

---

# AppleScript Language Guide

[Scripting & Automation](#) > [AppleScript](#)



2008-03-11



Apple Inc.  
© 2008 Apple Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, AppleScript, AppleScript Studio, AppleShare, AppleTalk, Bonjour, eMac, iTunes, iWork, Leopard, Logic, Mac, Mac OS, and Macintosh are trademarks of Apple Inc., registered in the United States and other countries.

Finder, Numbers, and Spotlight are trademarks of Apple Inc.

Intel and Intel Core are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Simultaneously published in the United States and Canada.

**Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

**IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.**

**THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.**

**Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.**

# Contents

## **Introduction**      **Introduction to AppleScript Language Guide 13**

---

- Who Should Read This Document? 14
- Organization of This Document 14
- Conventions Used in This Guide 15
- See Also 16

## **Chapter 1**      **AppleScript Lexical Conventions 17**

---

- Character Set 17
- Identifiers 18
- Keywords 18
- Comments 19
- The Continuation Character 20
- Literals and Constants 20
  - Boolean 20
  - Constant 20
  - List 21
  - Number 21
  - Record 21
  - Text 21
- Operators 22
- Variables 22
- Expressions 22
- Statements 23
- Commands 23
- Results 24
- Raw Codes 24

## **Chapter 2**      **AppleScript Fundamentals 25**

---

- Script Editor Application 25
- AppleScript and Objects 26
  - What Is in a Script Object 27
  - Properties 28
  - Elements 28
- Object Specifiers 29
  - What Is in an Object Specifier 29
  - Containers 30
  - Absolute and Relative Object Specifiers 30
  - Object Specifiers in Reference Objects 31
- Coercion (Object Conversion) 32

Scripting Additions	34
Commands Overview	34
Types of Commands	35
Target	35
Direct Parameter	36
Parameters That Specify Locations	36
AppleScript Error Handling	36
Global Constants in AppleScript	37
AppleScript Constant	37
current application Constant	39
missing value Constant	40
true, false Constants	40
The it and me Keywords	40
Aliases and Files	41
Specifying Paths	42
Working With Aliases	42
Working With Files	43
Remote Applications	44
Enabling Remote Applications	44
eppc-Style Specifiers	44
Targeting Remote Applications	44
Debugging AppleScript Scripts	45
Feedback From Your Script	45
Logging	46
Third Party Debuggers	46

### Chapter 3 **Variables and Properties** 47

---

Defining Properties	47
Declaring Variables	48
Local Variables	48
Global Variables	49
Using the copy and set Commands	49
Scope of Variables and Properties	51
Scope of Properties and Variables Declared in a Script Object	52
Scope of Variables Declared in a Handler	55

### Chapter 4 **Script Objects** 57

---

Defining Script Objects	57
Initializing Script Objects	59
Sending Commands to Script Objects	59
Inheritance in Script Objects	60
The AppleScript Inheritance Chain	60
Defining Inheritance Through the parent Property	61
Some Examples of Inheritance	61

Using the continue Statement in Script Objects 63

---

**Chapter 5 About Handlers 67**

---

Handler Basics 67

- Defining a Simple Handler 68
- Handlers with Labeled Parameters 68
- Handlers with Positional Parameters 69
- Handlers with Patterned Positional Parameters 70
- Recursive Handlers 71
- Errors in Handlers 71
- Passing by Reference Versus Passing by Value 71
- Calling Handlers in a tell Statement 72

Saving and Loading Libraries of Handlers 72

Handlers in Script Applications 73

- run Handlers 74
- open Handlers 75
- idle and quit Handlers for Stay-Open Applications 76

Calling a Script Application From a Script 77

---

**Chapter 6 Class Reference 79**

---

alias 79

application 80

boolean 82

class 83

constant 84

date 85

file 88

integer 89

list 89

number 92

POSIX file 93

real 93

record 94

reference 95

RGB color 96

script 97

text 97

unit types 103

---

**Chapter 7 Commands Reference 105**

---

activate 108

ASCII character 108

ASCII number 109

- beep 110
- choose application 110
- choose color 111
- choose file 112
- choose file name 114
- choose folder 115
- choose from list 116
- choose remote application 118
- choose URL 119
- clipboard info 120
- close access 120
- copy 121
- count 122
- current date 123
- delay 123
- display alert 124
- display dialog 125
- do shell script 128
- get 129
- get eof 131
- get volume settings 131
- info for 132
- launch 134
- list disks 134
- list folder 135
- load script 135
- localized string 136
- log 138
- mount volume 138
- offset 139
- open for access 140
- open location 141
- path to (application) 142
- path to (folder) 143
- path to resource 145
- random number 146
- read 147
- round 149
- run 151
- run script 151
- say 152
- scripting components 153
- set 154
- set eof 155
- set the clipboard to 156
- set volume 157

- store script 158
- summarize 159
- system attribute 160
- system info 161
- the clipboard 162
- time to GMT 163
- write 163

## Chapter 8 **Reference Forms 167**

---

- Arbitrary 167
- Every 168
- Filter 169
- ID 171
- Index 172
- Middle 173
- Name 173
- Property 174
- Range 175
- Relative 176

## Chapter 9 **Operators Reference 179**

---

- & (concatenation) 187
  - text 187
  - record 187
  - All Other Classes 187
- a reference to 188
  - Examples 188
- contains, is contained by 189
  - list 189
  - record 189
  - text 190
- equal, is not equal to 190
  - list 190
  - record 190
  - text 191
- greater than, less than 191
  - date 191
  - integer, real 191
  - text 191
- starts with, ends with 192
  - list 192
  - text 192

## Chapter 10 **Control Statements Reference 193**

---

- considering and ignoring Statements 193
  - considering / ignoring (text comparison) 193
  - considering / ignoring (application responses) 195
- error Statements 196
  - error 196
- if Statements 197
  - if (simple) 198
  - if (compound) 198
- repeat Statements 199
  - exit 199
  - repeat (forever) 199
  - repeat (number) times 200
  - repeat until 201
  - repeat while 202
  - repeat with loopVariable (from startValue to stopValue) 202
  - repeat with loopVariable (in list) 203
- tell Statements 205
  - tell (simple) 206
  - tell (compound) 206
- try Statements 207
  - try 207
- using terms from Statements 209
  - using terms from 209
- with timeout Statements 210
  - with timeout 210
- with transaction Statements 211
  - with transaction 211

## Chapter 11 **Handler Reference 213**

---

- continue 213
- return 214
- Handler Syntax (Labeled Parameters) 215
- Calling a Handler with Labeled Parameters 216
- Handler Syntax (Positional Parameters) 218
- Calling a Handler with Positional Parameters 218

## Chapter 12 **Folder Actions Reference 221**

---

- adding folder items to 222
- closing folder window for 223
- moving folder window for 223
- opening folder 224
- removing folder items from 225



**Appendix A      [AppleScript Keywords](#) 227**

---

**Appendix B      [Error Numbers and Error Messages](#) 233**

---

[AppleScript Errors](#) 233  
[Operating System Errors](#) 234

**Appendix C      [Working with Errors](#) 237**

---

[Catching Errors in a Handler](#) 237  
[Simplified Error Checking](#) 238

**Appendix D      [Double Angle Brackets](#) 241**

---

[When a Dictionary Is Not Available](#) 241  
[When AppleScript Displays Data in Raw Format](#) 241  
[Entering Script Information in Raw Format](#) 242  
[Sending Raw Apple Events From a Script](#) 242

**Appendix E      [Unsupported Terms](#) 243**

---

[List of Unsupported Terms](#) 243

**[Glossary](#) 245**

---

**[Document Revision History](#) 251**

---

**[Index](#) 253**

---



# Figures, Tables, and Listings

---

## Chapter 1 **AppleScript Lexical Conventions** 17

---

Table 1-1 AppleScript reserved words, listed alphabetically 18

---

## Chapter 2 **AppleScript Fundamentals** 25

---

Figure 2-1 The Finder dictionary in Script Editor (in Mac OS X v10.5) 26

Table 2-1 Default coercions supported by AppleScript 32

---

## Chapter 3 **Variables and Properties** 47

---

Table 3-1 Scope of property and variable declarations at the top level in a script object 53

Table 3-2 Scope of variable declarations within a handler 55

---

## Chapter 4 **Script Objects** 57

---

Listing 4-1 A pair of script objects with a simple parent-child relationship 61

---

## Chapter 6 **Class Reference** 79

---

Table 6-1 Special characters in text 100

Table 6-2 White space constants 100

---

## Chapter 7 **Commands Reference** 105

---

Figure 7-1 Bundle structure with localized string data 137

Figure 7-2 Key/value pair for localized string data 137

Table 7-1 AppleScript commands 105

---

## Chapter 8 **Reference Forms** 167

---

Table 8-1 Boolean expressions and tests in filter references 170

---

## Chapter 9 **Operators Reference** 179

---

Table 9-1 AppleScript operators 179

Table 9-2 Operator precedence 186

---

## Appendix A **AppleScript Keywords** 227

---

Table A-1 AppleScript reserved words, with descriptions 227

**Appendix B**      **Error Numbers and Error Messages**    **233**

---

Table B-1	AppleScript errors	233
Table B-2	Mac OS errors	234

# Introduction to AppleScript Language Guide

---

This document is a guide to the AppleScript language—its lexical conventions, syntax, keywords, and other elements. It is intended primarily for use with AppleScript 2.0 or later and Mac OS X version 10.5 or later.

AppleScript 2.0 can use scripts developed for any version of AppleScript from 1.1 through 1.10.7, any scripting addition created for AppleScript 1.5 or later for Mac OS X, and any scriptable application for Mac OS v7.1 or later. A script created with AppleScript 2.0 can be used by any version of AppleScript back to version 1.1, provided it does not use features of AppleScript, scripting additions, or scriptable applications that are unavailable in that version.

**Important:** Descriptions and examples for the terms in this document have been tested with AppleScript 2.0 in Mac OS X v10.5 (Leopard). Except for terms that are noted as being new in Leopard, most descriptions and examples work with previous system versions, but have not been tested against all of them.

If you need detailed information about prior system and AppleScript versions, see *AppleScript Release Notes (Mac OS X 10.4 and earlier)*.

## What Is AppleScript?

AppleScript is a scripting language created by Apple. It allows users to directly control scriptable Macintosh applications, as well as parts of Mac OS X itself. You can create scripts—sets of written instructions—to automate repetitive tasks, combine features from multiple scriptable applications, and create complex workflows.

**Note:** Apple also provides the Automator application, which allows users to automate common tasks by hooking together ready-made actions in a graphical environment. For more information, see Automator Documentation.

A scriptable application is one that can be controlled by a script. For AppleScript, that means being responsive to interapplication messages, called **Apple events**, sent when a script command targets the application. (Apple events can also be sent directly from other applications and Mac OS X.)

AppleScript itself provides a very small number of commands, but it provides a framework into which you can plug many task-specific commands—those provided by scriptable applications and scriptable parts of Mac OS X.

Most script samples and script fragments in this guide use scriptable features of the Finder application, scriptable parts of Mac OS X, or scriptable applications distributed with Mac OS X, such as TextEdit (located in `/Applications`).

## Who Should Read This Document?

You should use this document if you write or modify AppleScript scripts, or if you create scriptable applications and need to know how scripts should work.

*AppleScript Language Guide* assumes you are familiar with the high-level information about AppleScript found in *AppleScript Overview*.

## Organization of This Document

This guide describes the AppleScript language in a series of chapters and appendixes.

The first five chapters introduce components of the language and basic concepts for using it, then provide additional overview on working with script objects and handler routines:

- [“AppleScript Lexical Conventions”](#) (page 17) describes the characters, symbols, keywords, and other language elements that make up statements in an AppleScript script.
- [“AppleScript Fundamentals”](#) (page 25) describes basic concepts that underly the terminology and rules covered in the rest of this guide.
- [“Variables and Properties”](#) (page 47) describes common issues in working with variables and properties, including how to declare them and how AppleScript interprets their scope.
- [“Script Objects”](#) (page 57) describes how to define, initialize, send commands to, and use inheritance with script objects.
- [“About Handlers”](#) (page 67) provides information on using handlers (a type of function available in AppleScript) to factor and reuse code.

The following chapters provide reference for the AppleScript Language:

- [“Class Reference”](#) (page 79) describes the classes AppleScript defines for common objects used in scripts.
- [“Commands Reference”](#) (page 105) describes the commands that are available to any script.
- [“Reference Forms”](#) (page 167) describes the syntax for specifying an object or group of objects in an application or other container.
- [“Operators Reference”](#) (page 179) provides a list of the operators AppleScript supports and the rules for using them, along with sections that provide additional detail for commonly used operators.
- [“Control Statements Reference”](#) (page 193) describes statements that control when and how other statements are executed. It covers standard conditional statements, as well as statements used in error handling and other operations.
- [“Handler Reference”](#) (page 213) shows the syntax for defining and calling handlers and describes other statements you use with handlers.

The following chapter describes an AppleScript-related feature of Mac OS X:

- [“Folder Actions Reference”](#) (page 221) describes how you can write and attach script handlers to specific folders, such that the handlers are invoked when the folders are modified.

The following appendixes provide additional information about the AppleScript language and how to work with errors in scripts:

- [“AppleScript Keywords”](#) (page 227) lists the keywords of the AppleScript language, provides a brief description for each, and points to related information.
- [“Error Numbers and Error Messages”](#) (page 233) describes error numbers and error messages you may see in working with AppleScript scripts.
- [“Working with Errors”](#) (page 237) provides detailed examples of handling errors with [“try Statements”](#) (page 207) and [“error Statements”](#) (page 196).
- [“Double Angle Brackets”](#) (page 241) describes when you are likely to see double angle brackets (or chevrons—«») in scripts and how you can work with them.
- [“Unsupported Terms”](#) (page 243) lists terms that are no longer supported in AppleScript.

## Conventions Used in This Guide

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

**Important:** This document sometimes 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 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.
<i>placeholder</i>	Italic text indicates a placeholder that you replace with an appropriate value.
[optional]	Brackets indicate that the enclosed language element or elements are optional.
(a group)	Parentheses group elements together. However, the parentheses shown in <a href="#">“Handler Syntax (Positional Parameters)”</a> (page 218) are part of the syntax.
[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   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.
Filenames shown in scripts	Most filenames shown in examples in this document include extensions, such as <code>rtf</code> for a TextEdit document. Use of extensions in scripts is generally dependent on the “Show all file extensions” setting in the Advanced pane of Finder Preferences. To work with the examples on your computer, you may need to modify either that setting or the filenames.

## See Also

These Apple documents provide additional information for working with AppleScript:

- See *Getting Started with AppleScript* for a guided quick start, useful to both scripters and developers.
- See *AppleScript Overview*, including the chapter *Scripting With AppleScript*, for a high-level overview of AppleScript and its related technologies.
- See *Getting Started With Scripting & Automation* for information on the universe of scripting technologies available in Mac OS X.
- See [AppleScript Terminology and Apple Event Codes](#) for a list of many of the scripting terms defined by Apple.

For additional information on working with the AppleScript language and creating scripts, see one of the comprehensive third-party documents available in bookstores and online.



# AppleScript Lexical Conventions

---

This chapter provides an overview of the vocabulary and conventions of the AppleScript Language. It starts with the character set and introduces elements of increasing complexity.

After reading this chapter, you should have an understanding of the basic language components used to construct AppleScript expressions and statements.

AppleScript Lexical Conventions contains the following sections:

- [“Character Set”](#) (page 17)
- [“Identifiers”](#) (page 18)
- [“Keywords”](#) (page 18)
- [“Comments”](#) (page 19)
- [“The Continuation Character”](#) (page 20)
- [“Literals and Constants”](#) (page 20)
- [“Operators”](#) (page 22)
- [“Variables”](#) (page 22)
- [“Expressions”](#) (page 22)
- [“Statements”](#) (page 23)
- [“Commands”](#) (page 23)
- [“Results”](#) (page 24)
- [“Raw Codes”](#) (page 24)

## Character Set

Starting in Mac OS X v10.5 (AppleScript 2.0), the character set for AppleScript is Unicode. AppleScript preserves all characters correctly worldwide, and comments and text constants in scripts may contain any Unicode characters.

AppleScript syntax uses several non-ASCII characters, which can be typed using special key combinations. For information on characters that AppleScript treats specially, see the sections [“Identifiers”](#) (page 18), [“Comments”](#) (page 19), [“Text”](#) (page 21), [“The Continuation Character”](#) (page 20), and [“Raw Codes”](#) (page 24) in this chapter, as well as [Table 9-1](#) (page 179) in [“Operators Reference”](#) (page 179).

## Identifiers

An AppleScript **identifier** is a series of characters that identifies a class name, variable, or other language element, such as labels for properties and handlers.

An identifier must begin with a letter and can contain any of these characters:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789\_

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

AppleScript remembers and enforces the first capitalization it comes across for an identifier. So if it first encounters an identifier as `myAccount`, it will later, during compilation, change versions such as `MyAccount` and `myaccount` to `myAccount`.

The following are examples of valid identifiers: `areaOfCircle`, `Agent007`, `axis_of_rotation`.

The following are not valid identifiers: `C-`, `back&forth`, `999`, `Why^Not`.

AppleScript provides a loophole to the preceding rules: identifiers whose first and last characters are vertical bars (`|`) can contain any characters. The leading and trailing vertical bars are not considered part of the identifier.

**Important:** This use of vertical bars can make scripts difficult to read, and is not recommended.

The following are legal identifiers: `|back&forth|`, `|Right*Now!|`.

An identifier can contain additional vertical bars preceded by a backslash (`\`) character, as in the identifier `|This\|Or\|That|`. Use of the backslash character is described further in the Special String Characters section of the `text` (page 97) class.

## Keywords

A **keyword** is a reserved word in the AppleScript language. Keywords consist of lower-case, alphabetic characters: `abcdefghijklmnopqrstuvwxyz`. In a few cases, such as `aside from`, they come in pairs.

**Important:** You should not attempt to reuse keywords in your scripts for variable names or other purposes. Developers should not re-define keywords in the terminology for their scriptable applications.

Table 1-1 lists the keywords reserved in AppleScript 2.0 (which are the same as those used in AppleScript 1.x). For additional information, see [Table A-1](#) (page 227), which provides a brief description for each keyword and points to related information, where available.

**Table 1-1** AppleScript reserved words, listed alphabetically

about	above	after	against	and	apart from
around	as	aside from	at	back	before

beginning	behind	below	beneath	beside	between
but	by	considering	contain	contains	contains
continue	copy	div	does	eighth	else
end	equal	equals	error	every	exit
false	fifth	first	for	fourth	from
front	get	given	global	if	ignoring
in	instead of	into	is	it	its
last	local	me	middle	mod	my
ninth	not	of	on	onto	or
out of	over	prop	property	put	ref
reference	repeat	return	returning	script	second
set	seventh	since	sixth	some	tell
tenth	that	the	then	third	through
thru	timeout	times	to	transaction	true
try	until	where	while	whose	with
without					

## Comments

A **comment** is text that is ignored by AppleScript when a script is executed. You can use comments to describe what is happening in the script or make other kinds of notes. There are three kinds of comments:

- A block comment begins with the characters (`*` and ends with the characters `*`). Block comments must be placed between other statements. That means they can be placed on the same line at the beginning or end of a statement, but cannot be embedded within a simple (one-line) statement.
- An end-of-line comment begins with the characters `--` (two hyphens) and ends with the end of the line:
 

```
--end-of-line comments extend to the end of the line
```
- Starting in version 2.0, AppleScript also supports use of the `#` symbol as an end-of-line comment. This allows you to make a plain AppleScript script into a Unix executable by beginning it with the following line and giving it execute permission:

```
#!/usr/bin/osascript
```

Compiled scripts that use `#` will run normally on pre-2.0 systems, and if edited will display using `--`. Executable text scripts using `#!/usr/bin/osascript` will not run on pre-2.0 systems, since the `#` will be considered a syntax error.

You can nest comments—that is, comments can contain other comments, as in this example:

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

## The Continuation Character

A simple AppleScript statement must normally be entered on a single line. You can extend a statement to the next line by ending it with the **continuation character**, `↵`. With a U.S. keyboard, you can enter this character by typing Option-I (lower-case L). In Script Editor, you can type Option-Return, which inserts the continuation character and moves the insertion point to the next line.

Here is a single statement displayed on two lines:

```
display dialog "This is just a test." buttons {"Great", "OK"} ↵
default button "OK" giving up after 3
```

A continuation character within a quoted text string is treated like any other character.

## Literals and Constants

A **literal** is a value that evaluates to itself—that is, it is interpreted just as it is written. In AppleScript, for example, `"Hello"` is a text literal. A **constant** is a word with a predefined value. For example, AppleScript defines a number of enumerated constants for use with the `path to (folder)` (page 143) command, each of which specifies a location for which to obtain the path.

### Boolean

---

AppleScript defines the Boolean values `true` and `false` and supplies the `boolean` (page 82) class.

### Constant

---

[“Global Constants in AppleScript”](#) (page 37) describes constants that can be used throughout your scripts. For related information, see the `constant` (page 84) class.

## List

---

A list defines an ordered collection of values, known as items, of any class. As depicted in a script, a list consists of a series of expressions contained within braces and separated by commas, such as the following:

```
{1, 7, "Beethoven", 4.5}
```

A list can contain other lists. An empty list (containing no items) is represented by a pair of empty braces: {}.

AppleScript provides the `list` (page 89) class for working with lists.

## Number

---

A numeric literal is a sequence of digits, possibly including other characters, such as a unary minus sign, period (in reals), or "E+" (in exponential notation). The following are some numeric literals:

```
-94596  
3.1415  
9.9999999999E+10
```

AppleScript defines classes for working with `real` (page 93) and `integer` (page 89) values, as well as the `number` class, which serves as a synonym for either `real` or `integer`.

## Record

---

A record is an unordered collection of labeled properties. A record appears in a script as a series of property definitions contained within braces and separated by commas. Each property definition consists of a unique label, a colon, and a value for the property. For example, the following is a record with two properties:

```
{product:"pen", price:2.34}
```

## Text

---

A `text` literal consists of a series of Unicode characters enclosed in a pair of double quote marks, as in the following example:

```
"A basic string."
```

AppleScript `text` objects are instances of the `text` (page 97) class, which provides mechanisms for working with text. The Special String Characters section of that class describes how to use white space, backslash characters, and double quotes in text.

## Operators

An **operator** is a symbol, word, or phrase that derives a value from another value or pair of values. For example, the multiplication operator (`*`) multiplies two numeric operands, while the concatenation operator (`&`) joins two objects (such as text strings). The `is equal` operator performs a test on two Boolean values.

For detailed information on AppleScript's operators, see [“Operators Reference”](#) (page 179).

## Variables

A **variable** is a named container in which to store a value. Its name, which you specify when you create the variable, follows the rules described in [“Identifiers”](#) (page 18). You can declare and initialize a variable at the same time with a `copy` (page 121) or `set` (page 154) command. For example:

```
set myName to "John"  
copy 33 to myAge
```

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

When AppleScript encounters a variable, it evaluates the variable by getting its value. A variable is contained in a script and its value is normally lost when you close the script that contains it.

AppleScript variables can hold values of any class. For example, you can assign the integer value `17` to a variable, then later assign the Boolean value `true` to the same variable.

For more information, see [“Variables and Properties”](#) (page 47).

## Expressions

An **expression** is any series of lexical elements that has a value. Expressions are used in scripts to represent or derive values. The simplest kinds of expressions, called literal expressions, are representations of values in scripts. More complex expressions typically combine literals, variables, operators, and object specifiers.

When you run a script, AppleScript converts its expressions into values. This process is known as **evaluation**. For example, when the following simple expression is evaluated, the result is 21:

```
3 * 7 --result: 21
```

An object specifier specifies some or all of the information needed to find another object. For example, the following object specifier specifies a named document:

```
document named "FavoritesList"
```

For more information, see [“Object Specifiers”](#) (page 29).

## Statements

A **statement** is a series of lexical elements that follows a particular AppleScript syntax. Statements can include keywords, variables, operators, constants, expressions, and so on.

Every script consists of statements. When AppleScript executes a script, it reads the statements in order and carries out their instructions.

A **control statement** is a statement that determines when and how other statements are executed. AppleScript defines standard control statements such as `if`, `repeat`, and `while` statements, which are described in detail in [“Control Statements Reference”](#) (page 193).

A **simple statement** is one that can be written on a single line:

```
set averageTemp to 63 as degrees Fahrenheit
```

**Note:** You can use a continuation character (→) to extend a simple statement onto a second line.

A **compound statement** is written on more than one line, can contain other statements, and has the word `end` (followed, optionally, by the first word of the statement) in its last line. For example the following is a compound `tell` statement:

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

A compound statement can contain other compound statements.

## Commands

A **command** is a word or series of words used in an AppleScript statement 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 or an object in Mac OS X, but it can also be a `script` object or a value in the current script.

The following statement uses AppleScript’s `get` (page 129) command to obtain the name of a window; the target is the front window of the Finder application:

```
get name of front window of application "Finder"
```

For more information on command types, parameters, and targets, see [“Commands Overview”](#) (page 34).

## Results

The **result** of a statement is the value generated, if any, when the statement is executed. For example, executing the statement `3 + 4` results in the value `7`. The result of the statement `set myText to "keyboard"` is the text object `"keyboard"`. A result can be of any class. AppleScript stores the result in the globally available property `result`, described in [“AppleScript Constant”](#) (page 37).

## Raw Codes

When you open, compile, edit, or run scripts with a script editor, you may occasionally see terms enclosed in double angle brackets, or chevrons (`«»`), in a script window or in another window. These terms are called *raw format* or *raw codes*, because they represent the underlying Apple event codes that AppleScript uses to represent scripting terms.

For compatibility with Asian national encodings, `«` and `»` are allowed as synonyms for `«` and `»` (Option-`\` and Option-Shift-`\`, respectively, on a U.S. keyboard), since the latter do not exist in some Asian encodings.

For more information on raw codes, see [“Double Angle Brackets”](#) (page 241).



# AppleScript Fundamentals

---

This chapter describes basic concepts that underlie the terminology and rules covered in the rest of this guide.

- [“Script Editor Application”](#) (page 25)
- [“AppleScript and Objects”](#) (page 26)
- [“Object Specifiers”](#) (page 29)
- [“Coercion \(Object Conversion\)”](#) (page 32)
- [“Scripting Additions”](#) (page 34)
- [“Commands Overview”](#) (page 34)
- [“AppleScript Error Handling”](#) (page 36)
- [“Global Constants in AppleScript”](#) (page 37)
- [“The it and me Keywords”](#) (page 40)
- [“Aliases and Files”](#) (page 41)
- [“Remote Applications”](#) (page 44)
- [“Debugging AppleScript Scripts”](#) (page 45)

## Script Editor Application

The Script Editor application is located in `/Applications/AppleScript`. It provides the ability to edit, compile, and execute scripts, display application scripting terminologies, and save scripts in a variety of formats, such as compiled scripts, applications, and plain text.

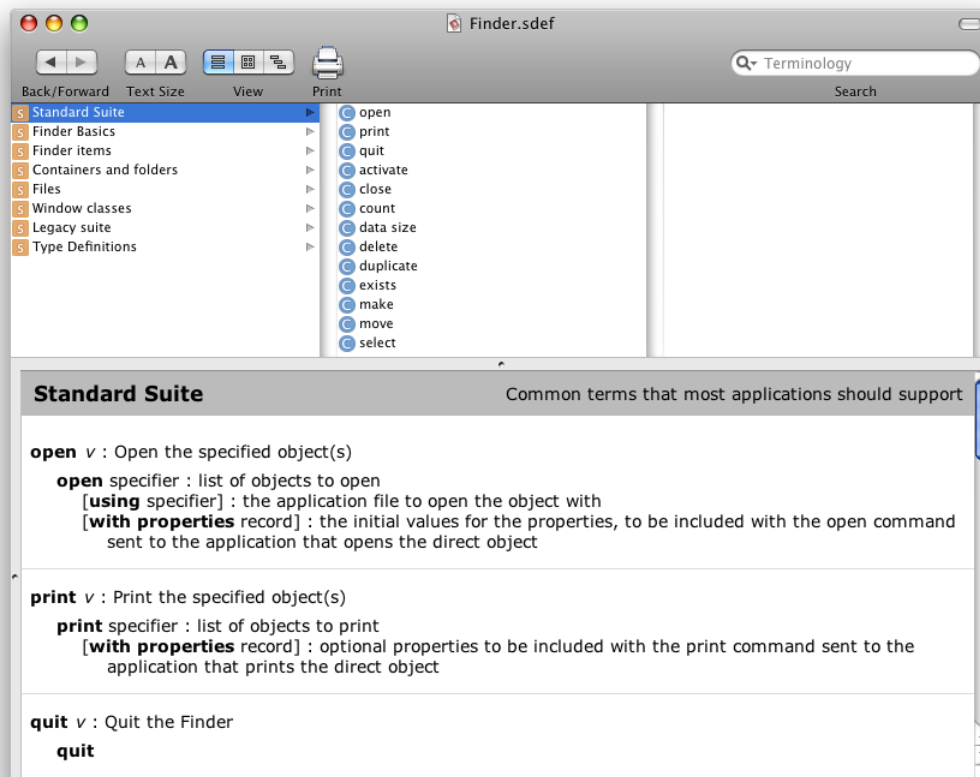
Script Editor can display the result of executing an AppleScript script and can display a log of the Apple events that are sent during execution of a script. In the Script Editor Preferences, you can also choose to keep a history of recent results or event logs.

Script Editor has text formatting preferences for various types of script text, such as language keywords, comments, and so on. You can also turn on or off the Script Assistant, a code completion tool that can suggest and fill in scripting terms as you type. In addition, Script Editor provides a contextual menu to insert many types of boilerplate script statements, such as conditionals, comments, and error handlers.

A **dictionary** is the part of a scriptable application that specifies the scripting terms it understands. You can choose `File > Open Dictionary` in Script Editor to display the dictionary of a scriptable application or scripting addition on your computer. Or you can drag an application icon to the Script Editor icon to display its dictionary (if it has one).

To display a list that includes just the scriptable applications and scripting additions provided by Mac OS X, choose Window > Library. Double-click an item in the list to display its dictionary. Figure 2-1 shows the dictionary for the Finder application in Mac OS X v10.5. The dictionary is labeled as “Finder.sdef”. The sdef format, along with other terminology formats, is described in “Specifying Scripting Terminology” in *AppleScript Overview*.

**Figure 2-1** The Finder dictionary in Script Editor (in Mac OS X v10.5)



There are also third-party editors for AppleScript.

## AppleScript and Objects

AppleScript is an object-oriented language. When you write, compile, and execute scripts, everything you work with is an object. An **object** is an instantiation of a class definition, which can include properties and actions. AppleScript defines classes for the objects you most commonly work with, starting with the top-level `script` (page 97) object, which is the overall script you are working in.

Within in a `script` object, you work with other objects, including:

- AppleScript objects:

AppleScript defines classes for boolean values, scripts, text, numbers, and other kinds of objects for working in scripts; for a complete list, see [“Class Reference”](#) (page 79).

- Mac OS X objects:

Scriptable parts of Mac OS X and applications distributed with it, such as Finder, System Events, and Database Events (located in `/System/Library/CoreServices`), define many useful classes.

- Application objects:

Third-party scriptable applications define classes that support a wide variety of features.

The following sections provide more detail about objects:

- [“What Is in a Script Object”](#) (page 27)
- [“Properties”](#) (page 28)
- [“Elements”](#) (page 28)

## What Is in a Script Object

---

When you enter AppleScript statements in script window in Script Editor, you are working in a top-level `script object`. All `script object` definitions follow the same syntax, except that a top-level `script object` does not have statements marking its beginning and end.

A `script object` can contain the following:

- Property definitions (optional):

A property is a labeled container in which to store a value.

- An explicit `run handler` (optional):

A `run handler` contains statements AppleScript executes when the script is run. (For more information, see [“run Handlers”](#) (page 74).)

- An implicit `run handler` (optional):

An implicit `run handler` consists of any statements outside of any contained handlers or `script objects`.

- Additional handlers (optional):

A handler is the equivalent of a subroutine. (For details, see [“About Handlers”](#) (page 67).)

- Additional `script objects` (optional):

A `script object` can contain nested `script objects`, each of which is defined just like a top-level `script object`, except that a nested `script object` is bracketed with statements that mark its beginning and end. (For details, see [“Script Objects”](#) (page 57).)

Here is an example of a simple script with one property, one handler, one nested `script object`, and an implicit `run handler` with two statements:

```
property defaultClientName : "Mary Smith"

on greetClient(nameOfClient)
    display dialog ("Hello " & nameOfClient & "!")
end greetClient
```

```
script testGreet
    greetClient(defaultClientName)
end script

run testGreet --result: "Hello Mary Smith!"
greetClient("Joe Jones") --result: "Hello Joe Jones!"
```

The first statement in the `run` handler is `run testGreet`, which runs the nested `script` object `testGreet`. That `script` object calls the handler `greetClient()`, passing the property `defaultClientName`. The handler displays a dialog, greeting the default client, Mary Smith.

The second statement in the `run` handler calls `greetClient()` directly, passing the string "Joe Jones".

## Properties

---

A **property** of an object is a characteristic that has a single value and a label, such as the `name` property of a window or the `month` property of a date. The definition for any AppleScript class includes the name and class for each of its properties. Property names must be unique within a class. Property values can be read/write or read only.

The AppleScript `date` (page 85) class, for example, defines both read/write and read only properties. These include the `weekday` property, which is read only, and the `month`, `day`, and `year` properties, which are read/write. That's because the value of the `weekday` property depends on the other properties—you can't set an arbitrary weekday for an actual date.

The class of a property can be a simple class such as `boolean` (page 82) or `integer` (page 89), a composite class such as a `point` class (made up of two integers), or a more complex class.

Most classes only support predefined properties. However, for the `script` (page 97) class, AppleScript lets you to define additional properties. For information on how to do this, see "Defining Properties" (page 47). You can also define properties for `record` (page 94) objects.

## Elements

---

An **element** is an object contained within another object. The definition for any AppleScript class includes the element types it can contain. An object can typically contain zero or more of each of its elements.

For a given element type, 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 `list` (page 89) object to contain no items (it can be an empty list). At a later time, the same list might contain many items.

Whether you can add elements to or remove elements from an object depends on the class and the element. For example, a `text` object is immutable—you cannot add or remove text once the object is created. For a `list` object, you cannot remove items, but you can use the `set` command to add an item to the beginning or end:

```
set myList to {1, "what", 3} --result: {1, "what", 3}
set beginning of myList to 0
set end of myList to "four"
myList --result: {0, 1, "what", 3, "four"}
```

## Object Specifiers

An **object specifier** specifies the information needed to find another object in terms of the objects in which it is contained. An object specifier 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.

An object specifier is fully evaluated (or resolved) only when a script is run, not when it is compiled. A script can contain a valid object specifier (such as `third document of application "TextEdit"` that causes an error when the script is executed (because, for example, there may be less than three documents open).

Applications typically return object specifiers in response to commands. For example, if you ask the Finder for a window, it returns information that specifies the window object your script asked for (if it exists). The top-level container in an object specifier is typically the application itself.

You create an object specifier every time your script uses a phrase that describes the path to an object or property, such as `name of window 1 of application "Finder"`. When you use the [a reference to](#) (page 188) operator, it creates a [reference](#) (page 95) object that wraps an object specifier.

The difference between an object specifier and the object it refers to is like the difference between a building address and the building itself. The address is a series of words and numbers, such as “2121 Oak Street, San Francisco, CA” that identifies a location (on a street, in a city, in a state). It is distinct from the building itself. If the building at that location is torn down and replaced with a new building, the address remains the same.

### What Is in an Object Specifier

---

An object specifier describes an object type, a location, and how to distinguish the object from other objects of the same type in that location. These three types of information—the type, or class; the location, or container; and the distinguishing information, or reference form—allow you to specify any object.

In the following example, the class of the object is `paragraph`. The container is the phrase `of document 1`. Because this phrase is inside a `tell` statement, the `tell` statement provides the top-level container, of application `"TextEdit"`. The distinguishing information (the reference form) is the combination of the class, `paragraph`, and an index value, `1`, which together indicate the first paragraph.

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

**Note:** If you examine the dictionary for the TextEdit application, you might think this script should say `paragraph 1 of text of document 1`. However, where the meaning is unambiguous, some applications make life easier for scripters by allowing them to omit a container from an object specifier. TextEdit uses this feature in supplying an *implicitly specified subcontainer* for the text in a document. That is, if an object specifier identifies an object, such as a word or paragraph, that is contained in a document’s text, TextEdit automatically supplies the `of text` part of the object specifier.

In addition to the index reference form, you can specify objects in a container by name, by range, by ID, and by the other forms described in [“Reference Forms”](#) (page 167).

## Containers

---

A container is an object that contains one or more objects or properties. In an object specifier, a 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 (from a Finder `tell` block):

```
folder "Applications" of startup disk
```

A container can be an object or a series of objects, listed from the innermost to the outermost containing object, as in the following:

```
tell application "Finder"
    first item of first folder of first disk
end tell
```

You can also use the possessive form (`'s`) to specify containers. In the following example, the innermost container is `first window` and the object it contains is a `name` property:

```
tell application "TextEdit"
    first window's name
end tell
```

In this example, the target of the `tell` statement (`"TextEdit"`) is the outer container for the object specifier.

## Absolute and Relative Object Specifiers

---

An **absolute object specifier** has enough information to identify an object or objects uniquely. It can be used unambiguously anywhere in a script. For a reference to an application object to be absolute, its outermost container must be the application itself, as in:

```
version of application "Finder" --result: "10.5.1"
```

In contrast, a **relative object specifier** does not specify enough information to identify an object or objects uniquely; for example:

```
name of item 1 of disk 2
```

When AppleScript encounters a relative object specifier in a `tell` statement, it attempts to use the default target specified by the statement to complete the object specifier. Though it isn't generally needed, this implicit target can be specified explicitly using the keyword `it`, which is described in [“The it and me Keywords”](#) (page 40).

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 get a name using the previous relative object specifier.

```
tell application "Finder"
    name of item 1 of disk 2
end tell
```

When AppleScript encounters a relative object specifier outside any `tell` statement, it tries to complete the object specifier by looking up the inheritance chain described in [“Inheritance in Script Objects”](#) (page 60).

## Object Specifiers in Reference Objects

---

When you can create a [reference](#) (page 95) object with the `a reference to` (page 188) operator, it contains an object specifier. For example:

```
tell application "TextEdit"
    set docRef to a reference to the first document
    --result: document 1 of application "TextEdit"
    -- an object specifier
    name of docRef --result: "New Report.rtf"
    -- name of the specified object
end tell
```

In this script, the variable `docRef` is a reference whose object specifier refers to the first document of the application `TextEdit`—which happens to be named “New Report.rtf” in this case. However, the object that `docRef` refers to can change. If you open a second `TextEdit` document called “Second Report.rtf” so that its window is in front of the previous document, then run this script again, it will return the name of the now-frontmost document, “Second Report.rtf”.

You could instead create a reference with a more specific object specifier:

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

If you run this script after opening a second document, it will still return the name of the original document, “New Report.rtf”, if the document exists.

After you create a reference object with the `a reference to` operator, you can use the `contents` property to get the value of the object that it refers to. That is, using the `contents` property causes the reference’s object specifier to be evaluated. In the following script, for example, the content of the variable `myWindow` is the window reference itself.

```
set myWindow to a ref to window "Q1.rtf" of application "TextEdit"
myWindow
-- result: window "Q1.rtf" of application "TextEdit" (object specifier)
contents of myWindow
--result: window id 283 of application "TextEdit" (an evaluated window)
get myWindow
-- result: window "Q1.rtf" of application "TextEdit" (object specifier)
```

Note that the result of the `get` command is to return the reference’s object specifier, not to resolve the specifier to the object it specifies.

When it can, AppleScript will implicitly dereference a reference object (without use of the `contents` property), as in the following example:

```
set myWindow to a ref to window 1 of application "TextEdit"
name of myWindow --result: "Q1.rtf" (if that is the first window's name)
```

For related information, see the Discussion section for the [reference](#) (page 95) class.

## Coercion (Object Conversion)

**Coercion** (also known as **object conversion**) is the process of converting objects from one class to another. AppleScript converts an object to a different class in either of these circumstances:

- in response to the `as` operator
- automatically, when an object is of a different class than was expected for a particular command or operation

Not all classes can be coerced to all other class types. Table 2-1 summarizes the coercions that AppleScript supports for commonly used classes. For more information about each coercion, see the corresponding class definition in “[Class Reference](#)” (page 79).

AppleScript provides many coercions, either as a built-in part of the language or through the Standard Additions scripting addition. You can use these coercions outside of a `tell` block in your script. However, coercion of application class types may be dependent on the application and require a `tell` block that targets the application.

The `as` operator specifies a specific coercion. For example, the following statement coerces the integer `2` into the text `"2"` before storing it in the variable `myText`:

```
set myText to 2 as text
```

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. If the conversion can't be performed, AppleScript reports an error.

When coercing text strings to values of class `integer`, `number`, or `real`, or vice versa, AppleScript uses the current Numbers settings in the Formats pane in International preferences to determine what separators to use in the string. When coercing strings to values of class `date` or vice versa, AppleScript uses the current Dates settings in the Formats pane.

**Table 2-1** Default coercions supported by AppleScript

Convert from class	To class	Notes
<a href="#">alias</a> (page 79)	<code>list (single-item)</code> <code>text</code>	
<a href="#">application</a> (page 80)	<code>list (single-item)</code>	This is both an AppleScript class and an application class.
<a href="#">boolean</a> (page 82)	<code>integer</code> <code>list (single-item)</code> <code>text</code>	
<a href="#">class</a> (page 83)	<code>list (single-item)</code> <code>text</code>	



Convert from class	To class	Notes
<code>constant</code> (page 84)	<code>list (single-item)</code> <code>text</code>	
<code>date</code> (page 85)	<code>list (single-item)</code> <code>text</code>	
<code>file</code> (page 88)	<code>list (single-item)</code> <code>text</code>	
<code>integer</code> (page 89)	<code>list (single-item)</code> <code>real</code> <code>text</code>	Coercing an <code>integer</code> to a number does not change its class.
<code>list</code> (page 89) (single-item)	any class to which the item can be coerced if it is not part of a list	
<code>list</code> (page 89) (multiple-item)	<code>text</code> , if each of the items in the list can be coerced to a <code>text</code> object	
<code>number</code> (page 92)	<code>integer</code> <code>list (single-item)</code> <code>real</code> <code>text</code>	Values identified as values of class <code>number</code> are really values of either class <code>integer</code> or class <code>real</code> .
<code>POSIX file</code> (page 93)	see <code>file</code>	<code>POSIX file</code> is a pseudo-class equivalent to the <code>file</code> class.
<code>real</code> (page 93)	<code>integer</code> <code>list (single-item)</code>	In coercing to <code>integer</code> , any fractional part is rounded. Coercing a <code>real</code> to a number does not change its class.
<code>record</code> (page 94)	<code>list</code>	All labels are lost in the coercion and the resulting list cannot be coerced back to a record.
<code>reference</code> (page 95)	any class to which the referenced object can be coerced	
<code>script</code> (page 97)	<code>list (single-item)</code>	
<code>text</code> (page 97)	<code>integer</code> <code>list (single-item)</code> <code>real</code>	Can coerce to <code>integer</code> or <code>real</code> only if the <code>text</code> object represents an appropriate number.

Convert from class	To class	Notes
<code>unit types</code> (page 103)	integer list (single-item) real text	Can coerce between unit types in the same category, such as inches to kilometers (length) or gallons to liters (liquid volume).

## Scripting Additions

A **scripting addition** is a file or bundle that provides handlers you can use in scripts to perform commands and coercions.

Many of the commands described in this guide are defined in the Standard Additions scripting addition in Mac OS X. These commands are stored in the file `StandardAdditions.osax` in `/System/Library/ScriptingAdditions`, and are available to any script. You can examine the terminology for the Standard Additions by opening this file in Script Editor.

**Note:** A script can obtain the location of the Standard Additions with this script statement, which uses the `path to (folder)` (page 143) command:

```
path to scripting additions as text
  --result: "Hard_Disk:System:Library:ScriptingAdditions:"
```

Scripting additions can be embedded within bundled script applets by placing them in a folder named `Scripting Additions` (note the space between “Scripting” and “Additions”) inside the bundle’s `Contents/Resources/` folder. Note that Script Editor does not look for embedded scripting additions when editing bundled applets. During script development, any required scripting additions must be properly installed in `/System/ScriptingAdditions`, `/Library/ScriptingAdditions`, or `~/Library/ScriptingAdditions` so that Script Editor can find them.

Developers can create their own scripting additions, as described in Technical Note TN1164, *Scripting Additions for Mac OS X*. For related conceptual information, see *AppleScript Overview*, particularly the section “Extending AppleScript with Coercions, Scripting Additions, and Faceless Background Applications” in the chapter Open Scripting Architecture.

## Commands Overview

A **command** is a word or a 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 often an **application object** (one that is stored in an application or its documents and managed by the application, such as a window or document) or an object in Mac OS X. However, it can also be a `script` object or a value in the current script.

