AppleScript Language Guide

English Dialect

Apple Computer, Inc.

© 1996 Apple Computer, Inc.
All rights reserved.

No part of this publication or the software described in it 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 Computer, Inc.

Printed in the United States of America.

The Apple logo is a trademark of Apple Computer, 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 book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple Macintosh computers.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, AppleTalk, HyperCard, HyperTalk, LaserWriter, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

AppleScript, Finder, Geneva and System 7 are trademarks of Apple Computer, Inc.

Adobe Illustrator and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

FileMaker is a registered trademark of Claris Corporation.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manuals distributed with an Apple product, Apple will replace the manuals at no charge to you, provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 90-day period after you purchased the software. In addition, Apple will replace damaged manuals for as long as the software is included in Apple's Media Exchange program. See your authorized Apple dealer for program coverage and details. In some countries the replacement period may be different; check with your authorized Apple dealer.

**ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTA-
BILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.**

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, 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 MANUAL, 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

Figures and Tables xiii

Preface About This Guide xv

Audience xv
Organization of This Guide xvi
Sample Applications and Scripts xvii
For More Information xviii
  Getting Started xviii
  Scripting Additions xviii
  Other AppleScript Dialects xviii
  Scriptable Applications xviii
Conventions Used in This Guide xix

Part 1 Introducing AppleScript 1

Chapter 1 AppleScript, Scripts, and Scriptable Applications 3

What Is AppleScript? 3
What Can You Do With Scripts? 5
  Automating Activities 5
  Integrating Applications 7
  Customizing Applications 7
Who Runs Scripts, and Who Writes Them? 9
Special Features of AppleScript 10
What Applications Are Scriptable? 11

Chapter 2 Overview of AppleScript 13

How Does AppleScript Work? 14
Statements 14
Commands and Objects 17
Dictionaries 18
Values 20
Expressions 21
Operations 21
Variables 22
Script Objects 23
Scripting Additions 23
Dialects 24
Other Features and Language Elements 24
Continuation Characters 25
Comments 26
Identifiers 27
Case Sensitivity 28
Abbreviations 29
Compiling Scripts With the Script Editor 30

Part 2 AppleScript Language Reference 31

Chapter 3 Values 33

Using Value Class Definitions 33
Literal Expressions 36
Properties 36
Elements 37
Operators 37
Commands Handled 37
Reference Forms 38
Coercions Supported 38

Value Class Definitions 38
Boolean 40
Class 41
Constant 42
Data 43
Date 43
Integer 47
List 48
Number 52
Real 53
Record 54
Reference 57
String 60
Styled Text 64
Text 66
Coercing Values 67

Chapter 4 Commands 71

Types of Commands 71
Application Commands 72
AppleScript Commands 73
Scripting Addition Commands 74
User-Defined Commands 76
Using Command Definitions 77
Syntax 78
Parameters 78
Result 79
Examples 79
Errors 79
Using Parameters 80
Parameters That Specify Locations 80
Coercion of Parameters 81
Raw Data in Parameters 81
Using Results 82
Double Angle Brackets in Results and Scripts 83

Command Definitions 84
Close 87
Copy 88
Count 92
Data Size 97
Delete 98
Duplicate 99
Exists 99
Get 100
Launch 103
Make 105
Move 106
Open 107
Print 108
Quit 109
Run 110
Save 112
Set 113

Chapter 5 Objects and References 119

Using Object Class Definitions 119
Properties 120
Element Classes 120
Commands Handled 120
Default Value Class Returned 122
References 122
Containers 123
Complete and Partial References 124
Reference Forms 125
Arbitrary Element 126
Every Element 127
Filter 129
ID 130
Index 131
Middle Element 133

Name 134
Property 135
Range 136
Relative 139
Using the Filter Reference Form 140
References to Files and Applications 143
References to Files 144
References to Applications 146
References to Local Applications 147
References to Remote Applications 148

Chapter 6 Expressions 149

Results of Expressions 149
Variables 150
Creating Variables 150
Using Variables 152
The "A Reference To" Operator 153
Data Sharing 154
Scope of Variables 155
Predefined Variables 156
Script Properties 156
Defining Script Properties 157
Using Script Properties 157
Scope of Script Properties 158
AppleScript Properties 158
Text Item Delimiters 158
Reference Expressions 160
Operations 161
Operators That Handle Operands of Various Classes 168
Equal, Is Not Equal To 168
Greater Than, Less Than 172
Starts With, Ends With 173
Contains, Is Contained By 175
Concatenation 177
Operator Precedence 178
Date-Time Arithmetic 180

Chapter 7 Control Statements 183

Characteristics of Control Statements 184
Tell Statements 185
Tell (Simple Statement) 188
Tell (Compound Statement) 189
If Statements 190
If (Simple Statement) 192
If (Compound Statement) 193
Repeat Statements 194
Repeat (forever) 197
Repeat (number) Times 198
Repeat While 199
Repeat Until 200
Repeat With (loopVariable) From (startValue) To (stopValue) 201
Repeat With (loopVariable) In (list) 202
Exit 204
Try Statements 204
Kinds of Errors 205
How Errors Are Handled 206
Writing a Try Statement 206
Try 207
Signaling Errors in Scripts 210
Error 210
Considering and Ignoring Statements 213
Considering/Ignoring 214
With Timeout Statements 217
With Timeout 218
With Transaction Statements 219
With Transaction 219

Chapter 8 Handlers 221

Using Subroutines 221
Types of Subroutines 223
Scope of Subroutine Calls in Tell Statements 224
Checking the Classes of Subroutine Parameters 225

Recursive Subroutines 225
Saving and Loading Libraries of Subroutines 226
Subroutine Definitions and Calls 228
Subroutines With Labeled Parameters 229
Subroutine Definition, Labeled Parameters 229
Subroutine Call, Labeled Parameters 230
Examples of Subroutines With Labeled Parameters 232
Subroutines With Positional Parameters 235
Subroutine Definition, Positional Parameters 235
Subroutine Call, Positional Parameters 236
Examples of Subroutines With Positional Parameters 238
The Return Statement 239
Return 240
Command Handlers 241
Command Handler Definition 241
Command Handlers for Script Applications 243
Run Handlers 243
Open Handlers 246
Handlers for Stay-Open Script Applications 247
Idle Handlers 248
Quit Handlers 249
Interrupting a Script Application's Handlers 250
Calling a Script Application 251
Scope of Script Variables and Properties 252
Scope of Properties and Variables Declared at the Top Level of a Script 254
Scope of Properties and Variables Declared in a Script Object 258
Scope of Variables Declared in a Handler 263

Chapter 9 Script Objects 265

About Script Objects 265
Defining Script Objects 267
Sending Commands to Script Objects 268
Initializing Script Objects 269
Inheritance and Delegation 271

Defining Inheritance 271
How Inheritance Works 272
The Continue Statement 277
Using Continue Statements to Pass Commands to Applications 280
The Parent Property and the Current Application 281
Using the Copy and Set Commands With Script Objects 283

Appendix A The Language at a Glance 289

Commands 289
References 294
Operators 296
Control Statements 299
Handlers 301
Script Objects 303
Variable and Property Assignments and Declarations 303
Predefined Variables 304
Constants 305
Placeholders 307

Appendix B Scriptable Text Editor Dictionary 313

About Text Objects 313
Elements of Text Objects 314
Special Properties of Scriptable Text Editor Text Objects 314
Text Styles 315
AppleScript and Non-Roman Script Systems 317
Scriptable Text Editor Object Class Definitions 318
Application 318
Character 321
Document/Window 323
File 328
Insertion Point 329
Paragraph 331
Selection 334
Text 336

