements. You specify containers with the reserved words `of` or `in`.

continuation character A character (¬), entered by typing Option-L, used in the Script Editor to extend a statement to the next line.

control statement A statement that controls when and how one or more other statements are executed. AppleScript control statements include Tell, If, Repeat, Considering and Ignoring, With Timeout, and With Transaction.

current application Either the default target application or whatever application is currently set as a script's parent property.

current directory The folder or volume whose contents you can see when you choose Open, Save, or a related command from an application's File menu.

current script The script from which a user-defined command is executed.

Data An AppleScript value class used for data that do not belong to any of the other AppleScript value classes. In AppleScript, a value that belongs to the class Data can be stored in a variable, but cannot be manipulated.

Date An AppleScript value class used for a value that specifies a time, day of the month, month, and year.

declaration The first occurrence of a variable or property identifier in a script. The form and location of the declaration determine how AppleScript treats the identifier in that script—for example, as a property, global variable, or local variable.

default target The object that receives a command if no object is specified or if the object is incompletely specified in the command. Default targets are specified in Tell statements.

delegation The use of a Continue statement to call a handler in a parent script object or the current application.

dialect A version of the AppleScript language that resembles a specific human language or programming language. As of AppleScript version 1.3, English is the only dialect supported.

dictionary The set of commands, objects, and other words that are understood by a particular application or by a version of the system software. Each application or version of the system software has its own dictionary.

direct parameter The parameter immediately following a command.

double angle brackets Characters («») typically used by AppleScript to enclose raw data. You can enter double angle brackets, also known as chevrons, in a script by typing Option-Backslash and Shift-Option-Backslash.

element An object contained within another object, or a type of object that can be contained in another object. For example, a word object is an element of a paragraph object, but it is possible to have a paragraph with no words.

empty list A list with no items.

error expression An expression, usually a string, that describes an error.

error handler A collection of statements that are executed in response to an error message.

error message A message that is returned by an application, by AppleScript, or by the Macintosh Operating System if an error occurs during the handling of a command.

error number An integer that identifies an error.

evaluation The conversion of an expression to a value.

Every Element reference form A reference form that specifies every object of a particular class in a container.

Exit statement A statement used in the body of a Repeat statement to exit the Repeat statement.

explicit Run handler A handler at the top level of a script or a script object that begins with `on run` and ends with `end`. A single script or script object can include an explicit Run handler or an implicit Run handler, but not both.

expression In AppleScript, any series of words that has a value.

filter A phrase, added to a reference to a system or application object, that specifies elements in a container that match one or more conditions.

Filter reference form A reference form that specifies all objects in a container that match one or more conditions specified in a Boolean expression.

formal parameter See **parameter variable.**

global variable A variable that is available anywhere in the script in which it is defined.

handler A collection of statements that AppleScript executes in response to a command or an error message.

identifier A series of characters that identifies a value or handler in AppleScript. Identifiers are used to name variables, subroutines, parameters, properties, and commands.

ID reference form A reference form that specifies an object by the value of its ID property.

If statement A control statement that contains one or more Boolean expressions whose results determine whether to execute other statements within the If statement.

Ignoring statement A control statement that lists a specific set of attributes to be ignored when AppleScript performs operations on strings or sends commands to applications.

implicit Run handler All the statements at the top level of a script except for property declarations, script object definitions, and other command handlers. A single script or script object can include an explicit Run handler or an implicit Run handler, but not both.

Index reference form A reference form that specifies an object or location by describing its position with respect to the beginning or end of the container.

infinite loop A Repeat statement that does not specify when repetition stops.

inheritance The ability of a child script object to take on the properties and handlers of a parent script object.

initializing a script object The process of creating a script object from the properties and handlers listed in a script object definition. AppleScript creates a script object when it runs a script or handler that contains a script object definition.

insertion point An object class, supported by many applications, that specifies a place where another object or objects can be added.

integer A positive or negative number without a fractional part. In AppleScript, Integer is a value class.