Commands often return results. For example, the `display dialog` (page 125) command returns a record that may contain text, a button name, and other information. Your script can examine this record to determine what to do next. You can assign the result of a command to a variable you define, or access it through the predefined AppleScript `result` variable.

## Types of Commands

---

Scripts can make use of the following kinds of commands:

- An **AppleScript command** is one that is built into the AppleScript language. There currently are five such commands: `get` (page 129), `set` (page 154), `count` (page 122), `copy` (page 121), and `run` (page 151). Except for `copy`, each of these commands can also be implemented by applications. That is, there is an AppleScript version of the command that works on AppleScript objects, but an application can define its own version that works on the object types it defines.
- A **scripting addition command** is one that is implemented through the mechanism described in “[Scripting Additions](#)” (page 34)). Although anyone can create a scripting addition (see Technical Note TN1164, *Scripting Additions for Mac OS X*), this guide documents only the scripting addition commands from the Standard Additions, supplied by Apple as part of Mac OS X. These commands are available to all scripts.
- A **user-defined command** is one that is implemented by a handler defined in a `script` object. To invoke a user-defined command outside of a `tell` statement, simply use its name and supply values for any parameters it requires. The command will use the current script as its target.

To invoke a user-defined command inside a `tell` statement, see “[Calling Handlers in a tell Statement](#)” (page 72).

- An **application command** is one that is defined by scriptable application to provide access to a scriptable feature. They are typically enclosed in a `tell` statement that targets the application. You can determine which commands an application supports by examining its dictionary in Script Editor.

Scriptable applications that ship with Mac OS X, such as the Finder and System Events applications (located in `/System/Library/CoreServices`), provide many useful scripting commands.

Third-party scriptable applications also provide commands you can use in scripts. Many support all or a subset of the Standard commands, described in Technical Note TN2106, *Scripting Interface Guidelines*. These include commands such as `delete`, `duplicate`, `exists`, and `move`, as well as application implementations of AppleScript commands, such as `get` and `set`.

## Target

---

There are two ways to specify an object as the target of a command: by supplying it as the direct parameter of the command (described in the next section) or by specifying it as the target of a `tell` statement that contains the command. If a script doesn't explicitly specify the target with a `tell` statement, and it isn't handled by a handler in the script or by AppleScript itself, it is sent to the next object in the inheritance chain (see “[The AppleScript Inheritance Chain](#)” (page 60)).

In the following script, the target of the `get` (page 129) command is the object specifier `name of first window`. Because the enclosing `tell` statement specifies the Finder application, the full specifier is `name of first window of application "Finder"`, and it is the Finder application which obtains and returns the requested information.

```
tell application "Finder"
```

```

    get name of first window
end tell

```

When a command targets an application, the result may be an application object. If so, subsequent statements that target the result object are sent to the application.

## Direct Parameter

---

The **direct parameter** is a value, usually an object specifier, 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, it is noted in the command's definition.

In the following statement, the object specifier `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 specifies a default target for all commands contained within it, so the direct parameter is optional. The following example has the same result as the previous example:

```

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

```

## 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.

In the following example, the `to` parameter specifies the location to which to move the first paragraph. The value of the `to` parameter of the `duplicate` command is the relative object specifier `before paragraph 4`, which is an insertion point. AppleScript completes the specifier with the target of the `tell` statement, `front document of application "TextEdit"`.

```

tell front document of application "TextEdit"
    duplicate paragraph 1 to before paragraph 4
end tell

```

The phrases `paragraph 1` and `before paragraph 4` are called **index** and **relative references**, respectively. For more information, see ["Reference Forms"](#) (page 167).

## AppleScript Error Handling

During script execution, errors may occur due to interaction with Mac OS X, problems encountered in an application script command, or problems caused by statements in the script itself. When an error occurs, AppleScript stops execution at the current location, signals an error, and looks up the calling chain for script statements that can handle the error. That is, it looks for the nearest error-handling code block that surrounds the location where the error occurred.

Scripts can handle errors by enclosing statements that may encounter an error within a `try` (page 207) statement. The `try` statement includes an `on error` section that is invoked if an error occurs. AppleScript passes information about the error, including an error number and an error message, to the `on error` section. This allows scripts to examine the error number and to display information about it.

If the error occurs within a handler that does not provide a `try` statement, AppleScript looks for an enclosing `try` statement where the handler was invoked. If none of the calls in the call chain is contained in a `try` statement, AppleScript stops execution of the script and displays an error message (for any error number other than -128, described below).

A script can use an `error` (page 196) statement to signal an error directly. Doing so invokes the AppleScript error handling mechanism, which looks for an enclosing `try` statement to handle the error.

Some “errors” are the result of the normal operation of a command. For example, commands such as `display dialog` (page 125) and `choose file` (page 112) signal `error -128` (User canceled), if the user clicks the Cancel button. Scripts routinely handle the user canceled error to ensure normal operation. For an example of how to do this, see the Examples section for the `display dialog` command. If no `try` statement in a script handles the -128 error, AppleScript halts execution of the script without displaying any error message.

For related information, see “Results” (page 24), “error Statements” (page 196), “try Statements” (page 207), “Error Numbers and Error Messages” (page 233), and “Working with Errors” (page 237).

## Global Constants in AppleScript

AppleScript defines a number of global constants that you can use anywhere in a script.

### AppleScript Constant

---

The global constant `AppleScript` provides access to properties you can use throughout your scripts.

You can use the `AppleScript` identifier itself to distinguish an AppleScript property from a property of the current target with the same name, as shown in the section “`version`” (page 39).

The following sections describe additional properties of `AppleScript`.

#### `pi`

---

This `mathematical` value represents the ratio of a circle's circumference to its diameter. It is defined as a real number with the value 3.14159265359.

For example, the following statement computes the area of a circle with radius 7:

```
set circleArea to pi * 7 * 7 --result: 153.9380400259
```

## result

---

When a statement is executed, AppleScript stores the resulting value, if any, in the predefined property `result`. The value remains there until another statement is executed that generates a value. Until a statement that yields a result is executed, the value of `result` is undefined. You can examine the result in Script Editor by looking in the Result pane of the script window.

**Note:** When an error occurs during script execution, AppleScript signals an error. It doesn't return error information in the `result` property. For more information, see [“AppleScript Error Handling”](#) (page 36).

## Text Constants

---

AppleScript defines the text properties `space`, `tab`, `return`, `linefeed`, and `quote`. You effectively use these properties as text constants to represent white space or a double quote (") character. They are described in the Special String Characters section of the `text` (page 97) class.

## text item delimiters

---

AppleScript provides the `text item delimiters` property for use in processing text. This property consists of a list of strings used as delimiters by AppleScript when it coerces a list to text or gets text items from text strings. AppleScript currently uses only the first delimiter in the list.

Because `text item delimiters` respect `considering` and `ignoring` attributes in AppleScript 2.0, delimiters are case-insensitive by default. Formerly, they were always case-sensitive. To enforce the previous behavior, add an explicit `considering case` statement.

You can get and set the current value of the `text item delimiters` property. Normally, AppleScript doesn't use any delimiters. For example, if the text delimiters have not been explicitly changed, the statement

```
{"bread", "milk", "butter", 10.45} as string
```

returns the following:

```
"breadmilkbutter10.45"
```

For printing or display purposes, it is usually preferable to set `text item 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 can be used 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 item delimiters` property in Script Editor, it remains changed until you restore its previous value or until you quit Script Editor and launch it again. If you change `text item 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 Script Editor or in other script applications you run.

Scripts commonly use an error handler to reset the `text item delimiters` property to its former value if an error occurs (for more on dealing with errors, see [“AppleScript Error Handling”](#) (page 36)):

```
set savedDelimiters to AppleScript's text item delimiters
try
    set AppleScript's text item delimiters to {"**"}
    --other script statements...
    --now reset the text item delimiters:
    set AppleScript's text item delimiters to savedDelimiters
on error m number n
    --also reset text item delimiters in case of an error:
    set AppleScript's text item delimiters to savedDelimiters
    --and resignal the error:
    error m number n
end try
```

## version

---

This property provides the current version of AppleScript. The following script shows how to check for a version greater than or equal to version 1.9. The `if` statement is wrapped in a `considering numeric strings` statement so that an AppleScript version such as 1.10.6 compares as larger than, say, version 1.9.

```
considering numeric strings
    if version of AppleScript as string ≥ "1.9" then
        -- Perform operations that depend on version 1.9 or greater
    else
        -- Handle case where version is not high enough
    end if
end considering
```

Applications can have their own `version` property, so to access the AppleScript version explicitly, you use the phrase `version of AppleScript`. This will work inside a `tell` block that targets another application, such as the following:

```
tell application "Finder"
    version --result: "10.5.1"
    version of AppleScript --result: "2.0"
end tell
```

## current application Constant

---

The `current application` constant refers to the application that is executing the current AppleScript script (for example, Script Editor). Because the current application is the parent of AppleScript (see [“The AppleScript Inheritance Chain”](#) (page 60)), it gets a chance to handle commands that aren't handled by the current script or by AppleScript.

The `current application` constant is an object specifier—if you ask AppleScript for its value, the result is the object specifier:

```
get current application --result: current application
```

However, if you ask for name of current application, AppleScript resolves the object specifier and returns the current application's name:

```
name of current application --result: "Script Editor"
```

## missing value Constant

---

The `missing value` constant is a placeholder for missing or uninitialized information.

For example, the following statements use the `missing value` constant to determine if a variable has changed:

```
set myVariable to missing value
    -- perform operations that might change the value of myVariable
if myVariable is equal to missing value then
    -- the value of the variable never changed
else
    -- the value of the variable did change
end if
```

## true, false Constants

---

AppleScript defines the Boolean constants `true` and `false`. These constants are described with the [boolean](#) (page 82) class.

## The it and me Keywords

AppleScript defines the keyword `me` to refer to the current script and the keyword `it` to refer to the current target. (The **current script** is the one that is currently being executed; the **current target** is the object that is the current default target for commands.) It also defines `my` as a synonym for `of me` and `its` as a synonym for `of it`.

If a script hasn't targeted anything, `it` and `me` refer to the same thing—the script—as shown in the following example:

```
-- At the top-level of the script:
me --result: «script» (the top-level script object)
it --result: «script» (same as it, since no target set yet)
```

A `tell` statement specifies a default target. In the following example, the default target is the Finder application:

```
-- Within a tell block:
tell application "Finder" -- sets target
    me --result: «script» (still the top-level script object)
```



```
    it --result: application "Finder" (target of the tell statement)
end tell
```

You can use the words `of me` or `my` to indicate that the target of a command is the current script and not the target of the `tell` statement. In the following example, the word `my` indicates that `minimumValue()` handler is defined by the script, not by Finder:

```
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
```

You can also use `of me` or `my` to distinguish script properties from object properties. Suppose there is a TextEdit document open named "Simple.rtf":

```
tell document 1 of application "TextEdit"
    name --result: "Simple.rtf" (implicitly uses target of tell)
    name of it --result: "Simple.rtf" (specifies target of tell)
    me --result: «script» (top-level script object, not target of tell)
end tell
```

The following example shows how to specify different version properties in a Finder `tell` statement. The Finder is the default target, but using `version of me`, `my version`, or `version of AppleScript` allows you to specify the version of the top-level script object. (The top-level script object returns the AppleScript version, because it inherits from AppleScript, as described in "The AppleScript Inheritance Chain" (page 60).)

```
tell application "Finder"
    version --result: "10.5.1" (Finder version is the default in tell block)
    its version --result: "10.5.1" (specifically asks for Finder version)
    version of me --result: "2.0" (AppleScript version)
    my version --result: "2.0" (AppleScript version)
    version of AppleScript --result: "2.0" (AppleScript version)
end tell
```

For information on using `it` in a filter reference, see the Discussion section for the "Filter" (page 169) reference form.

## Aliases and Files

To refer to items and locations in the Mac OS X file system, you use `alias` (page 79) objects and `file` (page 88) objects.

An `alias` object is a dynamic reference to an existing file system object. Because it is dynamic, it can maintain the link to its designated file system object even if that object is moved or renamed.

A `file` object represents a specific file at a specific location in the file system. It can refer to an item that does not currently exist, such as the name and location for a file that is to be created. A `file` object is not dynamic, and always refers to the same location, even if a different item is moved into that place. The `POSIX file` (page 93) pseudo-class is roughly synonymous with `file`: `POSIX file` specifiers evaluate to a `file` object, but they use different semantics for the name, as described in "Specifying Paths" (page 42).

The following is the recommended usage for these types:

- Use an `alias` object to refer to existing file system objects.
- Use a `file` object to refer to a file that does not yet exist.
- Use a `POSIX file` specifier if you want to specify the file using a POSIX path.

The following sections describe how to specify file system objects by path and how to work with them in your scripts.

## Specifying Paths

---

You can create `alias` objects and `file` objects by supplying a name specifier, where the name is the path to an item in the file system.

For `alias` and `file` specifiers, the path is an HFS path, which takes the form

"disk:item:subitem:subsubitem:...:item". For example, "Hard\_Disk:Applications:Mail.app" is the HFS path to the Mail application, assuming your boot drive is named "Hard\_Disk".

HFS paths with a leading colon, such as ":folder:file", are resolved relative to the HFS working directory. However, their use is discouraged, because the location of the HFS working directory is unspecified, and there is no way to control it from AppleScript.

For POSIX file specifiers, the path is a POSIX path, which takes the form

"/item/subitem/subsubitem/.../item". The disk name is not required for the boot disk. For example, "/Applications/Mail.app" is the POSIX path to the Mail application. You can see the POSIX path of an item in Finder in the "Where" field of its Get Info window. Despite the name, POSIX file specifiers may refer to folders or disks. Use of "~" to specify a home directory is not supported.

POSIX paths without a leading slash, such as "folder/file", are resolved relative to the POSIX working directory. This is supported, but only is useful for scripts run from the shell—the working directory is the current directory in the shell. The location of the POSIX working directory for applications is unspecified.

## Working With Aliases

---

AppleScript defines the `alias` (page 79) class to represent aliases. An alias can be stored in a variable and used throughout a script.

The following script first creates an alias to an existing file in the variable `notesAlias`, then uses the variable in a `tell` statement that opens the file. It uses a `try` (page 207) statement to check for existence of the alias before creating it, so that the alias is only created once, even if the script is run repeatedly.

```
try
    notesAlias -- see if we've created the alias yet
on error
    -- if not, create it in the error branch
    set notesAlias to alias "Hard_Disk:Users:myUser:Feb_Notes.rtf"
end try
-- now open the file from the alias:
tell application "TextEdit" to open notesAlias
```

Finding the object an alias refers to is called *resolving* an alias. AppleScript 2.0 attempts to resolve aliases only when you run a script. However, in earlier versions, AppleScript attempts to resolve aliases at compile time.

Once you run the previous example, creating the alias, the script will be able to find the original file when you run it again, even if the file's name or location changes. (However, if you run the script again after recompiling it, it will create a new alias.)

You can get the HFS path from an alias by coercing it to text:

```
notesAlias as text --result: "Hard_Disk:Users:myUser:Feb_Notes.rtf"
```

You can use the `POSIX path` property to obtain a POSIX-style path to the item referred to by an alias:

```
POSIX path of notesAlias --result: "/Feb_Notes.rtf"
```

If an alias doesn't refer to an existing file system object then it is broken. You can't create an alias to an object that doesn't exist, such as a file you plan to create. For that you use a `file` object, described in the next section.

For a sample script that shows how a script application can process a list of aliases it receives when a user drops one or more file icons on it, see [“open Handlers”](#) (page 75).

## Working With Files

---

AppleScript uses `file` objects to represent files in scripts. A `file` object can be stored in a variable and used throughout a script. The following script first creates a `file` object for an existing file in the variable `notesFile`, then uses the variable in a `tell` statement that opens the file:

```
set notesFile to POSIX file "/Users/myUser/Feb_Meeting_Notes.rtf"
tell application "TextEdit" to open notesFile
```

You can use a `file` object to specify a name and location for a file that may not exist:

```
set newFile to POSIX file "/Users/myUser/BrandNewFile.rtf"
```

Similarly, you can let a user specify a new file with the `choose file name` (page 114) command, then use the returned `file` object to create the file. In the following example, if the user cancels the `choose file name` dialog, the rest of the script is not executed. If the user does supply a file name, the script opens the file, creating it if necessary, then uses a `try` statement to make sure it closes the file when it is finished writing to it.

```
set theFile to choose file name
set referenceNumber to open for access theFile with write permission
try
    -- statements to write to the file
on error
    close access referenceNumber
end try
close access referenceNumber
```

Typically, when you pass a `file` object to a command that uses it to operate on a new or existing item in the file system, the components of the path must exist for the command to succeed.

## Remote Applications

A script can target an application on a remote computer if remote applications are enabled on that computer, and if the script specifies the computer with an eppc-style specifier.

### Enabling Remote Applications

---

For a script to send commands to a remote application, the following conditions must be satisfied:

- The computer that contains the application and the computer on which the script is run must be connected to each other through a network.
- Remote Apple Events (set in the Sharing preferences pane) must be enabled on the remote computer and user access must be provided (you can allow access for all users or for specified users only).
- If the specified remote application is not running, you must run it.
- You must authenticate as admin when you compile or run the script.

### eppc-Style Specifiers

---

An eppc-style specifier takes the following format:

```
eppc://[user[:password]@]IP_address
```

`ip_address`

Either a numeric IP address in dotted decimal form (four numbers, from 0 to 255, separated by periods; for example, 123.23.23.123) or a hostname. A hostname can be a Bonjour name.

The following are examples of valid eppc-style specifiers. If you supply the user name and password, no authentication is required. If you do not supply it, authentication may be required.

```
"eppc://myCoolMac.local" -- hostname, no user or pwd
"eppc://myUserName:pwd@myCoolMac.local" -- user, pwd, and hostname
"eppc://123.23.23.123" -- IP address, no user or pwd
"eppc://myUserName:pwd@123.23.23.123" -- user, pwd, and IP address
"eppc://myUserName@server.company.com" -- server address, user
```

**Important:** If a part of the eppc-style specifier contains non-UTF-8 characters or white space, it must be URL-encoded: for example, here is a user name that contains a space:

```
John%20Smith.
```

### Targeting Remote Applications

---

You can target an application that is running on a remote machine and you can launch applications on remote machines that are not currently running.

The following example uses an eppc-style specifier to target the Finder on a remote computer. It includes a user name and password, so no authentication is required.

```
set remoteMachine to "eppc://userName:pwd@MacName.local"
tell app "Finder" of machine remoteMachine to close front window
```

**Important:** If you compile an erroneous eppc-style address, you will have to quit and relaunch Script Editor for changes to that address to take effect.

In some cases, you'll need to use a [using terms from](#) (page 209) statement to tell AppleScript to compile against the local version of an application. The following example uses that technique in telling the remote Finder application to open the TextEdit application:

```
set remoteFinder to application "Finder" of machine ↵
    "eppc://myUserName:pwd@123.23.23.123"

using terms from application "Finder"
    tell remoteFinder
        open application file id "com.apple.TextEdit"
    end tell
end using terms from
```

If you omit the password (pwd) in the previous script, you will have to authenticate when you run the script.

## Debugging AppleScript Scripts

AppleScript does not include a built-in debugger, but it does provide several simple mechanisms to help you debug your scripts or just observe how they are working.

### Feedback From Your Script

---

You can insert various statements into a script to indicate the current location and other information. In the simplest case, you can insert a beep command in a location of interest:

```
beep 3 -- three beeps; a very important part of the script!
```

A [display dialog](#) (page 125) command can display information about what's happening in a script and, like a breakpoint, it halts execution until you dismiss it (or until it times out, depending on the parameters you pass). The following example displays the current script location and the value of a variable:

```
display dialog "In factorial routine; x = " & (x as string)
```

The [say](#) (page 152) command can get your attention by speaking the specified text. In the following example, `currentClient` is a text object that stores a client name:

```
say "I'm in the clientName handler. The client is " & currentClient
```

## Logging

---

Script Editor can display a log of the Apple events that are sent during execution of a script. In the Script Editor Preferences, you can also choose to keep a history of recent results or event logs.

In addition, you can insert `log` (page 138) statements into a script. Log output is shown in the Event Log pane of a script window, and also in the Event Log History window, if it is open.

The following simple example logs the current word in a `repeat with loopVariable (in list)` (page 203) statement:

```
set wordList to words in "Where is the hammer?"
repeat with currentWord in wordList
    log currentWord
    if contents of currentWord is equal to "hammer" then
        display dialog "I found the hammer!"
    end if
end repeat
```

The following shows how the words appear in the log when the script is run:

```
(*Where*)
(*is*)
(*the*)
(*hammer*)
```

## Third Party Debuggers

---

If you need full-featured debugging capabilities, there are powerful, third-party AppleScript debuggers available.

# Variables and Properties

---

Variables and properties are introduced in previous chapters in this document. You use them in `script` objects to store and manipulate values.

**Important:** In reading this chapter, you should be familiar with the information on implicit and explicit `run` handlers in [“run Handlers”](#) (page 74).

The following sections cover common issues in working with variables and properties, including how to declare them and how AppleScript interprets their scope in a script:

- [“Defining Properties”](#) (page 47)
- [“Declaring Variables”](#) (page 48)
- [“Scope of Variables and Properties”](#) (page 51)

## Defining Properties

Property labels follow the rules described in [“Identifiers”](#) (page 18).

Property definitions use the following syntax:

```
property propertyLabel : expression
```

*propertyLabel*

An identifier.

*expression*

An AppleScript expression that sets the initial value for the property. Property definitions are evaluated before variable assignments, so property definitions cannot contain variables.

The following are examples of valid property definitions:

```
property windowCount : 0
property defaultName : "Barry"
property strangeValue : (pi * 7)^2
```

After you define a property, you can change its value with the [copy](#) (page 121) or [set](#) (page 154) command.

The value set by a property definition is not reset each time the script is run; instead, it persists until the script is recompiled.

You cannot declare a property in a handler but a handler can access a property defined in its containing `script` object.

## Declaring Variables

Variable names follow the rules described in “[Identifiers](#)” (page 18).

To create a variable in AppleScript, you assign it a value using the `copy` (page 121) or `set` (page 154) command. For example, the following statements create and initialize two variables, one named `circumference` and one named `savedResult`:

```
set circumference to pi * 3.5 --result: 10.995574287564
copy circumference to savedResult --result: 10.995574287564 (copy of 1st variable)
```

As shown in this example, a variable assignment can make use of a previously defined variable. It can also make use of properties declared in the same `script` object.

There are some obvious, and some more subtle, differences in using `copy` and `set` to create a variable—see “[Using the copy and set Commands](#)” (page 49) for more information.

If you assign a new value to a variable that is already in use, it replaces the old value. You can assign a simple value, an expression, or an object specifier—expressions are evaluated and object specifiers are resolved to obtain the value to assign. To create a variable whose value is an object specifier itself, rather than the value of the object specified, use the `a reference to` (page 188) operator.

The next two sections describe how you can explicitly define a `local` or a `global` variable. These variable types differ primarily in their scope. Scope, which refers to where a variable is accessible within a script, is described in detail in “[Scope of Variables and Properties](#)” (page 51).

## Local Variables

---

You can declare explicit `local` variables using the following syntax:

```
local variableName [, variableName ]...
```

*variableName*

An identifier.

The following are examples of valid `local` variable declarations:

```
local windowCount -- defines one variable
local agentName, agentNumber, agentHireDate -- defines three variables
```

You cannot assign an initial value to a `local` variable in its declaration, nor can you declare a class for the variable. Instead, you use the `copy` (page 121) or `set` (page 154) command to initialize a variable and set its class. For example:

```
set windowCount to 0 -- initialize to zero; an integer
set agentName to "James Smith" -- assign agent name; a text string
set agentNumber to getAgentNumber(agentName) -- call handler; an integer
copy current date to agentHireDate -- call current date command; a date
```



## Global Variables

---

The syntax for `global` variables is nearly identical to that for `local` variables:

```
global variableName [, variableName]...
```

*variableName*

An identifier.

The following are examples of valid `global` variable declarations:

```
global gAgentCount
global gStatementDate, gNextAgentNumber
```

As with `local` variables, you use the `copy` (page 121) or `set` (page 154) command to initialize `global` variables and set their class types. For example:

```
set gAgentCount to getCurrentAgentCount() -- call handler to get count
set gStatementDate to current date -- get date from current date command
set gNextAgentNumber to getNextAvailNumber() -- call handler to get number
```

## Using the `copy` and `set` Commands

---

As its name implies, when you use the `copy` (page 121) command to create a variable, it always creates a separate copy (though note that a copy of an object specifier still specifies the same object). However, when you use the `set` (page 154) command to create a variable, the new variable always refers to the original object or value. You have essentially created another name for the same object.

When more than one variable refers to a changeable (or mutable) object, a change to the object is observable through any of the variables. The types of AppleScript objects that are mutable are `date` (page 85), `list` (page 89), `record` (page 94), and `script` (page 97) objects.

For objects that cannot be modified (immutable objects), variables created with the `set` command may seem like copies—there’s no way to change the object the variables point to, so they seem independent. This is demonstrated in the example in the next section that creates the variables `myName` and `yourName`.

## Declaring Variables with the `set` Command

---

You can use the `set` command to set a variable to any type of object. If the variable doesn’t exist, it is created; if it does exist, its current value is replaced:

```
set numClowns to 5 --result: 5
set myList to { 1, 2, "four" } --result: {1, 2, "four"}
tell application "TextEdit"
    set word1 to word 1 of front document --result: some word
end tell
```

The following example uses a mutable object. It creates two variables that refer to the same list, then modifies the list through one of the variables:

```
set myList to { 1, 2, 3 }
set yourList to myList
set item 1 of myList to 4
```

After executing these statements, the statements `item 1 of myList` and `item 1 of yourList` both yield 4, because both variables refer to the same list.

Now suppose you're working with an immutable object, such as a `text` object:

```
set myName to "Sheila"
set yourName to myName
```

Both variables refer to the same `text` object, but `text` objects are not mutable, so there is no way to change the value of `myName` such that it affects the value of `yourName`. (If you assign new text to one of the variables, you are just creating a new, separate `text` object.)

The `set` command can assign several variables at once using a pattern, which may be a list or record: a list or record of variables on one side, and a list or record of values on the other. Values are matched to variables based on their position for a list, or based on their keys for a record. Not having enough values is an error; if there are too many values, the extra ones are ignored. The order in which the values are evaluated and the variables are assigned is unspecified, but all values are evaluated before any assignments are made.

The Examples section of the `set` (page 154) command shows some simple pattern assignments. Here is an example with more complex patterns:

```
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" *)
```

In the final assignment statement above, `{lastName:r}` is a record that hasn't been used before in the script, and contains an item with label `lastName` and value `r` (a previously defined variable). The variable `x` has previously been set to have a record that has an item with label `lastName` and value `"Chapman"`. During the assignment, the value of the item labeled `lastName` in the new record is set to the value of the item labeled `lastName` in `x`—hence it now has the value `"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 set a pattern to a pattern, as long as the patterns match. For details, see the `set` (page 154) command.

**Note:** Using patterns with the `set` command is similar to using patterned parameters with handlers, which is described in [“Handlers with Patterned Positional Parameters”](#) (page 70).

## Declaring Variables with the `copy` Command

You can use the `copy` command to set a variable to any type of object. If the variable doesn't exist, it is created; if it does exist, its current value is replaced. The `copy` command creates a new copy that is independent of the original—a subsequent change does not change the original value (though note that a copy of an object specifier still specifies the same object).

To copy within an application, you should use the application's `duplicate` command, if it has one. To copy between applications, you can use the `get` (page 129) command to obtain information from one application and the `set` (page 154) command to set it in another.

The `copy` command creates a deep copy—that is, if you copy a nested data structure, such as a list that contains another list, the entire structure is copied, as shown in the following example. This example creates a record (`alpha`), then a list (`beta`), then a list that contains the first record and list (`gamma`), then finally a copy of `gamma` (`delta`). It then changes a property in the original record, `alpha`. The result shows that the property is changed wherever `alpha` appears, except in the copy, `delta`:

```
set alpha to {property1:10, property2:20}
set beta to {1, 2, "Hello"}
set gamma to {alpha, beta, "Goodbye"}
copy gamma to delta
set property1 of alpha to 42

{alpha, beta, gamma, delta} -- List variables to show contents
(*result: {{property1:42, property2:20}, {1, 2, "Hello"}, {{property1:42,
property2:20}, {1, 2, "Hello"}, "Goodbye"}, {{property1:10, property2:20}, {1,
2, "Hello"}, "Goodbye"}} *)
```

If you make a copy of a reference object, it refers to the same object as the original (because both contain the same object specifier):

```
set windowRef to a reference to window 1 of application "Finder"
name of windowRef --result: "Script testing folder"
copy windowRef to currentWindowRef --result: a new object specifier
name of currentWindowRef --result: "Script testing folder"
```

## Scope of Variables and Properties

The **declaration** of a variable or property identifier is the first valid occurrence of the identifier in a `script` object. The form and location of the declaration determine how AppleScript treats the identifier in that `script` object.

The **scope** is the range over which AppleScript recognizes a declared identifier within a `script` object. 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` object in which it is declared. After declaring a property, you can reuse the same identifier as a separate variable only if you first declare it as a `local` variable.

**Lifetime** refers to the period of time over which a variable or property is in existence. Only the values of properties and `global` variables can persist after a script is run.

In the discussions that follow, declarations and statements in a `script` object that occur outside of any handlers or nested `script` objects are identified as *outside*.

The following examples show the four basic forms for declaring variables and properties in AppleScript:

- `property x: 3`

The scope of a property definition is the `script` object in which it is declared, including any handlers or nested `script` objects. A property definition specifies an initial value. You cannot declare a property in a handler.

The value set by a property definition is not reset each time the script is run; instead, it persists until the script is recompiled.

- `global x`

The scope of a `global` variable can be limited to specific handlers or contained `script` objects or it can extend throughout a top-level `script` object. A `global` declaration doesn't set an initial value—it must be initialized by a `copy` (page 121) or `set` (page 154) command before a script can access its value.

The value of a `global` variable is not reset each time a script is run, unless its initialization statement is executed.

- `local x`

The scope of a `local` variable can be limited to specific handlers or contained `script` objects or it can extend throughout a top-level `script` object. A `local` declaration doesn't set an initial value—it must be initialized by a `copy` or `set` command before a script can access its value.

The value of a `local` variable is reset each time the handler is run (either the `run` handler for the script, or the specific handler in which the variable is declared).

- `set x to 3`

In the absence of a `global` variable declaration, the scope of a variable declared with the `copy` or `set` command is normally restricted to the `run` handler for the script, making it implicitly local to that run handler. However, a handler or nested `script` object can declare the same variable with a `global` declaration to gain access to it.

The value of a variable declared with the `copy` or `set` command is reset each time a script is run.

If you want to use the same identifier in several different places in a script, you should either declare it as a property or as a `global` variable.

It is often convenient to limit the scope of a particular identifier to a single handler or nested `script` object, which you can do by defining it as a `local` variable in the handler or `script` object. Outside, the identifier has no value associated with it and can be reused elsewhere in the script. When used this way, a `local` variable is said to *shadow* (or block access to) a `global` variable or property with the same name, making the global version inaccessible in the scope of the handler or `script` object where the `local` variable is declared.

**Note:** If you save a script as a script application, then run the application on read-only media, the value of a modified property or `global` variable is not saved.

The following sections provide additional information about scope:

- [“Scope of Properties and Variables Declared in a Script Object”](#) (page 52)
- [“Scope of Variables Declared in a Handler”](#) (page 55)

## Scope of Properties and Variables Declared in a Script Object

---

Table 3-1 shows the scope and lifetime for variables and properties that are declared at the top level in a `script` object (outside any handlers or nested `script` objects).

**Table 3-1** Scope of property and variable declarations at the top level in a script object

Declaration type	Scope (visibility)	Lifetime
property x: 3	Everywhere in script	Reset when script is recompiled
global x	Everywhere in script	Reset when reinitialized in script or when script is recompiled
local x	Within run handler only	Reset when script is run
set x to 3	Within run handler only	Reset when script is run

The scope of a property in a script object extends to any subsequent statements anywhere in the script. Consider the following 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` anywhere in this script, AppleScript associates it with the `currentCount` property.

The value of a property persists after the script in which the property is defined has been run. Thus, the value of `currentCount` is 0 the first time this script is run, 1 the next time it is run, and so on. The property's current value is saved with the script object and is not reset to 0 until the script is recompiled—that is, modified and then run again, saved, or checked for syntax.

The value of a global variable also persists after the script in which it is defined has been run. However, depending on how it is initialized, a global variable may be reset each time the script is run again. The next example shows how to initialize a global variable so that it is initialized only the first time a script is run, and thus produces the same result as using a property in the previous example:

```
global currentCount
increment()

on increment()
    try
        set currentCount to currentCount + 1
    on error
        set currentCount to 1
    end try
    display dialog "Count is now " & currentCount & "."
end increment
```

The first time the script is run, the statement `set currentCount to currentCount + 1` generates an error because the global variable `currentCount` has not been initialized. When the error occurs, the `on error` block initializes `currentCount`. When the script is run again, the variable has already been initialized, so the error branch is not executed, and the variable keeps its previous value. Persistence is accomplished, but not as simply as in the previous example.

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:

```
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 0.

In the absence of a `global` variable declaration, the scope of a variable declaration using the `set` command 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 is limited to the `run` handler for the script. Because this script has no explicit `run` handler, outside statements are part of its implicit `run` handler, as described in "run Handlers" (page 74). 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 with the same variable declared with the `set` command outside the handler, you can use a `global` declaration in the handler, as shown in the next example. (This approach also works to associate a variable in a nested script object.)

```
set currentCount to 0
on increment()
    global currentCount
    set currentCount to currentCount + 1
end increment

increment() --result: 1
currentCount --result: 1
```

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 declare the same variable as `global` results in an error.

If you declare an outside variable with the `set` command and then declare the same identifier as a property, the declaration with the `set` command overrides the property definition. For example, the following script returns 10, not 5. This occurs because AppleScript evaluates property definitions before it evaluates `set` command declarations:

```
set numClowns to 10 -- evaluated after property definition
property numClowns: 5 -- evaluated first
numClowns --result: 10
```

The next example, shows how to use a `global` variable declaration in a script object to associate a `global` variable with an outside property:

```
property currentCount : 0
script Paula
  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 Paula --result: 1
run Paula --result: 2
currentCount --result: 2
currentCount of Paula --result: 20
```

This script declares two separate `currentCount` properties: one outside any handlers (and script objects) in the main script and one in the script object `Paula` but outside of any handlers or script objects within `Paula`. 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.

## Scope of Variables Declared in a Handler

---

A handler can't declare a property, although it can refer to a property that is declared outside any handler in the script object. (A handler can contain script objects, but it can't contain another handler, except in a contained script object.)

[Table 3-2](#) (page 55) summarizes the scope of variables declared in a handler. Examples of each form of declaration follow.

**Table 3-2** Scope of variable declarations within a handler

Declaration type	Scope (visibility)	Lifetime
<code>global x</code>	Within handler only	Reset when script is recompiled; if initialized in handler, then reset when handler is run

Declaration type	Scope (visibility)	Lifetime
local x	Within handler only	Reset when handler is run
set x to 3	Within handler only	Reset when handler is run

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 outside any handler. However, the use of `currentCount` in any subsequent handler in the script is local to that handler unless the handler also explicitly declares `currentCount` as a global variable.

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 outside the handler:

```
property currentCount : 10
on increment()
    local currentCount
    set currentCount to 5
end increment

increment() --result: 5
currentCount --result: 10
```

The scope of a variable declaration using the `set` command in a handler is limited to that handler:

```
script Henry
    set currentCount to 10 -- implicit local variable in script object
    on increment()
        set currentCount to 5 -- implicit local variable in handler
    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 in the script object `Henry` is limited to the run handler for the script object (in this case, an implicit run handler, consisting of the last two statements in the script). The scope of the second `currentCount` declaration, within the `on increment` handler, is limited to that handler. The two instances of `currentCount` are independent variables.



# Script Objects

---

This chapter describes the `script` object, which is used to implement all AppleScript scripts. Before reading this chapter, you should be familiar with the information in [“AppleScript and Objects”](#) (page 26).

A **script object** is a user-defined object that can combine data (in the form of properties) and actions (in the form of handlers and additional `script` objects). Script objects support inheritance, allowing you to define a hierarchy of objects that share properties and handlers. You can also extend or modify the behavior of a handler in one `script` object when calling it from another `script` object.

The top-level `script` (page 97) object is the one that implements the overall script you are working on. Any `script` object can contain nested `script` objects, each of which is defined just like a top-level `script` object, except that a nested `script` object is bracketed with statements that mark its beginning and end.

This chapter describes `script` objects in the following sections:

- [“Defining Script Objects”](#) (page 57) shows the syntax for defining `script` objects and includes a simple example .
- [“Initializing Script Objects”](#) (page 59) describes how AppleScript creates a `script` object with the properties and handlers you have defined.
- [“Sending Commands to Script Objects”](#) (page 59) describes how you use `tell` statements to send commands to `script` objects.
- [“Inheritance in Script Objects”](#) (page 60) describes inheritance works and how you can use it to share functionality in the `script` objects you define.

## Defining Script Objects

Each `script` object definition (except for the top-level `script` object) begins with the keyword `script`, followed by a variable name, and ends with the keyword `end` (or `end script`). The statements in between can be any combination of property definitions, handler definitions, nested `script` object definitions, and other AppleScript statements.

The syntax of a `script` object definition is as follows:

```
script variableName
    [ (property | prop) parent : parentSpecifier ]
    [ (property | prop) propertyLabel : initialValue ]...
    [ handlerDefinition ]...
    [ statement ]...
```

`end [ script ]`

*variableName*

A variable identifier for the script. You can refer to a script object by this name elsewhere in a script.

*parentSpecifier*

Specifies the parent of the `script` object, typically another `script` object.

For more information, see [“Inheritance in Script Objects”](#) (page 60).

*propertyLabel*

An identifier, unique within the `script` object, that specifies a characteristic of the object; equivalent to an instance variable.

*initialValue*

The value that is assigned to the property each time the `script` object is initialized. `script` objects are initialized when compiled. *initialValue* is required in property definitions.

*handlerDefinition*

A handler for a command the `script` object can respond to; equivalent to a method. For more information, see [“About Handlers”](#) (page 67) and [“Handler Reference”](#) (page 213).

*statement*

Any AppleScript statement. Statements other than handler and property definitions are treated as if they were part of an implicit handler definition for the `run` command; they are executed when a `script` object receives the `run` command.

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 any other handler.

You can use a `tell` statement to send commands to a `script` object. For example, the following statement sends the `sayHello` command the `script` object defined above.

```
tell John to sayHello to "Herb" --result: "Hello Herb"
```

You can manipulate the properties of `script` objects by using the `get` command to get the value of a property and the `set` or `copy` command to change the value. The value of a property is persistent—it gets reset every time you compile the script, but not when you run it.

## Initializing Script Objects

When you define a `script` object, it can contain properties, handlers, and nested `script` object definitions. When you execute the script containing it, AppleScript creates a `script` object with the defined properties, handlers, and nested `script` objects. The process of creating an instance of a `script` object from its definition is called initialization. A `script` object must be initialized before it can respond to commands.

A top-level `script` object is initialized each time the script's `run` handler is executed. Similarly, if you define a script within a handler, 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 `thePoint`:

```
on makePoint(x, y)
    script thePoint
        property xCoordinate:x
        property yCoordinate:y
    end script
    return thePoint
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 `thePoint` when it executes the `makePoint` command. After the call to `makePoint`, the variable `myPoint` refers to this `script` object. The parameter variables in the `makePoint` handler, in this case, `x` and `y`, become local variables of the `script` object. The initial value of `x` is 10, and the initial value of `y` is 20, because those are the parameters passed to the `makePoint` handler that initialized the `script` object.

If you added the following line to the end of the previous script and ran it, the variable `myOtherPoint` would refer to a second instance of the `script` object `thePoint`, with different property values:

```
set myOtherPoint to makePoint(30,50)
```

The `makePoint` script is a kind of constructor function that creates `script` objects representing points.

## Sending Commands to Script Objects

You can use `tell` statements to send commands to `script` objects. For example, the following `tell` statement sends two `sayHello` commands to the `script` object `John` (defined below):

```
tell John
    sayHello to "Herb"
    sayHello to "Grace"
end tell
```

For a `script` object to respond to a command within a `tell` statement, either the `script` object or its parent object must have a handler for the command. For more information about parent objects, see [“Inheritance in Script Objects”](#) (page 60).

A `script` object definition may include an implicit `run` handler, consisting of all executable statements that are outside of any handler or nested `script` object, or it may include an explicit `run` handler that begins with `on run`, but it may not contain both—such a script will not compile. If a script has no `run` handler (for example, a script that serves as a library of handlers, as described in “[Saving and Loading Libraries of Handlers](#)” (page 72)), executing the script does nothing. However, sending it an explicit `run` command causes an error. For more information, see “[run Handlers](#)” (page 74).

The `display dialog` command in the following `script` object definition is the only executable statement at the top level, so it constitutes the `script` object’s implicit `run` handler and is executed when the script sends a `run` command to `script` object `John`, with the statement `tell John to run`.

```
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

tell John to run
```

You can also use the possessive to send a command to a `script` object. For example, either of the following two forms send the `sayHello` command to `script` object `John` (the first version compiles into the second):

```
John's sayHello to "Jake" --result: "Hello Jake"
sayHello of John to "Jake" --result: "Hello Jake"
```

## Inheritance in Script Objects

You can use the AppleScript 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 is described in the following sections:

- “[The AppleScript Inheritance Chain](#)” (page 60)
- “[Defining Inheritance Through the parent Property](#)” (page 61)
- “[Some Examples of Inheritance](#)” (page 61)
- “[Using the continue Statement in Script Objects](#)” (page 63)

### The AppleScript Inheritance Chain

---

The top-level `script` object is the parent of all other `script` objects, although any `script` object can specify a different parent object. The top-level `script` object also has a parent—AppleScript itself (the AppleScript component). And even AppleScript has a parent—the current application. The name of that application (which is typically Script Editor) can be obtained through the global constant `current application`. This hierarchy defines the **inheritance chain** that AppleScript searches to find the target for a command or the definition of a term.

Every `script` object has access to the properties, handlers, and script objects it defines, as well as to those defined by its parent, and those of any other object in the inheritance chain, including AppleScript. That's why the constants and properties described in [“Global Constants in AppleScript”](#) (page 37) are available to any script.

**Note:** There is an exception to the previous claim. An explicit `local` variable can *shadow* (or block access to) a `global` variable or property with the same name, making the global version inaccessible in the scope of the handler or `script` object. For related information, see [“Scope of Variables and Properties”](#) (page 51).

## Defining Inheritance Through the parent Property

---

When working with `script` (page 97) objects, **inheritance** is the ability of a child `script` object to take on the properties and handlers of a parent object. You specify inheritance with the `parent` property.

The object listed in a `parent` property definition is called the **parent 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, though if one is not specified, every `script` is a child of the top-level `script`, as described in [“The AppleScript Inheritance Chain”](#) (page 60). A `script` object can have many children, but a child `script` object can have only one parent. The parent object may be any object, such as a `list` (page 89) or an `application` (page 80) object, but it is typically another `script` object.

The syntax for defining a parent object is

```
(property | prop) parent : variable
```

*variable*

An identifier for a variable that refers to the parent 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 a parent `script` object (or a command that calls a function that creates a parent `script` object) must come before the definition of the child in the same script.

## Some Examples of Inheritance

---

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 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.

[Listing 4-1](#) (page 61) shows the definitions of a parent `script` object called Alex and a child `script` object called AlexJunior.

**Listing 4-1** A pair of script objects with a simple parent-child relationship

```
script Alex
  on sayHello()
    return "Hello, " & getName()
  end sayHello
  on getName()
```

## Script Objects

```

        return "Alex"
    end getName
end script

script AlexJunior
    property parent : Alex
    on getName()
        return "Alex Jr"
    end getName
end script

-- Sample calls to handlers in the script objects:
tell Alex to sayHello() --result: "Hello, Alex"
tell AlexJunior to sayHello() --result: "Hello, Alex Jr."

tell Alex to getName() --result: "Alex"
tell AlexJunior to getName() --result: "Alex Jr"

```

Each script object defines a `getName()` handler to return its name. The script object `Alex` also defines the `sayHello()` handler. Because `AlexJunior` declares `Alex` to be its parent object, it inherits the `sayHello()` handler.

Using a `tell` statement to invoke the `sayHello()` handler of script object `Alex` returns "Hello, Alex". Invoking the same handler of script object `AlexJunior` returns "Hello, Alex Jr"—although the same `sayHello()` handler in `Alex` is executed, when that handler calls `getName()`, it's the `getName()` in `AlexJunior` that is executed.

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 `JohnSon` in the following script inherits its `vegetable` property from script object `John`.

```

script John
    property vegetable : "Spinach"
end script
script JohnSon
    property parent : John
end script
set vegetable of John to "Swiss chard"
vegetable of JohnSon
--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 in the parent object also 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". For related information, see "The `it` and `me` Keywords" (page 40).