Text Item 339
Text Style Info 341
Window 342
Word 342
Scriptable Text Editor Commands 345
Copy 347
Cut 348
Data Size 349
Duplicate 349
Make 350
Move 351
Open 351
Paste 351
Revert 352
Save 353
Select 354
Scriptable Text Editor Errors 355

Appendix C Error Messages 357

Operating System Errors 358
Apple Event Errors 359
Apple Event Registry Errors 361
AppleScript Errors 362

Glossary 363

Index 371

Figures and Tables

Chapter 1 AppleScript, Scripts, and Scriptable Applications 3

Figure 1-1 Changing text style with the mouse and with a script 4
Figure 1-2 A script that performs a repetitive action 6
Figure 1-3 A script that copies information from one application to another 8
Figure 1-4 Different ways to run a script 9

Chapter 2 Overview of AppleScript 13

Figure 2-1 How AppleScript works 15
Figure 2-2 How AppleScript gets the Scriptable Text Editor dictionary 20

Chapter 3 Values 33

Figure 3-1 Value class definition for lists 34
Figure 3-2 Coercions supported by AppleScript 69

Table 3-1 AppleScript value class identifiers 39

Chapter 4 Commands 71

Figure 4-1 Command definition for the Move command 77
Figure 4-2 The Scriptable Text Editor document “simple” 95

Table 4-1 Standard application commands defined in this chapter 85
Table 4-2 AppleScript commands defined in this chapter 86

Chapter 5 Objects and References 119

Figure 5-1 The Scriptable Text Editor’s object class definition for paragraph objects 121
Figure 5-2 The Scriptable Text Editor document “simple” 137

Table 5-1 Reference forms 126
Table 5-2 Boolean expressions and tests in Filter references 142

Chapter 6 Expressions 149

Table 6-1 AppleScript operators 163
Table 6-2 Operator precedence 179

Chapter 8 Handlers 221

Figure 8-1 Scope of property and variable declarations at the top level of a script 254
Figure 8-2 Scope of property and variable declarations at the top level of a script object 258
Figure 8-3 Scope of variable declarations within a handler 263

Chapter 9 Script Objects 265

Figure 9-1 Relationship between a simple child script and its parent 273
Figure 9-2 Another child-parent relationship 273
Figure 9-3 A more complicated child-parent relationship 274

Appendix A The Language at a Glance 289

Table A-1 Command syntax 290
Table A-2 Reference form syntax 294
Table A-3 Container notation in references 296
Table A-4 Operators 297
Table A-5 Control statements 300
Table A-6 Handler definitions and calls 302
Table A-7 Script objects 303
Table A-8 Assignments and declarations 304
Table A-9 Predefined variables 305
Table A-10 Constants defined by AppleScript 305
Table A-11 Placeholders used in syntax descriptions 308

Appendix B Scriptable Text Editor Dictionary 313

Figure B-1 Bounds and Position properties of a Scriptable Text Editor window 327

Table B-1 Variations from standard behavior in Scriptable Text Editor versions of standard application commands 345
Table B-2 Other Scriptable Text Editor commands 347

About This Guide

The *AppleScript Language Guide: English Dialect* is a complete guide to the English dialect of the AppleScript language. AppleScript allows you to create sets of written instructions—known as scripts—to automate and customize your applications.

Audience

This guide is for anyone who wants to write new scripts or modify existing scripts.

Before using this guide, you should read *Getting Started With AppleScript* to learn what hardware and software you need to use AppleScript; how to install AppleScript; and how to run, record, and edit scripts.

To make best use of this guide, you should already be familiar with at least one of the following:

- another scripting language (such as HyperTalk, the scripting language for HyperCard, or a scripting language for a specific application)
- a computer programming language (such as BASIC, Pascal, or C)
- a macro language (such as a language used to manipulate spreadsheets)

If you're not already familiar with the basics of scripting and programming (such as variables, subroutines, and conditional statements such as If-Then), you may want additional information to help you get started. You can find a variety of introductory books on scripting and programming—including books specifically about AppleScript—in many bookstores.

Macintosh software developers who want to create scriptable and recordable applications should refer to *Inside Macintosh: Interapplication Communication*.

Organization of This Guide

This guide is divided into two parts:

- Part 1, "Introducing AppleScript," provides an overview of the AppleScript language and the tasks you can perform with it.
- Part 2, "AppleScript Language Reference," provides reference descriptions of all of the features of the AppleScript language.

Part 1 contains these chapters:

- Chapter 1, "AppleScript, Scripts, and Scriptable Applications," introduces AppleScript and its capabilities.
- Chapter 2, "Overview of AppleScript," provides an overview of the elements of the AppleScript language.

Part 2 contains the following chapters:

- Chapter 3, "Values," describes the classes of data that can be stored and manipulated in scripts and the coercions you can use to change a value from one class to another.
- Chapter 4, "Commands," describes the types of commands available in AppleScript, including application commands, AppleScript commands, scripting addition commands, and user-defined commands. It also includes descriptions of all AppleScript commands and standard application commands.
- Chapter 5, "Objects and References," describes objects and their characteristics and explains how to refer to objects in scripts.
- Chapter 6, "Expressions," describes types of expressions in AppleScript, how AppleScript evaluates expressions, and operators you use to manipulate values.

- Chapter 7, "Control Statements," describes statements that control when and how other statements are executed. It includes information about Tell, If, and Repeat statements.
- Chapter 8, "Handlers," describes subroutines, command handlers, error handlers, and the scope of variables and properties in handlers and elsewhere in a script. It includes the syntax for defining and calling subroutines and error handlers.
- Chapter 9, "Script Objects," describes how to define and use script objects. It includes information about object-oriented programming techniques such as using inheritance and delegation to define groups of related objects.

At the end of the guide are three appendixes, a glossary of AppleScript terms, and an index.

- Appendix A, "The Language at a Glance," is a collection of tables that summarize the features of the AppleScript language. It is especially useful for experienced programmers who want a quick overview of the language.
- Appendix B, "Scriptable Text Editor Dictionary," defines the words in the AppleScript language that are understood by the Scriptable Text Editor sample application.
- Appendix C, "Error Messages," lists the error messages returned by AppleScript.

Sample Applications and Scripts

A sample application, the Scriptable Text Editor, is included with AppleScript. The Scriptable Text Editor is scriptable; that is, it understands scripts written in the AppleScript language. It also supports recording of scripts: when you use the Record button in the Script Editor (the application you use to write and modify scripts), the actions you perform in the Scriptable Text Editor generate AppleScript statements for performing those actions. Scripts for performing tasks in the Scriptable Text Editor are used as examples throughout this guide.

For More Information

Getting Started

See the companion book *Getting Started With AppleScript* to learn what hardware and software you need to use AppleScript; how to install AppleScript; and how to run, record, and edit scripts.

Scripting Additions

Scripting additions are files that provide additional commands you can use in scripts. A standard set of scripting additions comes with AppleScript. Scripting additions are also sold commercially, included with applications, and distributed through electronic bulletin boards and user groups.

For information about using the scripting additions that come with AppleScript, see the companion book *AppleScript Scripting Additions Guide: English Dialect.*

Other AppleScript Dialects

A dialect is a version of the AppleScript language that resembles a particular language. This guide describes the English dialect of AppleScript (also called AppleScript English). This dialect uses words taken from the English language and has an English-like syntax. Other dialects can use words from other human languages, such as Japanese, and have a syntax that resembles a specific human language or programming language.