International Text A value class that represents an ordered series of bytes, beginning with a four-byte language code and a four-byte script code which together determine the format of the bytes that follow.

item A value in a list or record. An item is specified by its offset from the beginning or end of the list or record.

labeled parameter A parameter that is identified by a label. See also **positional parameter.**

list An ordered collection of values. Lists are enclosed by braces. The values in a list are separated by commas. List is an AppleScript value class.

literal expression An expression that evaluates to itself.

local variable A variable that is available only in the handler in which it is defined. Variables that are defined within subroutines, command handlers, and error handlers are local unless they are explicitly declared as global variables.

log statement A script statement that reports the value of one or more variables to the Script Editor's Event Log window.

loop A series of statements that is repeated.

loop variable A variable whose value controls the number of times the statements in a Repeat statement are executed.

Middle Element reference form A reference form that specifies the middle object of a particular class in a container.

Name reference form A reference form that specifies an object by the value of its Name property.

nested control statement A control statement that is contained within another control statement.

Number A synonym for the AppleScript value classes Integer and Real.

object An identifiable part of an application, or thing within an application, that can respond to commands by performing actions.

object class A category for objects that share characteristics such as properties and element classes and respond to the same commands.

operand A value from which an operator derives another value.

operation An expression that derives a new value from one or more other values. An operator, such as the addition operator (+), concatenation operator (&), or Contents Of, determines how the new value is derived.

operator An AppleScript language element (a word, series of words, or symbol) used in an expression to derive a value from another value or pair of values.

optional parameter A parameter that need not be included for a command to be successful.

parameter variable An identifier in a subroutine definition that represents the actual value of a parameter when the subroutine is called. Also called **formal parameter**.

parent script object A script object from which another script object, called the child, inherits properties and handlers.

partial reference A reference that does not include enough information to identify an object or objects uniquely. When AppleScript encounters a partial reference, it uses the default object specified in the Tell statement to complete the reference.

positional parameter A subroutine parameter that is identified by the order in which it is listed. In a subroutine call, positional parameters are enclosed in parentheses and separated by commas. They must be listed in the order in which they appear in the corresponding subroutine definition.

property A characteristic of an object that has a single value and is identified by a label. See also **script property.**

Property reference form A reference form that specifies a property of an application object, record or script object.

Range reference form A reference form that specifies a series of objects of the same class in the same container.

raw data Data of type event, property, class, data, preposition, keyform, constant, or script that AppleScript displays between double angle brackets (or chevrons) because it cannot display the data in its native format.

real A number that can include a decimal fraction. Real is an AppleScript value class.

record An unordered collection of properties. Properties within a record are identified by labels that are unique within the record. Record is an AppleScript value class.

recordable application An application that uses Apple events to report user actions for recording purposes. When recording is turned on, the Script Editor creates statements corresponding to any significant actions you perform in a recordable application.

recursive subroutine A subroutine that calls itself.

reference A phrase that specifies one or more objects using the reference forms defined by AppleScript. Reference is an AppleScript value class.

reference form The syntax for referring to objects. AppleScript defines reference forms for Arbitrary Element, Every Element, Filter, ID, Index, Middle Element, Name, Property, Range, and Relative.

Relative reference form A reference form that specifies an object or location by describing its position in relation to another object, known as the base, in the same container.

Repeat statement A control statement that contains a series of statements to be repeated and, in most cases, instructions that specify when the repetition stops.

required parameter A parameter that must be included for a command to be successful.

reserved words The words in system and application dictionaries, including object and command names, constants, parameters, and properties.

result A value generated when a command is executed or an expression evaluated.

scope The range over which AppleScript recognizes a variable or property, which determines where else in a script you may refer to the same variable or property. The scope of a variable depends on where you declare it and whether you declare it as global or local. The scope of a property extends to the entire script or script object in which it is declared.

script A series of written instructions that, when executed, cause actions in applications and on the desktop.

scriptable application An application that can respond to application commands sent to it when an application such as Script Editor runs a script.

script application An application whose only function is to run the script associated with it.

script code A constant that identifies a particular script system for use on Macintosh computers.

script editor An application used to create and modify scripts.