## Using the `continue` Statement in Script Objects

---

In a child script object, you can define a handler with the same name as a handler defined in its parent object. In implementing the child handler, you have several options:

- The handler in the child script object can be independent of the one in its parent. This allows you to call either handler, as you wish.
- The handler in the child can simply invoke the handler in its parent. This allows the child object to take advantage of the parent's implementation (as shown in the script objects below that contain a `on identify` handler).
- The handler in the child can invoke the handler in its parent, changing the values passed to it or executing additional statements before or after invoking the parent handler. This allows the child object to modify or add to the behavior of its parent, but still take advantage of the parent's implementation.

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.

This handing off of control to another 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 executed.

The syntax for a `continue` statement is shown in "continue" (page 213).

The following script includes two script object definitions, `Elizabeth` and `ChildOfElizabeth`.

```
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 this 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 this 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

```



## CHAPTER 4

### Script Objects

```
        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»
```



# About Handlers

---

When script developers want to factor and re-use their code, they can turn to handlers. A handler is a collection of statements that can be invoked by name. Handlers are also known as functions, subroutines, or methods.

This chapter describes how to work with handlers, in the following sections:

- [“Handler Basics”](#) (page 67)
- [“Saving and Loading Libraries of Handlers”](#) (page 72)
- [“Handlers in Script Applications”](#) (page 73)

For detailed reference information, see [“Handler Reference”](#) (page 213).

## Handler Basics

A **handler** is a collection of statements that can be invoked by name. Handlers are useful in scripts that perform the same action in more than one place. You can package statements that perform a specific task as a handler, give it a descriptive name, and call it from anywhere in the script. This makes the script shorter and easier to maintain.

A script can contain one or more handlers. However, you can not nest a handler definition within another handler (although a script object defined in a handler can contain other handlers).

The definition for a handler specifies the parameters it uses, if any. It does not specify the class for its parameters. However, most handlers expect each parameter to be of a specific class, so it is useful to add a comment that lists the expected class types.

When you call a handler, you must list its parameters according to how they are specified in its definition. Handlers may have labeled parameters or positional parameters, described in subsequent sections.

A handler definition can contain variable declarations and statements. It may use a `return` statement (described in detail in [“return”](#) (page 214)) to return a value and exit the handler.

A call to a handler must include all the parameters specified in the handler definition. There is no way to specify optional parameters.

The sections that follow provide additional information on working with handlers:

- [“Defining a Simple Handler”](#) (page 68)
- [“Handlers with Labeled Parameters”](#) (page 68)
- [“Handlers with Positional Parameters”](#) (page 69)
- [“Handlers with Patterned Positional Parameters”](#) (page 70)
- [“Recursive Handlers”](#) (page 71)

- [“Errors in Handlers”](#) (page 71)
- [“Passing by Reference Versus Passing by Value”](#) (page 71)
- [“Calling Handlers in a tell Statement”](#) (page 72)

## Defining a Simple Handler

---

The following is a definition for a simple handler that takes any parameter value that can be displayed as text (presumably one representing a date) and displays it in a dialog box. The handler name is `rock`; its parameter is `around the clock`, where `around` is a parameter label and `clock` is the parameter name (the `is` is an AppleScript filler for readability):

```
on rock around the clock
    display dialog (clock as text)
end rock
```

This handler allows an English-like calling statement:

```
rock around the current date -- call handler to display current date
```

A handler can have no parameters. To indicate that a handler has no parameters, you include a pair of empty parentheses after the handler name in both the handler definition and the handler call. For example, the following `helloWorld` script has no parameters.

```
on helloWorld()
    display dialog "Hello World"
end

helloWorld() -- Call the handler
```

## Handlers with Labeled Parameters

---

To define a handler with labeled parameters, you list the labels to use when calling the handler and the statements to be executed when it is called. (The syntax is shown in [“Handler Syntax \(Labeled Parameters\)”](#) (page 215).)

Handlers with labeled parameters can also have a direct parameter. With the exception of the direct parameter, which must directly follow the handler name, labeled parameters can appear in any order, with the labels from the handler definition identifying the parameter values. This includes parameters listed in `given`, `with`, and `without` clauses (of which there can be any number).

The `findNumbers` handler in the following example uses the special label `given` to define a parameter with the label `given` rounding.

```
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 then -- round the number
            -- Use copy so original list isn't modified.
            copy (round x) to x
        end if
    end repeat
end to findNumbers
```

```

        if x > minLimit then
            set end of resultList to x
        end if
    end repeat
    return resultList
end findNumbers

```

The next statements show how to call `findNumbers` by passing a predefined list variable:

```

set myList to {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}

```

You can also specify the value of the `rounding` parameter by using a `with` or `without` clause to indicate `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`.) These examples pass a list object directly, rather than using a list variable as in the previous case:

```

findNumbers of {5.1, 20.1, 20.5, 33} above 20 with rounding
--result: {33}

findNumbers of {5.1, 20.1, 20.5, 33.7} above 20 without rounding
--result: {20.1, 20.5, 33.7}

```

Here is another handler that uses parameter labels:

```

to check for yourNumber from startRange thru endRange
    if startRange ≤ yourNumber and yourNumber ≤ endRange then
        display dialog "Congratulations! Your number is included."
    end if
end check

```

The following statement calls the handler, causing it to display the "Congratulations!" message

```
check for 8 from 7 thru 10 -- call the handler
```

## Handlers with Positional Parameters

---

The definition for a handler with positional parameters shows the order in which to list parameters when calling the handler and the statements to be executed when the handler is called. The definition must include parentheses, even if it doesn't include any parameters. The syntax is shown in "[Handler Syntax \(Positional Parameters\)](#)" (page 218).

In the following example, the `minimumValue` routine returns the smaller of two values:

```

on minimumValue(x, y)
    if x < y then
        return x
    else
        return y
    end if
end minimumValue

-- To call minimumValue:

```

```
minimumValue(5, 105) --result: 5
```

The first line of the `minimumValue` handler specifies the parameters of the handler. To call a handler with positional parameters you list the parameters in the same order as they are specified in the handler definition.

If a handler call is part of an expression, AppleScript uses the value returned by the handler to evaluate the expression. For example, to evaluate the following expression, AppleScript first calls `minimumValue`, then evaluates the rest of the expression.

```
minimumValue(5, 105) + 50 --result: 55
```

## Handlers with Patterned Positional Parameters

---

You can create a handler whose positional parameters define a pattern to match when calling the handler. For example, the following handler takes a single parameter whose pattern consists of two items in a list:

```
on displayPoint({x, y})
    display dialog ("x = " & x & ", y = " & y)
end displayPoint

-- Calling the handler:
set testPoint to {3, 8}
displayPoint(testPoint)
```

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)."
```

The properties of a record passed to a handler with patterned parameters don't have to be given in the same order in which they are given in the handler's definition, as long as all the properties required to fit the pattern are present.

The following call to `minimumValue` uses the value from a handler call to `maximumValue` as its second parameter. The `maximumValue` handler (not shown) returns the larger of two passed numeric values.

```
minimumValue(20, maximumValue(1, 313)) --result: 20
```

## Recursive Handlers

---

A **recursive handler** is a handler that calls itself. For example, this recursive handler 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. The factorial of 0 is 1.)

```
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 handler `factorial` is called once, passing the value 10. The handler then calls itself recursively with a value of  $x - 1$ , or 9. Each time the handler 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 handlers, including the original `factorial` call.

When you call a recursive handler, AppleScript keeps track of the variables and pending statements in the original (partially executed) handler until the recursive handler has completed. Because each call uses some memory, the maximum number of pending handlers is limited by the available memory. As a result, a recursive handler may generate an error before the recursive calls complete.

In addition, a recursive handler may not be the most efficient solution to a problem. For example, the factorial handler shown above can be rewritten to use a `repeat` statement instead of a recursive call, as shown in the example in [repeat with loopVariable \(from startValue to stopValue\)](#) (page 202).

## Errors in Handlers

---

As with any AppleScript statements that may encounter an error, you can use a `try` statement to deal with possible errors in a handler. A `try` (page 207) statement includes two collections of statements: one to be executed in the general case, and a second to be executed only if an error occurs.

By using one or more `try` statements with a handler, you can combine the advantages of reuse and error handling in one package. For a detailed example that demonstrates this approach, see [“Working with Errors”](#) (page 237).

## Passing by Reference Versus Passing by Value

---

Within a handler, each parameter is like a variable, providing access to passed information. AppleScript passes all parameters by reference, which means that a passed variable is shared between the handler and the caller, as if the handler had created a variable using the `set` (page 154) command. However, it is important to remember a point raised in [“Using the copy and set Commands”](#) (page 49): only mutable objects can actually be changed.

As a result, a parameter’s class type determines whether information is effectively passed by value or by reference:

- For mutable objects (those whose class is `date` (page 85), `list` (page 89), `record` (page 94), or `script` (page 97)), information is passed *by reference*:

If a handler changes the value of a parameter of this type, the original object is changed.

- For all other class types, information is effectively passed *by value*:

Although AppleScript passes a reference to the original object, that object cannot be changed. If the handler assigns a new value to a parameter of this type, the original object is unchanged.

If you *want* to pass by reference with a class type other than `date`, `list`, `record`, or `script`, you can pass a reference object that refers to the object in question. Although the handler will have access only to a copy of the reference object, the specified object will be the same. Changes to the specified object in the handler will change the original object, although changes to the reference object itself will not.

## Calling Handlers in a tell Statement

To call a handler from within a `tell` statement, you must use the reserved words `of me` or `my` to indicate that the handler is part of the script and not a command that should be sent to the target of the `tell` statement.

For example, the following script calls the `minimumValue` handler defined in “[Handlers with Positional Parameters](#)” (page 69) from within a `tell` statement. If this call did not include the words `of me`, it would cause an error, because AppleScript would send the `minimumValue` command to `TextEdit`, which does not understand that message.

```
tell front document of application "TextEdit"
    minimumValue(12, 400) of me
    set paragraph 1 to result as text
end tell
--result: The handler call is successful.
```

Instead of using the words `of me`, you could insert the word `my` before the handler call:

```
my minimumValue(12, 400)
```

## Saving and Loading Libraries of Handlers

In addition to defining and calling handlers within a script, you can access handlers from other scripts. To make a handler available to another script, save it as a compiled script, then use the `load script` (page 135) command to load it in any script that needs to call the handler. You can use this technique to create libraries containing many handlers.

**Note:** The `load script` command loads the compiled script as a `script` object; for more information, see “[Script Objects](#)” (page 57).

For example, the following script contains two handlers: `areaOfCircle` and `factorial`:

```
-- This handler computes the area of a circle from its radius.
-- (The area of a circle is equal to pi times its radius squared.)
```



```

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 * radius * pi -- pi is predefined by AppleScript.
    else
        error "The parameter must be a real number or an integer"
    end if
end areaOfCircle

```

```

-- This handler 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

```

In Script Editor, save the script as a compiled Script (which has extension `scpt`) or Script Bundle (extension `scptd`) and name it “NumberLib.”

After saving the script as a compiled script, other scripts can use the `load script` command to load it. For example, the following script loads the compiled script `NumberLib.scpt`, storing the resulting script object in the variable `numberLib`. It then makes handler calls within a `tell` statement that targets the script object. The compiled script must exist in the specified location for this script to work.

```

set numberLibrary to (load script file "NumberLib.scpt")

tell numberLibrary
    factorial(10)           --result: 3628800
    areaOfCircle from 12   --result: 452.38934211693
end tell

```

## Handlers in Script Applications

A **script application** is an application whose only function is to run the script associated with it. Script applications contain handlers that allow them to respond to commands. For example, many script applications can respond to 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. It can also contain other handlers to respond to commands such as `quit` or `print`.

When saving a script in Script Editor, you can create a script application by choosing either Application or Application Bundle from the File Format options. Saving as Application results in a simple format that is compatible with Mac OS 9. Saving as Application Bundle results in an application that uses the modern bundle format, with its specified directory structure, which is supported back to Mac OS X v10.3.

When creating a script application, you can also specify whether a startup screen should appear before the application runs its script. Whatever you write in the Description pane of the script window in Script Editor is displayed in the startup screen. You can also specify in Script Editor whether a script application should stay open after running. The default is for the script to quit immediately after it is run.

You can run a script application from the Finder much like any other application. If it has a startup screen, the user must click the Run button or press the Return key before the script actually runs.

Consider the following simple script

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

What this script does as a script application depends on what you specify when you save it. If you don't specify a startup screen or tell it to stay open, it will automatically execute once, closing the front Finder window, and then quit.

If a script application modifies the value of a property, the changed value persists across launches of the application. For related information, see [“Scope of Variables and Properties”](#) (page 51).

For information about some common script application handlers, see the following sections:

- [“run Handlers”](#) (page 74)
- [“open Handlers”](#) (page 75)
- [“idle and quit Handlers for Stay-Open Applications”](#) (page 76)

See [“Handler Reference”](#) (page 213) for syntax information.

## run Handlers

---

When you run a script or launch a script application, its `run` handler is invoked. A script's `run` handler is defined in one of two ways:

- As an implicit `run` handler, which consists of all statements declared outside any handler or nested `script` object in a script.
  - Declarations for properties and `global` variables are not considered statements in this context—that is, they are not considered to be part of an implicit `run` handler.
- As an explicit `run` handler, which is enclosed within `on run` and `end` statements, similar to other handlers.

Having both an implicit and an explicit `run` handler is not allowed, and causes a syntax error during compilation. If a script has no `run` handler (for example, a script that serves as a library of handlers, as described in [“Saving and Loading Libraries of Handlers”](#) (page 72)), executing the script does nothing. However, sending it an explicit `run` command causes an error.

The following script demonstrates an implicit `run` handler. The script consists of a statement that invokes the `sayHello` handler, and the definition for the handler itself:

```
sayHello()

on sayHello()
    display dialog "Hello"
end sayHello
```

The implicit `run` handler for this script consists of the statement `sayHello()`, which is the only statement outside the handler. If you save this script as a script application and then run the application, the script receives a `run` command, which causes it to execute the one statement in the implicit `run` handler.

You can rewrite the previous script to provide the exact same behavior with an explicit `run` handler:

```
on run
    sayHello()
end run

on sayHello()
    display dialog "Hello"
end sayHello
```

Whether a script is saved as a script application or as a compiled script, its `run` handler is invoked when the script is run. You can also invoke a `run` handler in a script application from another script. For information about how to do this, see [“Calling a Script Application From a Script”](#) (page 77).

## open Handlers

---

Mac OS X applications, including script applications, receive an `open` command whenever the user drops file, folder, or disk icons on the application’s Finder icon, even if the application is already running.

If the script in a script application includes an `open` handler, the handler is executed when the application receives the `open` command. The `open` handler takes a single parameter which provides a list of all the items to be opened. Each item in the list is an `alias` (page 79) object.

For example, the following `open` handler makes a list of the pathnames of all items dropped on the script application’s icon and saves them in the frontmost TextEdit document:

```
on open names
    set pathNamesString to "" -- Start with empty text string.
    repeat with i in names
        -- In this loop, you can perform operations on each dropped item.
        -- For now, just get the name and append a return character.
        set iPath to (i as text)
        set pathNamesString to pathNamesString & iPath & return
    end repeat
    -- Store list in open document, to verify what was dropped.
    tell application "TextEdit"
        set paragraph 1 of front document to pathNamesString
    end tell
    return
end open
```

Files, folders, or disks are not moved, copied, or affected in any way by merely dropping them on a script application. However, the script application’s handler can tell Finder to move, copy, or otherwise manipulate the items. For examples that work with Finder items, see [“Folder Actions Reference”](#) (page 221).

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 77).

## idle and quit Handlers for Stay-Open Applications

---

By default, a script application that receives a `run` or `open` command handles that single command and then quits. In contrast, a stay-open script application (one saved as Stay Open in Script Editor) stays open after it is launched.

A stay-open script application can be useful for several reasons:

- 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 it is 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.

Two particular handlers that stay-open script applications often provide are an `idle` handler and a `quit` handler.

### idle Handlers

---

If a stay-open script application includes an `idle` handler, AppleScript sends the script application periodic `idle` commands—by default, every 30 seconds—allowing it to perform background tasks when it is not performing other actions.

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. You can return 0 to maintain the default delay of 30 seconds.

For example, when saved as a stay-open application, the following script beeps every 5 seconds:

```
on idle
    beep
    return 5
end idle
```

The result returned from a handler is just the result of the last statement, even if it doesn't include the word `return` explicitly. (See “[return](#)” (page 214) for more information.) For example, this handler gets called once a minute, because the value of the last statement is 60:

```
on idle
    set x to 10
    beep
    set x to x * 6 -- The handler returns the result (60).
end idle
```

### 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.

**Note:** The `continue` statement passes control back to the application's default `quit` handler. For more information, see “[continue](#)” (page 213).

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 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 to be impossible to quit the application. For example, if the handler shown above gets an error before the `continue quit` statement, the application won't quit. If necessary, you can use Force Quit (Command-Option-Esc) to halt the application.

## Calling a Script Application From a Script

A script can send commands to a script application just as it can to other applications. To launch a non-stay-open application and run its script, use a `launch` (page 134) 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 command, sends back a reply if necessary, and quits.

Similarly, to launch a non-stay-open application and run its `stringTest` handler (which takes a `text` object as a parameter), use a `launch` command followed by a `stringTest` command, like this:

```
tell application "NonStayOpen"
    launch
    stringTest("Some example text.")
end tell
```

For information on how to create script applications, see “[Handlers in Script Applications](#)” (page 73).



# Class Reference

---

A **class** is a category for objects that share characteristics. AppleScript defines classes for common objects used in AppleScript scripts, such as aliases, Boolean values, integers, text, and so on.

Each object in a script is an instance of a specific class and has the same properties (including the `class` property), can contain the same kinds of elements, and supports the same kinds of operations and coercions as other objects of that type. Objects that are instances of AppleScript types can be used anywhere in a script—they don't need to be within a `tell` block that specifies an application.

Scriptable applications also define their own classes, such as windows and documents, which commonly contain properties and elements based on many of the basic AppleScript classes described in this chapter. Scripts obtain these objects in the context of the applications that define them. For more information on the class types applications typically support, see “Standard Classes” in Technical Note TN2106, [Scripting Interface Guidelines](#).

## alias

---

A persistent reference to an existing file, folder, or volume in the file system.

For related information, see [file](#) (page 88), [POSIX file](#) (page 93), and [“Aliases and Files”](#) (page 41).

### Properties of alias objects

`class`

Access: read only

Class: [class](#) (page 83)

The class identifier for the object. The value is always `alias`.

`POSIX path`

Access: read only

Class: [text](#) (page 97)

The POSIX-style path to the object.

### Coercions Supported

AppleScript supports coercion of an `alias` object to a [text](#) (page 97) object or single-item [list](#) (page 89).

### Examples

```
set zApp to choose application as alias -- (then choose Finder.app)
--result: alias "Leopard:System:Library:CoreServices:Finder.app:"
class of zApp --result: alias
zApp as text --result: "Leopard:System:Library:CoreServices:Finder.app:"
zApp as list --result: {alias "Leopard:System:Library:CoreServices:Finder.app:"}
```

You can use the `POSIX path` property to obtain a POSIX-style path to the item referred to by an alias:

```
POSIX path of zApp --result: "/System/Library/CoreServices/Finder.app/"
```

### Discussion

You can only create an alias to a file or folder that already exists.

### Special Considerations

AppleScript 2.0 attempts to resolve aliases only when you run a script. However, in earlier versions, AppleScript attempts to resolve aliases at compile time.

## application

---

An application on a local machine or an available server.

An application object in a script has all of the properties described here, which are handled by AppleScript. It may have additional properties, depending on the specific application it refers to.

### Properties of application objects

`class`

Access: read only

Class: `class` (page 83)

The class identifier for the object. The value is always `application`.

`frontmost`

Access: read only

Class: `boolean` (page 82)

Is the application frontmost?

Starting in AppleScript 2.0, accessing an application's `frontmost` property returns a Boolean value without launching the application or sending it an event.

The value of `frontmost` for background-only applications, UI element applications such as System Events, and applications that are not running is always `false`.

`id`

Access: read only

Class: `text` (page 97)

The application's bundle identifier (the default) or its four-character signature code. (New in AppleScript 2.0.)

For example, the bundle identifier for the TextEdit application is `"com.apple.TextEdit"`. Its four-character signature code is `'ttxx'`. If you ask for an application object's `id` property, you will get the bundle identifier version, unless the application does not have a bundle identifier and does have a signature code.

`name`

Access: read only

Class: `text` (page 97)

The application's name.

Starting in AppleScript 2.0, accessing an application's `name` property returns the application name as text without launching the application or sending it an event.



```
running
```

Access: read only

Class: [boolean](#) (page 82)

Is the application running? (New in AppleScript 2.0.)

Accessing an application's `running` property returns a Boolean value without launching the application or sending it an event.

You can also ask the System Events utility application whether an application is running. While it requires more lines in your script to do so, that option is available in earlier versions of the Mac OS.

```
version
```

Access: read only

Class: [text](#) (page 97)

The application's version.

Starting in AppleScript 2.0, accessing this property returns the application version as text without launching the application or sending it an event.

### Coercions Supported

AppleScript supports coercion of an `application` object to a single-item [list](#) (page 89).

### Examples

You can determine whether an application on the current computer is running without launching it (this won't work if your target is on a remote computer):

```
tell application "iTunes" -- doesn't automatically launch app
  if it is running then
    pause
  end if
end tell
```

You can also use this format:

```
if application "iTunes" is running
  tell application "iTunes" to pause
end if
```

The following statements specify the TextEdit application by, respectively, its signature, its bundle id, and by a POSIX path to a specific version of TextEdit:

```
application id "ttxx"
application id "com.apple.TextEdit"
application "/Applications/TextEdit.app"
```

You can target a remote application with a `tell` statement. For details, see [“Remote Applications”](#) (page 44).

### Special Considerations

Starting in Mac OS X v10.5, there are several changes in application behavior:

- Applications launch hidden.

AppleScript has always launched applications if it needed to in order to send them a command. However, they would always launch visibly, which could be visually disruptive. AppleScript now launches applications hidden by default. They will not be visible unless the script explicitly says otherwise using `activate`.

- Applications are located lazily.

When running a script, AppleScript will not attempt to locate an application until it needs to in order to send it a command. This means that a compiled script or script application may contain references to applications that do not exist on the user's system, but AppleScript will not ask where the missing applications are until it encounters a relevant `tell` block. Previous versions of AppleScript would attempt to locate every referenced application before running the script.

When opening a script for editing, AppleScript will attempt to locate all the referenced applications in the entire script, which may mean asking where one is. Pressing the Cancel button only cancels the search for that application; the script will continue opening normally, though custom terminology for that application will display as raw codes. In older versions, pressing Cancel would cancel opening the script.

- Applications are located and re-located dynamically.

Object specifiers that refer to applications, including those in `tell` blocks, are evaluated every time a script runs. This alleviates problems with scripts getting "stuck" to a particular copy of an application.

In prior versions of AppleScript, use of the new built-in application properties will fall back to sending an event to the application, but the application may not handle these properties in the same way, or handle them at all. (Most applications will handle `name`, `version`, and `frontmost`; `id` and `running` are uncommon.) The other new features described above require AppleScript 2.0.

## boolean

---

A logical truth value.

A `boolean` object evaluates to one of the AppleScript constants `true` or `false`. A **Boolean expression** contains one or more `boolean` objects and evaluates to `true` or `false`.

### Properties of boolean objects

`class`

Access: read only

Class: `class` (page 83)

The class identifier for the object. The value is always `boolean`.

### Operators

The operators that take `boolean` objects as operands are `and`, `or`, `not`, `&`, `=`, and `<`, as well as their text equivalents: `is equal to`, `is not equal to`, `equals`, and `so on`.

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

The binary operators `and` and `or` take `boolean` objects 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`.

The concatenation operator (`&`) creates a list containing the two boolean values on either side of it; for example:

```
true & false --result: {true, false}
```

For additional information on these operators, see [“Operators Reference”](#) (page 179).

### Coercions Supported

AppleScript supports coercion of a `boolean` object to a single-item `list` (page 89), a `text` (page 97) object, or an `integer` (page 89).

### Examples

The following are simple Boolean expressions:

```
true
false
paragraphCount > 2
```

AppleScript supplies the Boolean constants `true` and `false` to serve as the result of evaluating a Boolean operation. But scripts rarely need to use these literals explicitly because a Boolean expression itself evaluates to a Boolean value. For example, consider the following two script snippets:

```
if companyName is equal to "Acme Baking" then
    return true
else
    return false
end if

return companyName is equal to "Acme Baking"
```

The second, simpler version, just returns the value of the Boolean comparison `companyName is equal to "Acme Baking"`, so it doesn't need to use a Boolean constant.

### Discussion

When you pass a Boolean value as a parameter to a command, the form may change when you compile the command. For example, the following line

```
choose folder showing package contents true
```

is converted to this when compiled by AppleScript:

```
choose folder with showing package contents
```

It is standard for AppleScript to compile parameter expressions from the Boolean form (such as `showing package contents true` or `invisibles false`) into the `with form` (`with showing package contents` or `without invisibles`, respectively).

## class

---

Specifies the class of an object or value.

All classes have a `class` property that specifies the class type. The value of the `class` property is an identifier.

**Properties of class objects**`class`

Access: read only

Class: `class` (page 83)The class identifier for the object. The value of this property is always `class`.**Operators**The operators that take class identifier values as operands are `&`, `=`, `,`, and `as`.The coercion operator `as` takes an object of one class type and coerces it to an object of a type specified by a class identifier. For example, the following statement coerces a `text` object into a corresponding `real`:

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

**Coercions Supported**AppleScript supports coercion of a class identifier to a single-item `list` (page 89) or a `text` (page 97) object.**Examples**Asking for the class of a type such as `integer` results in a value of `class`:

```
class of text --result: class
class of integer --result: class
```

Here is the class of a boolean literal:

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

And here are some additional examples:

```
class of "Some text" --result: text
class of {1, 2, "hello"} --result: list
```

**constant**

---

A word with a predefined value.

Constants are generally used for enumerated types. You cannot define constants in scripts; constants can be defined only by applications and by AppleScript. See [“Global Constants in AppleScript”](#) (page 37) for more information.**Properties of constant objects**`class`

Access: read-only

Class: `class` (page 83)The class identifier for the object. The value of this property is always `constant`.**Operators**The operators that take `constant` objects as operands are `&`, `=`, `,`, and `as`.

**Coercions Supported**

AppleScript supports coercion of a constant object to a single-item `list` (page 89) or a `text` (page 97) object.

**Examples**

One place you use constants defined by AppleScript is in text comparisons performed with `considering` or `ignoring` statements (described in [considering / ignoring \(text comparison\)](#) (page 193)). For example, in the following script statements, `punctuation`, `hyphens`, and `white space` are constants:

```
considering punctuation but ignoring hyphens and white space
    "bet-the farm," = "BetTheFarm," --result: true
end considering
class of hyphens --result: constant
```

The final statement shows that the class of `hyphens` is constant.

**Discussion**

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

Literal constants are defined in [“Literals and Constants”](#) (page 20).

In addition to the constants defined by AppleScript, applications often define enumerated types to be used for command parameters or property values. For example, the iTunes `search` command defines these constants for specifying the search area:

```
albums
all
artists
composers
displayed
songs
```

**date**


---

Specifies the day of the week, the date (month, day of the month, and year), and the time (hours, minutes, and seconds).

To get the current date, use the command `current date` (page 123):

```
set theDate to current date
--result: "Friday, November 9, 2007 11:35:50 AM"
```

You can get and set the different parts of a `date` object through the `date` and `time` properties described below.

When you compile a script, AppleScript displays date and time values according to the format specified in System Preferences.

**Properties of date objects**

```
class
Access: read only
Class: class (page 83)
```

The class identifier for the object. The value of this property is always `date`.

`day`

Access: read/write

Class: `integer` (page 89)

Specifies the day of the month of a `date` object.

`weekday`

Access: read only

Class: `constant` (page 84)

Specifies the day of the week of a `date` object, with one of these constants: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, or Sunday.

`month`

Access: read/write

Class: `constant` (page 84)

Specifies the month of the year of a `date` object, with one of the constants January, February, March, April, May, June, July, August, September, October, November, or December.

`year`

Access: read/write

Class: `integer` (page 89)

Specifies the year of a `date` object; for example, 2004.

`time`

Access: read/write

Class: `integer` (page 89)

Specifies the number of seconds since midnight of a `date` object; for example, 2700 is equivalent to 12:45 AM (2700 / 60 seconds = 45 minutes).

`date string`

Access: read only

Class: `text` (page 97)

A `text` object that specifies the date portion of a `date` object; for example, "Friday, November 9, 2007".

To obtain a compact version of the date, use `short date string`. For example, `short date string of (current date) --result: "1/27/08"`.

`time string`

Access: read only

Class: `text` (page 97)

A `text` object that specifies the time portion of a `date` object; for example, "3:20:24 PM".

### Operators

The operators that take `date` object 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.

AppleScript supports the following operations on `date` objects with the `+` and `-` operators:

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

where `timeDifference` is an `integer` (page 89) 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 "Friday, November 9, 2007" + 4 * days + 3 * hours + 2 * minutes
--result: date "Tuesday, November 13, 2007 3:02:00 AM"
```

To express a time difference in more convenient form, divide the number of seconds by the appropriate constant:

```
31449600 / weeks --result: 52.0
```

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 `time to GMT` (page 163) command.

### Coercions Supported

AppleScript supports coercion of a `date` object to a single-item `list` (page 89) or a `text` (page 97) object.

### Examples

The following expressions show some options for specifying a date, along with the results of compiling the statements. If you construct a date using only partial information, AppleScript fills in the missing pieces with default values. The actual format is based on the settings in System Preferences.

```
date "8/9/2007, 17:06"
--result: date "Thursday, August 9, 2007 5:06:00 PM"
date "7/16/70"
--result: date "Wednesday, July 16, 2070 12:00:00 AM"
date "12:06" -- specifies a time on the current date
--result: date "Friday, November 9, 2007 12:06:00 PM"
date "Sunday, December 12, 1954 12:06 pm"
--result: date "Sunday, December 12, 1954 12:06:00 PM"
```

The following statements access various date properties (results depend on the date the statements are executed):

```
set theDate to current date --using current date command
--result: date "Friday, November 9, 2007 11:58:38 AM"
weekday of theDate --result: Friday
day of theDate --result: 9
month of theDate --result: November
year of theDate --result: 2007
time of theDate --result: 43118 (seconds since 12:00:00 AM)
time string of theDate --result: "11:58:38 AM"
date string of theDate --result: "Friday, November 9, 2007"
```

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 "Jan 1, 2008"
--result: date "Tuesday, January 1, 2008 2:30:00 AM"
date "2:30 am" of date "Sun Jan 27, 2008"
--result: date "Sunday, January 27, 2008 2:30:00 AM"
date "Nov 19, 2007" relative to date "3PM"
--result: date "Monday, November 19, 2007 3:00:00 PM"
date "1:30 pm" in date "April 1, 2008"
--result: date "Tuesday, April 1, 2008 1:30:00 PM"
```

### Special Considerations

You can create a date object using a string that follows the date format specified in the Formats pane in International preferences. For example, in US English:

```
set myDate to date "3/4/2008"
```

When you compile this statement, it is converted to the following:

```
set myDate to date "Tuesday, March 4, 2008 12:00:00 AM"
```

## file

---

A reference to a file, folder, or volume in the file system. A `file` object has exactly the same attributes as an `alias` object, with the addition that it can refer to an item that does not exist.

For related information, see [alias](#) (page 79) and [POSIX file](#) (page 93). For a description of the format for a file path, see ["Aliases and Files"](#) (page 41).

### Coercions Supported

AppleScript supports coercion of a `file` object to a `text` (page 97) object or single-item `list` (page 89).

### Examples

```
set fp to open for access file "Leopard:Users:myUser:NewFile"
close access fp
```

### Discussion

You can create a `file` object that refers to a file or folder that does not exist. For example, you can use the `choose file name` (page 114) command to obtain a `file` object for a file that need not currently exist.



## integer

---

A number without a fractional part.

### Properties of integer objects

`class`

Access: read-only

Class: `class` (page 83)

The class identifier for the object. The value of this property is always `integer`.

### Operators

The operators that can have `integer` values as operands are `+`, `-`, `*`, `÷` (or `/`), `div`, `mod`, `^`, `=`, `,`, `>`, `≥`, `<`, and `≤`.

The `div` operator always returns an `integer` value as its result. The `+`, `-`, `*`, `mod`, and `^` operators return values of type `integer` or `real`.

### Coercions Supported

AppleScript supports coercion of an `integer` value to a single-item `list` (page 89), a `real` (page 93) number, or a `text` (page 97) object.

Coercion of an `integer` to a number does nothing:

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

### Examples

```
1
set myResult to 3 - 2
-1
1000
```

### Discussion

The biggest value (positive or negative) that can be expressed as an integer in AppleScript is  $\pm 536870911$ , which is equal to  $\pm(2^{29} - 1)$ . Larger integers are converted to real numbers, expressed in exponential notation, when scripts are compiled.

**Note:** The smallest possible `integer` value is actually  $-536870912$  ( $-2^{29}$ ), but it can only be generated as a result of an expression. If you enter it directly into a script, it will be converted to a real when you compile.

## list

---

An ordered collection of values. The values contained in a list are known as items. Each item can belong to any class.

A list appears in a script as a series of expressions contained within braces and separated by commas. An empty list is a list containing no items. It is represented by a pair of empty braces: `{}`.

**Properties of list objects**`class`

Access: read-only

Class: `class` (page 83)The class identifier for the object. The value of this property is always `list`.`length`

Access: read only

Class: `integer` (page 89)

Specifies the number of items in the list.

`rest`

Access: read only

Class: `list` (page 89)

A list containing all items in the list except the first item.

`reverse`

Access: read only

Class: `list` (page 89)

A list containing all items in the list, but in the opposite order.

**Elements of list objects**`item`

A value contained in the list. Each value contained in a list is an item and an item may itself be another list. You can refer to values by their item numbers. For example, `item 2` of `{"soup", 2, "nuts"}` is the integer `2`.

You can also refer to indexed list items by class. For example, `integer 1` of `{"oatmeal", 42, "new"}` returns `42`.

**Operators**

The operators that can have list values as operands are `&`, `=`, `,`, `starts with`, `ends with`, `contains`, and `is contained by`.

For detailed explanations and examples of how AppleScript operators treat lists, see [“Operators Reference”](#) (page 179).

**Commands Handled**

You can count the items in a list or the elements of a specific class in a list with the `count` (page 122) command. You can also use the `length` property of a list:

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

**Coercions Supported**

AppleScript supports coercion of a single-item list to any 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 `text` (page 97) object if each of the items in the list can be coerced to a `text` object, as in the following example:

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

The resulting `text` object 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, see “[text item delimiters](#)” (page 38).

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

### Examples

The following statement defines a list that contains a `text` object, an integer, and a Boolean value:

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

Each list item can be any valid expression. The following list has the same value as the previous list:

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

The following statements work with lists; note that the concatenation operator (&) joins two lists into a single list:

```
class of {"this", "is", "a", "list"} --result: list
item 3 of {"this", "is", "a", "list"} --result: "a"
items 2 thru 3 of {"soup", 2, "nuts"} --result: {2, "nuts"}
{"This"} & {"is", "a", "list"} --result: {"This", "is", "a", "list"}
```

For large lists, it is more efficient to use the `a` reference to operator when inserting a large number of items into a list, rather than to access the list directly. For example, using direct access, the following script takes about 10 seconds to create a list of 10,000 integers (results will vary depending on the computer and other factors):

```
set bigList to {}
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 bigList
    -- DON'T DO THE FOLLOWING--it's even slower!
    -- set bigList to bigList & n
end
set total to (time of (current date)) - t --End timing
```

But the following script, which uses the `a` reference to operator, creates a list of 100,000 integers (ten times the size) in just a couple of seconds (again, results may vary):

```
set bigList to {}
set bigListRef to a reference to bigList
set numItems to 100000
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
```

Similarly, accessing the items in the previously created list is much faster using a reference to—the following takes just a few seconds:

```
set t to (time of (current date)) --Start timing
repeat with n from 1 to numItems -- where numItems = 100,000
    item n of bigListRef
end repeat
set total to (time of (current date)) - t --End timing
```

However, accessing the list directly, even for only 4,000 items, can take over a minute:

```
set numItems to 4000
set t to (time of (current date)) --Start timing
repeat with n from 1 to numItems
    item n of bigList
end repeat
set total to (time of (current date)) - t --End timing
```

## number

---

An abstract class that can represent an `integer` or a `real`.

There is never an object whose class is `number`; the actual class of a "number" object is always one of the more specific types, `integer` (page 89) or `real` (page 93).

### Properties of number objects

`class`

Access: read-only

Class: `class` (page 83)

The class identifier for the object. The value of this property is always either `integer` or `real`.

### 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 89) or `real` (page 93) classes.

### Coercions Supported

Coercing an object to `number` results in an `integer` object if the result of the coercion is an `integer`, or a `real` object if the result is a non-integer number.

### Examples

Any valid literal expression for an `integer` or a `real` value is also a valid literal expression for a `number` value:

```
1
2
-1
1000
10.2579432
1.0
1.
```

## POSIX file

---

A pseudo-class equivalent to the `file` class.

There is never an object whose class is `POSIX file`; the result of evaluating a POSIX file specifier is a `file` object. The difference between `file` and `POSIX file` objects is in how they interpret name specifiers: a `POSIX file` object interprets "name" as a POSIX path, while a `file` object interprets it as an HFS path.

For related information, see [alias](#) (page 79) and [file](#) (page 88). For a description of the format for a POSIX path, see “Aliases and Files” (page 41).

### Properties of POSIX file objects

See [file](#) (page 88).

### Coercions Supported

See [file](#) (page 88).

### Examples

The following example asks the user to specify a file name, starting in the temporary directory `/tmp`, which is difficult to specify using a file specifier:

```
set fileName to choose file name default location (POSIX file "/tmp")
    -result: dialog starts in /tmp folder
```

## real

---

Numbers that can include a fractional part, such as 3.14159 and 1.0.

### Properties of real objects

`class`

Access: read-only

Class: `class` (page 83)

The class identifier for the object. The value of this property is always `real`.

### Operators

The operators that can have `real` values as operands are `+`, `-`, `*`, `÷` (or `/`), `div`, `mod`, `^`, `=`, `>`, `≥`, `<`, and `≤`.

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.

### Coercions Supported

AppleScript supports coercion of a `real` value to an `integer` value, rounding any fractional part.

AppleScript also supports coercion of a `real` value to a single-item [list](#) (page 89) or a [text](#) (page 97) object. Coercion to text uses the decimal separator specified in Numbers in the Formats pane in International preferences.

**Examples**

```
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
```

**Discussion**

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**


---

An unordered collection of labeled properties. The only AppleScript classes that support user-defined properties are `record` and `script`.

A record appears in a script as a series of property definitions contained within braces and separated by commas. Each property definition consists of a label, a colon, and the value of the property. For example, this is a record with two properties: `{product:"pen", price:2.34}`.

Each property in a record has a unique label which distinguishes it from other properties in the collection. The values assigned to properties can belong to any class. You can change the class of a property simply by assigning a value belonging to another class.

**Properties of record objects**

`class`

Access: read/write

Class: `class` (page 83)

The class identifier for the record. By default, the value is `record`.

If you define a `class` property explicitly in a record, the value you define replaces the implicit `class` value. In the following example, the class is set to `integer`:

```
set myRecord to {class:integer, min:1, max:10}
class of myRecord --result: integer
```

`length`

Access: read only

Class: `integer` (page 89)

Specifies the number of properties in the record.

**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 Reference”](#) (page 179).

**Commands Handled**

You can count the properties in a record with the `count` command:

```
count {name:"Robin", mileage:400} --result: 2
```

**Coercions Supported**

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

**Examples**

The following example shows how to change the value of a property in a record:

```
set myRecord to {product:"pen", price:2.34}
product of myRecord -- result: "pen"
```

```
set product of myRecord to "pencil"
product of myRecord -- result: "pencil"
```

AppleScript evaluates expressions in a record before using the record in other expressions. For example, the following two records are equivalent:

```
{ name:"Steve", height:76 - 1.5, weight:150 + 20 }
{ name:"Steve", height:74.5, weight:170 }
```

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

```
item 2 of { name:"Rollie", IQ:186, city:"Unknown" } --result: error
```

You can access the `length` property of a record to count the properties it contains:

```
length of {name:"Chris", mileage:1957, city:"Kalamazoo"} --result: 3
```

You can get the same value with the `count` (page 122) command:

```
count {name:"Chris", mileage:1957, city:"Kalamazoo"} --result: 3
```

**Discussion**

After you define a record, you cannot add additional properties to it. You can, however, concatenate records. For more information, see [& \(concatenation\)](#) (page 187).

## reference

An object that encapsulates an object specifier.

The result of the [a reference to](#) (page 188) operator is a reference object, and object specifiers returned from application commands are implicitly turned into reference objects.

A `reference object` “wraps” an object specifier. If you target a `reference object` with the `get` (page 129) command, the command returns the `reference object` itself. If you ask a `reference object` for its `contents` property, it returns the enclosed object specifier. All other requests to a `reference object` are forwarded to its enclosed object specifier. For example, if you ask for the `class` of a `reference object`, you get the `class` of the object specified by its object specifier.

For related information, see “Object Specifiers” (page 29).

### Properties of reference objects

Other than the `contents` property, all other property requests are forwarded to the enclosed object specifier, so the `reference object` appears to have all the properties of the referenced object.

`contents`

Access: depends on the referenced object or objects

Class: depends on the referenced object or objects

The enclosed object specifier.

### Operators

All operators are forwarded to the enclosed object specifier, so the `reference object` appears to support all the operators of referenced object.

The `a reference to` operator returns a `reference object` as its result.

### Coercions Supported

All coercions are forwarded to the enclosed object specifier, so the `reference object` appears to support all the coercions of referenced object.

### Examples

Reference objects are most often used to specify application objects. The following example creates a `reference` to a window within the `TextEdit` application:

```
set myWindow to a ref to window "top.rtf" of application "TextEdit"
--result: window "top.rtf" of application "TextEdit"
```

In subsequent script statements, you can use the variable `myWindow` in place of the longer term `window "top.rtf" of application "TextEdit"`.

Because all property requests other than `contents` of are forwarded to its enclosed specifier, the `reference object` appears to have all the properties of the referenced object. For example, both `class` of statements in the following example return `window`:

```
set myRef to a reference to window 1
class of contents of myRef -- explicit dereference using "contents of"
class of myRef -- implicit dereference
```

For additional examples, see the `a reference to` (page 188) operator.

### RGB color

---

A type definition for a three-item list of `integer` values, from 0 to 65535, that specify the red, green, and blue components of a color.



Otherwise, behaves exactly like a [list](#) (page 89) object.

### Examples

```
set whiteColor to {65535, 65535, 65535} -- white
set yellowColor to {65535, 65535, 0} -- yellow
yellowColor as string --result: "65535655350"
set redColor to {65535, 0, 0} -- red
set userColor to choose color default color redColor
```

## script

---

A collection of AppleScript declarations and statements that can be executed as a group.

The syntax for a `script` object is described in “[Defining Script Objects](#)” (page 57).

### Properties of script objects

`class`

Access: read-only

Class: `class` (page 83)

The class identifier for the object. The value of this property is always `script`.

### Commands Handled

You can copy a `script` object with the [copy](#) (page 121) command or create a reference to it with the [set](#) (page 154) command.

### Coercions Supported

AppleScript supports coercion of a `script` object to a single-item [list](#) (page 89).

### Examples

The following example shows a simple `script` object that displays a dialog. It is followed by a statement that shows how to run the script:

```
script helloScript
    display dialog "Hello."
end script

run helloScript -- invoke the script
```

### Discussion

A `script` object can contain other `script` objects, called child scripts, and can have a parent object. For additional information, including more detailed examples, see “[Script Objects](#)” (page 57).

## text

---

An ordered series of Unicode characters.

Starting in AppleScript 2.0, AppleScript is entirely Unicode-based. There is no longer a distinction between Unicode and non-Unicode text. Comments and text constants in scripts may contain any Unicode characters, and all text processing is done in Unicode, so all characters are preserved correctly regardless of the user’s language preferences.

For example, the following script works correctly in AppleScript 2.0, where it would not have in previous versions:

```
set jp to " "
set ru to " "
jp & " and " & ru -- returns "  and "
```

For information on compatibility with previous AppleScript versions, including the use of `string` and `Unicode text` as synonyms for `text`, see the Special Considerations section.

### Properties of text objects

`class`

Access: read-only

Class: `class` (page 83)

The class identifier for the object. The value of this property is always `text`.

`id`

Access: read-only

Class: `integer` (page 89) or `list` (page 89) of integer

A value (or list of values) representing the Unicode code point (or code points) for the character (or characters) in the `text` object. (A **Unicode code point** is a unique number that represents a character and allows it to be represented in an abstract way, independent of how it is rendered. A character in a `text` object may be composed of one or more code points.)