For information about a specific dialect, see the version of the *AppleScript Language Guide* for that dialect.

Scriptable Applications

Not all applications are scriptable. The advertising and packaging for an application usually mention if it is scriptable. The documentation for a scriptable application typically lists the AppleScript words that the application understands.

Conventions Used in This Guide

Words and sample scripts in `monospaced font` are AppleScript language elements that must be typed exactly as shown. Terms are shown in **boldface** where they are defined**.** You can also find these definitions in the glossary.

Here are some additional conventions used in syntax descriptions:

`language element`

Plain computer font indicates an element that you must type exactly as shown. If there are special symbols (for example, + or &), you must also type them exactly as shown.

placeholder — Italic text indicates a placeholder that you must replace with an appropriate value. (In some programming languages, placeholders are called nonterminals.)

[optional] — Brackets indicate that the enclosed language element or elements are optional.

(a group) — Parentheses group together elements. If parentheses are part of the syntax, they are shown in bold.

[optional]... — Three ellipsis points (. . .) after a group defined by brackets indicate that you can repeat the group of elements within brackets 0 or more times.

(a group)... — Three ellipsis points (. . .) after a group defined by parentheses indicate that you can repeat the group of elements within parentheses one or more times.

a | b | c — Vertical bars separate elements in a group from which you must choose a single element. The elements are often grouped within parentheses or brackets.

PART ONE

Introducing AppleScript

AppleScript, Scripts, and Scriptable Applications

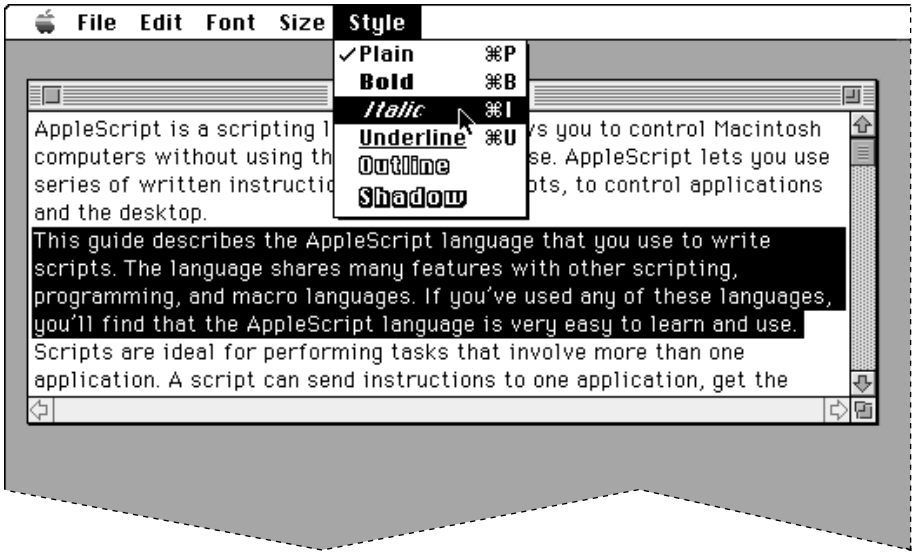
This chapter introduces the AppleScript scripting language. It answers these questions:

- What is AppleScript?
- What are scripts?
- Who runs scripts, and who writes them?
- How is AppleScript different from other scripting mechanisms?
- What can you do with scripts?
- What applications are scriptable?

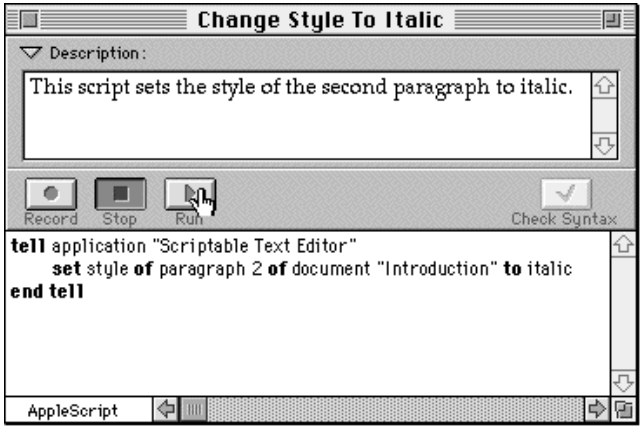
What Is AppleScript?

AppleScript is a scripting language that allows you to control Macintosh computers without using the keyboard or mouse. AppleScript lets you use series of written instructions, known as **scripts,** to control applications and the desktop. Figure 1-1 shows the difference between changing the text style of a paragraph with the mouse and performing the same task with a script.

Figure 1-1 Changing text style with the mouse and with a script



Changing the style of text with the mouse



Changing the style of text with a script

The script shown at the bottom of Figure 1-1 is written in AppleScript English, which is a **dialect** of the AppleScript scripting language that resembles English. This guide describes AppleScript English and how you can use it to write scripts. Other dialects, such as AppleScript Japanese and AppleScript French, are designed to resemble other human languages. Still others, such as the Programmer's Dialect, resemble other programming languages. For information about dialects other than AppleScript English, see the guide for the dialect you want to use. For information about installing dialects, see *Getting Started With AppleScript*.

All AppleScript dialects share many features with other scripting, programming, and macro languages. If you've used any of these languages, you'll find AppleScript dialects very easy to learn and use.

AppleScript comes with an application called **Script Editor** that you can use to create and modify scripts. You can also use Script Editor to translate scripts from one AppleScript dialect to another.

What Can You Do With Scripts?

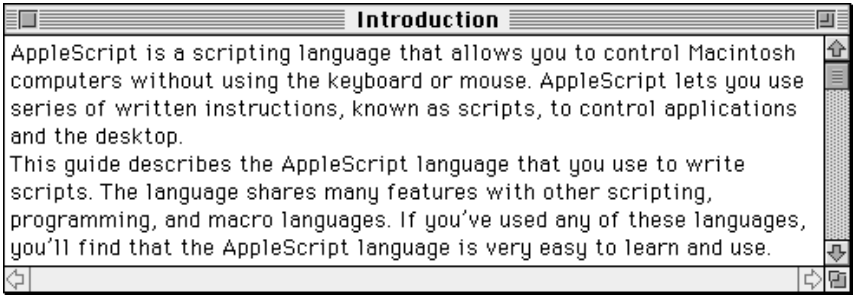
AppleScript lets you automate, integrate, and customize applications. The following sections provide examples.

Automating Activities

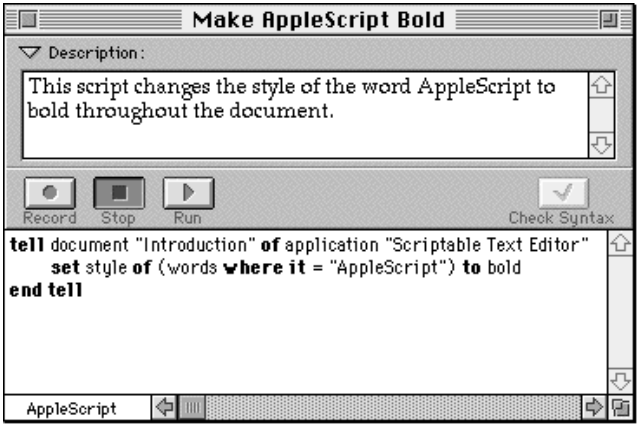
Scripts make it easy to perform repetitive tasks. For example, if you want to change the style of the word "AppleScript" to bold throughout a document named Introduction, you can write a script that does the job instead of searching for each occurrence of the word, selecting it, and changing it from the Style menu.