Script Editor The script-editing application distributed with AppleScript.

scripting addition A file that provides additional commands or coercions you can use in scripts. Each scripting addition contains one or more command handlers. If a scripting addition is located in the Scripting Additions folder (in the Extensions folder of the System Folder), the command handlers it provides are available for use by any script whose target is an application on that computer.

script object A user-defined object in a script that combines data (in the form of properties) and potential actions (in the form of handlers).

script object definition A compound statement that contains a collection of properties, handlers, and other AppleScript statements. A script object definition begins with the reserved word `script`, followed by an optional variable name, and ends with the keyword `end` (or `end script`).

script property A labeled container in which to store a value. Script properties are similar to variables, but they are persistent. Unlike variable values, script property values are saved when you save a script.

script system A collection of system software facilities that allow for the visual representation of a particular writing system. Script systems include Roman, Japanese, Hebrew, Greek, and Thai.

simple statement A statement that is contained on a single line and ends with a return character. See also **compound statement.**

simple value A value, such as an integer or a constant, that does not contain other values.

statement A series of AppleScript words, similar to an English sentence, that contains a request for an action or an expression to be evaluated. See also **compound statement, simple statement.**

string An ordered series of characters (a character string). String is an AppleScript value class.

Styled Text A synonym for the AppleScript value class String. A string referred to as Styled Text may include style and font information.

subroutine A collection of statements that is executed in response to a user-defined command.

suite A listing of the Apple event constructs (object class definitions, descriptor types, and so on) needed for performing a particular type of scriptable activity.

synonym An AppleScript word, phrase, or language element that has the same meaning as another AppleScript word, phrase, or language element. For example, the operator `does not equal` is a synonym for ≠.

syntax The arrangement of words in an AppleScript statement.

syntax description A template for using a command or control statement in a script.

system object An object that is part of a scriptable element of the Mac OS, such as a theme object from the Appearance control panel.

target The recipient of a command. Potential targets include application objects, script objects, the current script, and the current application.

Tell statement A control statement that specifies the default target for the statements it contains.

test A Boolean expression that specifies the conditions of a filter or an If statement.

Text A synonym for the AppleScript value class String.

Try statement A two-part compound statement that contains a series of AppleScript statements, followed by an error handler to be invoked if any of those statements cause an error.

unary operator An operator that derives a new value from a single value.

Unicode An international standard that uses a 16-bit encoding to uniquely specify the characters and symbols for all commonly used languages.

Unicode Text A value class that represents an ordered series of two-byte Unicode characters.

user-defined command A command that triggers the execution of a collection of statements, called a subroutine, elsewhere in the same script.

value A type of data that can be manipulated by and stored in scripts. The AppleScript value classes include Boolean, Class, Constant, Data, Date, Integer, List, Real, Record, Reference, String, Unicode Text, and so on.

value class A category of values with similar characteristics. Values that belong to the same class respond to the same operators.

variable A named container in which to store a value.

With Timeout statement A control statement that allows you to change the amount of time AppleScript waits for application commands to complete before stopping execution of the script.

With Transaction statement A control statement that allows you to take advantage of applications that support the notion of a transaction—a sequence of related events that should be performed as if they were a single operation.

Index

Symbols

A

B

C

D

E

F

G

H

I, J, K

L

M

N

O

P

Q

R

S

T

U

V

W, X

Y

Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

LEAD WRITER
Steve Evangelou

WRITERS
Sean Cotter, Mitchell Gass, Pegi Wheeler

DEVELOPMENTAL EDITORS
Jeanne Woodward, Beverly Zegarski

ILLUSTRATORS
Ruth Anderson, Deborah Dennis, Karin Stroud

PRODUCTION EDITORS
Lorraine Findlay, Gerri Gray

Special thanks to William Cook, Chris Espinosa, Warren Harris, Chris Nebel, Richard Nygord, and Sal Soghoian.

Acknowledgments to Andy Bachorski, Kathleen Carter, Dan Clifford, Sue Dumont, Tony Francis, Ron Karr, Donna Lee, Kazuhisa Ohta, Donald Olson, Jon Pugh, Laurel Rezeau, Brett Sher, Steve Smith, Peter Sparks, Jason Yeo, and the entire AppleScript team.