This property, added in AppleScript 2.0, can also be used as an address, which allows mapping between Unicode code point values and the characters at those code points. For example, `id of "A"` returns 65, and `character id 65` returns "A".

The `id` of `text` longer than one code point is a list of integers, and vice versa: for example, `id of "hello"` returns {104, 101, 108, 108, 111}, and `string id {104, 101, 108, 108, 111}` returns "hello". (Because of a bug, `text id ...` does not work; you must use one of `string`, `Unicode text`, or `character`.)

These uses of the `id` property obsolete the older `ASCII character` (page 108) and `ASCII number` (page 109) commands, since, unlike those, they cover the full Unicode character range and will return the same results regardless of the user's language preferences.

`length`

Access: read only

Class: `integer` (page 89)

The number of characters in the text.

`quoted form`

Access: read only

Class: `text` (page 97)

A representation of the text that is safe from further interpretation by the shell, no matter what its contents are. Mainly useful for passing a text string to the `do shell script` (page 128) command.

**Elements of text objects**

A `text` object can contain these elements (which may behave differently than similar elements used in applications):

`character`

Specify by: [“Arbitrary”](#) (page 167), [“Every”](#) (page 168), [“Index”](#) (page 172), [“Middle”](#) (page 173), [“Range”](#) (page 175)

One or more Unicode characters that make up the text.

Starting in AppleScript 2.0, elements of `text` object count a combining character cluster (also known as a Unicode grapheme cluster) as a single character. (This relates to a feature of Unicode that is unlikely to have an impact on most scripters: some “characters” may be represented as either a single entity or as a base character plus a series of combining marks.

For example, “é” may be encoded as either U+00E9 (LATIN SMALL LETTER E WITH ACUTE) or as U+0065 (LATIN SMALL LETTER E), U+0301 (COMBINING ACUTE ACCENT). Nonetheless, AppleScript 2.0 will count both as one character, where older versions counted the base character and combining mark separately.

`paragraph`

Specify by: [“Arbitrary”](#) (page 167), [“Every”](#) (page 168), [“Index”](#) (page 172), [“Middle”](#) (page 173), [“Range”](#) (page 175)

A series of characters beginning immediately after either the first character after the end of the preceding paragraph or the beginning of the text and ending with either a carriage return character (`\r`), a linefeed character (`\n`), a return/linefeed pair (`\r\n`), or the end of the text. The Unicode “paragraph separator” character (U+2029) is not supported.

Because `paragraph` elements are *separated* by a carriage return, linefeed, or carriage return/linefeed pair, text ending with a paragraph break specifies a following (empty) paragraph. For example, `"this\nthat\n"` has three paragraphs, not two: “this”, “that”, and “” (the empty paragraph after the trailing linefeed).

Similarly, two paragraph breaks in a row specify an empty paragraph between them:

```
paragraphs of "this\n\nthat" --result: {"this", "", "that"}
```

`text`

Specify by: [“Every”](#) (page 168), [“Name”](#) (page 173)

All of the text contained in the `text` object, including spaces, tabs, and all other characters.

You can use `text` to access contiguous characters (but see also the Discussion section below):

```
text 1 thru 5 of "Bring me the mouse." --result: "Bring"
```

`word`

Specify by: [“Arbitrary”](#) (page 167), [“Every”](#) (page 168), [“Index”](#) (page 172), [“Middle”](#) (page 173), [“Range”](#) (page 175)

A continuous series of characters, with word elements parsed according to the word-break rules set in the International preference pane.

Because the rules for parsing words are thus under user control, your scripts should not count on a deterministic text parsing of words.

### Operators

The operators that can have `text` objects as operands are `&`, `=`, `,`, `>`, `≥`, `<`, `≤`, `starts with`, `ends with`, `contains`, `is contained by`, and `as`.

In text comparisons, you can specify whether white space should be considered or ignored. For more information, see [“considering and ignoring Statements”](#) (page 193).

For detailed explanations and examples of how AppleScript operators treat `text` objects, see [“Operators Reference”](#) (page 179).

### Special String Characters

The backslash (`\`) and double-quote (`"`) characters have special meaning in text. AppleScript encloses text in double-quote characters and uses the backslash character to represent return (`\r`), tab (`\t`), and linefeed (`\n`) characters (described below). So if you want to include an actual backslash or double-quote character in a `text` object, you must use the equivalent two-character sequence. As a convenience, AppleScript also provides the text constant `quote`, which has the value `\`.

**Table 6-1** Special characters in text

Character	To insert in text
Backslash character ( <code>\</code> )	<code>\\</code>
Double quote ( <code>"</code> )	<code>\"</code> <code>quote</code> (text constant)

To declare a `text` object that looks like this when displayed:

```
He said "Use the \'\' character."
```

you can use the following:

```
"He said \"Use the '\\\' character.\""
```

White space refers to text characters that display as vertical or horizontal space. AppleScript defines the white space constants `return`, `linefeed`, `space`, and `tab` to represent, respectively, a return character, a linefeed character, a space character, and a tab character. (The `linefeed` constant became available in AppleScript 2.0.)

Although you effectively use these values as text constants, they are actually defined as properties of the global constant `AppleScript`.

**Table 6-2** White space constants

Constant	Value
<code>space</code>	<code>" "</code>
<code>tab</code>	<code>"\t"</code>
<code>return</code>	<code>"\r"</code>
<code>linefeed</code>	<code>"\n"</code>

To enter white space in a string, you can just type the character—that is, you can press the Space bar to insert a space, the Tab key to insert a tab character, or the Return key to insert a return. In the latter case, the string will appear on two lines in the script, like the following:

```
display dialog "Hello" & "
" & "Goodbye"
```

When you run this script, "Hello" appears above "Goodbye" in the dialog.

You can also enter a tab, return, or linefeed with the equivalent two-character sequences. When a `text` object containing any of the two-character sequences is displayed to the user, the sequences are converted. For example, if you use the following `text` object in a `display dialog` (page 125) command:

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

it is displayed like this (unless you enable "Escape tabs and line breaks in strings" in the Editing tab of the of Script Editor preferences):

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

To use the white space constants, you use the concatenation operator to join multiple `text` objects together, as in the following example:

```
"Year" & tab & tab & "Units sold" & return & "2006" & tab -
& tab & "300" & return & "2007" & tab & tab & "453"
```

When passed to `display dialog`, this text is displayed as follows:

```
Year      Units sold
2006      300
2007      453
```

### Coercions Supported

AppleScript supports coercion of an `text` object to a single-item `list` (page 89). If a `text` object represents an appropriate number, AppleScript supports coercion of the `text` object to an integer or a real number.

### Examples

You can define a `text` object in a script by surrounding text characters with quotation marks, as in these examples:

```
set theObject to "some text"
set clientName to "Mr. Smith"
display dialog "This is a text object."
```

Suppose you use the following statement to obtain a `text` object named `docText` that contains all the text extracted from a particular document:

```
set docText to text of document "MyFavoriteFish.rtf" of application "TextEdit"
```

The following statements show various ways to work with the `text` object `docText`:

```
class of docText --result: text
first character of docText --result: a character
every paragraph of docText --result: a list containing all paragraphs
paragraphs 2 thru 3 of docText --result: a list containing two paragraphs
```

The next example prepares a `text` object to use with the `display dialog` command. It uses the `quote` constant to insert `\` into the text. When this text is displayed in the dialog (above a text entry field), it looks like this: Enter the text in quotes ("text in quotes"):

```
set promptString to "Enter the text in quotes (" & quote ¬
    & "text in quotes" & quote & "): "
display dialog promptString default answer ""
```

The following example gets a POSIX path to a chosen folder and uses the `quoted form` property to ensure correct quoting of the resulting string for use with shell commands:

```
set folderName to quoted form of POSIX path of (choose folder)
```

Suppose that you choose the folder named `iWork '08` in your `Applications` folder. The previous statement would return the following result, which properly handles the embedded single quote and space characters in the folder name:

```
"/Applications/iWork '\''08/'"
```

### Discussion

To get a contiguous range of characters within a `text` object, use the `text` element. For example, the value of the following statement is the `text` object `"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 `text` object. For example, if you attempt to change the value of the first character of the `text` object `myName` as shown next, you'll get an error:

```
set myName to "Boris"
set character 1 of myName to "D"
--result: error: you cannot set the values of elements of text objects
```

However, you can achieve the same result 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"
```

This example doesn't actually modify the existing `text` object—it sets the variable `myName` to refer to a new `text` object with a different value.

### Special Considerations

For compatibility with versions prior to AppleScript 2.0, `string` and `Unicode text` are still defined, but are considered synonyms for `text`. For example, all three of these statements have the same effect:

```
someObject as text
someObject as string
someObject as Unicode text
```

In addition, `text`, `string`, and `Unicode text` will all compare as equal. For example, `class of "foo"` is `string` is true, even though `class of "foo"` returns `text`. However, it is still possible for applications to distinguish between the three different types, even though AppleScript itself does not.

Starting with AppleScript 2.0, there is no style information stored with `text` objects.

Because all `text` is `Unicode text`, scripts now always get the `Unicode text` behavior. This may be different from the former `string` behavior for some locale-dependent operations, in particular `word` elements. To get the same behavior with 2.0 and pre-2.0, add an explicit `as Unicode text` coercion, for example, `words of (someText as Unicode text)`.

Because `text item delimiters` (described in [“text item delimiters”](#) (page 38)) respect `considering` and `ignoring attributes` in AppleScript 2.0, delimiters are case-insensitive by default. Formerly, they were always case-sensitive. To enforce the previous behavior, add an explicit `considering case` statement.

Because AppleScript 2.0 scripts store all text as `Unicode`, any `text` constants count as a use of the former `Unicode text` class, which will work with any version of AppleScript back to version 1.3. A script that contains `Unicode`-only characters such as Arabic or Thai will run, but will not be correctly editable using versions prior to AppleScript 2.0: the `Unicode`-only characters will be lost.

## unit types

---

Used for working with measurements of length, area, cubic and liquid volume, mass, and temperature.

The unit type classes support simple objects that do not contain other values and have only a single property, the `class` property.

### Properties of unit type objects

`class`

Access: read only

Class: (varies; listed below)

The class identifier for the object. These are the available classes:

**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`

### Operators

None. You must explicitly coerce a unit type to a number type before you can perform operations with it.

**Coercions Supported**

You can coerce a unit type object to `integer` (page 89), single-item `list` (page 89), `real` (page 93), or `text` (page 97). 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.

**Examples**

The following statements calculate the area of a circle with a radius of 7 yards, then coerce the area to square feet:

```
set circleArea to (pi * 7 * 7) as square yards --result: square yards
153.9380400259
circleArea as square feet --result: square feet 1385.4423602331
```

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 5.0 as square kilometers --result: square kilometers 5.0
theArea as square miles --result: square miles 1.930510792712
theArea as square meters --result: square meters 5.0E+6
```

However, you cannot coerce an area measurement to a unit type in a different category:

```
set theArea to 5.0 as square meters --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 text object:

```
set myPounds to 2.2 as pounds --result: pounds 2.2
set textValue to myPounds as text --result: "2.2"
```



# Commands Reference

This chapter describes the commands available to perform actions in AppleScript scripts. For information on how commands work, see [“Commands Overview”](#) (page 34).

The commands described in this chapter are available to any script—they are either built into the AppleScript language or added to it through the standard scripting additions (described in [“Scripting Additions”](#) (page 34)).

**Note:** In the command descriptions below, if the first item in the Parameters list does not include a parameter name, it is the direct parameter of the command (described in [“Direct Parameter”](#) (page 36)).

Table 7-1 lists each command according to the suite (or related group) of commands to which it belongs and provides a brief description. Detailed command descriptions follow the table, in alphabetical order.

**Table 7-1** AppleScript commands

Command	Description
<b>AppleScript suite</b>	
<code>activate</code> (page 108)	Brings an application to the front, and opens it if it is on the local computer and not already running.
<code>log</code> (page 138)	In Script Editor, displays a value in the Event Log History window or in the Event Log pane of a script window.
<b>Clipboard Commands suite</b>	
<code>clipboard info</code> (page 120)	Returns information about the clipboard.
<code>set the clipboard to</code> (page 156)	Places data on the clipboard.
<code>the clipboard</code> (page 162)	Returns the contents of the clipboard.
<b>File Commands suite</b>	
<code>info for</code> (page 132)	Returns information for a file or folder.
<code>list disks</code> (page 134)	Returns a list of the currently mounted volumes. <b>Deprecated</b> Use <code>tell application "System Events"</code> to get the name of every disk.
<code>list folder</code> (page 135)	Returns the contents of a specified folder. <b>Deprecated</b> Use <code>tell application "System Events"</code> to get the name of every disk item of ....

Command	Description
<code>mount volume</code> (page 138)	Mounts the specified AppleShare volume.
<code>path to (application)</code> (page 142)	Returns the full path to the specified application.
<code>path to (folder)</code> (page 143)	Returns the full path to the specified folder.
<code>path to resource</code> (page 145)	Returns the full path to the specified resource.
<b>File Read/Write suite</b>	
<code>close access</code> (page 120)	Closes a file that was opened for access.
<code>get eof</code> (page 131)	Returns the length, in bytes, of a file.
<code>open for access</code> (page 140)	Opens a disk file for the <code>read</code> (page 147) and <code>write</code> (page 163) commands.
<code>read</code> (page 147)	Reads data from a file that has been opened for access.
<code>set eof</code> (page 155)	Sets the length, in bytes, of a file.
<code>write</code> (page 163)	Writes data to a file that was opened for access with write permission.
<b>Internet suite</b>	
<code>open location</code> (page 141)	Opens a URL with the appropriate program.
<b>Miscellaneous Commands suite</b>	
<code>current date</code> (page 123)	Returns the current date and time.
<code>do shell script</code> (page 128)	Executes a shell script using the <code>sh</code> shell.
<code>get volume settings</code> (page 131)	Returns the sound output and input volume settings.
<code>random number</code> (page 146)	Generates a random number.
<code>round</code> (page 149)	Rounds a number to an integer.
<code>set volume</code> (page 157)	Sets the sound output and/or input volume.
<code>system attribute</code> (page 160)	Gets environment variables or attributes of this computer.
<code>system info</code> (page 161)	Returns information about the system.
<code>time to GMT</code> (page 163)	Returns the difference between local time and GMT (Universal Time).
<b>Scripting suite</b>	
<code>load script</code> (page 135)	Returns a <code>script</code> object loaded from a file.
<code>run script</code> (page 151)	Runs a script or script file

Command	Description
<a href="#">scripting components</a> (page 153)	Returns a list of all scripting components.
<a href="#">store script</a> (page 158)	Stores a <code>script</code> object into a file.
<b>Standard suite</b>	
<a href="#">copy</a> (page 121)	Copies one or more values into variables.
<a href="#">count</a> (page 122)	Counts the number of elements in an object.
<a href="#">get</a> (page 129)	Returns the value of a script expression or an application object.
<a href="#">launch</a> (page 134)	Launches the specified application without sending it a <code>run</code> command.
<a href="#">run</a> (page 151)	For an application, launches it. For a script application, launches it and sends it the <code>run</code> command. For a script script object, executes its <code>run</code> handler.
<a href="#">set</a> (page 154)	Assigns one or more values to one or more script variables or application objects.
<b>String Commands suite</b>	
<a href="#">ASCII character</a> (page 108)	Converts a number to a character. <b>Deprecated</b> starting in AppleScript 2.0. Use the <code>id</code> property of the <code>text</code> (page 97) class instead.
<a href="#">ASCII number</a> (page 109)	Converts a character to its numeric value. <b>Deprecated</b> starting in AppleScript 2.0. Use the <code>id</code> property of the <code>text</code> (page 97) class instead.
<a href="#">localized string</a> (page 136)	Returns the localized string for the specified key.
<a href="#">offset</a> (page 139)	Finds one piece of text inside another.
<a href="#">summarize</a> (page 159)	Summarizes the specified text or text file.
<b>User Interaction suite</b>	
<a href="#">beep</a> (page 110)	Beeps one or more times.
<a href="#">choose application</a> (page 110)	Allows the user to choose an application.
<a href="#">choose color</a> (page 111)	Allows the user to choose a color.
<a href="#">choose file</a> (page 112)	Allows the user to choose a file.
<a href="#">choose file name</a> (page 114)	Allows the user to specify a new file reference.
<a href="#">choose folder</a> (page 115)	Allows the user to choose a folder.

Command	Description
<a href="#">choose from list</a> (page 116)	Allows the user to choose one or more items from a list.
<a href="#">choose remote application</a> (page 118)	Allows the user to choose a running application on a remote machine.
<a href="#">choose URL</a> (page 119)	Allows the user to specify a URL.
<a href="#">delay</a> (page 123)	Pauses for a fixed amount of time.
<a href="#">display alert</a> (page 124)	Displays an alert.
<a href="#">display dialog</a> (page 125)	Displays a dialog box, optionally requesting user input.
<a href="#">say</a> (page 152)	Speaks the specified text.

## activate

---

Brings an application to the front, launching it if necessary.

### Syntax

```
activate application required
```

### Parameters

#### *application*

The application to activate.

### Result

None.

### Examples

```
activate application "TextEdit"
tell application "TextEdit" to activate
```

### Discussion

The `activate` command does not launch applications on remote machines. For examples of other ways to specify an application, see the [application](#) (page 80) class and [“Remote Applications”](#) (page 44).

## ASCII character

---

Returns the character for a specified number.

**Important:** This command is deprecated starting in AppleScript 2.0—use the `id` property of the `text` class instead.

### Syntax

ASCII character                      *integer*                      required

### Parameters

*integer* (page 89)

The character code, an integer between 0 and 255.

### Result

A *text* (page 97) object containing the character that corresponds to the specified number.

Signals an error if *integer* is out of range.

### Examples

```
set theChar to ASCII character 65 --result: "A"
set theChar to ASCII character 194 --result: "→"
set theChar to ASCII character 2040 --result: invalid range error
```

### Discussion

The name “ASCII” is something of a misnomer. *ASCII character* uses the primary text encoding, as determined by the user’s language preferences, to map between integers and characters. If the primary language is English, the encoding is Mac OS Roman, if it is Japanese, the encoding is MacJapanese, and so on. For integers below 128, this is generally the same as ASCII, but for integers from 128 to 255, the results vary considerably.

Because of this unpredictability, *ASCII character* and *ASCII number* are deprecated starting in AppleScript 2.0. Use the *id* property of the *text* class instead, since it always uses the same encoding, namely Unicode.

## ASCII number

---

Returns the number associated with a specified character.

**Important:** This command is deprecated starting in AppleScript 2.0—use the *id* property of the *text* class instead.

### Syntax

ASCII number                      *text*                      required

### Parameters

*text* (page 97)

A *text* object containing at least one character. If there is more than one character, only the first one is used.

### Result

The character code of the specified character as an integer.

### Examples

```
set codeValue to ASCII number "→" --result: 194
```

**Discussion**

The result of `ASCII number` depends on the user's language preferences; see the Discussion section of [ASCII character](#) (page 108) for details.

`beep`


---

Plays the system alert sound one or more times.

**Syntax**

<code>beep</code>		required
	<i>integer</i>	optional

**Parameters**

*integer* (page 89)  
Number of times to beep.

*Default Value:*

1

**Result**

None.

**Examples**

Audible alerts can be useful when no one is expected to be looking at the screen:

```
beep 3 --result: three beeps, to get attention
display dialog "Something is amiss here!" -- to show message
```

`choose application`


---

Allows the user to choose an application.

**Syntax**

<code>choose application</code>		required
<code>with title</code>	<i>text</i>	optional
<code>with prompt</code>	<i>text</i>	optional
<code>multiple selections allowed</code>	<i>boolean</i>	optional
<code>as</code>	<i>class</i>	optional

**Parameters**

with title *text* (page 97)

Title text for the dialog.

*Default Value:*

"Choose Application"

with prompt *text* (page 97)

A prompt to be displayed in the dialog.

*Default Value:*

"Select an application:"

multiple selections allowed *boolean* (page 82)

Allow multiple items to be selected? If *true*, the results will be returned in a list, even if there is exactly one item.

*Default Value:*

false

as *class* (*application* (page 80) | *alias* (page 79))

Specifies the desired class of the result. If specified, the value must be one of *application* or *alias*.

*Default Value:*

*application*

**Result**

The selected application, as either an *application* or *alias* object; for example, *application "TextEdit"*. If multiple selections are allowed, returns a list containing one item for each selected application, if any.

Signals a "user canceled" error if the user cancels the dialog. For an example of how to handle such errors, see "[try Statements](#)" (page 207).

**Examples**

```
choose application with prompt "Choose a web browser:"
choose application with multiple selections allowed
choose application as alias
```

**Discussion**

The `choose application` dialog initially presents a list of all applications registered with the system. To choose an application not in that list, use the Browse button, which allows the user to choose an application anywhere in the file system.

```
choose color
```

---

Allows the user to choose a color from a color picker dialog.

**Syntax**

```
choose color required
default color RGB color optional
```

**Parameters**

default color *RGB color* (page 96)

The color to show when the color picker dialog is first opened.

*Default Value:*

{0, 0, 0}: black.

**Result**

The selected color, represented as a list of three integers from 0 to 65535 corresponding to the red, green, and blue components of a color; for example, {0, 65535, 0} represents green.

Signals a “user canceled” error if the user cancels the `choose color` dialog. For an example of how to handle such errors, see “[try Statements](#)” (page 207).

**Examples**

This example lets the user choose a color, then uses that color to set the background color in their home folder (when it is in icon view):

```
tell application "Finder"
  tell icon view options of window of home
    choose color default color (get background color)
    set background color to the result
  end tell
end tell
```

**choose file**


---

Allows the user to choose a file.

**Syntax**

<code>choose file</code>		required
<code>with prompt</code>	<i>text</i>	optional
<code>of type</code>	<i>list of text</i>	optional
<code>default location</code>	<i>alias</i>	optional
<code>invisibles</code>	<i>boolean</i>	optional
<code>multiple selections allowed</code>	<i>boolean</i>	optional
<code>showing package contents</code>	<i>boolean</i>	optional

**Parameters**

with prompt *text* (page 97)

The prompt to be displayed in the dialog.

*Default Value:*

None; no prompt is displayed.



of type *list* (page 89) of *text* (page 97)

A list of Uniform Type Identifiers (UTIs); for example, {"public.html", "public.rtf"}. Only files of the specified types will be selectable. For a list of system-defined UTIs, see *Uniform Type Identifiers Overview*. To get the UTI for a particular file, use *info for* (page 132).

**Note:** Four-character file type codes, such as "PICT" or "MooV", are also supported, but are deprecated. To get the file type code for a particular file, use *info for* (page 132).

*Default Value:*

None; any file can be chosen.

default location *alias* (page 79)

The folder to begin browsing in.

*Default Value:*

Browsing begins in the last selected location, or, if this is the first invocation, in the user's Documents folder.

invisibles *boolean* (page 82)

Show invisible files and folders?

*Default Value:*

true: This is only for historical compatibility reasons. Unless you have a specific need to choose invisible files, you should always use *invisibles false*.

multiple selections allowed *boolean* (page 82)

Allow multiple items to be selected? If true, the results will be returned in a list, even if there is exactly one item.

*Default Value:*

false

showing package contents *boolean* (page 82)

Show the contents of packages? If true, packages are treated as folders, so that the user can choose a file inside a package (such as an application).

*Default Value:*

false. Manipulating the contents of packages is discouraged unless you control the package format or the package itself.

## Result

The selected file, as an *alias*. If multiple selections are allowed, returns a list containing one *alias* for each selected file, if any.

Signals a "user canceled" error if the user cancels the dialog. For an example of how to handle such errors, see "*try Statements*" (page 207).

## Examples

```
set aFile to choose file with prompt "HTML or RTF:" -
  of type {"public.html", "public.rtf"} invisibles false
```

A UTI can specify a general class of files, not just a specific format. The following script allows the user to choose any image file, whether its format is JPEG, PNG, GIF, or whatever. It also uses the *default location* parameter combined with *path to (folder)* (page 143) to begin browsing in the user's Pictures folder:

```
set picturesFolder to path to pictures folder
```

```
choose file of type "public.image" with prompt "Choose an image:" ↵
    default location picturesFolder invisibles false
```

## choose file name

---

Allows the user to specify a new filename and location. This does not create a file—rather, it returns a file specifier that can be used to create a file.

### Syntax

choose file name		required
with prompt	<i>text</i>	optional
default name	<i>text</i>	optional
default location	<i>alias</i>	optional

### Parameters

with prompt *text* (page 97)

The prompt to be displayed near the top of the dialog.

*Default Value:*

"Specify new file name and location"

default name *text* (page 97)

The default file name.

*Default Value:*

"untitled"

default location *alias* (page 79)

The default file location. See [choose file](#) (page 112) for examples.

*Default Value:*

Browsing starts in the last location in which a search was made or, if this is the first invocation, in the user's Documents folder.

### Result

The selected location, as a file. For example:

```
file "HD:Users:currentUser:Documents:untitled"
```

Signals a "user canceled" error if the user cancels the dialog. For an example of how to handle such errors, see ["try Statements"](#) (page 207).

### Examples

The following example supplies a non-default prompt and search location:

```
set fileName to choose file name with prompt "Save report as:" ↵
    default name "Quarterly Report" ↵
    default location (path to desktop folder)
```

**Discussion**

If you choose the name of a file or folder that exists in the selected location, `choose file name` offers the choice of replacing the chosen item. However, choosing to replace does not actually replace the item.

`choose folder`


---

Allows the user to choose a directory, such as a folder or a disk.

**Syntax**

<code>choose folder</code>		required
<code>with prompt</code>	<i>text</i>	optional
<code>default location</code>	<i>alias</i>	optional
<code>invisibles</code>	<i>boolean</i>	optional
<code>multiple selections allowed</code>	<i>boolean</i>	optional
<code>showing package contents</code>	<i>boolean</i>	optional

**Parameters**

`with prompt` [text](#) (page 97)

The prompt to be displayed in the dialog.

*Default Value:*

None; no prompt is displayed.

`default location` [alias](#) (page 79)

The folder to begin browsing in.

*Default Value:*

Browsing begins in the last selected location, or, if this is the first invocation, in the user's Documents folder.

`invisibles` [boolean](#) (page 82)

Show invisible folders?

*Default Value:*

false

`multiple selections allowed` [boolean](#) (page 82)

Allow multiple items to be selected? If `true`, the results will be returned in a list, even if there is exactly one item.

*Default Value:*

false

`showing package contents` [boolean](#) (page 82)

Show the contents of packages? If `true`, packages are treated as folders, so that the user can choose a package folder, such as an application, or a folder inside a package.

*Default Value:*

false. Manipulating the contents of packages is discouraged unless you control the package format or the package itself.

**Result**

The selected directory, as an `alias`. If multiple selections are allowed, returns a list containing one `alias` for each selected directory, if any.

Signals a “user canceled” error if the user cancels the `choose folder` dialog. For an example of how to handle such errors, see “[try Statements](#)” (page 207).

**Examples**

The following example specifies a prompt and allows multiple selections:

```
set foldersList to choose folder ↵
    with prompt "Select as many folders as you like:" ↵
    with multiple selections allowed
```

The following example gets a POSIX path to a chosen folder and uses the `quoted form` property (of the `text` (page 97) class) to ensure correct quoting of the resulting string for use with shell commands:

```
set folderName to quoted form of POSIX path of (choose folder)
```

Suppose that you choose the folder named `iWork '08` in your `Applications` folder. The previous statement would return the following result, which properly handles the embedded single quote and space characters in the folder name:

```
"/Applications/iWork '\''08/'"
```

**choose from list**


---

Allows the user to choose items from a list.

**Syntax**

<code>choose from list</code>	<i>list</i>	required
<code>with title</code>	<i>text</i>	optional
<code>with prompt</code>	<i>text</i>	optional
<code>default items</code>	<i>list</i>	optional
<code>OK button name</code>	<i>text</i>	optional
<code>cancel button name</code>	<i>text</i>	optional
<code>multiple selections allowed</code>	<i>boolean</i>	optional
<code>empty selection allowed</code>	<i>boolean</i>	optional

**Parameters**

*list* (page 89) (of *number* (page 92) or *text* (page 97))

A list of numbers and/or `text` objects for the user to choose from.

with title *text* (page 97)

Title text for the dialog.

*Default Value:*

None; no title is displayed.

with prompt *text* (page 97)

The prompt to be displayed in the dialog.

*Default Value:*

"Please make your selection:"

default items *list* (page 89) (of *number* (page 92) or *text* (page 97))

A list of numbers and/or text objects to be initially selected. The list cannot include multiple items unless you also specify `multiple selections allowed true`. If an item in the default items list is not in the list to choose from, it is ignored.

*Default Value:*

None; no items are selected.

OK button name *text* (page 97)

The name of the OK button.

*Default Value:*

"OK"

cancel button name *text* (page 97)

The name of the Cancel button.

*Default Value:*

"Cancel"

multiple selections allowed *boolean* (page 82)

Allow multiple items to be selected?

*Default Value:*

false

empty selection allowed *boolean* (page 82)

Allow the user to choose OK with no items selected? If false, the OK button will not be enabled unless at least one item is selected.

*Default Value:*

false

### Result

If the user clicks the OK button, returns a `list` (page 89) of the chosen `number` (page 92) and/or `text` (page 97) items; if empty selection is allowed and nothing is selected, returns an empty list (`{}`). If the user clicks the Cancel button, returns `false`.

### Examples

This script selects from a list of all the people in Address Book who have defined birthdays, and gets the birthday of the selected one. Notice the `if the result is not false test (choose from list returns false if the user clicks Cancel)` and the `set aName to item 1 of the result (choose from list returns a list, even if it contains only one item)`.

```
tell application "Address Book"
    set bDayList to name of every person whose birth date is not missing value
    choose from list bDayList with prompt "Whose birthday would you like?"
    if the result is not false then
```

```

        set aName to item 1 of the result
        set theBirthday to birth date of person named aName
        display dialog aName & "'s birthday is " & date string of theBirthday
    end if
end tell

```

**Discussion**

For historical reasons, `choose from list` is the only dialog command that returns a result (`false`) instead of signaling an error when the user presses the “Cancel” button.

## choose remote application

---

Allows the user to choose a running application on a remote machine.

**Syntax**

<code>choose remote application</code>		required
<code>with title</code>	<i>text</i>	optional
<code>with prompt</code>	<i>text</i>	optional

**Parameters**

`with title` [text](#) (page 97)

Title *text* for the `choose remote application` dialog.

*Default Value:*

None; no title is displayed.

`with prompt` [text](#) (page 97)

The prompt to be displayed in the dialog.

*Default Value:*

"Select an application:"

**Result**

The selected application, as an [application](#) (page 80) object.

Signals a “user canceled” error if the user cancels the dialog. For an example of how to handle such errors, see [“try Statements”](#) (page 207).

**Examples**

```
set myApp to choose remote application with prompt "Choose a remote web browser:"
```

**Discussion**

The user may choose a remote machine using Bonjour or by entering a specific IP address. There is no way to limit the precise kind of application returned, so either limit your script to generic operations or validate the user’s choice. If you want your script to send application-specific commands to the resulting application, you will need a using terms from statement.

For information on targeting other machines, see [“Remote Applications”](#) (page 44).

## choose URL

---

Allows the user to specify a URL.

**Syntax**

choose URL		required
showing	<i>listOfServiceTypesOrTextStrings</i>	optional
editable URL	<i>boolean</i>	optional

**Parameters**

showing *list* (page 89) (of service types or *text* (page 97))

A list that specifies the types of services to show, if available. The list can contain one or more of the following service types, or one or more *text* objects representing Bonjour service types (described below), or both:

- Web servers: shows *http* and *https* services
- FTP Servers: shows *ftp* services
- Telnet hosts: shows *telnet* services
- File servers: shows *afp*, *nfs*, and *smb* services
- News servers: shows *nntp* services
- Directory services: shows *ldap* services
- Media servers: shows *rtsp* services
- Remote applications: shows *eppc* services

A *text* object is interpreted as a Bonjour service type—for example, "*\_ftp.\_tcp*" represents the file transfer protocol. These types are listed in [Technical Q&A 1312: Bonjour service types used in Mac OS X](#).

*Default Value:*

File servers

editable URL *boolean* (page 82)

Allow user to type in a URL? If you specify *editable URL false*, the text field in the dialog is inactive.

*choose URL* does not attempt to verify that the user-entered text is a valid URL. Your script should be prepared to verify the returned value.

*Default Value:*

*true*: the user can enter a text string. If *false*, the user is restricted to choosing an item from the Bonjour-supplied list of services.

**Result**

The URL for the service, as a *text* object. This result may be passed to [open location](#) (page 141) or to any application that can handle the URL, such as a browser for *http* URLs.

Signals a “user canceled” error if the user cancels the dialog. For an example of how to handle such errors, see [“try Statements”](#) (page 207).

**Examples**

The following script asks the user to choose an URL, either by typing in the text input field or choosing one of the Bonjour-located servers:

```
set myURL to choose URL
tell application Finder to open location myURL
```

`clipboard info`

---

Returns information about the current clipboard contents.

**Syntax**

```
clipboard info                                required
for class                                    optional
```

**Parameters**

for [class](#) (page 83)

Restricts returned information to only this data type.

*Default Value:*

None; returns information for all types of data as a list of lists, where each list represents a scrap flavor.

**Result**

A [list](#) (page 89) containing one entry {`class`, `size`} for each type of data on the clipboard. To retrieve the actual data, use the [the clipboard](#) (page 162) command.

**Examples**

```
clipboard info
clipboard info for Unicode text
```

`close access`

---

Closes a file opened with the `open for access` command.

**Syntax**

```
close access fileSpecifier                required
```

**Parameters**

([alias](#) (page 79) | [file](#) (page 88) | *file descriptor*)

The alias or file specifier or integer file descriptor of the file to close. A file descriptor must be obtained as the result of an earlier [open for access](#) (page 140) call.

**Result**

None.

Signals an error if the specified file is not open.



**Examples**

You should always close files that you open, being sure to account for possible errors while using the open file:

```
set aFile to choose file
set fp to open for access aFile
try
    --file reading and writing here
on error e number n
    --deal with errors here and don't resignal
end
close access fp
```

**Discussion**

Any files left open will be automatically closed when the application exits.

**copy**


---

Copies one or more values, storing the result in one or more variables. This command only copies AppleScript values, not application-defined objects.

**Syntax**

<code>copy</code>	<i>expression</i>	required
<code>to</code>	<i>variablePattern</i>	required

**Parameters***expression*

The expression whose value is to be copied.

`to` *variablePattern*

The name of the variable or pattern of variables in which to store the value or pattern of values. Patterns may be lists or records.

**Result**

The new copy of the value.

**Examples**

As mentioned in the Discussion, `copy` creates an independent copy of the original value, and it creates a deep copy. For example:

```
set alpha to {1, 2, {"a", "b"}}
copy alpha to beta
set item 2 of item 3 of alpha to "change" --change the original list
set item 1 of beta to 42 --change a different item in the copy
{alpha, beta}
--result: {{1, 2, {"a", "change"}}, {42, 2, {"a", "b"}}}
```

Each variable reflects only the changes that were made directly to that variable. Compare this with the similar example in `set` (page 154).

See the `set` (page 154) command for examples of using variable patterns. The behavior is the same except that the values are copied.

### Discussion

The `copy` command may be used to assign new values to existing variables, or to define new variables. See “Declaring Variables with the `copy` Command” (page 50) for additional details.

Using the `copy` command creates a new value that is independent of the original—a subsequent change to that value does not change the original value. The copy is a “deep” copy, so sub-objects, such as lists within lists, are also copied. Contrast this with the behavior of the `set` (page 154) command.

When using `copy` with an object specifier, the specifier itself is the value copied, not the object in the target application that it refers to. `copy` therefore copies the object specifier, but does not affect the application data at all. To copy the object in the target application, use the application’s `duplicate` command, if it has one.

### Special Considerations

The syntax `put expression into variablePattern` is also supported, but is deprecated. It will be transformed into the `copy` form when you compile the script.

## count

---

Counts the number of elements in another object.

### Syntax

```
(count | number of)          expression          required
```

### Parameters

#### *expression*

An expression that evaluates to an object with elements, such as a `list` (page 89), `record` (page 94), or application-defined container object. `count` will count the contained elements.

### Result

The number of elements, as an `integer` (page 89).

### Examples

In its simplest form, `count`, or the equivalent pseudo-property `number`, counts the `item` elements of a value. This may be an AppleScript value, such as a list:

```
set aList to {"Yes", "No", 4, 5, 6}
count aList --result: 5
number of aList --result: 5
```

...or an application-defined object that has `item` elements:

```
tell application "Finder" to count disk 1 --result: 4
```

If the value is an object specifier that evaluates to a list, `count` counts the items of that list. This may be an “Every” (page 168) specifier:

```
count every integer of aList --result: 3
count words of "hello world" --result: 2
```

```
tell application "Finder" to count folders of disk 1 --result: 4
```

...or a [“Filter”](#) (page 169) specifier:

```
tell application "Finder"
    count folders of disk 1 whose name starts with "A" --result: 1
end tell
```

...or similar. For more on object specifiers, see [“Object Specifiers”](#) (page 29).

## current date

---

Returns the current date and time.

### Syntax

```
current date required
```

### Result

The current date and time, as a [date](#) (page 85) object.

### Examples

```
current date --result: date "Tuesday, November 13, 2007 11:13:29 AM"
```

See the [date](#) (page 85) class for information on how to access the properties of a date, such as the day of the week or month.

## delay

---

Waits for a specified number of seconds.

### Syntax

```
delay required
      optional
      number
```

### Parameters

[number](#) (page 92)

The number of seconds to delay. The number may be fractional, such as 0.5 to delay half a second.

*Default Value:*

0

### Result

None.

### Examples

```
set startTime to current date
delay 3 --delay for three seconds
set elapsedTime to ((current date) - startTime)
```

```
display dialog ("Elapsed time: " & elapsedTime & " seconds")
```

**Discussion**

`delay` does not make any guarantees about the actual length of the delay, and it cannot be more precise than 1/60th of a second. `delay` is not suitable for real-time tasks such as audio-video synchronization.

**display alert**


---

Displays a standardized alert containing a message, explanation, and from one to three buttons.

**Syntax**

<code>display alert</code>	<i>text</i>	required
<code>message</code>	<i>text</i>	optional
<code>as</code>	<i>alertType</i>	optional
<code>buttons</code>	<i>list</i>	optional
<code>default button</code>	<i>buttonSpecifier</i>	optional
<code>cancel button</code>	<i>buttonSpecifier</i>	optional
<code>giving up after</code>	<i>integer</i>	optional

**Parameters*****text*** (page 97)

The alert text, which is displayed in emphasized system font.

**message** *text* (page 97)

An explanatory message, which is displayed in small system font, below the alert text.

**as** *alertType*

The type of alert to show. You can specify one of the following alert types:

`informational`: the standard alert dialog

`warning`: the alert dialog dialog is badged with a warning icon

`critical`: currently the same as the standard alert dialog

**Default Value:**

`informational`

**buttons** *list* (page 89) (of *text* (page 97))

A list of up to three button names.

If you supply one name, a button with that name serves as the default and is displayed on the right side of the alert dialog. If you supply two names, two buttons are displayed on the right, with the second serving as the default button. If you supply three names, the first is displayed on the left, and the next two on the right, as in the case with two buttons.

**Default Value:**

`{ "OK" }`: One button labeled "OK", which is the default button.

`default button` (*text* (page 97) or *integer* (page 89))

The name or number of the default button. This may be the same as the cancel button.

*Default Value:*

The rightmost button.

`cancel button` (*text* (page 97) or *integer* (page 89))

The name or number of the cancel button. See “Result” below. This may be the same as the default button.

*Default Value:*

None; there is no cancel button.

`giving up after` *integer* (page 89)

The number of seconds to wait before automatically dismissing the alert.

*Default Value:*

None; the dialog will wait until the user clicks a button.

### Result

If the user clicks a button that was not specified as the cancel button, `display alert` returns a record that identifies the button that was clicked—for example, `{button returned: "OK"}`. If the command specifies a giving up after value, the record will also contain a `gave up:false` item.

If the `display alert` command specifies a giving up after value, and the dialog is dismissed due to timing out before the user clicks a button, the command returns a record indicating that no button was returned and the command gave up: `{button returned:"", gave up:true}`

If the user clicks the specified cancel button, the command signals a “user canceled” error. For an example of how to handle such errors, see “try Statements” (page 207).

### Examples

```
set alertResult to display alert "Insert generic warning here." →
  buttons {"Cancel", "OK"} as warning →
  default button "Cancel" cancel button "Cancel" giving up after 5
```

For an additional example, see the Examples section for the `try` (page 207) statement.

## display dialog

---

Displays a dialog containing a message, one to three buttons, and optionally an icon and a field in which the user can enter text.

### Syntax

<code>display dialog</code>	<i>text</i>	required
<code>default answer</code>	<i>text</i>	optional
<code>hidden answer</code>	<i>boolean</i>	optional
<code>buttons</code>	<i>list</i>	optional
<code>default button</code>	<i>labelSpecifier</i>	optional

<code>cancel button</code>	<i>labelSpecifier</i>	optional
<code>with title</code>	<i>text</i>	optional
<code>with icon</code>	<i>resourceSpecifier</i>	optional
<code>with icon</code>	<i>iconTypeSpecifier</i>	optional
<code>with icon</code>	<i>fileSpecifier</i>	optional
<code>giving up after</code>	<i>integer</i>	optional

### Parameters

#### *text*

The dialog text, which is displayed in emphasized system font.

`default answer` *text* (page 97)

The initial contents of an editable text field. This edit field is not present unless this parameter is present; to have the field present but blank, specify an empty string: `default answer ""`

#### *Default Value:*

None; there is no edit field.

`hidden answer` *boolean* (page 82)

If true, any text in the edit field is obscured as in a password dialog: each character is displayed as a bullet.

#### *Default Value:*

false: text in the edit field is shown in cleartext.

`buttons` *list* (page 89) (of *text* (page 97))

A list of up to three button names.

#### *Default Value:*

If you don't specify any buttons, by default, Cancel and OK buttons are shown, with the OK button set as the default button.

If you specify any buttons, there is no default or cancel button unless you use the following parameters to specify them.

`default button` (*text* (page 97) | *integer* (page 89))

The name or number of the default button. This button is highlighted, and will be pressed if the user presses the Return or Enter key.

#### *Default Value:*

If there are no buttons specified using `buttons`, the OK button. Otherwise, there is no default button.

`cancel button` (*text* (page 97) | *integer* (page 89))

The name or number of the cancel button. This button will be pressed if the user presses the Escape key or Command-period.

#### *Default Value:*

If there are no buttons specified using `buttons`, the Cancel button. Otherwise, there is no cancel button.

with title *text* (page 97)

The dialog window title.

*Default Value:*

None; no title is displayed.

with icon (*text* (page 97) | *integer* (page 89))

The resource name or ID of the icon to display.

with icon (*stop* | *note* | *caution*)

The type of icon to show. You may specify one of the following constants:

- *stop* (or 0): Shows a stop icon
- *note* (or 1): Shows the application icon
- *caution* (or 2): Shows a warning icon, badged with the application icon

with icon (*alias* (page 79) | *file* (page 88))

An *alias* or *file* specifier that specifies a `.icns` file.

giving up after *integer* (page 89)

The number of seconds to wait before automatically dismissing the dialog.

*Default Value:*

None; the dialog will wait until the user presses a button.

#### Result

A record containing the button clicked and text entered, if any. For example:

```
{text returned:"Cupertino", button returned:"OK"}
```

If the dialog does not allow text input, there is no `text` returned item in the returned record.

If the user clicks the specified cancel button, the command signals a “user canceled” error. For an example of how to handle such errors, see “[try Statements](#)” (page 207).

If the `display dialog` command specifies a `giving up after` value, and the dialog is dismissed due to timing out before the user clicks a button, it returns a record indicating that no button was returned and the command gave up: `{button returned:"", gave up:true}`

#### Examples

The following example shows how to use many of the parameters to a `display dialog` command, how to process possible returned values, and one way to handle a user cancelled error. The dialog displays two buttons and prompts a user to enter a name, giving up if they do not make a response within fifteen seconds. It shows one way to handle the case where the user cancels the dialog, which results in AppleScript signaling an “error” with the error number -128. The script uses additional `display dialog` commands to show the flow of logic and indicate where you could add statements to handle particular outcomes.

```
set userCanceled to false
try
    set dialogResult to display dialog ↵
        "What is your name?" buttons {"Cancel", "OK"} ↵
        default button "OK" cancel button "Cancel" ↵
        giving up after 15 ↵
        default answer (long user name of (system info))
on error number -128
    set userCanceled to true
```

```

end try

if userCanceled then
    -- statements to execute when user cancels
    display dialog "User cancelled."
else if gave up of dialogResult then
    -- statements to execute if dialog timed out without an answer
    display dialog "User timed out."
else if button returned of dialogResult is "OK" then
    set userName to text returned of dialogResult
    -- statements to process user name
    display dialog "User name: " & userName
end if
end

```

The following example displays a dialog that asks for a password. It supplies a default answer of "wrong", and specifies that the default answer, as well as any text entered by the user, is hidden (displayed as a series of bullets). It gives the user up to three chances to enter a correct password.