Figure 1-2 shows the script and what happens when you run it.

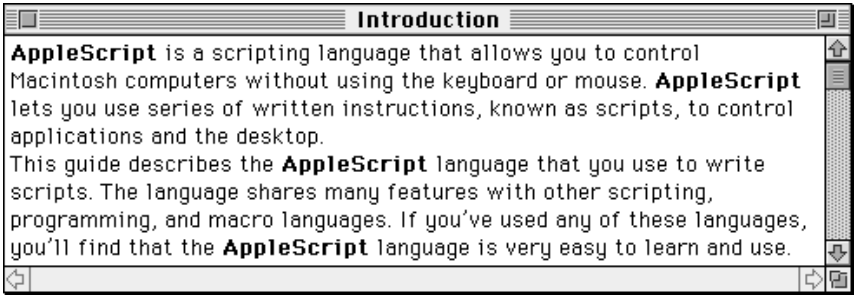
Figure 1-2 A script that performs a repetitive action



Introduction before running script



Make AppleScript Bold script



Introduction after running script

Integrating Applications

Scripts are ideal for performing tasks that involve more than one application. A script can send instructions to one application, get the resulting data, and then pass the data on to one or more additional applications. For example, a script can collect information from a database application and copy it to a spreadsheet application. Figure 1-3 shows a simple script that gets a value from the Count cell of an inventory database and copies it to the Inventory column of a spreadsheet.

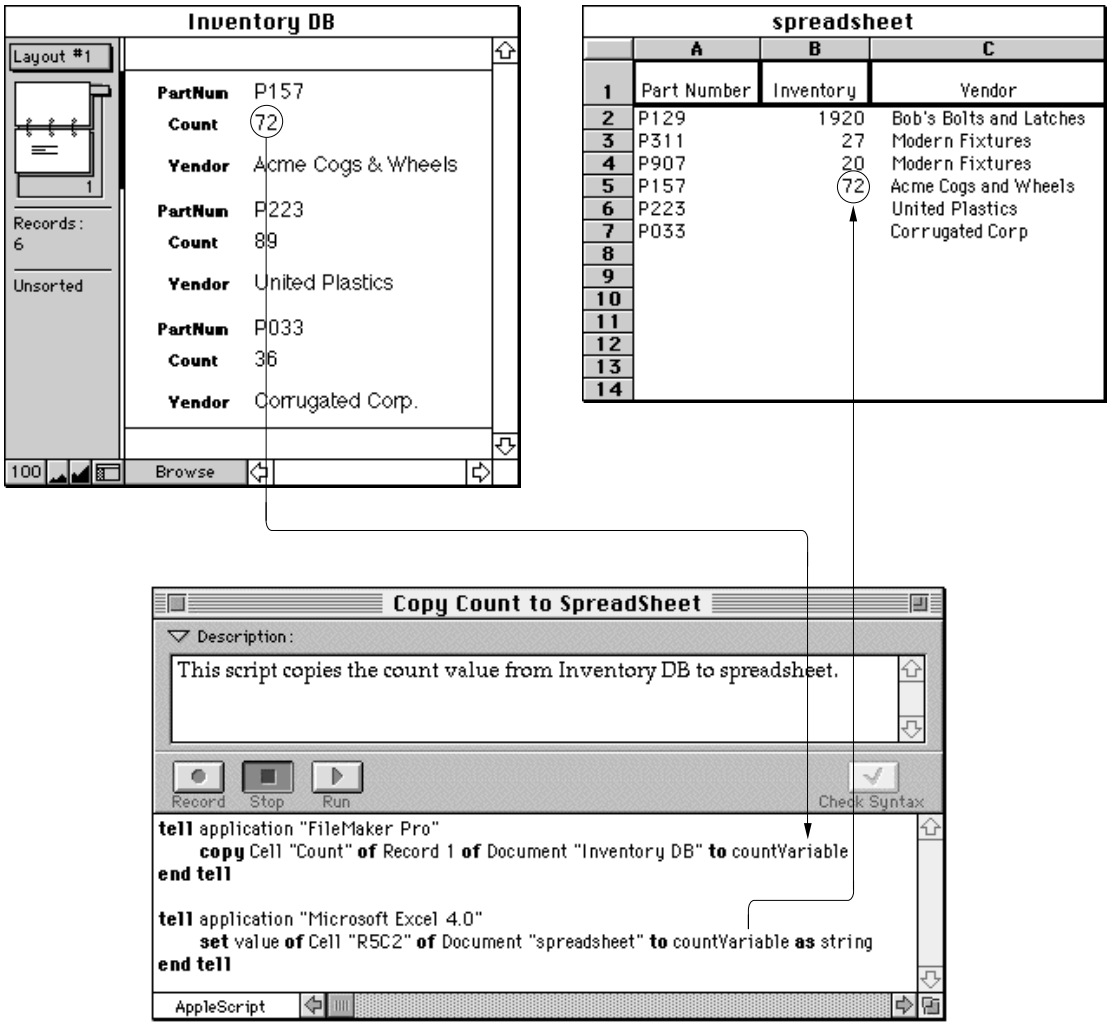
In the same way, a script can use one application to perform an action on data from another application. For example, suppose a word-processing application includes a spelling checker and also supports an AppleScript command to check spelling. You can check the spelling of a block of text from any other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

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

Customizing Applications

Scripts can add new features to applications. To customize an application, you add a script that is triggered by a particular action within the application, such as choosing a menu item or clicking a button. Whether you can add scripts to applications is up to each application, as are the ways you associate scripts with specific actions.

Figure 1-3 A script that copies information from one application to another

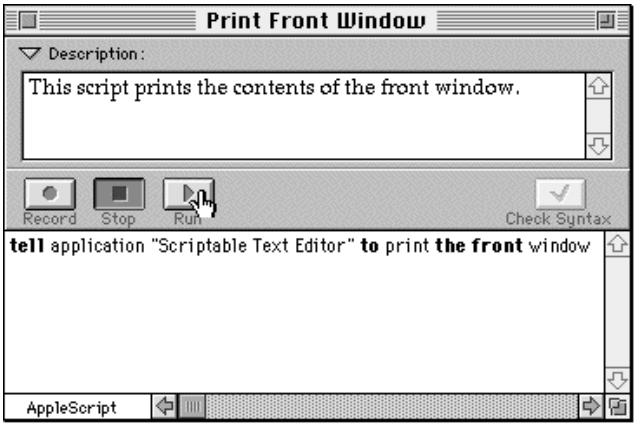


Who Runs Scripts, and Who Writes Them?

To run a script is to cause the actions the script describes to be performed. Everyone who uses a Macintosh computer can run scripts. Figure 1-4 illustrates two ways to run a script.

Figure 1-4 Different ways to run a script

Double-clicking a script application's icon



Clicking the Run button

If the script is a script application on the desktop, you can run it by double-clicking its icon. You can also run any script by clicking the Run button in the Script Editor window for that script.

Although everyone can run scripts, not everyone needs to write them. One person who is familiar with a scripting language can create sophisticated scripts that many people can use. For example, management information specialists in a business can write scripts for everyone in the business to use. Scripts are also sold commercially, included with applications, and distributed through electronic bulletin boards and user groups.