```

set prompt to "Please enter password:"
repeat 3 times
    set dialogResult to display dialog prompt ¬
        buttons {"Cancel", "OK"} default button 2 ¬
        default answer "wrong" with icon 1 with hidden answer
    set thePassword to text returned of dialogResult
    if thePassword = "magic" then
        exit repeat
    end if
end repeat
if thePassword = "magic" or thePassword = "admin" then
    display dialog "User entered valid password."
end if

```

The password text is copied from the return value `dialogResult`. The script doesn't check for a user cancelled error, so if the user cancels AppleScript stops execution of the script.

## do shell script

---

Executes a shell script using the `sh` shell.

### Syntax

<code>do shell script</code>	<i>text</i>	required
<code>as</code>	<i>class</i>	optional
<code>administrator privileges</code>	<i>boolean</i>	optional
<code>user name</code>	<i>text</i>	optional
<code>password</code>	<i>text</i>	optional
<code>altering line endings</code>	<i>boolean</i>	optional



**Parameters****text** (page 97)

The shell script to execute.

as **class** (page 83)

Specifies the desired type of the result. The raw bytes returned by the command will be interpreted as the specified class.

*Default Value:*«class utf8»: UTF-8 text. If there is no *as* parameter and the output is not valid UTF-8, the output will be interpreted as text in the primary encoding.administrator privileges **boolean** (page 82)

Execute the command as the administrator? Once a script is correctly authenticated, it will not ask for authentication again for five minutes. The elevated privileges and the grace period do not extend to any other scripts or to the rest of the system. For security reasons, you may not tell another application to do shell script with administrator privileges. Put the command outside of any tell block, or put it inside a tell me block.

*Default Value:*

false

user name **text** (page 97)

The name of an administrator account. You can avoid a password dialog by specifying a name in this parameter and a password in the password parameter. If you specify a user name, you must also specify a password.

password **text** (page 97)

An administrator password, typically used in conjunction with the administrator specified by the user name parameter. If user name is omitted, it is assumed to be the current user.

altering line endings **boolean** (page 82)

Should the do shell script command change all line endings in the command output to Mac-style and trim a trailing one? For example, the result of do shell script "echo foo; echo bar" is "foo\rbar", not the "foo\nbar\n" that the shell script actually returned.

*Default Value:*

true

**Result**

The output of the shell script.

Signals an error if the shell script exits with a non-zero status. The error number will be the status, the error message will be the contents of stderr.

**Examples**

do shell script "uptime"

**Discussion**For additional documentation and examples of the do shell script command, see Technical Note TN2065, [do shell script in AppleScript](#).

get

Evaluates an object specifier and returns the result.

The command name `get` is typically optional—expressions that appear as statements or operands are automatically evaluated as if they were preceded by `get`. However, `get` can be used to force early evaluation of part of an object specifier.

### Syntax

<code>get</code>	<i>specifier</i>	required
<code>as</code>	<i>class</i>	optional

### Parameters

#### *specifier*

An object specifier to be evaluated. If the specifier refers to an application-defined object, the `get` command is sent to that application. Technically, all values respond to `get`, but for all values other than object specifiers, `get` is an identity operation: the result is the exact same value.

#### as *class* (page 83)

The desired class for the returned data. If the data is not of the desired type, AppleScript attempts to coerce it to that type.

#### *Default Value:*

None; no coercion is performed.

### Result

The value of the evaluated expression. See “[Reference Forms](#)” (page 167) for details on what the results of evaluating various object specifiers are.

### Examples

`get` can get properties or elements of AppleScript-defined objects, such as lists:

```
get item 1 of {"How", "are", "you?"} --result: "How"
```

...or of application-defined objects:

```
tell application "Finder" to get name of home --result: "myname"
```

As noted above, the `get` is generally optional. For example, these statements are equivalent to the above two:

```
item 1 of {"How", "are", "you?"} --result: "How"
tell application "Finder" to name of home --result: "myname"
```

However, an explicit `get` can be useful for forcing early evaluation of part of an object specifier. Consider:

```
tell application "Finder" to get word 1 of name of home
--Finder got an error: Can't get word 1 of name of folder "myname" of folder
"Users" of startup disk.
```

This fails because Finder does not know about elements of text, such as words. AppleScript does, however, so the script has to make Finder get only the name of ... part:

```
tell application "Finder" to get word 1 of (get name of home)
--result: "myname"
```

The explicit `get` forces that part of the specifier to be evaluated; Finder returns a `text` result, from which AppleScript can then get `word 1`.

For more information on specifiers, see “Object Specifiers” (page 29).

`get eof`

---

Returns the length of a file, in bytes.

#### Syntax

```
get eof fileSpecifier required
```

#### Parameters

([alias](#) (page 79) | [file](#) (page 88) | *file descriptor*)

The file to obtain the length for, as an alias, a file specifier, or an [integer](#) (page 89) file descriptor. A file descriptor must be obtained as the result of an earlier [open for access](#) (page 140) call.

#### Result

The logical size of the file, that is, the length of its contents in bytes.

#### Examples

This example obtains an alias to a desktop picture folder and uses `get eof` to obtain its length:

```
set desktopPicturesFolderPath to ¬
    (path to desktop pictures folder as text) & "Flow 1.jpg" as alias
--result: alias "Leopard:Library:Desktop Pictures:Flow 1.jpg"
get eof desktopPicturesFolderPath --result: 531486
```

`get volume settings`

---

Returns the sound output and input volume settings.

#### Syntax

```
get volume settings required
```

#### Result

A record containing the sound output and input volume settings. All the integer settings are between 0 (silent) and 100 (full volume):

`output volume` (an [integer](#) (page 89))

The base output volume.

`input volume` (an integer)

The input volume.

`alert volume` (an integer)

The alert volume. 100 for this setting means “as loud as the output volume.”

`output muted` (a [boolean](#) (page 82))

Is the output muted? If true, this overrides the output and alert volumes.

**Examples**

```
set volSettings to get volume settings
--result: {output volume:43, input volume:35, alert volume:78, output muted:false}
```

**info for**


---

Return information for a file or folder.

**Syntax**

<code>info for</code>	<i>fileSpecifier</i>	required
<code>size</code>	<i>boolean</i>	optional

**Parameters**

([alias](#) (page 79) | [file](#) (page 88))

An alias or file specifier for the file or folder.

`size` [boolean](#) (page 82)

Return the size of the file or folder? For a file, its “size” is its length in bytes; for a folder, it is the sum of the sizes of all the files the folder contains.

*Default Value:*

`true`: Because getting the size of a folder requires getting the sizes of all the files inside it, `size true` may take a long time for large folders such as `/System`. If you do not need the size, ask to not get it using `size false`. Alternatively, target the Finder or System Events applications to ask for the specific properties you want.

**Result**

A record containing information about the specified file or folder, with the following fields. Some fields are only present for certain kinds of items:

`name` (a [text](#) (page 97) object)

The item’s full name, as it appears in the file system. This always includes the extension, if any. For example, `"OmniOutliner Professional.app"`.

`displayed name` (a [text](#) (page 97) object)

The item’s name as it appears in Finder. This may be different than the `name` if the extension is hidden or if the item has a localized name. For example, `"OmniOutliner Professional"`.

`short name` (a [text](#) (page 97) object, applications only)

The application’s `CFBundleName`, which is the name displayed in the menu bar when the application is active. This is often, but not always, the same as the displayed name. For example, `"OmniOutliner Pro"`.

`name extension` (a [text](#) (page 97) object)

The extension part of the item name. For example, the name extension of the file `"foo.txt"` is `"txt"`.

`bundle identifier` (a [text](#) (page 97) object)

The package’s bundle identifier. If the package is an application, this is the application’s `id`.

`type identifier` (a [text](#) (page 97) object)

The item’s type, as a Uniform Type Identifier (UTI). This is the preferred form for identifying item types, and may be used with `choose file`.

`kind` (a `text` (page 97) object)

The item's type, as displayed in Finder. This may be localized, and should only be used for display purposes.

`default application` (an `alias` (page 79) object)

The application that will open this item.

`creation date` (a `date` (page 85) object)

The date the item was created.

`modification date` (a `date` (page 85) object)

The date the item was last modified. Folder modification dates do not change when an item inside them changes, though they do change when an item is added or removed.

`file type` (a `text` (page 97) object)

The item's type, as a four-character code. This is the classic equivalent of the type identifier, but less accurate and harder to interpret; use `type identifier` if possible.

`file creator` (a `text` (page 97) object)

The item's four-character creator code. For applications, this is the classic equivalent of the bundle identifier, and will work for referencing an application by id. For files, this can be used to infer the default application, but not reliably; use `default application` if possible.

`short version` (a `text` (page 97) object)

The item's short version string, as it appears in a Finder "Get Info" window. Any item may have this attribute, but typically only applications do.

`long version` (a `text` (page 97) object)

The item's long version string, as it appears in a Finder "Get Info" window. Any item may have this attribute, but typically only applications do.

`size` (an `integer` (page 89))

The item's size, in bytes. For more details, see the `size` parameter.

`alias` (a `boolean` (page 82))

Is the item an alias file?

`folder` (a `boolean` (page 82))

Is the item a folder? This is true for packages, such as application packages, as well as normal folders.

`package folder` (a `boolean` (page 82))

Is the item a package folder, such as an application? A package folder appears in Finder as if it is a file.

`extension hidden` (a `boolean` (page 82))

Is the item's name extension hidden?

`visible` (a `boolean` (page 82))

Is the item visible? Typically, only special system files are invisible.

`locked` (a `boolean` (page 82))

Is the item locked?

`busy status` (a `boolean` (page 82))

Is the item currently in use?

If true, the item is reliably busy. If false, the item may still be busy, because this status may not be supported by some applications or file systems.

`folder window` (rectangle, folders only)

The folder's window's bounding rectangle, as list of four integers: {top, left, bottom, right}.

**Examples**

```
set downloadsFolder to path to downloads folder
  --result: alias "HD:Users:me:Downloads:"
info for downloadsFolder
  --result: {name:"Downloads", folder:true, alias:false, ...}
```

**Special Considerations**

Because `info for` returns so much information, it can be slow, and because it only works on one file at a time, it can be difficult to use. The recommended technique is to use System Events or Finder to ask for the particular properties you want.

## launch

---

Launches an application, if it is not already running, but does not send it a `run` command.

If an application is already running, sending it a `launch` command has no effect. That allows you to open an application without performing its usual startup procedures, such as opening a new window or, in the case of a script application, running its script. For example, you can use the `launch` command when you don't want an application to open and close visibly. This is less useful in AppleScript 2.0, which launches applications as hidden by default (even with the `run` (page 151) command).

See the [application](#) (page 80) class reference for information on how to use an `application` object's `is running` property to determine if it is running without having to launch it.

**Syntax**

```
launch application required
```

**Parameters***application*

The application to launch.

**Result**

None.

**Examples**

```
launch application "TextEdit"
tell application "TextEdit" to launch
```

**Discussion**

The `launch` command does not launch applications on remote machines. For examples of other ways to specify an application, see the [application](#) (page 80) class.

Many applications also support the `reopen` command, which reactivates a running application or launches it if it isn't running. If the application is already running, this command has the same effect as double-clicking the application icon in the Finder. Each application determines how it will implement the `reopen` command—some may perform their usual startup procedures, such as opening a new window, while others perform no additional operations.

## list disks

---

Returns the names of the currently mounted volumes.

**Important:** This command is deprecated; use `tell application "System Events"` to get the name of every disk.

**Syntax**

```
list disks required
```

**Result**

A [list](#) (page 89) of text objects, one for each currently mounted volume.

```
list folder
```

---

Returns the names of the items in a specified folder.

**Important:** This command is deprecated; use `tell application "System Events"` to get the name of every disk item of ....

**Syntax**

```
list folder fileSpecifier required  

invisibles boolean optional
```

**Parameters**

([alias](#) (page 79) | [file](#) (page 88))  
 Specifies the folder to list.

`invisibles` [boolean](#) (page 82)  
 Show invisible files and folders?

*Default Value:*  
 true

**Result**

A [list](#) (page 89) of [text](#) (page 97) objects, one for each item in the specified folder.

```
load script
```

---

Returns a `script` object loaded from a specified file.

**Syntax**

```
load script fileSpecifier required
```

**Parameters**

([alias](#) (page 79) | [file](#) (page 88))

An `alias` or `file specifier` that specifies a `script` object. The file must be a compiled script (with extension `scpt`) or script bundle (with extension `scptd`).

**Result**

The `script` object. You can get this object's properties or call its handlers as if it were a local `script` object.

**Examples**

For examples, see [“Saving and Loading Libraries of Handlers”](#) (page 72) in [“About Handlers”](#) (page 67).

`localized string`


---

Returns the localized text for the specified key.

**Syntax**

<code>localized string</code>	<i>text</i>	required
<code>from table</code>	<i>text</i>	optional
<code>in bundle</code>	<i>fileSpecifier</i>	optional

**Parameters**

*text* (page 97)

The key for which to obtain the localized text.

`from table` *text* (page 97)

The name of the strings file excluding the `.strings` suffix.

*Default Value:*

"Localizable"

`in bundle` (*alias* (page 79) | *file* (page 88))

An *alias* or *file* specifier that specifies the strings file.

*Default Value:*

The current script bundle for a document-based script (a `scptd` bundle); otherwise, the current application.

**Result**

A *text* (page 97) object containing the localized text, or the original key if there is no localized text for that key.

**Examples**

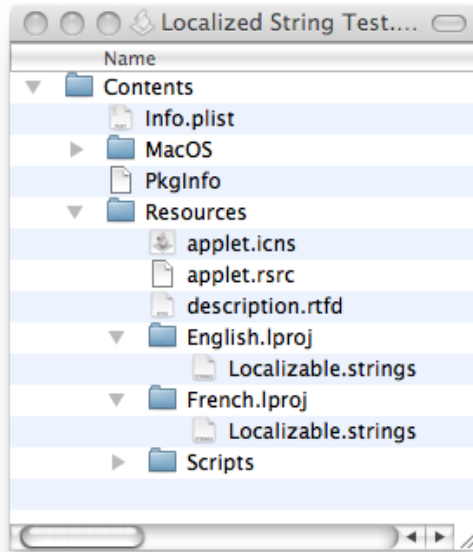
In order for `localized string` to be useful, you must create localized string data for it to use:

1. Save your script as an application bundle or script bundle.



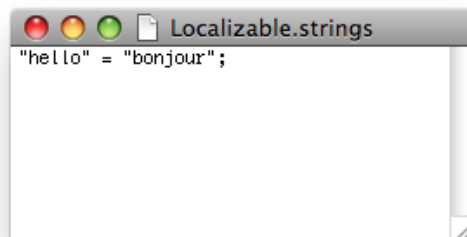
2. Create `lproj` folders in the `Resources` directory of the bundle for each localization: for example, `English.lproj`, `French.lproj`. Create files named `Localized.strings` in each one. When you are done, the folder structure should look like this:

Figure 7-1 Bundle structure with localized string data



3. Add key/value pairs to each `Localized.strings` file. Each pair is a line of text `"key" = "value" ;`, for example:

Figure 7-2 Key/value pair for localized string data



Now `localized string` will return the appropriate values, as defined in your files. For example, when running in French:

```
localized string "hello" --result: "bonjour"
```

## log

---

In Script Editor, displays a value in the Event Log History window or in the Event Log pane of a script window.

### Syntax

log		required
	<i>value</i>	optional

### Parameters

#### *value*

The value to display. Expressions are evaluated but object specifiers are not resolved.

The displayed value is enclosed in block comment characters—for example, (*\*window 1\**).

If you do not specify a value, log will display just the comment characters: (*\*\**).

### Result

None.

### Examples

The following shows a simple use of logging:

```
set area to 7 * 43 as square feet
log area -- result (in Event Log pane): (*square feet 301.0*)
```

Log statements can be useful for tracking a script's progress. For an example that shows how to log statements in a repeat loop, see [“Logging”](#) (page 46).

## mount volume

---

Mounts the specified network volume.

### Syntax

mount volume	<i>text</i>	required
on server	<i>text</i>	(see parameter description)
in AppleTalk zone	<i>text</i>	(see parameter description)
as user name	<i>text</i>	optional
with password	<i>text</i>	optional

### Parameters

#### *text* (page 97)

The name or URL (for example, `afp://server/volume/`) of the volume to mount.

#### on server *text* (page 97)

The server on which the volume resides; omit if URL path provided in direct parameter.

in AppleTalk zone *text* (page 97)

The AppleTalk zone in which the server resides; omit if URL path provided.

as user name *text* (page 97)

The user name with which to log in to the server; omit for guest access.

with password *text* (page 97)

The password for the user name; omit for guest access.

#### Result

None.

#### Examples

```
mount volume "afp://myserver.com/" -- guest access
mount volume "http://idisk.mac.com/myname/Public"
mount volume "http://idisk.mac.com/somebody" ↵
  as user name "myname" with password "mypassword"
```

#### Discussion

The `mount volume` command can connect to any file server that is supported by the Finder's `Connect To...` command, including Windows (smb), Samba, and FTP servers. On some kinds of servers, the `as user name` and `with password` parameters may not bypass the login dialog, but encoding the name and password in the URL (for example, `smb://myname:passwd@server.domain.com/sharename`) will mount it silently.

## offset

---

Finds one piece of text inside another.

#### Syntax

<code>offset</code>		required
<code>of</code>	<i>text</i>	required
<code>in</code>	<i>text</i>	required

#### Parameters

of *text* (page 97)

The source text to find the position of.

in *text* (page 97)

The target text to search in.

#### Result

An *integer* (page 89) value indicating the position, in characters, of the source text in the target, or 0 if not found.

#### Examples

```
set myString to "Yours, mine, and ours"
offset of "yours" in myString --result: 1, because case is ignored by default
offset of "mine" in myString --result: 8
offset of "theirs" in myString --result: 0, because "theirs" doesn't appear
considering case
  offset of "yours" in myString -- result: 0, because case is now considered
```

end considering

### Discussion

`offset` compares text as the `equals` operator does, including considering and ignoring conditions. The values returned are counted the same way character elements of text are counted—for example, offset of "c" in "école" is always 2, regardless of whether "école" is in Normalization Form C or D. The result of matching part of a character cluster is undefined.

## open for access

---

Opens a file for reading and writing.

### Syntax

open for access	<i>fileSpecifier</i>	required
write permission	<i>boolean</i>	optional

### Parameters

(*alias* (page 79) | *file* (page 88))

An *alias* or file specifier that specifies the file to open. You can only use an *alias* if the file exists.

write permission *boolean* (page 82)

Should writing to the file be allowed?

*Default Value:*

false: write and set eof commands on this file will fail with an error.

### Result

A file descriptor, as an *integer* (page 89). This file descriptor may be used with any of the other file commands: `read` (page 147), `write` (page 163), `get eof` (page 131), `set eof` (page 155), and `close access` (page 120).

### Examples

The following example opens a file named "NewFile" in the specified location `path to desktop`, but does not ask for write access:

```
set theFile to (path to desktop as text) & "NewFile"
set referenceNumber to open for access theFile
```

To open the file with write access, you would substitute the following line:

```
set referenceNumber to open for access theFile with write permission
```

### Discussion

Opening a file using `open for access` is not the same as opening a file using Finder. It is "open" only in the sense that AppleScript has access to read (and optionally write) its contents; it does not appear in one of the target application's windows, and it does not even have to be one of the target application's files. `open for access` and the associated file commands (`read`, `write`, `get eof`, `set eof`) are typically used with text files. They can also read and write arbitrary binary data, but this is not recommended unless you create the file yourself or have detailed knowledge of the file format.

Calling `open for access` on a file returns an integer, termed a *file descriptor*, which represents an open communication channel to the file's data. This file descriptor remains open until the script calls `close access` on it (or on the same file). Each file descriptor maintains a *file pointer*, which marks the current position within the file and is initially set to the beginning of the file. `read` and `write` commands begin reading or writing at the file pointer, unless instructed otherwise using a `from` or `starting at` parameter, and advance the file pointer by the number of bytes read or written, so the next operation will begin where the previous one left off.

A single file may be opened more than once, and therefore have several different file descriptors. Each file descriptor maintains its own file pointer, and each must be closed separately. If you open more than one channel at once with write permission, behavior is unspecified.

It is not strictly necessary to use `open for access`—all the other file commands can accept an alias; if the file is not open, they will open it, do the operation, and then close it. Explicitly opening and closing the file does have two potential advantages, however.

One is performance: if you are performing a number of operations on the same file, opening and closing it repeatedly could become expensive. It is cheaper to explicitly open the file, do the work, and then explicitly close it.

Two is ease of sequential read and write operations: because the file pointer tracks the progress through the file, reading or writing several pieces of data from the same file is a simple matter. Doing the same thing without using the file pointer requires calculating the data size yourself, which is not even possible in some cases.

## `open location`

---

Opens a URL with the appropriate program.

### Syntax

<code>open location</code>	<i>text</i>	required
<code>error reporting</code>	<i>boolean</i>	optional

### Parameters

*text* (page 97)

The URL to open.

`error reporting` *boolean* (page 82)

This parameter exists only for historical reasons; it is no longer supported.

### Result

None.

### Examples

This example opens an Apple web page:

```
open location "http://www.apple.com"
```

## path to (application)

---

Returns the location of the specified application.

### Syntax

```

path to application required
      application optional
as class optional

```

### Parameters

#### *application*

The application to locate. See the [application](#) (page 80) class reference for possible ways to specify an application. You may also use one of the following identifiers:

`current application`

The application executing the script, such as Script Editor.

`frontmost application`

The frontmost application.

`me`

The script itself. For script applications, this is the same as `current application`, but for script documents, it is the location of the document.

**Note:** Some older applications may treat `me` identically to `current application`.

`it`

The application of the current target.

*Default Value:*

`it`

as [class](#) (page 83) ([alias](#) (page 79) | [text](#) (page 97))

The class of the returned location. If specified, must be one of `alias` or `text`.

*Default Value:*

[alias](#) (page 79)

### Result

The location of the specified application, as either an `alias` or a `text` object containing the path.

### Examples

```

path to application "TextEdit"
  --result: alias "Leopard:Applications:TextEdit.app:"
path to --result: alias "Leopard:Applications:AppleScript:Script Editor.app:"
path to me --result: same as above
path to it --result: same as above
path to frontmost application --result: same as above
path to current application
  --result: same, but could be different for a script application

```

`path to (folder)`

---

Returns the location of the specified special folder.

**Syntax**

<code>path to</code>	<i>folder constant</i>	required
<code>from</code>	<i>domain constant</i>	optional
<code>as</code>	<i>class</i>	optional
<code>folder creation</code>	<i>boolean</i>	optional

**Parameters***folder constant*

The special folder for which to return the path. You may specify one of the following folders:

- application support
- applications folder
- desktop
- desktop pictures folder
- documents folder
- downloads folder
- favorites folder
- Folder Action scripts
- fonts
- help
- home folder
- internet plugins
- keychain folder
- library folder
- modem scripts
- movies folder
- music folder
- pictures folder
- preferences
- printer descriptions
- public folder
- scripting additions
- scripts folder
- shared documents
- shared libraries
- sites folder
- startup disk
- startup items
- system folder
- system preferences
- temporary items
- trash
- users folder
- utilities folder
- workflows folder

The following folders are also defined, but are only meaningful when used with `from Classic domain`:

- apple menu
- control panels
- control strip modules
- extensions
- launcher items folder
- printer drivers
- printmonitor
- shutdown folder
- speakable items
- stationery
- voices



from *domain constant*

The domain in which to look for the specified folder. You may specify one of the following domains:

system domain

A folder in /System.

local domain

A folder in /Library.

network domain

A folder in /Network.

user domain

A folder in ~, the user's home folder.

Classic domain

A folder in the Classic Mac OS system folder. Only meaningful on systems that support Classic.

*Default Value:*

The default domain for the specified folder. This varies depending on the folder.

as *class* (page 83) (*alias* (page 79) | *text* (page 97))

The class of the returned location.

*Default Value:*

*alias* (page 79)

folder creation *boolean*

Create the folder if it doesn't exist? Your script may not have permission to create the folder (for example, asking to create something in the system domain), so your script should be prepared for that error.

*Default Value:*

true

### Result

The location of the specified folder, as either an *alias* or a *text* object containing the path.

### Examples

```
path to desktop --result: alias "Leopard:Users:johndoe:Desktop:"
```

```
path to desktop as string --result: "Leopard:Users:johndoe:Desktop:"
```

## path to resource

---

Returns the location of the specified resource.

### Syntax

path to resource	<i>text</i>	required
in bundle	<i>fileSpecifier</i>	optional
in directory	<i>text</i>	optional

**Parameters***text*

The name of the requested resource.

in bundle (*alias* (page 79) | *file* (page 88))An *alias* or *file* specifier that specifies the bundle containing the resource.*Default Value:*The current script bundle for a document-based script (a `scptd` bundle); otherwise, the current application.in directory *text* (page 97)The name of a subdirectory in the bundle's `Resources` directory.**Result**The location of the specified resource, as an *alias* (page 79).**Examples**The following example shows how you can get the path to a `.icns` file—in this case, in the Finder application.

```
tell application "Finder"
set gearIconPath to path to resource "Gear.icns"
end
--result: alias
"HD:System:Library:CoreServices:Finder.app:Contents:Resources:Gear.icns"
```

**random number**

Returns a random number.

**Syntax**

<code>random number</code>		required
<code>from</code>	<i>number</i>	optional
<code>to</code>	<i>number</i>	optional
<code>with seed</code>	<i>number</i>	optional

**Parameters**from *number* (page 92)

The lowest number to return. Can be negative.

*Default Value:*

0.0

to *number* (page 92)

The highest number to return. Can be negative.

*Default Value:*

1.0

with seed [integer](#) (page 89)

An initial seed for the random number generator. Once called with any particular seed value, `random number` will always generate the same sequence of numbers. This can be useful when testing randomized algorithms: you can force it to behave the same way every time.

#### Result

A number between the `from` and `to` limits, including the limit values. Depending on the limit values, the result may be an integer or a real. If at least one limit is specified, and all specified limits are integers, the result is an integer. Otherwise, the result is a real, and may have a fractional part.

#### Examples

```
random number --result: 0.639215561057
random number from 1 to 10 --result: 8
```

#### Discussion

Random numbers are, by definition, random, which means that you may get the same number twice (or even more) in a row, especially if the range of possible numbers is small.

The numbers generated are only pseudo-random, and are not considered cryptographically secure.

If you need to select one of a set of objects in a relationship, use some *object* rather than *object* (`random number from 1 to count objects`). See the “[Arbitrary](#)” (page 167) reference form for more details.

## read

---

Reads data from a file.

#### Syntax

<code>read</code>	<i>fileSpecifier</i>	required
<code>from</code>	<i>integer</i>	optional
<code>for</code>	<i>integer</i>	optional
<code>to</code>	<i>integer</i>	optional
<code>before</code>	<i>text</i>	optional
<code>until</code>	<i>text</i>	optional
<code>using delimiters</code>	<i>text</i>	optional
<code>as</code>	<i>class</i>	optional

#### Parameters

([alias](#) (page 79) | [file](#) (page 88) | *file descriptor*)

The file to read from, as an alias, a file specifier, or an [integer](#) (page 89) file descriptor. A file descriptor must be obtained as the result of an earlier [open for access](#) (page 140) call.

from *integer* (page 89)

The byte position in the file to start reading from. The position is 1-based, so 1 is the first byte of the file, 2 the second, and so on. Negative integers count from the end of the file, so -1 is the last byte, -2 the second-to-last, and so on.

*Default Value:*

The current file pointer (see *open for access* (page 140)) if the file is open, or the beginning of the file if not.

for *integer* (page 89)

The number of bytes to read.

*Default Value:*

Read until the end of the file.

to (*integer* (page 89) | *eof*)

Stop reading at this byte position in the file; use *eof* to indicate the last byte. The position is 1-based, like the *from* parameter.

before *text* (page 97)

A single character; read up to the next occurrence of that character. The *before* character is also read, but is not part of the result, so the next *read* will start just after it.

until *text* (page 97)

A single character; read up to and including the next occurrence of that character.

using delimiter *text* (page 97)

A delimiter, such as a tab or return character, used to separate the data read into a list of text objects. The resulting items consist of the text between occurrences of the delimiter text. The delimiter is considered a separator, so a leading or trailing delimiter will produce an empty string on the other side. For example, the result of reading "axbxcx" using a delimiter of "x" would be {"a", "b", "c", ""}.

*Default Value:*

None; *read* returns a single item.

using delimiters *list* (page 89) of *text* (page 97)

As using *delimiter* above, but all of the strings in the list count as delimiters.

as *class* (page 83)

Interpret the raw bytes read as this class. The most common ones control the use of three different text encodings:

`text` or `string`

The primary text encoding, as determined by the user's language preferences set in the International preference panel. (For example, Mac OS Roman for English, MacJapanese for Japanese, and so on.)

`Unicode text`

UTF-16.

`<<class utf8>>`

UTF-8. (See "Double Angle Brackets" (page 241) for information on chevron or "raw" syntax.)

Any other class is possible, for example `date` or `list`, but is typically only useful if the data was written using a `write` statement specifying the same value for the `as` parameter.

*Default Value:*

`text`

**Result**

The data read from the file. If the file is open, the file pointer is advanced by the number of bytes read, so the next `read` command will start where the previous one left off.

**Examples**

The following example opens a file for read access, reads up to (and including) the first occurrence of ".", closes the file, and displays the text it read. (See the Examples section for the `write` (page 163) command for how to create a similar file for reading.)

```
set fp to open for access file "Leopard:Users:myUser:NewFile"
set myText to read fp until "."
close access fp
display dialog myText
```

To read all the text in the file, replace `set myText to read fp until "."` with `set myText to read fp`.

**Discussion**

At most one of `to`, `for`, `before`, and `until` is allowed. Use of `before`, `until`, or using `delimiter(s)` will interpret the file first as text and then coerce the text to whatever is specified in the `as` parameter. Otherwise, it is treated as binary data (which may be interpreted as text if so specified.)

`read` cannot automatically detect the encoding used for a text file. If a file is not in the primary encoding, you must supply an appropriate `as` parameter.

When reading binary data, `read` always uses big-endian byte order. This is only a concern if you are reading binary files produced by other applications.

**round**


---

Rounds a number to an integer.

**Syntax**

<code>round</code>	<i>real</i>	required
<code>rounding</code>	<i>roundingDirection</i>	optional

**Parameters**

*real* (page 93)

The number to round.

rounding *roundingDirection*

The direction to round. You may specify one of the following rounding directions:

up

Rounds to the next largest integer. This is the same as the math “ceiling” function.

down

Rounds down to the next smallest integer. This is the same as the math “floor” function.

toward zero

Rounds toward zero, discarding any fractional part. Also known as truncation.

to nearest

Rounds to the nearest integer; .5 cases are rounded to the nearest even integer. For example, 1.5 rounds to 2, 0.5 rounds to 0. Also known as “unbiased rounding” or “bankers’ rounding.” See Discussion for details.

as taught in school

Rounds to the nearest integer; .5 cases are rounded away from zero. This matches the rules commonly taught in elementary mathematics classes.

*Default Value:*

to nearest

### Result

The rounded value, as an [integer](#) (page 89) if it is within the allowable range ( $\pm 2^{29}$ ), or as a [real](#) (page 93) if not.

### Examples

Rounding up or down is not the same as rounding away from or toward zero, though it may appear so for positive numbers. For example:

```
round 1.1 rounding down --result: 1
round -1.1 rounding down --result: -2
```

To round to the nearest multiple of something other than 1, divide by that number first, round, and then multiply. For example, to round a number to the nearest 0.01:

```
set x to 5.1234
set quantum to 0.01
(round x/quantum) * quantum --result: 5.12
```

### Discussion

The definition of `to nearest` is more accurate than `as taught in school`, but may be surprising if you have not seen it before. For example:

```
round 1.5 --result: 2
round 0.5 --result: 0
```

Rounding 1.5 to 2 should come as no surprise, but `as taught in school` would have rounded 0.5 up to 1. The problem is that when dealing with large data sets or with many subsequent rounding operations, always rounding up introduces a slight upward skew in the results. The round-to-even rule used by `to nearest` tends to reduce the total rounding error, because on average an equal portion of numbers will round down as will round up.

run

---

Executes the `run` handler of the specified target.

To run an application, it must be on a local or mounted volume. If the application is already running, the effect of the `run` command depends on the application. Some applications are not affected; others repeat their startup procedures each time they receive a `run` command.

The `run` command launches an application as hidden; use [activate](#) (page 108) to bring the application to the front.

For a `script` object, the `run` command causes either the explicit or the implicit `run` handler, if any, to be executed. For related information, see [“run Handlers”](#) (page 74).

### Syntax

```
run                                runTarget                            optional
```

### Parameters

runTarget *script*

A [script](#) (page 97) or [application](#) (page 80) object.

*Default Value:*

`it` (the current target)

### Result

The result, if any, returned by the specified object's `run` handler.

### Examples

```
run application "TextEdit"
tell application "TextEdit" to run
run myScript --where myScript is a script object
```

For information about using the `run` command with `script` objects, see [“Sending Commands to Script Objects”](#) (page 59).

### Discussion

To specify an application to run, you can supply a string with only the application name, as shown in the Examples section. Or you can specify a location more precisely, using one of the forms described in [“Aliases and Files”](#) (page 41). For examples of other ways to specify an application, see the [application](#) (page 80) class.

It is not necessary to explicitly tell an application to `run` before sending it other commands; AppleScript will do that automatically. To launch an application without invoking its usual startup behavior, use the [launch](#) (page 134) command. For further details, see [“Calling a Script Application From a Script”](#) (page 77).

run script

---

Runs a specified script or script file.

See also [store script](#) (page 158).

### Syntax

<code>run script</code>	<i>scriptTextOrFileSpecifier</i>	required
<code>with parameters</code>	<i>listOfParameters</i>	optional
<code>in</code>	<i>text</i>	optional

**Parameters**

([text](#) (page 97) | [alias](#) (page 79) | [file](#) (page 88))

The script text, or an `alias` or `file` specifier that specifies the script file to run.

`with parameters` [list](#) (page 89) of *anything*

A list of parameter values to be passed to the script.

`in` [text](#) (page 97)

The scripting component to use.

*Default Value:*

"AppleScript"

**Result**

The result of the script's `run` handler.

**Examples**

The following script targets the application Finder, escaping the double quotes around the application name with the backslash character (for more information on using the backslash, see the Special String Characters section in the [text](#) (page 97) class description):

```
run script "get name of front window of app \"Finder\"" --result: a window name
```

This example executes a script stored on disk:

```
set scriptAlias to "Leopard:Users:myUser:Documents:savedScript.scptd:" as alias
run script scriptAlias --result: script is executed
```

`say`

---

Speaks the specified text.

**Syntax**

<code>say</code>	<i>text</i>	required
<code>displaying</code>	<i>text</i>	optional
<code>using</code>	<i>text</i>	optional
<code>waiting until completion</code>	<i>boolean</i>	optional
<code>saving to</code>	<i>fileSpecifier</i>	optional



**Parameters****text** (page 97)

The text to speak.

displaying **text** (page 97)

The text to display in the feedback window, if different from the spoken text. This parameter is ignored unless Speech Recognition is turned on (in System Preferences).

using **text** (page 97)

The voice to speak with—for example: "Zarvox".

You can use any of the voices from the System Voice pop-up on the Text to Speech tab in the Speech preferences pane.

*Default Value:*

The current System Voice (set in the Speech panel in System Preferences).

waiting until completion **boolean** (page 82)

Should the command wait for speech to complete before returning? This parameter is ignored unless Speech Recognition is turned on (in System Preferences).

*Default Value:*

true

saving to (**alias** (page 79) | **file** (page 88))An **alias** or **file** specifier to an AIFF file (existing or not) to contain the sound output. You can only use an **alias** specifier if the file exists. If this parameter is specified, the sound is not played audibly, only saved to the file.*Default Value:*

None; the text is spoken out loud, and no file is saved.

**Result**

None.

**Examples**

say "You are not listening to me!" using "Bubbles" -- result: spoken in Bubbles

The following example saves the spoken text into a sound file:

```
set soundFile to choose file name -- specify name ending in ".aiff"
--result: a file URL
say "I love oatmeal." using "Victoria" saving to soundFile
--result: saved to specified sound file
```

## scripting components

Returns a list of the names of all currently available scripting components, such as the AppleScript component.

**Syntax**

```
scripting components required
```

**Result**A **list** (page 89) of **text** (page 97) items, one for each installed scripting component.

**Examples**

```
scripting components --result: {"AppleScript"}
```

**Discussion**

A scripting component is a software component, such as AppleScript, that conforms to the Open Scripting Architecture (OSA) interface. The OSA provides an abstract interface for applications to compile, execute, and manipulate scripts without needing to know the details of the particular scripting language. Each scripting language corresponds to a single scripting component.

**set**


---

Assigns one or more values to one or more variables.

**Syntax**

set	<i>variablePattern</i>	required
to	<i>expression</i>	optional

**Parameters***variablePattern*

The name of the variable or pattern of variables in which to store the value or pattern of values. Patterns can be lists or records.

to *expression*

The expression whose value is to be set. It can evaluate to any type of object or value.

**Result**

The value assigned.

**Examples**

set may be used to create new variables:

```
set myWeight to 125
```

...assign new values to existing variables:

```
set myWeight to myWeight + 23
```

...change properties or elements of objects, such as lists:

```
set intList to {1, 2, 3}
set item 3 of intList to 42
```

...or application-defined objects:

```
tell application "Finder" to set name of startup disk to "Happy Fun Ball"
```

As mentioned in the Discussion, setting one variable to another makes both variables refer to the exact same object. If the object is mutable, that is, it has writable properties or elements, changes to the object will appear in both variables:

```
set alpha to {1, 2, {"a", "b"}}
set beta to alpha
```

```
set item 2 of item 3 of alpha to "change" --change the original variable
set item 1 of beta to 42 --change a different item in the new variable
{alpha, beta}
--result: {{42, 2, {"a", "change"}}, {42, 2, {"a", "change"}}}
```

Both variables show the same changes, because they both refer to the same object. Compare this with the similar example in [copy](#) (page 121). Assigning a new object to a variable is not the same thing as changing the object itself, and does not affect any other variables that refer to the same object. For example:

```
set alpha to {1, 2, 3}
set beta to alpha --result: beta refers to the same object as alpha
set alpha to {4, 5, 6}
--result: assigns a new object to alpha; this does not affect beta.
{alpha, beta}
--result: {{4, 5, 6}, {1, 2, 3}}
```

`set` can assign several variables at once using a pattern, which may be a list or a record. For example:

```
tell application "Finder" to set {x, y} to position of front window
```

Since `position of front window` evaluates to a list of two integers, this sets `x` to the first item in the list and `y` to the second item.

You can think of pattern assignment as shorthand for a series of simple assignments, but that is not quite accurate, because the assignments are effectively simultaneous. That means that you can use pattern assignment to exchange two variables:

```
set {x, y} to {1, 2} --now x is 1, and y is 2.
set {x, y} to {y, x} --now x is 2, and y is 1.
```

To accomplish the second statement using only simple assignments, you would need a temporary third variable.

For more information on using the `set` command, including a more complex pattern example, see [“Declaring Variables with the set Command”](#) (page 49).

### Discussion

Using the `set` command to assign a value to a variable causes the variable to refer to the original value. In a sense, it creates a new name for the same object. If multiple variables refer to a mutable object (that is, one with writable properties or elements, such as a list or `script` object), changes to the object are observable through any of the variables. If you want a separate copy, use the [copy](#) (page 121) command. This sharing only applies to values in AppleScript itself; it does not apply to values in other applications. Changing the object a variable refers to is not the same as altering the object itself, and does not affect other variables that refer to the same object.

```
set eof
```

---

Sets the length of a file, in bytes.

### Syntax

<code>set eof</code>	<i>fileSpecifier</i>	required
<code>to</code>	<i>integer</i>	required

**Parameters**

(*alias* (page 79) | *file* (page 88) | *file descriptor*)

The file to set the length of, as an alias, a file specifier, or as an integer file descriptor, which must be obtained as the result of an earlier `open for access` (page 140) call.

to *integer* (page 89)

The new length of the file, in bytes. If the new length is shorter than the existing length of the file, any data beyond that position is lost. If the new length is longer, the contents of the new bytes are unspecified.

**Result**

None.

Signals a “write permission” error if the file was opened using `open for access` without write permission.

**Examples**

If you want to completely replace the contents of an existing file, the first step must be to change its length to zero:

```
set theFile to choose file with prompt "Choose a file to clobber:"
set eof theFile to 0
```

```
set the clipboard to
```

---

Places data on the clipboard.

**Syntax**

```
set the clipboard to anything required
```

**Parameters**

*anything*

The data (of any type) to place on the clipboard.

**Result**

None.

**Examples**

The following script places text on the clipboard, then retrieves the text in TextEdit with a `the clipboard` (page 162) command:

```
set the clipboard to "Important new text."
tell application "TextEdit"
    activate --make sure TextEdit is running
    set clipText to the clipboard --result: "Important new text."
    --perform operations with retrieved text
end tell
```

**Discussion**

It is not necessary to use the clipboard to move data between scriptable applications. You can simply `get` the data from the first application into a variable and `set` the appropriate data in the second application.

## set volume

---

Sets the sound output, input, and alert volumes.

**Syntax**

set volume		required
	<i>number</i>	optional
output volume	<i>integer</i>	optional
input volume	<i>integer</i>	optional
alert volume	<i>integer</i>	optional
output muted	<i>boolean</i>	optional

**Parameters**

*number* (page 92)

The sound output volume, a real number from 0 to 7.

**Important:** This parameter is deprecated; if specified, all other parameters will be ignored.

output volume *integer* (page 89)

The sound output volume, an integer from 0 to 100.

*Default Value:*

None; the output volume is not changed.

input volume *integer* (page 89)

The sound input volume, an integer from 0 to 100.

*Default Value:*

None; the input volume is not changed.

alert volume *integer* (page 89)

The alert input volume, an integer from 0 to 100.

*Default Value:*

None; the alert volume is not changed.

output muted *boolean* (page 82)

Should the sound output be muted?

*Default Value:*

None; the output muting is not changed.

**Result**

None.

**Examples**

The following example saves the current volume settings, before increasing the output volume, saying some text, and restoring the original value:

```

set savedSettings to get volume settings
-- {output volume:32, input volume:70, alert volume:78, output muted:false}
set volume output volume 90
say "This is pretty loud."
set volume output volume (output volume of savedSettings)
delay 1
say "That's better."

```

## store script

---

Stores a script object into a file.

See also [run script](#) (page 151).

### Syntax

store script	<i>script</i>	required
in	<i>fileSpecifier</i>	optional
replacing	<i>replacingConstant</i>	optional

### Parameters

#### *script*

The script object to store.

in ([alias](#) (page 79) | [file](#) (page 88))

An [alias](#) or [file](#) specifier that specifies the file to store the `script` object in.

#### *Default Value:*

None; a standard Save As dialog will be presented to allow the user to choose where to save the `script` object.

replacing *replacingConstant*

Allow overwriting an existing file? You may specify one of the following constants:

yes

Overwrite without asking.

no

Never overwrite; signal an error if the file exists.

ask

Present a dialog asking the user what to do; the options are Replace (overwrite the file), Cancel (signal a “user canceled” error), or Save As (save to a different location).

#### *Default Value:*

ask

### Result

None.

**Examples**

This example stores a script on disk, using the Save As dialog to specify a location on the desktop and the name `storedScript`. It then creates an alias to the stored script and runs it with `run script`:

```
script test
    display dialog "Test"
end script

store script test --specify "Leopard:Users:myUser:Desktop:storedScript"

set localScript to alias "Leopard:Users:myUser:Desktop:storedScript" run script
localScript --result: displays dialog "Test"
```

The `store script` command stores only the contents of the script—in this case, the one statement, `display dialog "Test"`. It does not store the beginning and ending statements of the script definition.

**summarize**


---

Summarizes the specified text or text file.

**Syntax**

<code>summarize</code>	<i>textSpecifier</i>	required
<code>in</code>	<i>integer</i>	optional

**Parameters***textSpecifier*

The [text](#) (page 97), or an [alias](#) (page 79) to a text file, to summarize.

in *integer* (page 89)

The number of sentences desired in the summary.

*Default Value:*

1

**Result**

A [text](#) (page 97) object containing a summarized version of the text or file.

**Examples**

This example summarizes Lincoln's famous Gettysburg Address down to one sentence—a tough job even for AppleScript:

```
set niceSpeech to "Four score and seven years ago our fathers brought forth on
    this continent a new nation, conceived in Liberty, and dedicated to the
    proposition that all men are created equal.
    Now we are engaged in a great civil war, testing whether that nation, or any
    nation, so conceived and so dedicated, can long endure. We are met on a great
    battle-field of that war. We have come to dedicate a portion of that field, as
    a final resting place for those who here gave their lives that that nation
    might live. It is altogether fitting and proper that we should do this.
    But, in a larger sense, we can not dedicate—we can not consecrate—we can not
    hallow—this ground. The brave men, living and dead, who struggled here, have
    consecrated it, far above our poor power to add or detract. The world will little
```

```

note, nor long remember what we say here, but it can never forget what they
did here. It is for us the living, rather, to be dedicated here to the unfinished
work which they who fought here have thus far so nobly advanced. It is rather
for us to be here dedicated to the great task remaining before us—that from
these honored dead we take increased devotion to that cause for which they gave
the last full measure of devotion—that we here highly resolve that these dead
shall not have died in vain—that this nation, under God, shall have a new birth
of freedom—and that government of the people, by the people, for the people,
shall not perish from the earth."
set greatSummary to summarize niceSpeech in 1
display dialog greatSummary --result: displays one inspiring sentence

```

## system attribute

---

Get environment variables or attributes of this computer.

### Syntax

```
system attribute          attribute                optional
```

### Parameters

#### *attribute*

The attribute to test: either a Gestalt value or a shell environment variable name. Gestalt values are described in *Gestalt Manager Reference*.

#### *Default Value:*

If the attribute is omitted, `system attribute` will return a list of the names of all currently defined environment variables.

has [integer](#) (page 89)

For Gestalt values, an integer mask that is bitwise-ANDed with the Gestalt response. If the result is non-zero, `system attribute` returns `true`, otherwise `false`.

For environment variables, this parameter is ignored.

#### *Default Value:*

None; `system attribute` returns the original Gestalt response code.

### Result

If the attribute specified is a Gestalt selector, either the Gestalt response code or `true` or `false` depending on the `has` parameter.

If the attribute specified is an environment variable, the value of that variable, or an empty string (" ") if it is not defined.

If no attribute is supplied, a list of all defined environment variables.

### Examples

To get the current shell:

```
system attribute "SHELL" --result: "/bin/bash" (for example)
```

To get a list of all defined environment variables:

```
system attribute
```



```
(* result: (for example)
{"PATH", "TMPDIR", "SHELL", "HOME", "USER", "LOGNAME", "DISPLAY", "SSH_AUTH_SOCK",
 "Apple_PubSub_Socket_Render", "__CF_USER_TEXT_ENCODING", "SECURITYSESSIONID",
 "COMMAND_MODE"}
*)
```

## system info

---

Gets information about the system.

### Syntax

```
system info required
```

### Result

A record containing various information about the system and the current user. This record contains the following fields:

AppleScript version (a [text](#) (page 97) object)

The version number of AppleScript, for example, "2.0". This can be useful for testing for the existence of AppleScript features. When comparing version numbers, use `considering numeric strings` to make them compare in numeric order, since standard lexicographic ordering would consider "1.9" to come after "1.10".

AppleScript Studio version (a [text](#) (page 97) object)

The version number of AppleScript Studio, for example, "1.5".

system version (a [text](#) (page 97) object)

The version number of Mac OS X, for example, "10.5.1".

short user name (a [text](#) (page 97) object)

The current user's short name, for example, "hoser". This is set in the Advanced Options panel in the Accounts preference pane, or in the "Short Name" field when creating the account. This is also available from System Events using `name of current user`.

long user name (a [text](#) (page 97) object)

The current user's long name, for example, "Random J. Hoser". This is the "User Name" field in the Accounts preference pane, or in the "Name" field when creating the account. This is also available from System Events using `full name of current user`.

user ID (an [integer](#) (page 89))

The current user's user ID. This is set in the Advanced Options panel in the Accounts preference pane.

user locale (a [text](#) (page 97) object)

The current user's locale code, for example "en\_US".

home directory (an [alias](#) (page 79) object)

The location of the current user's home folder. This is also available from Finder's `home` property, or System Events' `home folder` property.

boot volume (a [text](#) (page 97) object)

The name of the boot volume, for example, "Macintosh HD". This is also available from Finder or System Events using `name of startup disk`.

computer name (a [text](#) (page 97) object)

The computer's name, for example "mymac". This is the "Computer Name" field in the Sharing preference pane.

host name (a [text](#) (page 97) object)

The computer's DNS name, for example "mymac.local".

IPv4 address (a [text](#) (page 97) object)

The computer's IPv4 address, for example "192.201.168.13".

primary Ethernet address (a [text](#) (page 97) object)

The MAC address of the primary Ethernet interface, for example "00:1c:63:91:4e:db".

CPU type (a [text](#) (page 97) object)

The CPU type, for example "Intel 80486".

CPU speed (an [integer](#) (page 89))

The clock speed of the CPU in MHz, for example 2400.

physical memory (an [integer](#) (page 89))

The amount of physical RAM installed in the computer, in megabytes (MB), for example 2048.

### Examples

```
system info --result: long record of information
```

## the clipboard

---

Returns the contents of the clipboard.

### Syntax

the clipboard		required
as	<i>class</i>	optional

### Parameters

as [class](#) (page 83)

The type of data desired. `the clipboard` will attempt to find that "flavor" of data on the clipboard; if it is not found, it will attempt to coerce whatever flavor is there.

### Result

The data from the clipboard, which can be of any type.

### Examples

The following script places text on the clipboard, and then appends the clipboard contents to the frontmost TextEdit document:

```
set the clipboard to "Add this sentence at the end."
tell application "TextEdit"
    activate --make sure TextEdit is running
    make new paragraph at end of document 1 with data (return & the clipboard)
end tell
```

### Discussion

It is not necessary to use the clipboard to move data between scriptable applications. You can simply get the data from the first application into a variable and set the appropriate data in the second application.

## time to GMT

---

Returns the difference between local time and GMT (Greenwich Mean Time) or Universal Time, in seconds.

### Syntax

```
time to GMT required
```

### Result

The [integer](#) (page 89) number of seconds difference between the current time zone and Universal Time.

### Examples

The following example computes the time difference between the current location and Cupertino:

```
set localOffset to time to GMT --local difference, in seconds
set cupertinoOffset to -8.0 * hours
  --doesn't account for Daylight Savings; may actually be -7.0.
set difference to (localOffset - cupertinoOffset) / hours
display dialog ("Hours to Cupertino: " & difference)
```

## write

---

Writes data to a specified file.

### Syntax

```
write anything required
to fileSpecifier required
starting at integer optional
for integer optional
as class optional
```

### Parameters

#### *anything*

The data to write to the file. This is typically `text`, but may be of any type. When reading the data back, the `read` command must specify the same type, or the results are undefined.

#### to ( *alias* (page 79) | *file* (page 88) | *file descriptor* )

The file to write to, as an alias, a file specifier, or an [integer](#) (page 89) file descriptor. A file descriptor must be obtained as the result of an earlier [open for access](#) (page 140) call.

starting at (*integer* (page 89) | *eof*)

The byte position in the file to start reading from. The position is 1-based, so 1 is the first byte of the file, 2 the second, and so on. Negative integers count from the end of the file, so -1 is the last byte, -2 the second-to-last, and so on. The constant *eof* is the position just after the last byte; use this to append data to the file.

*Default Value:*

The current file pointer (see *open for access* (page 140)) if the file is open, or the beginning of the file if not.

for *integer* (page 89)

The number of bytes to write.

*Default Value:*

Write all the data provided.

as *class* (page 83)

Write the data as this class. The most common ones control the use of three different text encodings:

*text* or *string*

The primary text encoding, as determined by the user's language preferences set in the International preference panel. (For example, Mac OS Roman for English, MacJapanese for Japanese, and so on.)

Unicode text

UTF-16.

«class utf8»

UTF-8.

Any other class is possible, for example *date* or *list*, but is typically only useful if the data will be read using a *read* statement specifying the same value for the *as* parameter.

*Default Value:*

The class of the supplied data. See Special Considerations.

### Result

None. If the file is open, *write* will advance the file pointer by the number of bytes written, so the next *write* command will start writing where the last one ended.

Signals an error if the file is open without write permission, or if there is any other problem that prevents writing to the file, such as a lack of disk space.

### Examples

The following example opens a file with write permission, creating it if it doesn't already exist, writes text to it, and closes it.

```
set fp to open for access file "HD:Users:myUser:NewFile" with write permission
write "Some text. And some more text." to fp
close access fp
```

**Special Considerations**

As specified above, `write` with `no as` parameter writes as the class of the supplied data, which means that in AppleScript 2.0 `write` always writes `text` data using the primary encoding. Prior to 2.0, `string` and `Unicode text` were distinct types, which meant that it would use primary encoding for `string` and UTF-16 for `Unicode text`. For reliable results when creating scripts that will run on both 2.0 and pre-2.0, always specify the encoding explicitly using `as text` or `as Unicode text`, as appropriate.



# Reference Forms

---

This chapter describes AppleScript reference forms. A **reference form** specifies the syntax for identifying an object or group of objects in an application or other container—that is, the syntax for constructing an object specifier (described in “[Object Specifiers](#)” (page 29)).

For example, the following object specifier (from a script targeting the Finder) uses several index reference forms, which identify an object by its number within a container:

```
item 1 of second folder of disk 1
```

**Important:** When you use a reference form, you specify the container in which the referenced object or objects reside. This takes the form *referenceForm* of *containerObject*. You can also enclose a reference form in a `tell` statement, which then serves to specify the outer container. For more information, see “[Absolute and Relative Object Specifiers](#)” (page 30).

Some of the examples of reference forms shown in this chapter will not compile as shown. To compile them, you may need to add an enclosing `tell` statement, targeting the Finder or the word processing application TextEdit.

## Arbitrary

---

Specifies an arbitrary object in a container. This form is useful whenever randomness is desired.

Because an arbitrary item is, by its nature, random, this form is not useful for operations such as processing each item in a group of files, words, or other objects.

### Syntax

```
some class
```

### Placeholders

```
class
```

The class for an arbitrary object.

### Examples

The following creates a new Mail message with a random signature (and depends on the user having at least one signature):

```
tell application "Mail"
  activate
  set randomSignature to some signature
  set newMessage to make new outgoing message ↵
    at end of outgoing messages with properties ↵
      {subject:"Guess who?", content:"Welcome aboard.", visible:true}
  set message signature of newMessage to randomSignature
end tell
```

The following simply gets a random word from a TextEdit document:

```
tell application "TextEdit"
    some word of document 1 -- any word from the first document
end tell
```

## Every

---

Specifies every object of a particular class in a container.

### Syntax

```
every class
```

*pluralClass*

### Placeholders

*class*

A singular class (such as word or paragraph).

*pluralClass*

The plural form for a class (such as words or paragraphs).

### Value

The value of an every object specifier is a list of the objects from 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 uses an every object specifier to specify every word contained in a text string:

```
set myText to "That's all, folks"
every word of myText --result: {"That's", "all", "folks"} (a list of three words)
```

The following object specifier specifies the same list:

```
words of myText
```

The following example specifies a list of all the items in the Users folder of the startup disk (boot partition):

```
tell application "Finder"
    every item of folder "Users" of startup disk
end tell
```

The following specifies the same list as the previous example:

```
tell application "Finder"
    items of folder "Users" of startup disk
end tell
```

### Discussion

Use of the every reference form implies the existence of an index property for the specified objects.



If you specify an `every` object specifier as the container from which to obtain 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.

## Filter

Specifies all objects in a container that match a condition, or test, specified by a Boolean expression.

The filter form specifies application objects only. It cannot be used to filter the AppleScript objects `list` (page 89), `record` (page 94), or `text` (page 97). A term that uses the filter form is also known as a `whose` clause.

**Note:** You can use the words `where` or `that` as synonyms for `whose`.

A filter reference form can often be replaced by a `repeat` statement, or vice versa. For example, the following script closes every TextEdit window that isn't named "Old Report.rtf":

```
tell application "TextEdit"
    close every window whose name is not "Old Report.rtf"
end tell
```

You could instead obtain a list of open windows and set up a `repeat` statement that checks the name of each window and closes the window if it isn't named "Old Report.rtf". However, a `whose` clause is often the fastest way to obtain the desired information.

The following is an abbreviated form of the previous script:

```
windows of application "TextEdit" whose name is not "Old Report.rtf"
```

For related information, see [“repeat Statements”](#) (page 199).

### Syntax

```
objectSpecifier ( whose | where ) booleanTest
```

#### Placeholders

*objectSpecifier*

Specifies the container in which to look for objects that match the Boolean test.

*whose* | *where*

These words have the same meaning, and refer to all of the objects in the specified container that match the conditions in the specified Boolean expression.

*booleanTest*

Any Boolean expression (see the `boolean` (page 82) class definition).

#### Value

The value of a filter reference form is a list of the objects that pass the test. If no objects pass the test, the list is an empty list: `{}`.

#### Examples

The following example shows an object specifier for all open Finder windows that do not have the name "AppleScript Language Guide".

```
tell application "Finder"
    every window whose name is not "AppleScript Language Guide"
end tell
```

**Discussion**

In effect, a filter reduces the number of objects in a container. Instead of specifying every Finder window, the following object specifier specifies just the windows that are currently zoomed:

```
every window whose zoomed is true
```

To specify a container after a filter, you must enclose the filter and the object specifier it applies to in parentheses, as in this example:

```
tell application "Finder"
    (files whose file type is not "APPL") in folder "HD:SomeFolder:"
end tell
```

Within a test in a filter reference, the direct object is the object being tested. Though it isn't generally needed, this implicit target can be specified explicitly using the keyword `it`, which is described in [“The it and me Keywords”](#) (page 40).

The following example shows several equivalent ways of constructing a filter reference to find all the files in a folder that whose name contains the word “AppleScript”. While the term `it` refers to the Finder application outside of the filter statements, within them `of it` refers to the current file being tested. The result of each filter test is the same and is not changed by including or omitting the term `of it`:

```
tell application "Finder"
    it --result: application "Finder" (target of tell statement)
    set myFolder to path to home folder
        --result: alias "Leopard:Users:myUser:"
    files in myFolder --result: a list of Finder document files
    files in myFolder where name of it contains "AppleScript"
    (* result: document file "AppleScriptLG.pdf" of folder "myUser"
       of folder "Users" of startup disk of application "Finder"*)
    files in myFolder where name contains "AppleScript" -- same result
    every file in myFolder whose name contains "AppleScript" -- same result
    every file in myFolder where name of it contains "AppleScript"
        -- same result
end tell
```

A filter reference form 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 8-1](#) (page 170) shows some filter references, the Boolean expressions they contain, and what is being tested in each reference.

**Table 8-1** Boolean expressions and tests in filter references

Filter reference form	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

Filter reference form	Boolean expression	What is being tested
files whose creator type is "OMGR"	creator type is "OMGR"	The creator type property of each file

A test can be any Boolean expression. You can link multiple tests, as in the following statement:

```
windows whose zoomed is true and floating is false
```

## ID

---

Specifies an object by the value of its `id` property.

You can use the ID reference form only with application objects that have an ID property.

### Syntax

```
class id expression
```

### Placeholders

```
expression
```

The id value.

### Examples

The following examples use the ID reference form to specify an application by ID and a disk object by ID.

```
tell application id "com.apple.finder"
-- specifies an application (Finder) by its ID
disk id -100 -- specifies a Finder disk object by ID
name of disk id -100 --result: "Leopard_GM" (gets name from ID specifier)
end tell
```

### Discussion

Use of the `id` reference form implies the existence of a `id` property for the specified objects.

Although `id` properties are most often integers, an `id` property can belong to any class. An application that supports `id` properties for its scriptable objects 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 typically 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. Similarly, you could keep track of an item by its index, but indexes can change when items in a container are added, deleted, or even renamed.

**Note:** A good way to keep track of files and folders is to use an [alias](#) (page 79).

Starting in AppleScript 2.0, objects of class `application` (page 80) have an `id` property, which represents the application's bundle identifier (the default) or its four-character signature code.

Also starting in AppleScript 2.0, objects of class `text` (page 97) have an `id` property, representing the Unicode code point or points for the character or characters in the object. Because a `text` object's ID is based on the characters it contains, these IDs are not guaranteed to be unique, and in fact will be identical for two `text` objects that store the same characters. And in fact, there is no way to tell two such objects apart by inspection.

## Index

---

Specifies an object by describing its position with respect to the beginning or end of a container.

For related information, see [“Relative”](#) (page 176).

### Syntax

*class* [ *index* ] *integer*

*integer* (*st* | *nd* | *rd* | *th*) *class*

(*first* | *second* | *third* | *fourth* | *fifth* | *sixth* | *seventh* | *eighth* | *ninth* | *tenth*) *class*

(*last* | *front* | *back*) *class*

### Placeholders

*class*

The class of the indexed object to obtain.

*integer*

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).

*st* | *nd* | *rd* | *th*

Appended to the appropriate integer to form an index. For example, 1st, 2nd, 3rd.

*first* | *second* | *third* | *fourth* | *fifth* | *sixth* | *seventh* | *eighth* | *ninth* | *tenth*

Specify one of the ordinal indexes.

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*, and so on. (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.)

*last* | *front* | *back*

The *front* form (for example, *front window*) is equivalent to *class 1* (*window 1*) or *first class* (*first window*). The *last* and *back* forms (for example, *last word* and *back window*) refer to the last object in a container. They are equivalent to *class -1* (for example, *window -1*).

### Examples

Each of the following object specifiers specifies the first item on the startup disk:

```
item 1 of the startup disk
item index 1 of the startup disk -- "index" is usually omitted
the first item of the startup disk
```

The following object specifiers 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 object specifiers specify the last word in the third paragraph:

word -1 of paragraph 3  
 last word of paragraph 3

The following object specifiers specify the next-to-last word in the third paragraph.

word -2 of paragraph 3  
 -2th word of paragraph 3

### Discussion

Indexes are 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 `word 4` from a paragraph, the word no longer exists. But there may still be a `word 4`—the word that was formerly `word 5`. After `word 4` is deleted, any words with an index higher than 4 will also have a new index. So the object an index specifies can change.

For a unique, persistent object specifier, you can use the `id` reference form (see “ID” (page 171)), if the application supports it for the class of object you are working with. And for keeping track of a file, you can use an `alias` (page 79) object.

## Middle

---

Specifies the middle object of a particular class in a container. This form is rarely used.

### Syntax

`middle class`

### Placeholders

`class`

The class of the middle object to obtain.

### Examples

```
tell application "TextEdit"
    middle paragraph 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
```

### Discussion

The `middle` reference form generally works only when the `index` form also works.

AppleScript calculates the middle object by taking half the count, then rounding up. For example, the middle word of a paragraph containing ten words is the fifth word; the middle of eleven words is the sixth.

## Name

---

Specifies an object by name.

**Syntax**

```
class [ named ] nameText
```

**Placeholders**

*class*

The class for the specified object.

*nameText*

The value of the object's name property.

**Examples**

The following statements identify objects by name:

```
document "Report.rtf"
window named "logs"
```

**Discussion**

Use of the `name` reference form implies the existence of a `name` property for the specified objects.

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"
    item 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.

## Property

---

Specifies a property of an object.

**Syntax**

*propertyLabel*

**Placeholders**

*propertyLabel*

The label for the property.

**Examples**

The following example is an object specifier to a property of a Finder window. It lists the label for the window's property (`zoomed`) and its container (`front window`). `zoomed` is a Boolean property.

```
zoomed of front window -- e.g., false, if the window isn't zoomed
```

For many objects, you can obtain a list of properties:

```
tell app "Finder"
    properties of window 1 --result: a list of properties and their values
```

```
end tell
```

The following example is an object specifier to the `UnitPrice` property of a `record` (page 94) object. The label of the property is `UnitPrice` and the container is the `record` object.

```
UnitPrice of {Product:"Super Snack", UnitPrice:0.85, Quantity:10} --result: 0.85
```

### Discussion

Property labels are listed in 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 specify the property—there is no need to specify the class of the property.

## Range

---

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 (integers from integer 1 to integer 3).

### Syntax

```
every class from boundarySpecifier1 to boundarySpecifier2
```

```
pluralClass from boundarySpecifier1 to boundarySpecifier2
```

```
class startIndex ( thru | through ) stopIndex
```

```
pluralClass startIndex ( thru | through ) stopIndex
```

### Placeholders

*class*

A singular class (such as `window` or `word`).

*pluralClass*

A plural class (such as `windows` or `words`).

*boundarySpecifier1* and *boundarySpecifier2*

Specifiers to objects that bound the range. The range includes the boundary objects. You can use the reserved word `beginning` in place of *boundarySpecifier1* to indicate the position before the first object of the container. Similarly, you can use the reserved word `end` in place of *boundarySpecifier2* to indicate the position after the last object in the container.

*startIndex* and *stopIndex*

The indexes of the first and last object of the range (such as 1 and 10 in `words 1 thru 10`).

Though integer indexes are the most common class, the start and stop indexes can be of any class. An application determines which index classes are meaningful to it.

### Value

The value of a range reference form is a list of the objects in the range. If the specified container does not contain objects of the specified class, or if the range is out of bounds, an error is returned. For example, the following range specifier results in an error because there are no words in the list:

```
words 1 thru 3 of {1, 2, 3} --result: an error
```

**Examples**

The following example shows the boundary object form of a range specifier. When you compile this statement, Script Editor converts from integer 1 to integer 2 to the form integers 1 thru 2.

```
set intList to integers from integer 1 to integer 2 of {17, 33, 24}
--result: {17, 33}
```

In the next example, the phrase folders 3 thru 4 is a range specifier that specifies a list of two folders in the container startup disk:

```
tell application "Finder"
    folders 3 thru 4 of startup disk
end tell
--result: a list of folders (depends on contents of startup disk)
```

**Discussion**

If you specify a range specifier as the container for a property or object, as in

```
name of folders 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.

To obtain a contiguous series of characters—instead of a list—from a text object, use the text class:

```
text from word 1 to word 4 of "We're all in this together"
--result: "We're all in this"
words 1 thru 4 of "We're all in this together"
--result: {"We're", "all", "in", "this"}
```

## Relative

---

Specifies an object or an insertion point in a container by describing a position in relation to another object, known as the base, in the same container.

**Syntax**

```
[class](before|[in] front of) baseSpecifier
```

```
[class](after|[in] back of|behind) baseSpecifier
```

**Placeholders***class*

The class identifier of the specified object. If you omit this parameter, the specifier refers to an insertion point.

*baseSpecifier*

A specifier for the object.

*before|[in] front of*

These forms are equivalent, and refer to the object immediately preceding the base object.

*after|[in] back of|behind*

These forms are equivalent, and refer to the object immediately after the base.



beginning | front

These forms are equivalent, and refer to the first insertion point of the container (insertion point 1).

end | back

These 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 relative specifiers in the following `tell` block specify the same file by identifying its position relative to another file on a disk:

```
tell application "Finder"
    item before item 3 of startup disk --result: e.g., a specifier
    item after item 1 of startup disk --result: e.g., a specifier
end tell
```

The following example shows how to use various relative specifiers in a word processing document:

```
tell first document of application "TextEdit"
    copy word 1 to before paragraph 3
    copy word 3 to in back of paragraph 4
    copy word 1 of the last paragraph to behind the third paragraph
end tell
```

### Discussion

The `relative` reference form generally works only when the `index` form also works.

You can specify only a single object with a relative specifier—an object that is either before or after the base object.



# Operators Reference

---

This chapter describes AppleScript operators. An **operator** is a symbol, word, or phrase that derives a value from another value or pair of values. An **operation** is the evaluation of an expression that contains an operator. An **operand** is an expression from which an operator derives a value.

AppleScript provides logical and mathematical operators, as well as operators for containment, concatenation, and obtaining a reference to an object. Operators that operate on two values are called **binary operators**, while operators that operate on a single value are known as **unary operators**.

The first part of this chapter contains two tables: Table 9-1 summarizes all of the operators that AppleScript uses, and Table 9-2 (page 186) shows the order in which AppleScript evaluates operators within expressions. The rest of the chapter shows how AppleScript evaluates representative operators in script expressions.

**Table 9-1** AppleScript operators

AppleScript operator	Description
and	<p>Logical conjunction.</p> <p>A binary logical operator that combines two Boolean values. The result is <code>true</code> only if both operands evaluate to <code>true</code>.</p> <p>AppleScript checks the left-hand operand first and, if it is <code>false</code>, ignores the right-hand operand. (This behavior is called short-circuiting.)</p> <p>Class of operands: <code>boolean</code> (page 82)</p> <p>Class of result: <code>boolean</code></p>
or	<p>Logical disjunction.</p> <p>A binary logical operator that combines two Boolean values. The result is <code>true</code> if either operand evaluates to <code>true</code>.</p> <p>AppleScript checks the left-hand operand first and, if it is <code>true</code>, ignores the right-hand operand. (This behavior is called short-circuiting.)</p> <p>Class of operands: <code>boolean</code> (page 82)</p> <p>Class of result: <code>boolean</code></p>

AppleScript operator	Description
&	<p><b>Concatenation.</b></p> <p>A binary operator that joins two values. If the left-hand operand is a <code>text</code> object, the result is a <code>text</code> object (and only in this case does AppleScript try to coerce the value of the right-hand operand to match that of the left).</p> <p>If the operand to the left is a record, the result is a record. If the operand to the left belongs to any other class, the result is a list.</p> <p>For more information, see <a href="#">&amp; (concatenation)</a> (page 187).</p> <p>Class of operands: any</p> <p>Class of result: <a href="#">list</a> (page 89), <a href="#">record</a> (page 94), <a href="#">text</a> (page 97)</p>
= is equal equals [is] equal to	<p><b>Equality.</b></p> <p>A binary comparison operator that results in <code>true</code> if both operands have the same value. The operands can be of any class.</p> <p>For more information, see <a href="#">equal</a>, <a href="#">is not equal to</a> (page 190).</p> <p>Class of operands: <a href="#">boolean</a> (page 82)</p> <p>Class of result: <code>boolean</code></p>
(Option-equal sign on U.S. keyboard) is not isn't isn't equal [to] is not equal [to] doesn't equal does not equal	<p><b>Inequality.</b></p> <p>A binary comparison operator that results in <code>true</code> if its two operands have different values. The operands can be of any class.</p> <p>For more information, see <a href="#">equal</a>, <a href="#">is not equal to</a> (page 190).</p> <p>Class of operands: <a href="#">boolean</a> (page 82)</p> <p>Class of result: <code>boolean</code></p>
> [is] greater than comes after is not less than or equal [to] isn't less than or equal [to]	<p><b>Greater than.</b></p> <p>A binary comparison operator that results in <code>true</code> if the value of the left-hand operand is greater than the value of the right-hand operand.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the right-hand operand to the class of the left-hand operand.</p> <p>For more information, see <a href="#">greater than</a>, <a href="#">less than</a> (page 191).</p> <p>Class of operands: <a href="#">date</a> (page 85), <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93), <a href="#">text</a> (page 97)</p> <p>Class of result: <code>boolean</code> (page 82)</p>

AppleScript operator	Description
<p>&lt;</p> <p>[is] less than comes before</p> <p>is not greater than or equal [to]</p> <p>isn't greater than or equal [to]</p>	<p>Less than.</p> <p>A binary comparison operator that results in <code>true</code> if the value of the left-hand operand is less than the value of the right-hand operand.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the right-hand operand to the class of the operand to the left.</p> <p>For more information, see <a href="#">greater than</a>, <a href="#">less than</a> (page 191).</p> <p>Class of operands: <a href="#">date</a> (page 85), <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93), <a href="#">text</a> (page 97)</p> <p>Class of result: <a href="#">boolean</a> (page 82)</p>
<p>≥ (Option-period on U.S. keyboard)</p> <p>&gt;=</p> <p>[is] greater than or equal [to]</p> <p>is not less than</p> <p>isn't less than</p> <p>does not come before</p> <p>doesn't come before</p>	<p>Greater than or equal to.</p> <p>A binary comparison operator that results in <code>true</code> if the value of the left-hand operand is greater than or equal to the value of the right-hand operand.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the right-hand operand to the class of the operand to the left.</p> <p>The method AppleScript uses to determine which value is greater depends on the class of the operands.</p> <p>Class of operands: <a href="#">date</a> (page 85), <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93), <a href="#">text</a> (page 97)</p> <p>Class of result: <a href="#">boolean</a> (page 82)</p>
<p>≤ (Option-comma on U.S. keyboard)</p> <p>&lt;=</p> <p>[is] less than or equal [to]</p> <p>is not greater than</p> <p>isn't greater than</p> <p>does not come after</p> <p>doesn't come after</p>	<p>Less than or equal to.</p> <p>A binary comparison operator that results in <code>true</code> if the value of the left-hand operand is less than or equal to the value of the right-hand operand.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the right-hand operand to the class of the operand to the left.</p> <p>The method AppleScript uses to determine which value is greater depends on the class of the operands.</p> <p>Class of operands: <a href="#">date</a> (page 85), <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93), <a href="#">text</a> (page 97)</p> <p>Class of result: <a href="#">boolean</a> (page 82)</p>

AppleScript operator	Description
<p><code>start[s] with</code>  <code>begin[s] with</code></p>	<p><b>Starts with.</b></p> <p>A binary containment operator that results in <code>true</code> if the <code>list</code> or <code>text</code> object to its right matches the beginning of the <code>list</code> or <code>text</code> object to its left.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the right-hand operand to the class of the operand to the left.</p> <p>For more information, see <a href="#">starts with</a>, <a href="#">ends with</a> (page 192).</p> <p>Class of operands: <a href="#">list</a> (page 89), <a href="#">text</a> (page 97)</p> <p>Class of result: <a href="#">boolean</a> (page 82)</p>
<p><code>end[s] with</code></p>	<p><b>Ends with.</b></p> <p>A binary containment operator that results in <code>true</code> if the <code>list</code> or <code>text</code> object to its right matches the end of the <code>list</code> or <code>text</code> object to its left.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the right-hand operand to the class of the operand to the left.</p> <p>For more information, see <a href="#">starts with</a>, <a href="#">ends with</a> (page 192).</p> <p>Class of operands: <a href="#">list</a> (page 89), <a href="#">text</a> (page 97)</p> <p>Class of result: <a href="#">boolean</a> (page 82)</p>
<p><code>contain[s]</code></p>	<p><b>Containment.</b></p> <p>A binary containment operator that results in <code>true</code> if the <code>list</code>, <code>record</code>, or <code>text</code> object to its right matches any part of the <code>list</code>, <code>record</code>, or <code>text</code> object to its left.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the right-hand operand to the class of the operand to the left.</p> <p>For more information, see <a href="#">contains</a>, <a href="#">is contained by</a> (page 189).</p> <p>Class of operands: <a href="#">list</a> (page 89), <a href="#">record</a> (page 94), <a href="#">text</a> (page 97)</p> <p>Class of result: <a href="#">boolean</a> (page 82)</p>

AppleScript operator	Description
<p>does not contain doesn't contain</p>	<p><b>Non-containment.</b></p> <p>A binary containment operator that results in <code>true</code> if the <code>list</code>, <code>record</code>, or <code>text</code> object to its right does not match any part of the <code>list</code>, <code>record</code>, or <code>text</code> object to its left.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the right-hand operand to the class of the left-hand operand.</p> <p>For more information, see <a href="#">contains</a>, <a href="#">is contained by</a> (page 189).</p> <p>Class of operands: <a href="#">list</a> (page 89), <a href="#">record</a> (page 94), <a href="#">text</a> (page 97)</p> <p>Class of result: <a href="#">boolean</a> (page 82)</p>
<p>is in is contained by</p>	<p><b>Containment.</b></p> <p>A binary containment operator that results in <code>true</code> if the <code>list</code>, <code>record</code>, or <code>text</code> object to its left matches any part of the <code>list</code>, <code>record</code>, or <code>text</code> object to its right.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the left-hand operand to the class of the right-hand operand.</p> <p>For more information, see <a href="#">contains</a>, <a href="#">is contained by</a> (page 189).</p> <p>Class of operands: <a href="#">list</a> (page 89), <a href="#">record</a> (page 94), <a href="#">text</a> (page 97)</p> <p>Class of result: <a href="#">boolean</a> (page 82)</p>
<p>is not in is not contained by isn't contained by</p>	<p><b>Non-containment.</b></p> <p>A binary containment operator that results in <code>true</code> if the <code>list</code>, <code>record</code>, or <code>text</code> object to its left does not match any part of the <code>list</code>, <code>record</code>, or <code>text</code> object to its right.</p> <p>Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the left-hand operand to the class of the right-hand operand.</p> <p>For more information, see <a href="#">contains</a>, <a href="#">is contained by</a> (page 189).</p> <p>Class of operands: <a href="#">list</a> (page 89), <a href="#">record</a> (page 94), <a href="#">text</a> (page 97)</p> <p>Class of result: <a href="#">boolean</a> (page 82)</p>
<p>*</p>	<p><b>Multiplication.</b></p> <p>A binary arithmetic operator that multiplies the number to its left and the number to its right.</p> <p>Class of operands: <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93)</p> <p>Class of result: <code>integer</code>, <code>real</code></p>

AppleScript operator	Description
+	<p>Addition.</p> <p>A 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.</p> <p>As a unary operator, + has no effect and is removed on compile.</p> <p>Class of operands: <a href="#">date</a> (page 85), <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93)</p> <p>Class of result: <code>date</code>, <code>integer</code>, <code>real</code></p>
-	<p>Subtraction.</p> <p>A binary or unary arithmetic operator.</p> <p>The binary operator subtracts the number to its right from the number or date to its left.</p> <p>The unary operator makes the number to its right negative.</p> <p>Only integers can be subtracted from dates. AppleScript interprets such an integer as a number of seconds.</p> <p>Class of operands: <a href="#">date</a> (page 85), <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93)</p> <p>Class of result: <code>date</code>, <code>integer</code>, <code>real</code></p>
/ ÷ (Option-slash on U.S. keyboard)	<p>Division.</p> <p>A binary arithmetic operator that divides the number to its left by the number to its right.</p> <p>Class of operands: <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93)</p> <p>Class of result: <code>real</code></p>
div	<p>Integral division.</p> <p>A 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.</p> <p>Class of operands: <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93)</p> <p>Class of result: <code>integer</code></p>
mod	<p>Remainder.</p> <p>A binary arithmetic operator that divides the number to its left by the number to its right and returns the remainder as its result.</p> <p>Class of operands: <a href="#">integer</a> (page 89), <a href="#">real</a> (page 93)</p> <p>Class of result: <code>integer</code>, <code>real</code></p>



AppleScript operator	Description
<code>^</code>	<p>Exponentiation.</p> <p>A binary arithmetic operator that raises the number to its left to the power of the number to its right.</p> <p>Class of operands: <code>integer</code> (page 89), <code>real</code> (page 93)</p> <p>Class of result: <code>real</code></p>
<code>as</code>	<p>Coercion (or <i>object conversion</i>).</p> <p>A binary operator that converts the left-hand operand to the class listed to its right.</p> <p>Not all values can be coerced to all classes. The coercions that AppleScript can perform are listed in “<a href="#">Coercion (Object Conversion)</a>” (page 32). The additional coercions, if any, that an application can perform is listed in its dictionary.</p> <p>Class of operands: The right-hand operand must be a class identifier; the left-hand operand must be a value that can be converted to that class.</p> <p>Class of result: The class specified by the class identifier to the right of the operator</p>
<code>not</code>	<p>Negation.</p> <p>A unary logical operator that results in <code>true</code> if the operand to its right is <code>false</code>, and <code>false</code> if the operand is <code>true</code>.</p> <p>Class of operand: <code>boolean</code> (page 82)</p> <p>Class of result: <code>boolean</code></p>
<code>[a] (ref [to]   reference to)</code>	<p>A reference to.</p> <p>A unary operator that causes AppleScript to return a <a href="#">reference</a> (page 95) object that specifies the location of the operand to its right. A reference is evaluated at run time, not at compile time.</p> <p>See a <a href="#">reference to</a> (page 188) for more information.</p> <p>Class of operand: any class type</p> <p>Class of result: <code>reference</code></p>

When evaluating expressions, AppleScript uses operator precedence to determine which operations are evaluated first. In the following expression, for example, AppleScript does not simply perform operations from left to right—it performs the multiplication operation `2 * 5` first, because multiplication has higher precedence than addition.

```
12 + 2 * 5 --result: 22
```

[Table 9-2](#) (page 186) shows the order in which AppleScript performs operations. The column labeled “Associativity” indicates the order in the case where there are two or more operands 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.”

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))`.

You can enforce the order in which AppleScript performs operations by grouping expressions in parentheses, which are evaluated first, starting with the innermost pair of parentheses.

**Table 9-2** 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 (note that this is different from standard math, in which exponentiation takes precedence over unary plus or minus)
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 and
12	or	Left to right	Logical or

The following sections provide additional detail about how AppleScript evaluates operators in expressions:

- [& \(concatenation\)](#) (page 187)
- [a reference to](#) (page 188)

- [contains, is contained by](#) (page 189)
- [equal, is not equal to](#) (page 190)
- [greater than, less than](#) (page 191)
- [starts with, ends with](#) (page 192)

## & (concatenation)

---

The concatenation operator (&) concatenates `text` objects, joins `record` objects into a record, and joins other objects into a list.

[Table 9-1](#) (page 179) summarizes the use of use of this operator.

### text

The concatenation of two `text` objects joins the characters from the left-hand `text` object to the characters from the right-hand `text` object, without intervening spaces. For example, `"dump" & "truck"` evaluates to the `text` object `"dumptruck"`.

If the left-hand operand is a `text` object, but the right-hand operand is not, AppleScript attempts to coerce the right-hand operand to a `text` object. For example, when AppleScript evaluates the expression `"Route " & 66` it coerces the integer `66` to the `text` object `"66"`, and the result is the `text` object `"Route 66"`.

However, you get a different result if you reverse the order of the operands:

```
66 & "Route " --result: {66, "Route "} (a list)
```

In the following example, the left-hand operand is a `text` object and the right-hand operand is a list, so concatenation results in a `text` object:

```
item 1 of {"This"} & {"and", "that"} -- "Thisandthat"
```

### record

The concatenation of two records joins the properties of the left-hand record to the properties of the right-hand record. If both records contain properties with the same name, the value of the property from the left-hand record 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

Except for the cases described above for `text` objects and `record` objects, the concatenation operator (&) joins lists. A non-list operand is considered to be a list containing that operand. The following example shows concatenation of two integers, a list and a text string, and a list and a record, respectively:

```
1 & 2 --result: {1, 2}
{"this"} & "hello" --result: {"this", "hello"}
{"this"} & {a:1, b:2} --result: {"this", 1, 2}
```

If both the operands to be concatenated are lists, then the result is a list containing all the items in the left-hand list, followed by all the items in the right-hand list. For example:

```

{"This"} & {"and", "that"} --result: {"This", "and", "that"}
{"This"} & item 1 of {"and", "that"} --result: {"This", "and"}

```

To join two lists and create a list of lists, rather than a single list, you can enclose each list in two sets of brackets:

```

{{1, 2}} & {{3, 4}} --result: {{1, 2}, {3, 4}}

```

For information on working efficiently with large lists, see [list](#) (page 89).

## a reference to

---

The `a reference to` operator is a unary operator that returns a reference object. You can abbreviate this operator to `a ref to`, or `ref to`, or even just `ref`.

For related information, see the [reference](#) (page 95) class and “[Object Specifiers](#)” (page 29).

### Examples

The following statement creates a reference object that contains an object specifier to the Finder startup disk:

```

tell app "Finder" to set diskRef to a ref to startup disk
--result: startup disk of application "Finder"

```

The following shows how to obtain a reference object that refers to an item in a list:

```

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

```

In the final line, AppleScript automatically resolves the object specifier contained in the reference `itemRef` and obtains its value to use in the addition operation. To cause AppleScript to explicitly resolve a reference object, you can use its `contents` property:

```

contents of itemRef --result: 755

```

The next examples demonstrate how using a reference object can result in a different outcome than accessing an object directly. The first example obtains a current track object from iTunes, gets the name, changes the track, then gets the name again:

```

tell application "iTunes"
    set curTrack to current track
    --result: file track id 2703 of user playlist id 2425
    --      of source id 46 of application "iTunes"
    display dialog (name of curTrack as string) -- "Shattered"
    next track -- play next song
    display dialog (name of curTrack as string) -- "Shattered"
end tell

```

Because `curTrack` is a specific track object, its name doesn't change when the current track changes. But observe the result when using a reference to the current track:

```
tell application "iTunes"
  set trackRef to a reference to current track
  --result: current track of application "iTunes"
  display dialog (name of trackRef as string) -- "Shattered"
  next track -- play next song
  display dialog (name of trackRef as string) -- "Strange Days"
end tell
```

Because `trackRef` is a reference object containing an object specifier, the specifier identifies the new track when the current track changes.

`contains`, `is contained by`

---

The `contains` and `is contained by` operators work with lists, records, and text objects.

Table 9-1 (page 179) summarizes the use of these operators and their synonyms.

#### list

A list `contains` another list if the right-hand list is a sublist of the left-hand list. 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, the following statement is `true` because `1 + 1` evaluates to `2`, so that all the items in the right-hand list appear, in the same order, in the left-hand list:

```
{ "this", "is", 1 + 1, "cool" } contains { "is", 2 }
```

The following statement is `false` because the items in the right-hand list are not in the same order as the matching items in the left-hand list:

```
{ "this", "is", 2, "cool" } contains { 2, "is" }
```

A list `is contained by` another list if the left-hand list is a sublist of the right-hand list. 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—as with the concatenation operator (`&`), single values are coerced to one-item lists. For example, both of the following expressions evaluate to `true`:

```
{ "this", "is", 2, "cool" } contains 2
2 is contained by { "this", "is", 2, "cool" }
```

However, the following expressions, containing nested lists, both evaluate to `false`:

```
{"this", "is", {2}, "cool"} contains 2 -- false
{"this", "is", {2}, "cool"} contains {2} -- false
```

#### record

A record contains another record if all the properties in the right-hand record are included in the left-hand record, and the values of properties in the right-hand record are equal to the values of the corresponding properties in the left-hand record. A record is contained by another record if all the properties in the left-hand

record are included in the right-hand record, and the values of the properties in the left-hand record are equal to the values of the corresponding properties in the right-hand record. The order in which the properties appear does not matter. For example, the following is `true`:

```
{ name:"Matt", mileage:"8000", description:"fast" } ~
  contains { description:"fast", name:"Matt" }
```

### text

A `text` object contains another `text` object if the characters in the right-hand `text` object are equal to any contiguous series of characters in the left-hand `text` object. For example,

```
"operand" contains "era"
```

is `true`, but

```
"operand" contains "dna"
```

is `false`.

A `text` object is contained by another `text` object if the characters in the left-hand `text` object are equal to any series of characters in the right-hand `text` object. For example, this statement is `true`:

```
"era" is contained by "operand"
```

Text comparisons can be affected by considering and ignoring statements, as described in the Text section of [equal, is not equal to](#) (page 190).

### equal, is not equal to

---

The `equal` and `is not equal to` operators can handle operands of any class. Two expressions of different classes are generally not equal, although for scalar operands, such as booleans, integers, and reals, two operands are the same if they have the same value.

[Table 9-1](#) (page 179) summarizes the use of these operators and their synonyms.

### list

Two lists are equal if they both contain the same number of items and if the value of an item in one list is identical to the value of the item at the corresponding position in the other list:

```
{ 7, 23, "Hello" } = {7, 23, "Goodbye"} --result: false
```

### record

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" }
```

**text**

Two `text` objects are equal if they are both the same series of characters. They are not equal if they are different series of characters. For related information, see the [text](#) (page 97) class.

Text comparisons can be affected by `considering` and `ignoring` statements, which instruct AppleScript to selectively consider or ignore attributes of characters or types of characters. For example, unless you use an `ignoring` statement, AppleScript compares `text` objects by considering all characters and punctuation.

AppleScript does not distinguish uppercase from lowercase letters unless you use a `considering` statement to consider the `case` attribute. For example:

```
"DUMPtruck" is equal to "dumptruck" --result: true
considering case
  "DUMPtruck" is equal to "dumptruck" --result: false
end considering
```

When comparing two `text` objects, if the test is not enclosed in a `considering` or `ignoring` statement, then the comparison uses default values for `considering` and `ignoring` attributes (described in [considering / ignoring \(text comparison\)](#) (page 193)).

**greater than, less than**


---

The `greater than` and `less than` operators work with dates, integers, real numbers, and `text` objects.

[Table 9-1](#) (page 179) summarizes the use of these operators and their synonyms.

**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, real**

An integer or a real number is greater than another integer or real number if it represents a larger number. It is less than another integer or real number if it represents a smaller number.

**text**

To determine the ordering of two `text` objects, AppleScript uses the collation order set in the Language pane of International preferences. A `text` object is greater than (comes after) another `text` object based on the lexicographic ordering of the user's language preference. With the preference set to English, the following two statements both evaluate to `true`:

```
"zebra" comes after "aardvark"
"zebra" > "aardvark"
```

The following two statements also evaluate to `true`:

```
"aardvark" comes before "zebra"
"aardvark" < "zebra"
```

Text comparisons can be affected by `considering` and `ignoring` statements, as described in the Text section of [equal, is not equal to](#) (page 190).

## starts with, ends with

---

The `starts with` and `ends with` operators work with lists and text objects.

[Table 9-1](#) (page 179) summarizes the use of these operators and their synonyms.

### list

A list `starts with` the items in a second list if all the items in the second list are found at the beginning of the first list. A list `ends with` the items in a second list if all the items in the second list are found at the end of the first list. 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" }
```

### text

A text object `starts with` the text in a second text object if all the characters in the second object are found at the beginning of the first object. A text object `ends with` the text in a second text object if all the characters in the second object are found at the end of the first object. For example, the following expression is `true`:

```
"operand" starts with "opera"
```

A text object `ends with` another text object if the characters in the right-hand text object are the same as the characters at the end of the left-hand text object. For example, the following expression is `true`:

```
"operand" ends with "and"
```

Text comparisons can be affected by `considering` and `ignoring` statements, as described in the Text section of [`equal`, `is not equal to`](#) (page 190).