Special Features of AppleScript

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

- AppleScript makes it easy to refer to data within applications. Scripts can use familiar names to refer to familiar objects. For example, a script can refer to paragraph, word, and character objects in a word-processing document and to row, column, and cell objects in a spreadsheet.
- You can control several applications from a single script. Although many applications include built-in scripting or macro languages, most of these languages work for only one application. In contrast, you can use AppleScript to control any of the applications that support it. You don't have to learn a new language for each application.
- You can write scripts that control applications on more than one computer. A single script can control any number of applications, and the applications can be on any computer on a given network.
- You can create scripts by recording. The Script Editor application includes a recording mechanism that takes much of the work out of creating scripts. When recording is turned on, you can perform actions in a recordable application and the Script Editor creates corresponding instructions in the AppleScript language. To learn how to turn recording on and off, refer to *Getting Started With AppleScript*.
- AppleScript supports multiple dialects, or representations of the AppleScript language that resemble various human languages and programming languages. This guide describes the AppleScript English dialect. You can use Script Editor to convert a script from one dialect to another without changing what happens when you run the script.

What Applications Are Scriptable?

Applications that understand one or more AppleScript commands are called **scriptable applications.** Not all applications are scriptable. The advertising and packaging for an application usually mention if it is scriptable. The documentation for a scriptable application typically lists the AppleScript words that the application understands.

Some scriptable applications are also **recordable.** For every significant action you can perform in a recordable application, the Script Editor can record a series of corresponding instructions in the AppleScript language. With recordable applications, you can create a script simply by performing actions in the application.

Finally, some scriptable applications are also attachable. An **attachable application** is one that can be customized by attaching scripts to specific objects in the application, such as buttons and menu items. These scripts are triggered by specific user actions, such as choosing a menu item or clicking a button.

Overview of AppleScript

AppleScript is a dynamic, object-oriented script language. At its heart is the ability to send commands to objects in many different applications. These objects, which are familiar items such as words or paragraphs in a text-editing application or shapes in a drawing application, respond to commands by performing actions. AppleScript determines dynamically—that is, whenever necessary—which objects and commands an application recognizes based on information it obtains from each scriptable application.

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

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

This chapter provides an overview of AppleScript. It includes a summary of how AppleScript works and brief descriptions of the AppleScript language elements. Part 2 of this book, “AppleScript Language Reference,” describes the elements of the AppleScript language in more detail.

How Does AppleScript Work?

AppleScript works by sending messages, called **Apple events,** to applications. When you write a script, you write one or more groups of instructions called **statements.** When you run the script, the Script Editor sends these statements to the AppleScript extension, which interprets the statements and sends Apple events to the appropriate applications. Figure 2-1 shows the relationship between the Script Editor, the AppleScript extension, and the application.

The parts that you use—the Script Editor and the application—are shown to the left of the dotted line in Figure 2-1. The parts that work behind the scenes—the AppleScript extension and Apple events—are shown to the right of the dotted line.

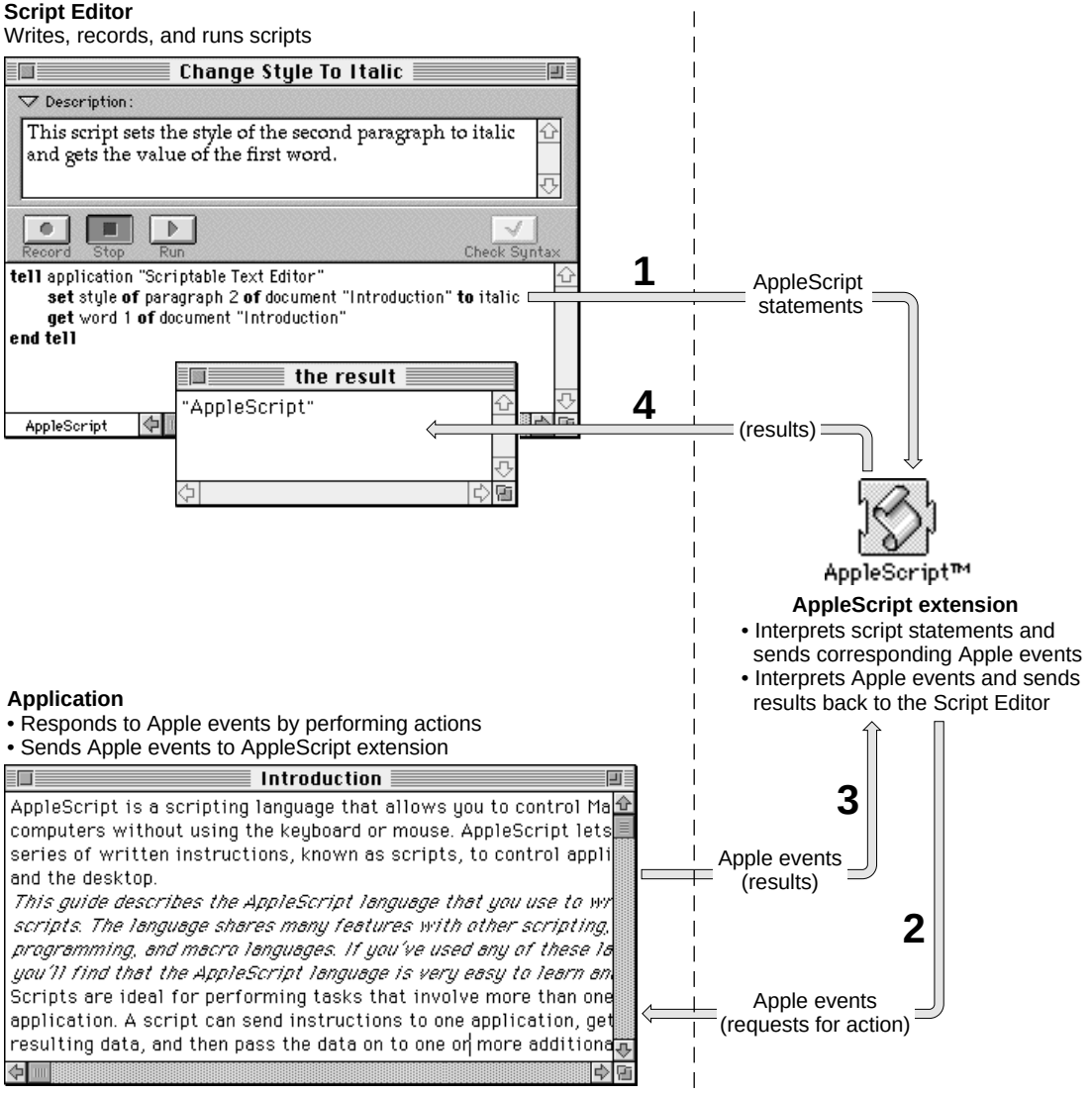
Applications respond to Apple events by performing actions, such as changing a text style, getting a value, or opening a document. Applications can also send Apple events back to the AppleScript extension to report results. The AppleScript extension sends the final results to the Script Editor, where they are displayed in the result window.

When you write scripts, you needn't be concerned about Apple events or the AppleScript extension. All you need to know is how to use the AppleScript language to request the actions or results that you want.

Statements

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

Figure 2-1 How AppleScript works



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

```
tell application "Scriptable Text Editor" to print the front window
```

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

```
tell application "Scriptable Text Editor"
   print the front window
end tell
```

The compound Tell statement includes the lines `tell application "Scriptable Text Editor"` and `end tell`, and all statements between these two lines.