# Control Statements Reference

---

This chapter describes AppleScript control statements. A **control statement** is a statement that determines when and how other statements are executed or how expressions are evaluated. For example, a control statement may cause AppleScript to skip or repeat certain statements.

**Simple statements** can be written on one line, while **compound statements** can contain other statements, including multiple clauses with nested and multi-line statements. A compound statement is known as a **statement block**.

Compound statements begin with one or more reserved words, such as `tell`, that identify the type of control statement. The last line of a compound statement always starts with `end`, and can optionally include the word that begins the control statement (such as `end tell`).

## considering and ignoring Statements

---

The `considering` and `ignoring` statements cause AppleScript to consider or ignore specific characteristics as it executes groups of statements. There are two kinds of `considering` and `ignoring` statements:

- Those that specify attributes to be considered or ignored in performing text comparisons.
- Those that specify whether AppleScript should consider or ignore responses from an application.

### `considering / ignoring (text comparison)`

---

Specify how AppleScript should treat attributes, such as case, in performing text comparisons.

#### 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

### Placeholders

#### *attribute*

A characteristic of the text:

#### case

If this attribute is ignored, uppercase letters are not distinguished from lowercase letters. See Special Considerations below for related information. See also [greater than](#), [less than](#) (page 191) for a description of how AppleScript sorts letters, punctuation, and other symbols.

#### diacriticals

If this attribute is ignored, text objects are compared as if no diacritical marks (such as `´`, `ˆ`, `˜`, and `¨`) are present; for example, "résumé" is equal to "resume".

#### hyphens

If this attribute is ignored, text objects are compared as if no hyphens are present; for example "anti-war" is equal to "antiwar".

#### numeric strings

By default, this attribute is ignored, and text strings are compared according to their character values. For example, if this attribute is considered, "1.10.1" > "1.9.4" evaluates as true; otherwise it evaluates as false. This can be useful in comparing version strings.

#### punctuation

If this attribute is ignored, text objects are compared as if no punctuation marks (such as `.`, `?`, `:`, `;`, `!`, `'`, `"`, ```) are present; for example "What? he inquired." is equal to "what he inquired".

#### white space

If this attribute is ignored, the text objects are compared as if spaces, tab characters, and return characters were not present; for example "Brick house" would be considered equal to "Brickhouse".

#### *Default Value:*

Case and numeric strings are ignored; all others are considered.

#### *statement*

Any AppleScript statement.

### Examples

The following examples show how considering and ignoring statements for various attributes can change the value of text comparisons.

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

```
"BOB" = "bob" --result: true
considering case
  "BOB" = "bob" --result: false
end considering
```

```

"a" = "á" --result: false
ignoring diacriticals
  "a" = "á" --result: true
end considering

"Babs" = "bábs" --result: false

ignoring case
  "Babs" = "bábs" --result: false
end ignoring

ignoring case and diacriticals
  "Babs" = "bábs" --result: true
end ignoring

```

**Discussion**

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. 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.

**Special Considerations**

Because `text item delimiters` (described in “[version](#)” (page 39)) respect `considering` and `ignoring` attributes in AppleScript 2.0, delimiters are case-insensitive by default. Formerly, they were always case-sensitive. To enforce the previous behavior, add an explicit `considering case` statement.

## considering / ignoring (application responses)

---

Permits a script to continue without waiting for an application to respond to commands that target it.

**Syntax**

```

considering | ignoring application responses
  [ statement ]...
end[considering | ignoring]

```

**Placeholders**

*statement*  
Any AppleScript statement.

**Examples**

The following example shows how to use an `ignoring` statement so that a script needn't wait while Finder is performing a potentially lengthy task:

```

tell application "Finder"
  ignoring application responses
    empty the trash
  end ignoring
end tell

```

Your script may want to ignore most responses from an application, but wait for a response to a particular statement. You can do so by nesting `considering` and `ignoring` statements:

```

tell application "Finder"
  ignoring application responses

```

```

empty the trash
-- other statements that ignore application responses
considering application responses
    set itemName to name of first item of startup disk
end considering
-- other statements that ignore application responses
end ignoring
end tell

```

**Discussion**

A response to an application command indicates whether the command completed successfully, and also returns results and error messages, if there are any. When you use an `ignoring application responses` block, you forego this information.

Results and error messages from AppleScript commands, scripting additions, and expressions are not affected by the `application responses` attribute.

## error Statements

---

During script execution, errors can occur in the operating system (for example, when a specified file isn't found), in an application (for example, when the script specifies an object that doesn't exist), and in the script itself. An **error message** is a message that is supplied by an application, AppleScript, or Mac OS X 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 `text` object, that describes the error; and other information.

A script can signal an error—which can then be handled by an error handler—with the `error` statement. This allows scripts to supply their own messages for errors that occur within the script. For example, a script can prepare to handle anticipated errors by using a `try` (page 207) statement. In the `on error` branch of a `try` statement, a script may be able to recover gracefully from the error. If not, it can use an `error` statement to resignal the error message it receives, modifying the message as needed to supply information specific to the script.

---

`error`

---

Signals an error in a script.

**Syntax**

```

error [ errorMessage ] [ number errorNumber ] ↵
    [ partial result resultList ] ↵
    [ from offendingObject ] [ to expectedType ]

```

**Placeholders***errorMessage*

A `text` object describing the error. Although this parameter is optional, you should provide descriptions for errors wherever possible. If you do not include an error description, an empty `text` object ("" ) is passed to the error handler.

*errorNumber*

The error number for the error. This is an optional parameter. If you do not include a number parameter, the value -2700 (unknown error) is passed to the error handler.

If the error you are signaling is a close match for one that already has an AppleScript error constant, you can use that constant. If you need to create a new number for the error, avoid using one that conflicts with error numbers defined by AppleScript, Mac OS X, and the Apple Event Manager. In general, you should use positive numbers from 500 to 10,000. For more information, see [“Error Numbers and Error Messages”](#) (page 233).

*resultList*

A list of objects. 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. This is rarely supported by applications.

*offendingObject*

A reference to the object, if any, that caused the error.

*expectedType*

A class. 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 a `try` (page 207) statement to handle a simple error, and demonstrates how you can use an `error` statement to catch an error, then resignal the error exactly as it was received, causing AppleScript to display an error dialog (and halt execution):

```
try
    word 5 of "one two three"
on error eStr number eNum partial result rList from badObj to expectedType
    -- statements that take action based on the error
    display dialog "Doing some preliminary handling..."
    -- then resignal the error
    error eStr number eNum partial result rList from badObj to expectedType
end try
```

In the next example, an `error` statement resignals an error, but omits any original error information and supplies its own message to appear in the error dialog:

```
try
    word 5 of "one two three"
on error
    -- statements to execute in case of error
    error "There are not enough words."
end try
```

For more comprehensive examples, see [“Working with Errors”](#) (page 237).

## if Statements

---

An `if` statement allows you to define statements or groups of statements that are executed only in specific circumstances, based on the evaluation of one or more Boolean expressions.

An `if` statement is also called a conditional statement. Boolean expressions in `if` statements are also called tests.

## if (simple)

---

Executes a statement if a Boolean expression evaluates to `true`.

### Syntax

```
if boolean then statement
```

### Placeholders

*boolean*

A Boolean expression.

*statement*

Any AppleScript statement.

### Examples

This script displays a dialog if the value of the Boolean expression `ageOfCat > 1` is `true`. (The variable `ageOfCat` is set previously.)

```
if ageOfCat > 1 then display dialog "This is not a kitten."
```

## if (compound)

---

Executes a group (or groups) of statements if a Boolean expression (or expressions) evaluates to `true`.

### Syntax

```
if boolean [ then ]  
  [ statement ]...
```

```
[ else if boolean [ then ]  
  [ statement ]... ]...
```

```
[ else  
  [ statement ]... ]
```

```
end [ if ]
```

### Placeholders

*boolean*

A Boolean expression.

*statement*

Any AppleScript statement.

### Examples

The following example uses a compound `if` statement, with a final `else` clause, to display a statement based on the current temperature (obtained separately):

```

if currentTemp < 60 then
    set response to "It's a little chilly today."
else if currentTemp > 80 then
    set response to "It's getting hotter today."
else
    set response to "It's a nice day today."
end if
display dialog response

```

**Discussion**

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.

**repeat Statements**

---

You use a `repeat` statement to create loops or execute groups of repeated statements in scripts.

There are a number of types of `repeat` statement, each differing in the way it terminates the loop. Each of the options, from repeating a loop a specific number of times, to looping over the items in a list, to looping until a condition is met, and so on, lends itself to particular kinds of tasks.

For information on testing and debugging `repeat` statements, see [“Debugging AppleScript Scripts”](#) (page 45).

**exit**

---

Terminates a `repeat` loop and resumes execution with the statement that follows the `repeat` statement.

You can only use an `exit` statement inside a `repeat` statement. Though most commonly used with the `repeat (forever)` form, you can also use an `exit` statement with other types of `repeat` statement.

**Syntax**

```
exit [repeat]
```

**Examples**

See the example in [`repeat \(forever\)`](#) (page 199).

```
repeat (forever)
```

---

Repeats a statement (or statements) until an `exit` statement is encountered.

**Important:** A `repeat (forever)` statement will never complete unless you cause it to do so.

To terminate a `repeat (forever)` statement, you can:

- Use an `exit` (page 199) statement and design the logic so that it eventually encounters the `exit` statement.
- Use a “`return`” (page 214) statement, which exits the handler or script that contains the loop, and therefore the loop as well.
- Use a `try` (page 207) statement and rely on an error condition to exit the loop.

**Syntax**

```
repeat
```

```
    [ statement ]...
```

```
end [ repeat ]
```

**Placeholders**

*statement*

Any AppleScript statement.

**Examples**

This form of the `repeat` statement is similar to the `repeat until` (page 201) form, except that instead of putting a test in the `repeat` statement itself, you determine within the loop when it is time to exit. You might use this form, for example, to wait for a lengthy or indeterminate operation to complete:

```
repeat
    -- perform operations
    if someBooleanTest then
        exit repeat
    end if
end repeat
```

In a script application that stays open, you can use an `idle` handler to perform periodic tasks, such as checking for an operation to complete. See “[idle Handlers](#)” (page 76) for more information.

```
repeat (number) times
```

---

Repeats a statement (or statements) a specified number of times.

**Syntax**

```
repeat integer [ times ]
```

```
    [ statement ]...
```



end [ repeat ]

#### Placeholders

##### *integer*

Specifies the number of times to repeat the statements in the body of the loop. Instead of an integer, you can specify any value that can be coerced to an integer. If the value is less than one, the body of the `repeat` statement is not executed.

##### *statement*

Any AppleScript statement.

#### Examples

The following handler uses the `repeat (number) times` form of the `repeat` statement to raise a passed number to the passed power:

```
on raiseToTheNth(x, power)
    set returnVal to x
    repeat power - 1 times
        set returnVal to returnVal * x
    end repeat
    return returnVal
end raiseToTheNth
```

repeat until

---

Repeats a statement (or statements) until a condition is met. Tests the condition before executing any statements.

#### Syntax

repeat until *boolean*

[ *statement* ]...

end [ repeat ]

#### Placeholders

##### *boolean*

A Boolean expression. If it has the value `true` when entering the loop, the statements in the loop are not executed.

##### *statement*

Any AppleScript statement.

#### Examples

The following example uses the `repeat until` form of the `repeat` statement to allow a user to enter database records. The handler `enterDataRecord()`, which is not shown, returns `true` if the user is done entering records:

```
set userDone to false
repeat until userDone
    set userDone to enterDataRecord()
```

```
end repeat
```

```
repeat while
```

---

Repeats a statement (or statements) as long as a condition is met. Tests the condition before executing any statements. Similar to the `repeat until` form, except that it continues *while* a condition is `true`, instead of *until* it is `true`.

**Syntax**

```
repeat while boolean
```

```
  [ statement ]...
```

```
end [ repeat ]
```

**Placeholders***boolean*

A Boolean expression. If it has the value `false` when entering the loop, the statements in the loop are not executed.

*statement*

Any AppleScript statement.

**Examples**

The following example uses the `repeat while` form of the `repeat` statement to allow a user to enter database records. In this case, we've just reversed the logic shown in the `repeat until` (page 201) example. Here, the handler `enterDataRecord()`, which is not shown, returns `true` if the user is *not* done entering records:

```
set userNotDone to true
repeat while userNotDone
    set userNotDone to enterDataRecord()
end repeat
```

```
repeat with loopVariable (from startValue to stopValue)
```

---

Repeats a statement (or statements) until the value of the controlling loop variable exceeds the value of the predefined stop value.

**Syntax**

```
repeat with loopVariable from startValue to stopValue [by stepValue]
```

```
  [ statement ]...
```

end [ repeat ]

#### Placeholders

##### *loopVariable*

Controls the number of iterations. It can be a previously defined variable or a new variable you define in the `repeat` statement.

##### *startValue*

Specifies a value that is assigned to *loopVariable* when the loop is entered.

You can specify an integer or any value that can be coerced to an integer.

##### *stopValue*

Specifies an value. When that value is exceeded by the value of *loopVariable*, iteration ends. If *stopValue* is less than *startValue*, the body is not executed.

You can specify an integer or any value that can be coerced to an integer.

##### *stepValue*

Specifies a value that is added to *loopVariable* after each iteration of the loop. You can assign an integer or a real value; a real value is rounded to an integer.

*Default Value:*

1

##### *statement*

Any AppleScript statement.

#### Examples

The following handler uses the `repeat with loopVariable` (from *startValue* to *stopValue*) form of the `repeat` statement to compute a factorial value (the factorial of a number is the product of all the positive integers from 1 to that number):

```
on factorial(x)
    set returnVal to 1
    repeat with n from 2 to x
        set returnVal to returnVal * n
    end repeat
    return returnVal
end factorial
```

#### Discussion

You can use an existing variable as the loop variable in a `repeat with loopVariable` (from *startValue* to *stopValue*) statement or define a new one in the 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. As a result, if you change the values in the body of the loop, it doesn't change the execution of the loop.

---

`repeat with loopVariable (in list)`

Loops through the items in a specified 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 a reference to the first item in *list*, in the second iteration, it is a reference to the second item in *list*, and so on.

### Syntax

```
repeat with loopVariable in list
```

```
    [ statement ]...
```

```
end [ repeat ]
```

### Placeholders

*loopVariable*

Any previously defined variable or a new variable you define in the `repeat` statement (see Discussion).

*list*

A list or a object specifier (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 Discussion).

*statement*

Any AppleScript statement.

### Examples

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. Note that within the loop, the loop variable (`currentWord`) is a reference to an item in a list, so in the test statement (`if contents of currentWord is equal to "hammer" then`) it must be cast to text (as `text`).

```
set wordList to words in "Where is the hammer?"
repeat with currentWord in wordList
    log currentWord
    if contents of currentWord is equal to "hammer" then
        display dialog "I found the hammer!"
    end if
end repeat
```

The statement `log currentWord` logs the current list item to Script Editor’s log window. For more information, see [“Debugging AppleScript Scripts”](#) (page 45).

### Discussion

You can use an existing variable as the loop variable in a `repeat with loopVariable (in list)` statement or define a new one in the `repeat with... 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 an object specifier that takes on the value of 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 listItem to i
end repeat
```

```
--result: item 4 of {1, 2, 3, 4} --result: an object specifier
```

To set a variable to the value of an item in the list, rather than a reference to the item, use the `contents of property`:

```
repeat with i in {1, 2, 3, 4}
    set listItem to contents of i
end repeat
--result: 4
```

You can also use the list items directly in expressions:

```
set total to 0
repeat with i in {1, 2, 3, 4}
    set total to total + 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}`.

## tell Statements

---

A `tell` statement specifies the default target—that is, the object to which commands are sent if they do not include a direct parameter. Statements within a `tell` statement that use terminology from the targeted object are compiled against that object’s dictionary.

The object of a `tell` statement is typically a reference to an application object or a script object. For example, the following `tell` statement targets the Finder application:

```
tell application "Finder"
    set frontWindowName to name of front window
    -- any number of additional statements can appear here
end tell
```

You can nest `tell` statements inside other `tell` statements, as long as you follow the syntax and rules described in [tell \(compound\)](#) (page 206).

When you need to call a handler from within a `tell` statement, there are special terms you use to indicate that the handler is part of the script and not a command that should be sent to the object of the `tell` statement. These terms are described in [“The it and me Keywords”](#) (page 40) and in [“Calling Handlers in a tell Statement”](#) (page 72).

A `tell` statement that targets a local application doesn’t cause it to launch, if it is not already running. For example, a script can examine the `is running` property of the targeted `application` (page 80) object to determine if the application is running before attempting to send it any commands. If it is not running it won’t be launched.

If a `tell` statement targets a local application and executes any statements that require a response from the application, then AppleScript will launch the application if it is not already running. The application is launched as hidden, but the script can send it an `activate` (page 108) command to bring it to the front, if needed.

A `tell` statement that targets a remote application will not cause it to launch—in fact, it will not compile or run unless the application is already running. Nor is it possible to access the `is running` property of an application on a remote computer.

## tell (simple)

---

Specifies a target object and a command to send to it.

### Syntax

```
tell referenceToObject to statement
```

### Placeholders

*referenceToObject*

Any object. Typically an object specifier or a `reference object` (which contains an object specifier).

*statement*

Any AppleScript statement.

### Examples

This simple `tell` statement closes the front Finder window:

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

For more information on how to specify an application object, see the [application](#) (page 80) class.

## tell (compound)

---

Specifies a target object and one or more commands to send to it. A compound `tell` statement is different from a simple `tell` statement in that it always includes an `end` statement.

### Syntax

```
tell referenceToObject
```

```
  [ statement ]...
```

```
end [tell]
```

### Placeholders

*referenceToObject*

Any object. Typically an object specifier or a `reference object` (which contains an object specifier).

*statement*

Any AppleScript statement, including another `tell` statement.

### Examples

The following statements show how to close a window using first a compound `tell` statement, then with two variations of a simple `tell` statement:

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

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

The following example shows a nested `tell` statement:

```
tell application "Finder"
    tell document 1 of application "TextEdit"
        set newName to word 1 -- handled by TextEdit
    end tell
    set len to count characters in newName -- handled by AppleScript
    if (len > 2) and (len < 15) then -- comparisons handled by AppleScript
        set name of first item of disk "HD" to newName -- handled by Finder
    end if
end tell
```

This example works because in each case the terminology understood by a particular application is used within a `tell` block targeting that application. However, it would not compile if you asked the Finder for word 1 of a document, or told TextEdit to `set name` of the first item on a disk, because those applications do not support those terms.

## try Statements

---

A `try` statement provides the means for scripts to handle potential errors. It attempts to execute one or more statements and, if an error occurs, executes a separate set of statements to deal with the error condition. If an error occurs and there is no `try` statement in the calling chain to handle it, AppleScript displays an error and script execution stops.

For related information, see [“error Statements”](#) (page 196) and [“AppleScript Error Handling”](#) (page 36).

`try`

---

Attempts to execute a list of AppleScript statements, calling an error handler if any of the statements results in an error.

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. After the error handler completes, control passes to the statement immediately following the end of the `try` statement.

### Syntax

`try`

`[ statement ]...`

`[ on error [ errorMessage ] [ number errorNumber ] [ from offendingObject ] ↵`

`[ partial result resultList ] [ to expectedType ]`

`[ statement ]...`

`end [ error | try ]`

### Placeholders

*statement*

Any AppleScript statement.

*errorMessage*

A `text` object, that describes the error.

*errorNumber*

The error number, an integer. For possible values, see [“Error Numbers and Error Messages”](#) (page 233).

*offendingObject*

A reference to the object, if any, that caused the error.

*resultList*

A list that provides partial results for objects that were handled before the error occurred. The list can contain values of any class. This parameter applies only to commands that return results for multiple objects. This is rarely supported by applications.

*expectedType*

The expected class. If the error was caused by a coercion failure, the value of this variable is the class of the coercion that failed. (The second example below shows how this works in a case where AppleScript is unable to coerce a `text` object into an `integer`.)

*variable*

Either a global variable or a local variable that can be used in the handler. A variable can contain any class of value. The scope of a local variable is the handler. The scope of a global variable extends to any other part of the script, including other handlers and `script` objects. For related information about local and global variables, see [“version”](#) (page 39).

**Examples**

The following example shows how you can use a `try` statement to handle the “Cancel” button for a `display alert` (page 124) command. Canceling returns an error number of -128, but is not really an error. This test handler just displays a dialog to indicate when the user cancels or when some other error occurs.

```
try
    display alert "Hello" buttons {"Cancel", "Yes", "No"} cancel button 1
on error errText number errNum
    if (errNum is equal to -128) then
        -- User cancelled.
        display dialog "User cancelled."
    else
        display dialog "Some other error: " & errNum & return & errText
    end if
end try
```

You can also use a simplified version of the `try` statement that checks for just a single error number. In the following example, only error -128 is handled. Any other error number is ignored by this `try` statement, but is automatically passed up the calling chain, where it may be handled by other `try` statements.

```
try
    display alert "Hello" buttons {"Cancel", "Yes", "No"} cancel button 1
on error number -128
    -- Either do something special to handle Cancel, or just ignore it.
end try
```

The following 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"
        -- do something that depends on variable "i"
```



```

    end repeat
on error from obj to newClass
    log {obj, newClass} -- Display from and to info in log window.
end try

```

This `repeat` statement fails because the text object "Toronto" cannot be coerced to an `integer` (page 89). The error handler simply writes the values of `obj` (the offending value, "Toronto") and `newClass` (the class of the coercion that failed, `integer`) to Script Editor's Event Log History window (and to the script window's Event Log pane). The result is "(\*Toronto, integer\*);", indicating the error occurred while trying to coerce "Toronto" to an integer.

For additional examples, see ["Working with Errors"](#) (page 237).

## using terms from Statements

---

A `using terms from statement` lets you specify which terminology AppleScript should use in compiling the statements in a script. Whereas a `tell` statement specifies the default target (often an application) to which commands are sent *and* the terminology to use, a `using terms from statement` specifies only the terminology.

A `using terms from statement` can be useful in writing application event handler scripts, such as Mail rules.

Another use for this type of statement is with a script that targets an application on a remote computer that may not be available when you compile the script (or the application may not be running). Or, you might be developing locally and only want to test with the remote application at a later time. In either case, you can use a `using terms from statement` to specify a local application (presumably with a terminology that matches the one on the remote computer) to compile against.

Even if a statement contained within a `using terms from statement` compiles, the script may fail when run because the target application's terminology may differ from that used in compiling.

You can nest `using terms from statements`. When you do so, each script statement is compiled against the terminology of the application named in the innermost enclosing `using terms from statement`.

### using terms from

---

Instructs AppleScript to use the terminology from the specified application in compiling the enclosed statements.

#### Syntax

```
using terms from application
```

```
    [ statement ]...
```

```
end[using terms from]
```

#### Placeholders

##### *application*

A specifier for an application object.

##### *statement*

Any AppleScript statement.

**Examples**

The following example shows how to use a `using terms from` statement in writing a Mail rule action script. These scripts take the following form:

```
using terms from application "Mail"
  on perform mail action with messages theMessages for rule theRule
    tell application "Mail"
      -- statements to process each message in theMessages
    end tell
  end perform mail action with messages
end using terms from
```

To use the script, you open Preferences for the Mail application, create or edit a rule, and assign the script as the action for the rule.

For an example that works with an application on a remote machine, see [“Targeting Remote Applications”](#) (page 44).

**with timeout Statements**

You can use a `with timeout` statement to control how long AppleScript waits for a command to execute before timing out. By default, when an application fails to respond to a command, AppleScript waits for two minutes before reporting an error and halting execution.

**with timeout**

Specifies how long AppleScript waits for a response to a command that is sent to another application.

**Syntax**

```
with timeout [ of ] integerExpression second[s]
  [ statement ]...
end [ timeout ]
```

**Placeholders***integerExpression*

The amount of time, in seconds, AppleScript should wait before timing out (and interrupting the command).

*statement*

Any AppleScript statement.

**Examples**

The following script tells TextEdit to close its first document; if the document has been modified, it asks the user if the document should be saved. It includes the statement `with timeout of 20 seconds`, so that if the user doesn't complete the `close` operation within 20 seconds, the operation times out.

```
tell application "TextEdit"
  with timeout of 20 seconds
    close document 1 saving ask
  end timeout
end tell
```

**Discussion**

When a command fails to complete in the allotted time (whether the default of two minutes, or a time set by a `with timeout` statement), 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. If you want the script to continue, you can wrap the statements in a `try` (page 207) statement. However, whether your script can send a command to cancel an offending lengthy operation after a timeout is dependent on the application that is performing the command.

A `with timeout` statement applies only to commands sent to application objects, not to commands sent to the application that is running the script.

In some situations, you may want to use an `ignoring application responses` statement (instead of a `with timeout` statement) so that your script needn't wait for application commands to complete. For more information, see "[considering and ignoring Statements](#)" (page 193).

**with transaction Statements**

---

When you execute a script, AppleScript may send one or more Apple events to targeted applications. A transaction is a set of operations that are applied as a single unit—either all of the changes are applied or none are. This mechanism works only with applications that support it.

**with transaction**

---

Associates a single transaction ID with any events sent to a target application as a result of executing commands in the body of the statement.

**Syntax**

```
with transaction [ session ]
```

```
    [ statement ]...
```

```
end [ transaction ]
```

**Placeholders**

*session*

An object that identifies a specific session.

*statement*

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. (In the following examples, "Small DB" and "Super DB" are representative database applications.)

```
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.

A `with transaction` statement works only with applications that explicitly support it. 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
```

# Handler Reference

---

This chapter provides reference for handlers, which are defined and introduced in “[About Handlers](#)” (page 67). It describes the types of parameters you can use with handlers and how you invoke them. It also describes the `continue` and `return` statements, which you use to control the flow of execution in handlers.

## continue

---

A `continue` statement causes AppleScript to invoke the handler with the same name in the parent of the current handler. If there is no such handler in the parent, AppleScript looks up the parent chain, ending with the current application.

A `continue` statement is like a handler call, in that after execution completes in the new location, it resumes with the statement after the `continue` statement.

### Syntax

```
continue handlerName [ parameterList ]
```

### Placeholders

#### *handlerName*

A required identifier that specifies the name of the current handler (which is also the name of the handler in which to continue execution).

#### *parameterList*

The list of parameters to be passed to *handlerName*.

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 the parameter variables that were specified in the original command (and thus the original values) or you can list values that may differ from those of the original variables.

### Examples

You can write a handler that overrides an AppleScript command but uses a `continue` statement to pass control on to the AppleScript command if desired:

```
on beep numTimes
    set x to display dialog "Start beeping?" buttons {"Yes", "No"}
    if button returned of x is "Yes" then ↵
        continue beep numTimes -- Let AppleScript handle the beep.
        -- In this example, nothing to do after returning from the continue.
    end beep
```

```
beep 3 --result: local beep handler invoked; shows dialog before beeping
tell my parent to beep 3 -- result: AppleScript beep command invoked
```

When AppleScript encounters the statement `beep 3`, it invokes the local `beep` handler, which displays a dialog. If the user clicks Yes, the handler uses a `continue` statement to pass the `beep` command to the script's parent (AppleScript), which handles the command by beeping. If the user clicks No, it does not continue the `beep` command, and no sound is heard.

The final statement, `tell my parent to beep 3`, shows how to directly invoke the AppleScript `beep` command, rather than the local handler.

For an example that uses a `continue` statement to exit a script handler and return control to the application's default `quit` handler, see [“quit Handlers”](#) (page 76).

For additional examples, see [“Using the continue Statement in Script Objects”](#) (page 63).

## return

---

A `return` statement exits a handler and optionally returns a specified value. Execution continues at the place in the script where the handler was called.

### Syntax

```
return [ expression ]
```

### Placeholders

*expression*

Represents the value to return.

### Examples

The following statement, inserted in the body of a handler, returns the integer 2:

```
return 2 -- returns integer value 2
```

If you include a `return` statement without an expression, AppleScript exits the handler immediately and no value is returned:

```
return -- no value returned
```

See other sections throughout [“Handler Reference”](#) (page 213) for more examples of scripts that use the `return` statement.

### Discussion

If a handler does not include a `return` statement, AppleScript returns the value returned by the last statement. If the last statement doesn't return a value, AppleScript returns nothing.

When AppleScript has finished executing a handler (that is, when it executes a `return` statement or the last statement in the handler), it passes control to the place in the script immediately after the place where the handler was called. If a handler call is part of an expression, AppleScript uses the value returned by the handler to evaluate the expression.

It is often considered good programming practice to have just one `return` statement and locate it at the end of a handler. Doing so can provide 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` handler in “[Handler Syntax \(Positional Parameters\)](#)” (page 218) is a simple script that uses two `return` statements.

For related information, see “[AppleScript Error Handling](#)” (page 36).

## Handler Syntax (Labeled Parameters)

---

A handler is a collection of statements that can be invoked by name. This section describes the syntax for handlers that use labeled parameters.

Labeled parameters are identified by their labels and can be listed in any order.

### Syntax

```
( on | to ) handlerName ↵
  [ [ of | in ] directParamName ] ↵
  [ ASLabel userParamName ]... ↵
  [ given userLabel : userParamName [, userLabel : userParamName ]... ]
  [ statement ]...
end [ handlerName ]
```

### Placeholders

#### *handlerName*

An identifier that names the handler.

#### *directParamName*

An identifier for the direct parameter variable. If it is included, *directParamName* must be listed immediately after the command name. The word `of` or `in` before *directParamName* is required in user-defined handlers, but is optional in terminology-defined handlers (for example, those defined by applications).

If a user-defined handler includes a direct parameter, the handler must also include at least one variable parameter.

#### *ASLabel*

An AppleScript-defined label. The available labels are: `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 are the only labels that can be used without the special label `given`. Each label must be unique among the labels for the handler (that is, you cannot use the same label for more than one parameter).

#### *userLabel*

An identifier for a user-defined label, associated with a user-defined parameter. Each label must be unique.

The first *userLabel*-*userParamName* pair must follow the word `given`; any additional pairs are separated by commas.

#### *userParamName*

An identifier for a parameter variable.

*statement*

Any AppleScript statement. These statements can include definitions of `script` objects, each of which, like any `script` object, can contain handlers and other `script` objects. However, you cannot declare another handler within a handler, except within a `script` object.

Handlers often contain a “[return](#)” (page 214) statement.

**Examples**

For examples and related conceptual information, see “[Handlers with Labeled Parameters](#)” (page 68).

**Discussion**

A handler written to respond to an application command (like those in “[Handlers in Script Applications](#)” (page 73)) need not include all of the possible parameters defined for that command. For example, an application might define a command with up to five possible parameters, but you could define a handler for that command with only two of the parameters.

If a script calls a handler with more parameters than are specified in the handler definition, the extra parameters are ignored.

## Calling a Handler with Labeled Parameters

---

This section describes the syntax for calling a handler with labeled parameters.

**Syntax**

*handlerName* ↪

[ [ *of* | *in* ] *directParam* ] ↪

[ [ *ASLabel paramValue* ... ] ↪

[ [ *with labelForTrueParam* [, *labelForTrueParam* ]... ↪

[ ( *and* | , ) *labelForTrueParam* ] ] ↪

[ [ *without labelForFalseParam* [, *labelForFalseParam* ]... ] ↪

[ ( *and* | , ) *labelForFalseParam* ] ] ↪

[ [ *given userLabel:paramValue* [, *userLabel:paramValue* ]... ]... ]

**Placeholders**

*handlerName*

An identifier that names the handler.

*directParam*

Any valid expression. The expression for the direct parameter must be listed first if it is included at all.



*ASLabel*

One of the following AppleScript-defined labels used in the definition of the handler: 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.

*paramValue*

The value of a parameter, which can be any valid expression.

*labelForTrueParam*

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, see the `findNumbers` handler in “[Handlers with Labeled Parameters](#)” (page 68).

*labelForFalseParam*

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.

*paramLabel*

Any parameter label used in the definition of the handler that is not among the labels for *ASLabel*. You must use the special label `given` to specify these parameters. For an example, see the `findNumbers` handler below.

**Examples**

For examples, see “[Handlers with Labeled Parameters](#)” (page 68).

**Discussion**

When you call a handler with labeled parameters, you supply the following:

1. The handler name.
2. A value for the direct parameter, if the handler has one. It must directly follow the handler name.
3. One label-value pair for each AppleScript-defined label and parameter defined for the handler.
4. One label-value pair for each user-defined label and parameter defined for the handler that *is not* a boolean value.

The first pair is preceded by the word `given`; a comma precedes each additional pair. The order of the pairs does not have to match the order in the handler definition.

5. For each user-defined label and parameter defined for the handler that *is* a boolean value, you can either:
  - a. Supply the label, followed by a boolean expression (as with non-boolean parameters); for example:

```
given rounding:true
```

- b. Use a combination of `with` and `without` clauses, as shown in the following examples:

```
with rounding, smoothing and curling
with rounding without smoothing, curling
```

**Note:** AppleScript automatically converts between some forms when you compile. For example, `given rounding:true` is converted to `with rounding`, and `with rounding, smoothing` is converted to `with rounding and smoothing`.

## Handler Syntax (Positional Parameters)

---

A handler is a collection of statements that can be invoked by name. This section describes the syntax for handlers that use positional parameters.

**Important:** The parentheses that surround the parameter list in the following definition are part of the syntax.

### Syntax

```
on | to handlerName ( [ userParamName [, userParamName ]... )
```

```
    [ statement ]...
```

```
end [ handlerName ]
```

### Placeholders

*handlerName*

An identifier that names the handler.

*userParamName*

An identifier for a user-defined parameter variable.

*statement*

Any AppleScript statement, including global or local variable declarations. For information about the scope of local and global variables, see [“Scope of Variables and Properties”](#) (page 51).

### Examples

For examples and related conceptual information, see [“Handlers with Positional Parameters”](#) (page 69).

## Calling a Handler with Positional Parameters

---

A call for a handler with positional parameters must list the parameters in the same order as they are specified in the handler definition.

### Syntax

```
handlerName ( [ paramValue [, paramValue ]... ] )
```

### Placeholders

*handlerName*

An identifier that names the handler.

*paramValue*

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 handler definition.

### Examples

For examples, see [“Handlers with Positional Parameters”](#) (page 69)

### Discussion

When you call a handler with positional parameters, you supply the following:

1. The handler name.
2. An opening and closing parenthesis.
3. If the handler has any parameters, then you also list, within the parentheses, the following:  
One value for each parameter defined for the handler. The value can be any valid expression.



# Folder Actions Reference

---

Folder Actions is a feature of Mac OS X that lets you associate AppleScript scripts with folders. A Folder Action script is executed when the folder to which it is attached is opened or closed, moved or resized, or has items added or removed. The script provides a handler that matches the appropriate format for the action, as described in this chapter.

Folder Actions make it easy to create hot folders that respond to external actions to trigger a workflow. For example, you can use a Folder Action script to initiate automated processing of any photo dropped in a targeted folder. A well written Folder Action script leaves the hot folder empty. This avoids repeated application of the action to the same files, and allows Folder Actions to perform more efficiently.

You can Control-click a folder to access some Folder Action features with the contextual menu in the Finder. Or you can use the Folder Actions Setup application, located in `/Applications/AppleScript`. This application lets you perform tasks such as the following:

- Enable or disable Folder Actions.
- View the folders that currently have associated scripts
- View and edit the script associated with a folder.
- Add folders to or remove folders from the list of folders.
- Associate one or more scripts with a folder.
- Enable or disable all scripts associated with a folder.
- Enable or disable individual scripts associated with a folder.
- Remove scripts associated with a folder.

Folder Actions Setup looks for scripts located in `/Library/Scripts/Folder Action Scripts` and `~/Library/Scripts/Folder Action Scripts`. You can use the sample scripts located in `/Library/Scripts/Folder Action Scripts` or any scripts you have added to these locations, or you can navigate to other scripts.

A Folder Action script provides a handler (see [“Handler Reference”](#) (page 213)) that is invoked when the specified action takes place. When working with Folder Action handlers, keep in mind that:

- You do not invoke Folder Actions directly. Instead, when a triggering action takes place on a folder, the associated handler is invoked automatically.
- When a Folder Action handler is invoked, none of the parameters is optional.
- A Folder Action handler does not return a value.

Here’s how you can use a Folder Action script to perform a specific action whenever an image file is dropped on a specific image folder:

1. Create a script with Script Editor or another script application.

2. In that script, write a handler that conforms to the syntax documented here for the “[adding folder items to](#)” (page 222) folder action. Your handler can use the aliases that are passed to it to access the image files dropped on the folder.
3. Save the script as a compiled script or script bundle.
4. Put a copy of the script in `/Library/Scripts/Folder Action Scripts` or `~/Library/Scripts/Folder Action Scripts`.
5. Use the Folder Actions Setup application, located in `/Applications/AppleScript`, to:
  - a. Enable folder actions for your image folder.
  - b. Add a script to that folder, choosing the script you created.

## adding folder items to

---

A script handler that is invoked after items are added to its associated folder.

### Syntax

```
on adding folder items to alias after receiving listOfAlias
  [statement]...
end[adding folder items to]
```

### Placeholders

#### *alias*

An [alias](#) (page 79) that identifies the folder that received the items.

#### *listOfAlias*

List of aliases that identify the items added to the folder.

#### *statement*

Any AppleScript statement.

### Examples

The following Folder Action handler is triggered when items are added to the folder to which it is attached. It makes an archived copy, in ZIP format, of the individual items added to the attached folder. Archived files are placed in a folder named `Done` within the attached folder.

```
on adding folder items to this_folder after receiving these_items
  tell application "Finder"
    if not (exists folder "Done" of this_folder) then
      make new folder at this_folder with properties {name:"Done"}
    end if
    set the destination_folder to folder "Done" of this_folder as alias
    set the destination_directory to POSIX path of the destination_folder
  end tell
  repeat with i from 1 to number of items in these_items
    set this_item to item i of these_items
    set the item_info to info for this_item
    if this_item is not the destination_folder and ¬
      the name extension of the item_info is not in {"zip", "sit"} then
```

```

        set the item_path to the quoted form of the POSIX path of this_item
        set the destination_path to the quoted form of -
            (destination_directory & (name of the item_info) & ".zip")
        do shell script ("/usr/bin/ditto -c -k -rsrc --keepParent " -
            & item_path & " " & destination_path)
    end if
end repeat
end adding folder items to

```

## closing folder window for

---

A script handler that is invoked after a folder's associated window is closed.

### Syntax

```

on closing folder window for alias
    [ statement ]...
end[closing folder window for]

```

### Placeholders

*alias*

An [alias](#) (page 79) that identifies the folder that was closed.

*statement*

Any AppleScript statement.

### Examples

The following Folder Action handler is triggered when the folder to which it is attached is closed. It closes any open windows of folders within the targeted folder.

```

-- This script is designed for use with Mac OS X v10.2 and later.
on closing folder window for this_folder
    tell application "Finder"
        repeat with EachFolder in (get every folder of folder this_folder)
            try
                close window of EachFolder
            end try
        end repeat
    end tell
end closing folder window for

```

## moving folder window for

---

A script handler that is invoked after a folder's associated window is moved or resized. Not currently available.

### Syntax

```

on moving folder window for alias from bounding rectangle
    [ statement ]...
end[moving folder window for]

```

### Placeholders

*alias*

An [alias](#) (page 79) that identifies the folder that was moved or resized.

You can use this alias to obtain the folder window's new coordinates from the Finder.

*bounding rectangle*

The previous coordinates of the window of the folder that was moved or resized. The coordinates are provided as a list of four numbers, {left, top, right, bottom}; for example, {10, 50, 500, 300} for a window whose origin is near the top left of the screen (but below the menu bar, if present).

*statement*

Any AppleScript statement.

**Examples**

```
on moving folder window for this_folder from original_coordinates
  tell application "Finder"
    set this_name to the name of this_folder
    set the bounds of the container window of this_folder to
      to the original_coordinates
  end tell
  display dialog "Window \"\" & this_name & "\"" has been returned to it's
  original size and position." buttons {"OK"} default button 1
end moving folder window for
```

**Special Considerations**

**Warning:** In Mac OS X v10.5, and possibly in previous OS versions, Folder Actions does not activate attached moving folder window for scripts when the folder is moved.

**opening folder**

A script handler that is invoked when its associated folder is opened in a window.

**Syntax**

```
on opening folder alias
```

```
[ statement ]...
```

```
end [opening folder]
```

**Placeholders***alias*

An *alias* (page 79) that identifies the folder that was opened.

*statement*

Any AppleScript statement.

**Examples**

The following Folder Action handler is triggered when the folder it is attached to is opened. It displays any text from the Spotlight Comments field of the targeted folder. (Prior to Mac OS X v10.4, this script displays text from the Comments field of the specified folder.)

```
-- This script is designed for use with Mac OS X v10.2 and later.
property dialog_timeout : 30 -- set the amount of time before dialogs auto-answer.
```

```
on opening folder this_folder
  tell application "Finder"
    activate
    set the alert_message to the comment of this_folder
    if the alert_message is not "" then
```



```

        display dialog alert_message buttons {"Open Comments", "Clear
Comments", "OK"} default button 3 giving up after dialog_timeout
        set the user_choice to the button returned of the result
        if the user_choice is "Clear Comments" then
            set comment of this_folder to ""
        else if the user_choice is "Open Comments" then
            open information window of this_folder
        end if
    end if
end tell
end opening folder

```

### Special Considerations

Spotlight was introduced in Mac OS X v10.4. In prior versions of the Mac OS, the example script shown above works with the Comments field of the specified folder, rather than the Spotlight Comments field.

## removing folder items from

---

A script handler that is invoked after items have been removed from its associated folder.

### Syntax

on removing folder items from *alias* after losing *listOfAliasOrText*

[ *statement* ]...

end[removing folder items from]

### Placeholders

*alias*

An [alias](#) (page 79) that identifies the folder from which the items were removed.

*listOfAliasOrText*

List of aliases that identify the items lost (removed) from the folder. For permanently deleted items, only the names are provided (as text strings).

*statement*

Any AppleScript statement.

### Examples

The following Folder Action handler is triggered when items are removed from the folder to which it is attached. It displays an alert containing the number of items removed.