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

```
tell application "Scriptable Text Editor"
   print front window
   close front window
end tell
```

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

Note

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

Here's another example of a compound statement:

```
if the number of windows is greater than 0 then
   print front window
end if
```

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

```
tell application "Scriptable Text Editor"
if the number of windows is greater than 0 then
   print front window
end if
end tell
```

Commands and Objects

Commands are the words or phrases you use in AppleScript statements to request actions or results. Every command is directed at a target, which is the object that responds to the command. The target of a command is usually an application object. **Application objects** are objects that belong to an application, such as windows, or objects in documents, such as the words and paragraphs in a text document. Each application object has specific information associated with it and can respond to specific commands.

For example, in the Scriptable Text Editor, window objects understand the Print command. The following example shows how to use the Print command to request that the Scriptable Text Editor print the front window.

```
tell application "Scriptable Text Editor"
   print front window
end tell
```

The Print command is contained within a Tell statement. Tell statements specify default targets for the commands they contain. The **default target** is the object that receives commands if no other object is specified or if the object is

specified incompletely in the command. In this case, the statement containing the Print statement does not contain enough information to uniquely identify the window object, so AppleScript uses the application name listed in the Tell statement to determine which object receives the Print command.

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

```
front window of application "Scriptable Text Editor"
```

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

Objects can contain other objects, called **elements.** In the previous example, the front window is an element of the Scriptable Text Editor application object. Similarly, in the next example, a word element is contained in a specific paragraph element, which is contained in a specific document.

```
word 1 of paragraph 3 of document "Try This"
```

Every object belongs to an **object class,** which is simply a name for objects with similar characteristics. Among the characteristics that are the same for the objects in a class are the commands that can act on the objects and the elements they can contain. An example of an object class is the Document object class in the Scriptable Text Editor. Every document created by the Script Editor belongs to the Document object class. The Script Editor's definition of the document object class determines which classes of elements, such as paragraphs and words, a document object can contain. The definition also determines which commands, such as the Close command, a document object can respond to.

Dictionaries

To examine a definition of an object class, a command, or some other word supported by an application, you can open that application's dictionary from the Script Editor. A **dictionary** is a set of definitions for words that are understood by a particular application. Unlike other scripting languages,

AppleScript does not have a single fixed set of definitions for use with all applications. Instead, when you write scripts in AppleScript, you use both definitions provided by AppleScript and definitions provided by individual applications to suit their capabilities.

Dictionaries tell you which objects are available in a particular application and which commands you can use to control them. Typically, the documentation for a scriptable application includes a complete list of the words in its dictionary. For example, Appendix B of this book contains a complete list of the words in the Scriptable Text Editor dictionary. In addition, if you are using the Script Editor, you can view the list of commands and objects for a particular application in a Dictionary window. For more information, see *Getting Started With AppleScript*.

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

```
tell application "Scriptable Text Editor"
   print front window
   close front window
end tell
```

AppleScript reads the words in the application's dictionary at the beginning of the Tell statement and uses them to interpret the statements in the Tell statement. For example, AppleScript uses the words in the Scriptable Text Editor dictionary to interpret the Print and Close commands in the Tell statement shown in the example.

Another way to use an application's dictionary is to specify the application name completely in a simple statement:

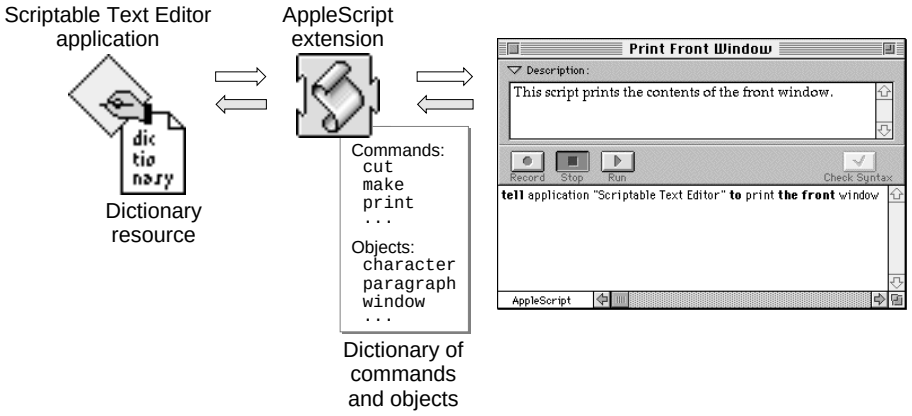
```
print front window of application "Scriptable Text Editor"
```

In this case, AppleScript uses the words in the Scriptable Text Editor dictionary to interpret the words in this statement only.

When you use a Tell statement or specify an application name completely in a statement, the AppleScript extension gets the dictionary resource for the application and reads its dictionary of commands, objects, and other words. Every scriptable application has a dictionary resource that defines the commands, objects, and other words script writers can use in scripts to control

the application. Figure 2-2 shows how AppleScript gets the words in the Scriptable Text Editor's dictionary.

Figure 2-2 How AppleScript gets the Scriptable Text Editor dictionary



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

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

Values

A **value** is a simple data structure that can be represented, stored, and manipulated within AppleScript. AppleScript recognizes many types of values, including character strings, real numbers, integers, lists, and dates. Values are fundamentally different from application objects, which can be manipulated from AppleScript, but are contained in applications or their documents. Values can be created in scripts or returned as results of commands sent to applications.

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

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

Expressions

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

Two common types of expressions are operations and variables. An **operation** is an expression that derives a new value from one or two other values. A **variable** is a named container in which a value is stored. The following sections introduce operations and variables. For more information about these and other types of expressions, see Chapter 6, "Expressions."

Operations

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

```
3 + 4                              --value: 7

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

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

You can use operations within AppleScript statements, such as:

```
tell application "Scriptable Text Editor"
   delete word 3 + 4 of document "Test"
end tell
```

When you run this script, AppleScript evaluates the expression `3 + 4` and uses the result to determine which word to delete.

Variables

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

```
copy "Mitch" to myName
```

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

```
set myName to "Mitch"
```

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

You can retrieve the value in a variable with a Get command. Run the following script and then display the result:

```
set myName to "Mitch"
get myName
```

You see that the value in `myName` is the value you stored with the Set command.

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

```
set myName to "Mitch"
set myName to "Pegi"
get myName
```

AppleScript does not distinguish uppercase letters from lowercase variables in variable names; the variables `myName`, `myname`, and `MYNAME` all represent the same value.

Script Objects

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

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

Scripting Additions

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

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

A single scripting addition file can contain several commands. For example, the File Commands scripting addition includes the commands Path To, List Folder, List Disks, and Info For. The scripting additions provided by Apple Computer, Inc., are described in the book *AppleScript Scripting Additions Guide.* Scripting additions are also sold commercially, included with applications, and distributed through electronic bulletin boards and user groups.

Dialects

AppleScript scripts can be displayed in several different **dialects,** or representations of AppleScript that resemble human languages or programming languages. The dialects available on a given computer are determined by the Dialects folder, a folder in the Scripting Additions folder (which in turn is located in the Extensions folder of the System Folder) that contains one dialect file for each AppleScript dialect installed on your computer.

You can select any of the available dialects from the Script Editor. You can tell which dialects are available by examining the pop-up menu in the lower-left corner of a Script Editor window. You can change the dialect in which a script is displayed by selecting a different dialect from the pop-up menu. The behavior of a script when you run it is not affected by the dialect in which it is displayed.

For more information about selecting dialects and formatting options from the Script Editor, see *Getting Started With AppleScript.*

Other Features and Language Elements

So far, you've been introduced to the key elements of the AppleScript language, including statements, objects, commands, expressions, and script objects. The reference section of this guide discusses these elements in more detail and describes how to use them in scripts. Before you continue to the reference section, however, you'll need to know about a few additional elements and features of the AppleScript scripting language that are not described in the reference:

- continuation characters
- comments
- identifiers
- case sensitivity
- abbreviations
- compiling scripts

Continuation Characters

A simple AppleScript statement must normally be on a single line. If a statement is longer than will fit on one line, you can extend it by including a **continuation character**, ¬ (Option-L or Option-Return), at the end of one line and continuing the statement on the next. For example, the statement