```

on removing folder items from this_folder after losing these_items
    tell application "Finder"
        set this_name to the name of this_folder
    end tell
    set the item_count to the count of these_items
    display dialog (item_count as text) & " items have been removed " & "from
folder \"" & this_name & "\"." buttons {"OK"} default button 1
end removing folder items from

```



# AppleScript Keywords

This appendix lists AppleScript keywords (or *reserved words*), provides a brief description for each, and points to related information, where available. (See also “Keywords” (page 18) in “AppleScript Lexical Conventions” (page 17).)

The keywords in [Table A-1](#) (page 227) are part of the AppleScript language. You should not attempt to reuse them in your scripts for variable names or other purposes. Developers should not re-define keywords in the terminology for their scriptable applications. You can view many additional scripting terms defined by Apple, but not part of the AppleScript language, in [AppleScript Terminology and Apple Event Codes](#).

**Table A-1** AppleScript reserved words, with descriptions

about	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
above	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
after	used to describe position in the “ <a href="#">Relative</a> ” (page 176) reference form; used as part of operator (comes after, does not come after) with classes such as <a href="#">date</a> (page 85), <a href="#">integer</a> (page 89), and <a href="#">text</a> (page 97)
against	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
and	logical <i>and</i> operator—see <a href="#">Table 9-1</a> (page 179)
apart from	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
around	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
as	coercion operator—see <a href="#">Table 9-1</a> (page 179)
aside from	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
at	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
back	used with “ <a href="#">Index</a> ” (page 172) and “ <a href="#">Relative</a> ” (page 176) reference forms; in back of is synonymous with after and behind
before	used to describe position in the “ <a href="#">Relative</a> ” (page 176) reference form; used as an operator (comes before, does not come before) with classes such as <a href="#">date</a> (page 85), <a href="#">integer</a> (page 89), and <a href="#">text</a> (page 97); synonymous with in front of
beginning	specifies an insertion location at the beginning of a container—see the boundary specifier descriptions for the “ <a href="#">Range</a> ” (page 175) reference form
behind	synonymous with after and in back of
below	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)

beneath	handler parameter label—see <a href="#">“Handler Syntax (Labeled Parameters)”</a> (page 215)
beside	handler parameter label—see <a href="#">“Handler Syntax (Labeled Parameters)”</a> (page 215)
between	handler parameter label—see <a href="#">“Handler Syntax (Labeled Parameters)”</a> (page 215)
but	used in <a href="#">“considering and ignoring Statements”</a> (page 193)
by	used with binary containment operator <code>contains</code> , <code>is contained by</code> (page 189); also used as handler parameter label—see <a href="#">“Handler Syntax (Labeled Parameters)”</a> (page 215)
considering	a control statement—see <a href="#">“considering and ignoring Statements”</a> (page 193)
contain, contains	binary containment operator—see <code>contains</code> , <code>is contained by</code> (page 189)
continue	changes the flow of execution—see <a href="#">“continue”</a> (page 213)
copy	an AppleScript command—see <a href="#">copy</a> (page 121)
div	division operator—see <a href="#">Table 9-1</a> (page 179)
does	used with operators such as <code>does not equal</code> , <code>does not come before</code> , and <code>does not contain</code> —see <a href="#">Table 9-1</a> (page 179)
eighth	specifies a position in a container—see <a href="#">“Index”</a> (page 172) reference form
else	used with <code>if</code> control statement—see <a href="#">“if Statements”</a> (page 197)
end	marks the end of a script or handler definition, or of a compound statement, such as a <code>tell</code> or <code>repeat</code> statement; also specifies an insertion location at the end of a container—see the boundary specifier descriptions for the <a href="#">“Range”</a> (page 175) reference form
equal, equals	binary comparison operator—see <code>equal</code> , <code>is not equal to</code> (page 190)
error	<a href="#">error</a> (page 196) control statement; also used with <code>try</code> (page 207) statement
every	specifies every object in a container—see <a href="#">“Every”</a> (page 168) reference form
exit	terminates a <code>repeat</code> loop—see <a href="#">exit</a> (page 199)
false	a Boolean literal—see <a href="#">“Boolean”</a> (page 20)
fifth	specifies a position in a container—see <a href="#">“Index”</a> (page 172) reference form
first	specifies a position in a container—see <a href="#">“Index”</a> (page 172) reference form
for	handler parameter label—see <a href="#">“Handler Syntax (Labeled Parameters)”</a> (page 215)
fourth	specifies a position in a container—see <a href="#">“Index”</a> (page 172) reference form

from	used in specifying a range of objects in a container—see “Range” (page 175) reference form; also used as handler parameter label—see “Handler Syntax (Labeled Parameters)” (page 215)
front	in front of is used to describe position in the “Relative” (page 176) reference form; synonymous with before
get	an AppleScript command—see <a href="#">get</a> (page 129)
given	a special handler parameter label—see “Handler Syntax (Labeled Parameters)” (page 215)
global	specifies the scope for a variable (see also local)—see “Global Variables” (page 49)
if	a control statement—see “if Statements” (page 197)
ignoring	a control statement—see “considering and ignoring Statements” (page 193)
in	used in construction object specifiers—see “Containers” (page 30); also used with the “Relative” (page 176) reference form—for example in front of and in back of
instead of	handler parameter label—see “Handler Syntax (Labeled Parameters)” (page 215)
into	put into is a deprecated synonym for the <a href="#">copy</a> (page 121) command; also used as handler parameter label—see “Handler Syntax (Labeled Parameters)” (page 215)
is	used with various comparison operators—see <a href="#">Table 9-1</a> (page 179)
it	refers to the current target (of it)—see “The it and me Keywords” (page 40)
its	synonym for of it—see “The it and me Keywords” (page 40)
last	specifies a position in a container—see “Index” (page 172) reference form
local	specifies the scope for a variable (see also global)—see “Local Variables” (page 48)
me	refers to the current script (of me)—see “The it and me Keywords” (page 40)
middle	specifies a position in a container—see “Index” (page 172) reference form
mod	remainder operator—see <a href="#">Table 9-1</a> (page 179)
my	synonym for of me—see “The it and me Keywords” (page 40)
ninth	specifies a position in a container—see “Middle” (page 173) reference form
not	logical negation operator—see <a href="#">Table 9-1</a> (page 179)
of	used in construction object specifiers—see “Containers” (page 30); used with or as part of many other terms, including of me, in front of, and so on
on	handler parameter label—see “Handler Syntax (Labeled Parameters)” (page 215)

onto	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
or	logical <i>or</i> operator—see <a href="#">Table 9-1</a> (page 179)
out of	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
over	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
prop, property	prop is an abbreviation for property—see “ <a href="#">The it and me Keywords</a> ” (page 40)
put	put into is a deprecated synonym for the <code>copy</code> (page 121) command
ref/reference	ref is an abbreviation for reference—see <a href="#">reference</a> (page 95)
repeat	a control statement—see “ <a href="#">repeat Statements</a> ” (page 199)
return	exits from a handler—see “ <a href="#">return</a> ” (page 214)
returning	deprecated
script	used to declare a script object; also the class of a script object—see the <code>script</code> (page 97) class and “ <a href="#">Script Objects</a> ” (page 57)
second	specifies a position in a container—see “ <a href="#">Index</a> ” (page 172) reference form
set	an AppleScript command—see <a href="#">set</a> (page 154)
seventh	specifies a position in a container—see “ <a href="#">Index</a> ” (page 172) reference form
since	handler parameter label—see “ <a href="#">Handler Syntax (Labeled Parameters)</a> ” (page 215)
sixth	specifies an index position in a container—see “ <a href="#">Index</a> ” (page 172) reference form
some	specifies an object in a container—see “ <a href="#">Arbitrary</a> ” (page 167) reference form
tell	a control statement—see “ <a href="#">tell Statements</a> ” (page 205)
tenth	specifies a position in a container—see “ <a href="#">Index</a> ” (page 172) reference form
that	synonym for <i>whose</i>
the	syntactic no-op, used to make script statements look more like natural language
then	used with <code>if</code> control statement—see “ <a href="#">if Statements</a> ” (page 197)
third	specifies a position in a container—see “ <a href="#">Index</a> ” (page 172) reference form
through, thru	used in specifying a range of objects in a container—see “ <a href="#">Range</a> ” (page 175) reference form
timeout	used with <code>with timeout</code> control statement—see <a href="#">with timeout</a> (page 210)
times	used with <code>repeat</code> control statement—see <a href="#">repeat (number) times</a> (page 200)

to	used in many places, including <code>copy</code> (page 121) and <code>set</code> (page 154) commands; in the “Range” (page 175) reference form; by operators such as <code>is equal to</code> and a reference <code>to</code> ; with the control statement <code>repeat with loopVariable (from startValue to stopValue)</code> (page 202); with the partial result parameter in “try Statements” (page 207)
transaction	used with <code>with transaction</code> control statement—see <code>with transaction</code> (page 211)
true	a Boolean literal—see “Boolean” (page 20)
try	an error-handling statement—see “try Statements” (page 207)
until	used with <code>repeat</code> control statement—see <code>repeat until</code> (page 201)
where	used with the “Filter” (page 169) reference form to specify a Boolean test expression (synonymous with <code>whose</code> )
while	used with <code>repeat</code> control statement—see <code>repeat while</code> (page 202)
whose	used with the “Filter” (page 169) reference form to specify a Boolean test expression (synonymous with <code>where</code> )
with	used in commands to specify various kinds of parameters, including <code>true</code> for some Boolean for parameters—see, for example, the <code>with prompt and multiple selections allowed</code> parameters to the <code>choose from list</code> (page 116) command; also used with application <code>make</code> commands to specify properties ( <code>with properties</code> )
without	used in commands to specify <code>false</code> for a Boolean for a parameter—see, for example, the <code>multiple selections allowed</code> parameter to the <code>choose from list</code> (page 116) command





# Error Numbers and Error Messages

This appendix describes error numbers and error messages provided by AppleScript, as well as certain Mac OS error numbers that may be of interest to scripters.

## 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.

**Table B-1** AppleScript errors

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.

Error number	Error message
-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.

## 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 often there is nothing you can do about them in a script, other than report them. A few, such as "User canceled", make sense for scripts to handle—as shown, for an example, in the Examples section for the [display dialog](#) (page 125) command.

**Table B-2** Mac OS errors

Error number	Error message
0	No error.
-34	Disk <name> 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 cancelled.
-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 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>.



# Working with Errors

---

This appendix provides a detailed example of handling errors with “try Statements” (page 207) and “error Statements” (page 196). It shows how to use a `try` statement to check for bad data and other errors, and an `error` statement to pass on any error that can’t be handled. It also shows how to check for just a particular error number that you are interested in.

## Catching Errors in a Handler

The `SumIntegerList` handler expects a list of integers. If any item in the passed list is not an integer, `SumIntegerList` signals error number 750 and returns 0. The handler includes an error handler that displays a dialog if the error number is equal to 750; if the error number is not equal to 750, the handler resignals the error with an `error` statement so that other statements in the call chain can handle the unknown error. 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
        end repeat
        return integerSum -- Successful completion of handler.
    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 errStr number errorNumber
        end if
    end try
end SumIntegerList
```

The `SumIntegerList` handler handles various error conditions. For example, the following call completes without error:

```
set sumList to {1, 3, 5}
set listTotal to SumIntegerList from sumList --result: 9
```

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 handler didn't total the list;
    -- do something to handle the error (not shown)
end if
```

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 (not shown).

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.

Finally, suppose the caller has its own error handler, so that if the handler 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 text
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.

## Simplified Error Checking

AppleScript provides a mechanism to streamline the way you can catch and handle individual errors. It is often necessary for a script to handle a particular error, but not others. It is possible to catch an error, check for the error number you are interested in, and use an error statement to resignal for other errors. For example:

```
try
    open for access file "MyFolder:AddressData" with write permission
on error msg number n from f to t partial result p
    if n = -49 then -- File already open error
        display dialog "I'm sorry but the file is already open."
    else
        error msg number n from f to t partial result p
    end if
end try
```

This script tries to open a file with write permission, but if the file is already opened, it just displays a dialog. However, you can instead implement this more concisely as:

```
try
    open for access file "MyFolder:AddressData" with write permission
```

**Working with Errors**

```
on error number -49
    display dialog "I'm sorry but the file is already open."
end try
```

In this version, there is no need to list the `message`, `from`, `to`, or `partial result` parameters, in order to pass them along. If the error is not `-49` (`file <name> is already open`), this error handler will not catch the error, and AppleScript will pass the error to the next handler in an outer scope.

One drawback to this approach is that you must use a literal constant for the error number in the `on error` parameter list. You can't use global variable or property names because the number must be known when the script is compiled.





# Double Angle Brackets

---

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 to conform to settings in Script Editor’s Formatting preferences.

When you open, compile, edit, or run scripts with Script Editor, you may occasionally see terms enclosed in double angle brackets, or chevrons («»). For example, you might see the term «event sysodlog» as part of a script—this is the event code representation for a `display dialog` (page 125) command. The event code representation is also known as **raw format**.

For compatibility with Asian national encodings, “⌘” and “⇧” are allowed as synonyms for “«” and “»” (Option-⌘ and Option-Shift-⌘, respectively, on a U.S. keyboard), since the latter do not exist in some Asian encodings.

The following sections provide more information about when chevrons appear in scripts.

## When a Dictionary Is Not Available

AppleScript uses double angle brackets in a Script Editor script window when it can’t identify a term. That happens when it encounters a term that isn’t part of the AppleScript language and isn’t defined in an application or scripting addition dictionary that is available when the script is opened or compiled.

For example, if a script is compiled on one machine and later opened on another, the dictionary may not be available, or may be from an older version of the application or scripting addition that does not support the term.

This can also happen if the file `StandardAdditions.osax` is not present in `/System/ScriptingAdditions`. Then, scripting addition commands such as `display dialog` will not be present and will be replaced with chevron notation («event sysodlog») when you compile or run the script.

## 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 scriptObjectJoe to Joe
scriptObjectJoe
--result: «script Joe»
```

Similarly, if Script Editor can't display a variable's 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.

This may occur because an application command returns a value that does not belong to any of the normal AppleScript classes. You can store such data in variables and send them as parameters to other commands, but Script Editor cannot display the data in its native format.

## 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. You might want to do this if you're working on a script that needs to use terminology that isn't available on your current machine—for example, if you're working at home and don't have the latest dictionary for a scriptable application you are developing, but you know the codes for a supported term.

You can also use AppleScript to display the underlying codes for 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 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, cancel the dialog.

Script Editor can compile the script, but displays chevron format for any terms that rely on a missing dictionary.

## Sending Raw Apple Events From a Script

The term `«event sysodlog»` is actually the raw form for an Apple event with event class `'syso'` and event ID `'dlog'` (the `display dialog` command). For a list of many of the four-character codes and their related terminology used by Apple, see *AppleScript Terminology and Apple Event Codes Reference*.

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, providing detailed documentation for how to do so is beyond the scope of this guide.

# Unsupported Terms

---

This appendix lists scripting terms that are not supported by AppleScript. Though you may see these terms in a dictionary, script, or scripting addition, you should not count on their behavior.

## List of Unsupported Terms

handle CGI request

This command is not supported.

internet address

An Internet or intranet address for the TCP/IP protocol. Only used for compatibility with WebSTAR AppleScript CGI scripts, this term is not supported by AppleScript itself.

web page

An HTML page. This class is not supported.



# Glossary

---

**absolute object specifier** An object specifier that has enough information to identify an object or objects uniquely. For an object specifier to an application object to be complete, its outermost container must be the application itself. See [relative object specifier](#).

**Apple event** An interprocess message that encapsulates a high-level task in a single package that can be passed across process boundaries, performed, and responded to with a reply event. When an AppleScript script is executed, a statement that targets a scriptable application may result in an Apple event being sent to that application.

**AppleScript** A scripting language that makes possible direct control of scriptable applications and scriptable parts of Mac OS X.

**AppleScript command** A script command provided by AppleScript. AppleScript commands do not have to be included in `tell` statements.

**application command** A command that is defined by scriptable application to provide access to a scriptable feature. 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 reference form** A reference form that specifies an arbitrary object in a container.

**assignment statement** A statement that assigns a value to a variable. Assignment statements use the `copy` or `set` commands.

**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; see the `boolean` 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 object, called the parent.

**class** (1) A category for objects that share characteristics such as properties and elements and respond to the same commands. (2) The label for the AppleScript `class` property—a reserved word that specifies the class to which an object belongs.

**coercion** The process of converting an object 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. Also known as object conversion.

**command** A word or series of words that requests an action. See also [handler](#).

**comment** Text that remains in a script after compilation but is ignored by AppleScript when the 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.

**composite value** A value that contains other values. Lists, records, and strings are examples of composite values.

**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`. See also [simple statement](#).

**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; see the `constant` 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 used in Script Editor to extend a statement to the next line. With a U.S. keyboard, you can enter this character by typing Option-I (lower-case L).

**continue statement** A statement that controls when and how other statements are executed. AppleScript defines standard control statements such as `if`, `repeat`, and `while`.

**control statement** A statement that causes AppleScript to exit the current handler and transfer execution to the handler with the same name in the parent. A `continue` statement can also be used to invoke an inherited handler in the local context.

**current application** The application that is using the AppleScript component to compile and execute scripts (typically, Script Editor).

**current script** The script currently being executed.

**current target** The object that is the current default target for commands.

**data** A class used for data that do not belong to any of the other AppleScript classes; see the `data` class.

**date** A class that specifies a time, day of the month, month, and year; see the `date` class.

**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 (or implicit) targets are specified in `tell` statements.

**delegation** The handing off of control to another object. In AppleScript, the use of a `continue` statement to call a handler in a parent object or the current application.

**dialect** A version of the AppleScript language that resembles a specific human language or programming language. As of AppleScript 1.3, English is the only dialect supported.

**dictionary** The set of commands, objects, and other terminology that is understood by an application or other scriptable entity. You can display an application's dictionary with Script Editor.

**direct parameter** The parameter immediately following a command, which typically specifies the object to which the command is sent.

**double angle brackets** Characters («») typically used by AppleScript to enclose raw data. With a U.S. keyboard, you can enter double angle brackets (also known as chevrons) by typing Option-Backslash and Shift-Option-Backslash.

**element** An object contained within another object. An object can typically contain zero or more of each of its elements.

**empty list** A list containing no items. See the `list` class.

**error expression** An expression, usually a `text` object, that describes an error.

**error handler** A collection of statements that are executed in response to an error message. See the `try` statement.

**error message** A message that is supplied by an application, by AppleScript, or by Mac OS X when 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 reference form** A reference form that specifies every object of a particular type 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` object that begins with `on run` and ends with `end`. A single `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 a condition specified by 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 can be invoked by name. See also [command](#).

**identifier** A series of characters that identifies a value or handler in AppleScript. Identifiers are used to name variables, handlers, 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 text strings or sends commands to applications.

**implicit run handler** All the statements at the top level of a script except for property definitions, `script` object definitions, and other handlers. A single `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 by describing its position with respect to the beginning or end of a container.

**inheritance** The ability of a child `script` object to take on the properties and handlers of a parent object.

**inheritance chain** The hierarchy of objects that AppleScript searches to find the target for a command or the definition of a term.

**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** A location where another object or objects can be added.

**integer** A positive or negative number without a fractional part; see the `integer` class.

**item** A value in a list or record. An item can be specified by its offset from the beginning or end of the list or record.

**keyword** A word that is part of the AppleScript language. Synonymous with [reserved word](#).

**labeled parameter** A parameter that is identified by a label. See also [positional parameter](#).

**lifetime** The period of time over which a variable or property is in existence.

**list** An ordered collection of values; see the `list` class.

**literal** A value 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 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 Event Log pane of a script window, and to the Event Log History window, if it is open.

**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 reference form** A reference form that specifies the middle object of a particular class in a container. (This form is rarely used.)

**name reference form** A reference form that specifies an object by name—that is, 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 classes `integer` and `real`.

**object** An instantiation of a class definition, which can include properties and actions.

**object conversion** See [coercion](#).

**object specifier** A phrase specifies the information needed to find another object in terms of the objects in which it is contained. See also [absolute object specifier](#), [relative object specifier](#), and [reference form](#).

**operand** An expression from which an operator derives a value.

**operation** The evaluation of an expression that contains an operator.

**operator** A symbol, word, or phrase that derives a value from another value or pair of values.

**optional parameter** A parameter that need not be included for a command to be successful.

**outside property, variable, or statement** A property, variable, or statement in a `script` object but occurs outside of any handlers or nested `script` objects.

**parameter variable** An identifier in a handler definition that represents the actual value of a parameter when the handler is called. Also called a *formal parameter*.

**parent object** An object from which another `script` object, called the child, inherits properties and handlers. A parent object may be any object, such as a `list` or an `application` object, but it is typically another `script` object.

**positional parameter** A handler parameter that is identified by the order in which it is listed. In a handler 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 handler definition.

**property** A labeled container in which to store a value. Properties can specify characteristics of objects.

**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 format** AppleScript terms enclosed in double angle brackets, or chevrons («»). AppleScript uses raw format because it cannot find a script term in any available dictionary, or cannot display data in its native format.

**real** A number that can include a decimal fraction; see the `real` class.

**record** An unordered collection of properties, identified by unique labels; see the `record` class.



**recordable application** An application that uses Apple events to report user actions for recording purposes. When recording is turned on, Script Editor creates statements corresponding to any significant actions you perform in a recordable application.

**recursive handler** A handler that calls itself.

**reference** An object that encapsulates an object specifier.

**reference form** The syntax for identifying an object or group of objects in an application or other container—that is, the syntax for constructing an object specifier. AppleScript defines reference forms for arbitrary, every, filter, ID, index, middle, name, property, range, and relative.

**relative object specifier** An object specifier that does not include enough information to identify an object or objects uniquely. When AppleScript encounters a partial object specifier, it uses the default object specified in the enclosing `tell` statement to complete the reference. See [absolute object specifier](#).

**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 word** A word that is part of the AppleScript language. Synonymous with [keyword](#).

**result** A value generated when a command is executed or an expression evaluated.

**return statement** A statement that exits a handler and optionally returns a specified value.

**scope** The range over which AppleScript recognizes a variable or property, which determines where else in a script you may refer to that variable or property.

**script** A series of written instructions that, when executed, cause actions in applications or Mac OS X.

**scriptable application** An application that can be controlled by a script. For AppleScript, that means being responsive to interapplication messages, called Apple events, sent when a script command targets the application.

**script application** An application whose only function is to run the script associated with it.

**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. If a scripting addition is located in the Scripting Additions folder, its terminology is available for use by any script.

**scripting addition command** A command that is implemented as a scripting addition.

**script object** A user-defined object that can combine data (in the form of properties) and actions (in the form of handlers and additional `script` objects).

**script object definition** A compound statement that contains a collection of properties, handlers, and other AppleScript statements.

**simple statement** One that can be written on a single line. See also [compound statement](#).

**simple value** A value, such as an integer or a constant, that does not contain other values.

**Standard suite** A set of standard AppleScript terminology that a scriptable application should support if possible. The Standard suite contains commands such as `count`, `delete`, `duplicate`, and `make`, and classes such as `application`, `document`, and `window`.

**statement** A series of lexical elements that follows a particular AppleScript syntax. Statements can include keywords, variables, operators, constants, expressions, and so on. See also [compound statement](#), [simple statement](#).

**statement block** One or more statements enclosed in a compound statement and having an `end` statement.

**string** A synonym for the `text` class.

**styled text** Text that may include style and font information. Not supported in AppleScript 2.0.

**suite** Within an application's scriptability information, a grouping of terms associated with related operations.

**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** The rules for constructing a valid AppleScript statement of a particular type.

**system object** An object that is part of a scriptable element of Mac OS X.

**target** The recipient of a command. Potential targets include `application objects`, `script objects` (including 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** An ordered series of characters (a text string); see the `text` class.

**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 code point** A unique number that represents a character and allows it to be represented in an abstract way, independent of how it is rendered.

**Unicode text** A class that represents an ordered series of two-byte Unicode characters.

**user-defined command** A command that is implemented by a handler defined in a `script` object.

**using terms from statement** A control statement that instructs AppleScript to use the terminology from the specified application in compiling the enclosed statements.

**variable** A named container in which to store a value.

**with timeout statement** A control statement that specifies 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, such that either all of the changes are applied or none are.

# Document Revision History

---

This table describes the changes to *AppleScript Language Guide*.

Date	Notes
2008-03-11	Updated to describe AppleScript features through Mac OS X v10.5 and AppleScript 2.0.
	The previous release of <i>AppleScript Language Guide</i> was on May 5, 1999.

**REVISION HISTORY**

Document Revision History

# Index

---

## Symbols

---

- \* operator [183](#)
- + operator [184](#)
- / operator [184](#)
- = operator [180](#)
- > operator [180](#)
- >= operator [181](#)
- & (concatenation) operator [187](#)
- & operator [180](#)
- & operator [187](#)
- < operator [181](#)
- <= operator [181](#)
- " character [100](#)
- \ character [100](#)
- ^ operator [185](#)
- { } characters [89](#)
- | in identifiers [18](#)
- | in syntax definitions [15](#)
- «» characters [241–242](#)
- ¬ character [20](#)
- ÷ operator [184](#)
- operator [184](#)
- operator [180](#)
- ≤ operator [181](#)
- ≥ operator [181](#)

## A

---

- a reference to operator [31](#), [95](#), [185](#), [188](#)
- about handler parameter label [215](#)
- above handler parameter label [215](#)
- absolute object specifiers [30](#)
- activate command [108](#)
- adding folder items to Folder Actions handler [222](#)
- addition operator [184](#)
- addition
  - of date values [86](#)
- administrator privileges parameter
  - of command do shell script [129](#)

- after reserved word [176](#)
- against handler parameter label [215](#)
- alert volume parameter
  - of command set volume [157](#)
- alias class [79](#)
- alias
  - specifying a file by [41](#)
- aliases and files [41–43](#)
- aliases
  - working with [42](#)
- altering line endings parameter
  - of command do shell script [129](#)
- and operator [179](#)
- angle brackets in scripts [241–242](#)
- apart from handler parameter label [215](#)
- Apple event code [24](#)
- Apple events [13](#)
- AppleScript character set (Unicode) [17](#)
- AppleScript constant [37](#)
  - AppleScript [37](#)
  - current application [39](#)
- AppleScript global constants [37](#)
- AppleScript property
  - missing value [40](#)
  - pi constant [37](#)
  - result [38](#)
  - text constants [38](#)
  - text item delimiters [38](#)
  - version [39](#)
- AppleScript suite [105](#)
- AppleScript
  - commands [35](#)
  - constants [37](#)
  - defined [13](#)
  - error numbers [233](#), [234](#)
  - fundamentals [25–46](#)
  - keywords [18](#), [227–231](#)
  - lexical conventions [17–24](#)
  - script objects [57–65](#)
  - unsupported terms [243](#)
  - variables and properties [47–56](#)
- application class [80](#)

application commands [35](#)  
 application object [34](#)  
 applications  
   remote [44](#)  
 arbitrary reference form [167](#)  
 arithmetic, date-time [86](#)  
 around handler parameter label [215](#)  
 as operator [32, 185](#)  
 as parameter  
   of command choose application [111](#)  
   of command display alert [124](#)  
   of command do shell script [129](#)  
   of command get [130](#)  
   of command path to (application) [142](#)  
   of command path to (folder) [145](#)  
   of command read [148](#)  
   of command the clipboard [162](#)  
   of command write [164](#)  
 as user name parameter  
   of command mount volume [139](#)  
 ASCII character command [108](#)  
 ASCII number command [109](#)  
 aside from handler parameter label [215](#)  
 assignment statement [22](#)  
 associativity, of operators [185](#)  
 at handler parameter label [215](#)

## B

---

back of reserved words [176](#)  
 back reserved word [172, 177](#)  
 backslash character in text [100](#)  
 beep command [110](#)  
 before parameter  
   of command read [148](#)  
 before reserved word [176](#)  
 beginning reserved word [177](#)  
 begins with operator [182](#)  
 behind reserved word [176](#)  
 below handler parameter label [215](#)  
 beneath handler parameter label [215](#)  
 beside handler parameter label [215](#)  
 between handler parameter label [215](#)  
 binary operator [179](#)  
 Bonjour  
   and remote applications [44, 118](#)  
   service types [119](#)  
 boolean class [82](#)  
 Boolean constants [20, 40, 83](#)  
 boolean expressions [197](#)  
 brackets [15](#)  
 but keyword [193](#)

buttons parameter  
   of command display alert [124](#)  
   of command display dialog [126](#)  
 by handler parameter label [215](#)

## C

---

cancel button name parameter  
   of command choose from list [117](#)  
 cancel button parameter  
   of command display alert [125](#)  
   of command display dialog [126](#)  
 case attribute [194](#)  
 character element [99](#)  
 character  
   elements of a text object [99](#)  
 chevrons [24, 241](#)  
 child script objects [61](#)  
 choose application command [110](#)  
 choose color command [111](#)  
 choose file command [112](#)  
 choose file name command [114](#)  
 choose folder command [115](#)  
 choose from list command [116](#)  
 choose remote application command [118](#)  
 choose URL command [119](#)  
 class class [83](#)  
 class property [79, 80, 82, 84, 85, 89, 90, 92, 93, 94, 97, 98, 103](#)  
 class  
   defined [79](#)  
   reference [79–104](#)  
 classes  
   mutable [49](#)  
 Clipboard Commands suite [105](#)  
 clipboard info command [120](#)  
 close access command [120](#)  
 closing folder window for Folder Actions handler [223](#)  
 coercion operator (as) [185](#)  
 coercion  
   see object conversion [32](#)  
 comes after operator [180](#)  
 comes before operator [181](#)  
 commands  
   AppleScript [35](#)  
   application [35](#)  
   defined [105](#)  
   direct parameter of [36](#)  
   reference [105–165](#)  
   scripting addition [35](#)  
   target of [35](#)

- user-defined 35
- waiting for completion of 211
- comments 19
  - block 19
  - end-of-line 19
- completion
  - of commands 211
- compound statements 23
- concatenation operator (&) 180, 187
- considering / ignoring (application responses) control statement 195
- considering / ignoring (text comparison) control statement 193
- considering and ignoring statements 193
- considering statements (application responses) 195
- considering statements (string comparison) 193
- constant class 84
- constant
  - defined 20
- constants
  - AppleScript 37
  - Boolean 20, 40, 83
  - days of the week 86
  - months of the year 86
  - text 100
  - white space 100
- constructor functions 59
- containers 30
- contains operator 182, 189
- contains, is contained by operator 189
- contents property 31, 96
- continue statement
  - defined 213
  - in script objects 63
- control statements reference 193–212
- conventions in this book 15
- copy command 121
- count command 122
- current application and parent property 39
- current application constant 39
- current date command 123
- current script 40
- current target 40

## D

---

- date class 85
- date string property 86
- date, relative 88
- date-time arithmetic 86
- day property 86
- days of the week constants 86

- debugging tips 45
  - flow of control 45
  - log statements 46
  - third party debuggers 46
- default answer parameter
  - of command display dialog 126
- default button parameter
  - of command display alert 125
  - of command display dialog 126
- default color parameter
  - of command choose color 112
- default items parameter
  - of command choose from list 117
- default location parameter
  - of command choose file 113
  - of command choose file name 114
  - of command choose folder 115
- default name parameter
  - of command choose file name 114
- delay command 123
- delegation 63
- diacriticals attribute 194
- dictionary
  - defined 25
  - displaying 25
  - when not available 241
- direct parameter of commands 36
- display alert command 124
- display dialog command 125
- displaying parameter
  - of command say 153
- div operator 184
- division operator (÷) 184
- do shell script command 128
- does not come after operator 181
- does not come before operator 181
- does not contain operator 183
- does not equal operator 180
- double angle brackets 241–242
- double-quote character 100

## E

---

- editable URL parameter
  - of command choose URL 119
- eighth reserved word 172
- elements of objects 28
- ellipsis in syntax definitions 15
- else clause 199
- else if clause 199
- empty list 89
- empty selection allowed parameter

- of command choose from list 117
- enabling remote applications 44
- end reserved word 177
- ends with operator 182, 192
- eppc-style specifier 44
- equal operator 190
- equal, is not equal to operator 190
- equals operator 180
- error control statement 196
- error numbers
  - AppleScript 233, 234
  - defined 196
- error reporting parameter
  - of command open location 141
- error
  - expression 196
  - handlers 207
  - handling 36
  - message 196
  - user cancelled 37
- errors
  - resignaling in scripts 237
  - signaling in scripts 196
  - types of 37
  - working with 237–239
- evaluation
  - defined 22
  - of expressions 22
- Event Log History window 209
- event timed out error message 211
- every reference form 168
- every reserved word 168
- exit control statement 199
- exit from repeat loop 199
- explicit run handlers 74
- exponent operator (^) 185
- expressions 22
  - boolean 197
  - evaluation of 22

## F

---

- false constant 40, 83
- fifth reserved word 172
- file class 88
- File Commands suite 105
- File Read/Write suite 106
- files and aliases 41–43
- files, specifying
  - by alias 41
  - by name 43
  - by pathname 43

- filter reference form 169
- first reserved word 172
- Folder Actions reference 221–225
- folder creation parameter
  - of command path to (folder) 145
- for handler parameter label 215
- for parameter
  - of command clipboard info 120
  - of command read 148
  - of command write 164
- fourth reserved word 172
- from handler parameter label 215
- from parameter
  - of command path to (folder) 145
  - of command random number 146
  - of command read 148
- from reserved word 175
- from table parameter
  - of command localized string 136
- front of reserved words 176
- front reserved word 172, 177
- frontmost property 80

## G

---

- get command 129
- get eof command 131
- get volume settings command 131
- given handler parameter label 215
- giving up after parameter
  - of command display alert 125
  - of command display dialog 127
- global constants
  - of AppleScript 37
- global variables 49, 52
  - persistence of 53
  - scope of 51
- greater than operator 180, 191
- greater than or equal to operator 181
- greater than, less than operator 191

## H

---

- handle CGI request (unsupported) 243
- handlers
  - call syntax
    - labeled parameters 216
    - positional parameters 218
  - calling from a tell statement 72
  - defined 67



- defining simple 68
- defining syntax
  - labeled parameters 215
  - no parameters 68
  - positional parameters 218
- errors in 71
- for errors 207
- for stay-open script applications 76–77
- idle 76
- in script applications 73
- libraries of 72
- open 75
- overview 67–77
- quit 76
- recursive 71
- reference 213–219
- run 74
- scope of identifiers declared within 55
- has parameter
  - of command system attribute 160
- hidden answer parameter
  - of command display dialog 126
- hyphens attribute 194

---

## I

- id property 80, 98
- ID reference form 171
- id reserved word 171
- identifiers 18
- idle handlers 76
- if (compound) control statement 198
- if (simple) control statement 198
- ignoring statements (application responses) 195
- ignoring statements (string comparison) 193
- implicit run handlers 74
- implicitly specified subcontainers 29
- in AppleTalk zone parameter
  - of command mount volume 139
- in back of reserved words 176
- in bundle parameter
  - of command localized string 136
  - of command path to resource 146
- in directory parameter
  - of command path to resource 146
- in front of reserved words 176
- in parameter
  - of command offset 139
  - of command run script 152
  - of command store script 158
  - of command summarize 159
- in

- for specifying a container 30
  - with date objects 88
- index reference form 172
- index reserved word 172
- info for command 132
- inheritance 60–65
  - examples of 61
- initializing script objects 59
- input volume parameter
  - of command set volume 157
- insertion point 36
- insertion point object
  - and index reference form 177
  - and relative reference form 176
- instead of handler parameter label 215
- integer class 89
- integral division operator 184
- internet address (unsupported) 243
- Internet suite 106
- into handler parameter label 215
- invisibles parameter
  - of command choose file 113
  - of command choose folder 115
  - of command list folder 135
- is contained by operator 183, 189
- is equal to operator 180
- is not contained by operator 183
- is not equal to operator 190
- is not greater than operator 181
- is not less than operator 181
- is not operator 180
- is operator 180
- it keyword 40
- item element 90
- items 89, 96
- its reserved word 40

---

## K

- keywords, AppleScript 18, 227

---

## L

- labeled parameters, of handlers 68
- language elements in syntax definitions 15
- large lists
  - inserting in 91
- last reserved word 172
- launch command 134
- length property 90, 94, 98

less than operator 181, 191  
 less than or equal to operator 181  
 libraries of handlers 72  
 lifetime of variables and properties 51  
 linefeed constant 100  
 list class 89  
 list disks command 134  
 list folder command 135  
 lists  
   inserting in large 91  
   merging 91  
 literal expressions 20  
 load script command 72, 135  
 local variables 48, 52, 59  
   scope of 51  
 localized string command 136  
 location parameters 36  
 log command 138  
 log statements 46  
 loop variable 202, 203  
 lowercase letters 194

## M

---

me keyword 40  
 merging lists 91  
 message parameter  
   of command display alert 124  
 middle reference form 173  
 middle reserved word 173  
 minus symbol (-) 184  
 Miscellaneous Commands suite 106  
 missing value constant 40  
 mod operator 184  
 month property 86  
 months of the year constants 86  
 mount volume command 138  
 moving folder window for Folder Actions handler  
   223  
 multiple selections allowed parameter  
   of command choose application 111  
   of command choose file 113  
   of command choose folder 115  
   of command choose from list 117  
 multiplication operator (\*) 183  
 mutable classes 49  
 my reserved word 64  
 my  
   in tell statements 72

## N

---

name property 80  
 name reference form 173  
 name  
   specifying a file by 43  
 named reserved word 174  
 nested tell statements 205  
   examples 206  
 ninth reserved word 172  
 not operator 185  
 number class 92  
 numeric literal 21

## O

---

object conversion (coercion) 32  
 object conversion  
   table of supported conversions 32  
 object specifiers 22, 29  
   absolute 30  
   contents of 29  
   evaluating with contents property 31  
   implicitly specified subcontainers 29  
   in reference objects 31  
   relative 30  
 objects  
   elements of 28  
   properties of 28  
   script  
     initializing 59  
     parent 61–65  
     sending commands to 59  
   using in AppleScript 26  
 of me  
   in tell statements 72  
 of my keyword 40  
 of parameter  
   of command offset 139  
 of type parameter  
   of command choose file 113  
 of  
   for specifying a container 30  
   with date objects 88  
 offset command 139  
 OK button name parameter  
   of command choose from list 117  
 on handler parameter label 215  
 on server parameter  
   of command mount volume 138  
 onto handler parameter label 215  
 open for access command 140

open handlers 75  
 open location command 141  
 opening folder Folder Actions handler 224  
 operators  
   binary 179  
   defined 179  
   listed, with descriptions 179–185  
   precedence 185  
   reference 179–192  
   unary 179  
 or operator 179  
 out of handler parameter label 215  
 output muted parameter  
   of command set volume 157  
 output volume parameter  
   of command set volume 157  
 over handler parameter label 215

## P

---

paragraph element 99  
 parameter variables 59, 213  
 parameters  
   direct 36  
   in continue statements 213  
   labeled 68  
   location 36  
   passing by reference versus value 71  
   patterned 70  
   positional 69  
 parent property 61  
 parent script objects 61–65  
 password parameter  
   of command do shell script 129  
 path to (application) command 142  
 path to (folder) command 143  
 path to resource command 145  
 pathname  
   specifying a file by 43  
 paths, specifying a file with 42  
 patterned parameters 70  
 persistence  
   of global variables 53  
   of script properties 53  
 pi constant 37  
 placeholders in syntax definitions 15  
 plural object names 168  
 plus symbol (+) 184  
 positional parameters, of handlers 69  
 POSIX file class 93  
 POSIX files  
   using with files and aliases 36–43

POSIX path property 79  
 possessive notation ('s) 30  
 possessive object names 30  
 precedence  
   of attributes 195  
   of operations 185  
 properties  
   declaring 47  
   lifetime of 51  
   of objects 28  
   of script objects 58  
   scope of 51  
 property reference form 174  
 punctuation attribute 194  
 put, (Deprecated--use copy) 230

## Q

---

quit handlers 76  
 quoted form property 98

## R

---

random number command 146  
 range reference form 175  
 raw apple events 242  
 raw data  
   displayed by AppleScript 241  
   entering in a script 242  
 raw format 241  
 read command 147  
 real class 93  
 record class 94  
 recursion 71  
 recursive handlers 71  
 reference class 95  
 reference forms 167–177  
   arbitrary 167  
   defined 167  
   every 168  
   filter 169  
   ID 171  
   index 172  
   middle 173  
   name 173  
   property 174  
   range 175  
   relative 176  
 relative object specifiers 30  
 relative reference form 176

- relative to
  - with date objects [88](#)
- remainder operator [184](#)
- remote applications [44](#)
  - choosing [118](#)
  - enabling [44](#)
  - targeting [44](#)
- removing folder items from Folder Actions handler [225](#)
- reopen command [134](#)
- repeat (forever) control statement [199](#)
- repeat (number) times control statement [200](#)
- repeat control statements [199](#)
- repeat until control statement [201](#)
- repeat while control statement [202](#)
- repeat with loopVariable (from startValue to stopValue) control statement [202](#)
- repeat with loopVariable (in list) control statement [203](#)
- replacing parameter
  - of command store script [158](#)
- reserved words (see keywords) [227](#)
- rest of property [90](#)
- rest property [90](#)
- Result pane [24, 38](#)
- result property [38](#)
- result variable [24](#)
- result, of statement [24](#)
- return character
  - in text objects [101](#)
- return constant [100](#)
- return statement [214](#)
  - in handler definition [67](#)
- returning, Deprecated reserved word [230](#)
- reverse property [90](#)
- RGB color class [96](#)
- round command [149](#)
- rounding parameter
  - of command round [150](#)
- run command [151](#)
- run handlers [74](#)
  - explicit [74](#)
  - implicit [74](#)
  - in script objects [58, 60](#)
- run script command [151](#)
- running property [81](#)
- runTarget parameter
  - of command run [151](#)
  - of command say [153](#)
- say command [152](#)
- scope
  - of variables and properties [51](#)
  - shadowing [52, 61](#)
- script applications [73](#)
  - calling [77](#)
  - handlers for [73](#)
  - Mac OS 9 compatible [73](#)
  - modern bundle format [73](#)
  - startup screen in [73](#)
  - stay-open [73](#)
- script class [97](#)
- Script Editor
  - Event Log History window [46, 209](#)
  - location in system [25](#)
  - overview [25](#)
- script objects [57–65](#)
  - child [61](#)
  - contents of [27](#)
  - defined [57](#)
  - initializing [59](#)
  - parent [61–65](#)
  - scope of identifiers declared at top level of [52](#)
  - sending commands to [59](#)
  - syntax of [57](#)
- script properties
  - persistence of [53](#)
  - scope of [51](#)
- script, current [40](#)
- scripting addition
  - command [35](#)
  - overview [34](#)
- scripting components command [153](#)
- Scripting suite [106](#)
- second reserved word [172](#)
- set command [154](#)
- set eof command [155](#)
- set the clipboard to command [156](#)
- set volume command [157](#)
- seventh reserved word [172](#)
- short-circuiting, during evaluation [179](#)
- showing package contents parameter
  - of command choose file [113](#)
  - of command choose folder [115](#)
- showing parameter
  - of command choose URL [119](#)
- simple statements [23](#)
- since handler parameter label [215](#)
- sixth reserved word [172](#)
- size parameter
  - of command info for [132](#)
- slash symbol (/) [184](#)

## S

---

saving to parameter

- some reserved word 167
- space constant 100
- special characters
  - in identifiers 18
  - in text 100
- Standard suite 107
- starting at parameter
  - of command write 164
- starts with operator 182, 192
- starts with, ends with operator 192
- startup screen in script applications 73
- statements 23
  - compound 23
  - simple 23
- stay-open script applications 73
- store script command 158
- storing values in variables 22
- string class 102
- String Commands suite 107
- subtraction of date values 86
- subtraction operator (-) 184
- suites
  - AppleScript 105
  - Clipboard Commands 105
  - File Commands 105
  - File Read/Write 106
  - Internet 106
  - Miscellaneous Commands 106
  - Scripting 106
  - Standard 107
  - String Commands 107
  - User Interaction 107
- summarize command 159
- synonyms for whose 169
- system attribute command 160
- system info command 161

## T

---

- tab character
  - in text objects 101
- tab constant 100
- target, current 40
- target
  - of commands 35
- targeting remote applications 44
- tell (compound) control statement 206
- tell (simple) control statement 206
- tell statements 36, 205
  - nested 205
  - nested, examples of 206
- tenth reserved word 172

- terminating
  - handler execution 214
  - repeat statement execution 199
- test
  - Boolean 197
  - in filter reference form 169
- text class 97
- text element 99
- text item delimiters
  - AppleScript property 38
- text literal 21
- text
  - as replacement for string 97
  - constants 38, 100
  - special characters in 100
- that reserved word 169
- the clipboard command 162
- the reserved word (syntactic no-op) 230
- then reserved word 198
- third reserved word 172
- through handler parameter label 215
- through reserved word 175
- thru handler parameter label 215
- thru reserved word 175
- time property 86
- time string property 86
- time to GMT command 163
- timeout, default value 210
- times reserved word 201
- to parameter
  - of command copy 121
  - of command random number 146
  - of command read 148
  - of command set 154
  - of command set eof 156
  - of command write 163
- transaction reserved word 211
- true constant 40, 83
- try control statement 207
- try statements 207

## U

---

- unary operators 179
- under handler parameter label 215
- Unicode text class 102
- unit types class 103
- Unix executable
  - making script into 19
- unsupported terms 243
- until parameter
  - of command read 148

uppercase letters [194](#)  
 user cancelled error [37](#)  
 User Interaction suite [107](#)  
 user name parameter  
   of command do shell script [129](#)  
 user-defined commands [35](#)  
 using delimiter parameter  
   of command read [148](#)  
 using delimiters parameter  
   of command read [148](#)  
 using parameter  
   of command say [153](#)  
 using terms from control statement [209](#)

## V

---

variables [22](#)  
   declaring [48](#)  
   declaring with copy command [50](#)  
   declaring with set command [49](#)  
   defined [22](#)  
   global [49, 51, 52](#)  
   lifetime of [51](#)  
   local [48, 51, 52, 59](#)  
   scope of [51](#)  
 version property [39, 81](#)  
 vertical bar character (|) in identifiers [18](#)  
 vertical bars (|)  
   in syntax definitions [15](#)

## W

---

waiting until completion parameter  
   of command say [153](#)  
 web page (unsupported) [243](#)  
 weekday property [86](#)  
 where reserved word [169](#)  
 while reserved word [202](#)  
 white space attribute [194](#)  
 white space constants [100](#)  
 whose reserved word [169](#)  
 whose  
   synonyms for [169](#)  
 with clause [217](#)  
 with icon parameter  
   of command display dialog [127](#)  
 with parameters parameter  
   of command run script [152](#)  
 with password parameter  
   of command mount volume [139](#)

with prompt parameter  
   of command choose application [111](#)  
   of command choose file [112](#)  
   of command choose file name [114](#)  
   of command choose folder [115](#)  
   of command choose from list [117](#)  
   of command choose remote application [118](#)  
 with seed parameter  
   of command random number [147](#)  
 with timeout control statement [210](#)  
 with timeout statements [210, 211](#)  
 with title parameter  
   of command choose application [111](#)  
   of command choose from list [117](#)  
   of command choose remote application [118](#)  
   of command display dialog [127](#)  
 with transaction control statement [211](#)  
 without clause [217](#)  
 word element [99](#)  
 working with errors [237](#)  
 write command [163](#)  
 write permission parameter  
   of command open for access [140](#)

## Y

---

year property [86](#)