```
delete word 1 of paragraph 3 of document "Learning AppleScript"
```

can appear on two lines:

```
delete word 1 of paragraph 3 of document ¬
   "Learning AppleScript"
```

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

```
--this statement causes an error:
delete word 1 of paragraph 3 of document "Fundamentals ¬
   of Programming"
```

Note
The characters `--` in the example indicate that the first line is a comment. A comment is text that is ignored by AppleScript when a script is run. Comments are added to help you understand scripts. They are explained in the next section, "Comments." ◆

If a string extends beyond the end of the line, you can continue typing without pressing Return (the text never wraps to the next line), or you can break the string into two or more strings and use the concatenation operator (&) to join them:

```
delete word 1 of paragraph 3 of document "Fundamentals " ¬
& "of Programming"
```

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

Comments

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

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

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

Here are some sample comments:

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

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

copy result to theCount--stores the result in theCount

(* The following subroutine, findString, searches for a
string in a list of Scriptable Text Editor files *)

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

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

```
--the following block comment is illegal
tell application "Scriptable Text Editor"
   get (* word 1 of *) paragraph 1 of front document
end tell
```

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

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

If you later remove `(*` and `*)`, the handler is once again available.

Identifiers

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

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

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

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

```
Yes
Agent99
Just_Do_It
```

The following are not valid identifiers:

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

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

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

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

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

Identifiers cannot be the same as any reserved words—that is, words in the system dictionary or words in the dictionary of the application named in the Tell statement. For example, you cannot create a variable whose identifier is `Yes` within a Tell statement to the Scriptable Text Editor, because `Yes` is a constant from the Scriptable Text Editor dictionary. In this case, AppleScript returns a syntax error if you use `Yes` as a variable identifier.

Case Sensitivity

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

The one exception to this rule is string comparisons. Normally, AppleScript does not distinguish uppercase from lowercase letters when comparing strings, but if you want AppleScript to consider case, you can use a special statement called a Considering statement. For more information, see “Considering and Ignoring Statements” on page 213.

Most of the examples in this chapter and throughout this guide are in lowercase letters. Sometimes words are capitalized to improve readability. For example, in the following variable assignment, the “N” in `myName` is capitalized to make it easier to see that two words have been combined to form the name of the variable.

```
set myName to "Pegi"
```

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

```
MYNAME
myname
MyName
mYName
```

When interpreting strings, such as `"Pegi"`, AppleScript preserves the case of the letters in the string, but does not use it in comparisons. For example, the value of the variable `myName` defined earlier is always `"Pegi"`, but the value of the expression `myName = "PEGI"` is `true`.

Abbreviations

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

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

```
tell application "Scriptable Text Editor"
   print the front window
end tell

tell app "Scriptable Text Editor"
   print the front window
end tell
```

Compiling Scripts With the Script Editor

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

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

PART TWO

AppleScript Language Reference

Values

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

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

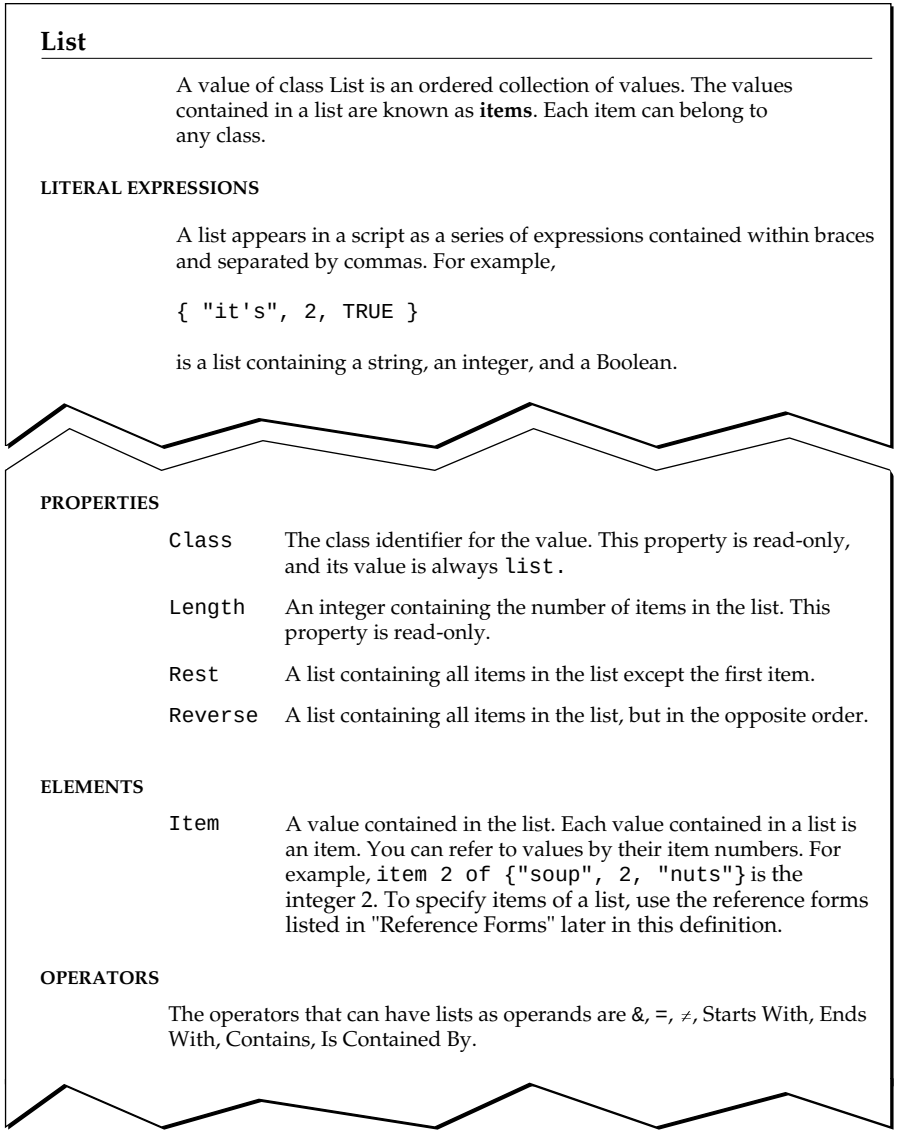
This chapter describes how to interpret value class definitions, discusses the common characteristics of all value classes, and presents definitions of the value classes supported in AppleScript. It also describes how to coerce values.

Using Value Class Definitions

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

Figure 3-1 shows the definition for the List value class, a composite value. The figure shows seven types of information: examples, properties, elements, operators, commands handled, reference forms, and coercions supported. The sections following the figure explain each type of information. Some definitions end with notes (not shown in Figure 3-1) that provide additional information.

Figure 3-1 Value class definition for lists



List

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

LITERAL EXPRESSIONS

A list appears in a script as a series of expressions contained within braces and separated by commas. For example,

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

is a list containing a string, an integer, and a Boolean.

PROPERTIES

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

ELEMENTS

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

OPERATORS

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

Figure 3-1 Value class definition for lists (continued)

COMMANDS HANDLED

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

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

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

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

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

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

REFERENCE FORMS

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

- *Property*. For example, `class of {"this", "is", "a", "list"}` specifies `list`.
- *Index*. For example, `item 3 of {"this", "is", "a", "list"}` specifies `"a"`.

COERCIONS SUPPORTED

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

AppleScript also supports coercion of an entire list to a string if all items in the list can be coerced to a string. The resulting string concatenates all the items:

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

Literal Expressions

A **literal expression** is an expression that evaluates to itself. The "Literal Expressions" section of a value class definition shows examples of how values of a particular class are represented in AppleScript—that is, typical literal expressions for values of that class. For example, in AppleScript and many other programming languages, the literal expression for a string is a series of characters enclosed in quotation marks. The quotation marks are not part of the string value; they are a notation that indicates where the string begins and ends. The actual string value is a data structure stored in AppleScript.

The sample value class definition in Figure 3-1 shows literal expressions for list values. As with the quotation marks in a string literal expression, the braces that enclose a list and the commas that separate its items are not part of the actual list value; they are notations that represent the grouping and items of the list.

Properties

A property of a value is a characteristic that is identified by a unique label and has a single value. Simple values have only one property, called Class, that identifies the class of the value. Composite values have a Class property, a Length property, and in some cases additional properties.

Use the Name reference form to specify properties of values. For example, the following reference specifies the Class property of an integer.

```
class of 101
--result: integer
```

The following reference specifies the Length property of a list.

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

You can optionally use the Get command with the Name reference form to get the value of a property for a specified value. In most cases, you can also use the Set command to set the additional properties listed in the definitions of composite values. If a property cannot be set with the Set command, its definition specifies that it is read-only.

Elements

Elements of values are values contained within other values. Composite values have elements; simple values do not. The sample value class definition in Figure 3-1 shows one element, called an item.

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

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

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

Operators

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

For complete descriptions of operators and how to use them in expressions, see "Operations," which begins on page 161.

Commands Handled

Commands are requests for action. Simple values cannot respond to commands, but composite values can. For example, lists can respond to the Count command, as shown in the following example.

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

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

Reference Forms

A reference is a compound name for an object or a value. You can use references to specify values within composite values or properties of simple values. You cannot use references to refer to simple values.

The "Reference Forms" section is included in composite value class definitions only. It lists the reference forms you can use to specify elements of a composite value. For complete descriptions of the AppleScript reference forms, see Chapter 5, "Objects and References."

Coercions Supported

AppleScript can change a value of one class into a value of another class. This is called **coercion.** The "Coercions Supported" section of a value class definition describes the classes to which values of that class can be coerced.

Because a list consists of one or more values, any value can be added to a list or coerced to a single-value list. The definition in Figure 3-1 also lists the value classes to which individual items in a list can be coerced.

For more information about coercions, see "Coercing Values," which begins on page 68. For a summary of the coercions provided by AppleScript, see Figure 3-2 on page 70.

Value Class Definitions

This section describes the AppleScript value classes. Table 3-1 summarizes the class identifiers recognized by AppleScript.

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

Table 3-1 AppleScript value class identifiers

Value class identifier	Description of corresponding value
Boolean	A logical truth value
Class	A class identifier
Constant	A reserved word defined by an application or AppleScript
Data	Raw data that cannot be represented in AppleScript, but can be stored in a variable
Date	A string that specifies a day of the week, day of the month, month, year, and time
Integer	A positive or negative number without a fractional part
List	An ordered collection of values
Number	Synonym for class Integer or class Real; a positive or negative number that can be either of class Integer or of class Real
Real	A positive or negative number that can have a fractional part
Record	A collection of properties
Reference	A reference to an object
String	An ordered series of characters
Styled Text	Synonym for a special string that includes style and font information
Text	Synonym for class string

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

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

Although definitions for value class synonyms are included in the sections that follow, they do not correspond to separate value classes. For more information about coercing values using synonyms, see “Coercing Values,” which begins on page 68.

Boolean

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

LITERAL EXPRESSIONS

```
true

false
```

PROPERTY

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

ELEMENTS

None

OPERATORS

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

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

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

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

COERCIONS SUPPORTED

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

Class

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

LITERAL EXPRESSIONS

```
string
integer
real
boolean
class
```

PROPERTY

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

ELEMENTS

None

OPERATORS

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

The operator As takes a value of one class and coerces it to a value of a class specified by a class identifier. For example,

```
"1.5" as real
```

coerces a string into the corresponding real number, `1.5`. For more information about coercing values, see Chapter 6, "Expressions."

COERCIONS SUPPORTED

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

Constant

A value of class **Constant** is a reserved word defined by AppleScript or an application in its dictionary. Applications define sets of values that can be used for parameters of a particular command; for example, the value of the `saving` parameter of a Close command must be one of the three constants `yes`, `no`, and `ask`.

LITERAL EXPRESSIONS

```
yes
no
ask
```

PROPERTY

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

ELEMENTS

None

OPERATORS

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

COERCIONS HANDLED

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

NOTES

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

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

Data

A value of class **Data** is data returned by an application (in response to a command) that does not belong to any of the other value classes defined in this section. A value of class Data is raw data that can only be stored in a variable.

PROPERTY

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

ELEMENTS

None

OPERATORS

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

COERCIONS SUPPORTED

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

Date

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

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

LITERAL EXPRESSIONS

```
date "12/5/54, 12:06 PM"

date "12/05/53, 12:06"

date "12/05/54"

date "12:06"

date "Sunday, December 5, 1954 12:06 pm"
```

PROPERTIES

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

ELEMENTS

None

OPERATORS

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

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

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

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

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

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

Here's an example:

```
date "Apr 15, 1992" + 4 * days + 3 * hours + 2 * minutes
```

For more information about the way AppleScript operators treat Date values, see "Date-Time Arithmetic," which begins on page 180.

REFERENCE FORMS

You can refer to properties of a Date value using the Property reference form.

```
weekday of date "May 3, 1993"
--result: Monday

time of date "May 3, 1993"
--result: "12:00 AM"
```

Note that AppleScript fills in a default time property for the date specified in the second example.

If you want to specify a time relative to a date, you can do so as follows:

```
date "2:30 am" of date "May 3, 1993"
--result: date "May 3, 1993 2:30 AM"

date "Sept. 27, 1993" relative to date "3PM"
--result: date "September 27, 1993 3:00 PM"
```

In addition to `of`, you can also use the synonyms `relative to` or `in` for this purpose.

COERCIONS SUPPORTED

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

NOTES

Regardless of the format you use when you type a date in a script, AppleScript always displays Date values in the format shown in the following example, which includes the full name of the day of the week and month and no leading zeros for the date.

```
date "Sunday, January 3, 1992 12:05 PM"
```

If you don't specify a complete date, day, and time when typing a Date value, AppleScript fills in information as needed. If you don't specify the date information, AppleScript uses the date when the script is compiled. If you

don't specify the time information, 12:00 AM (midnight) is the default. If you omit AM or PM, AM is the default; however, if you specify 12:00 without AM or PM, 12:00 PM is the default. If you specify the time using 24-hour time, AppleScript converts it to the equivalent time using AM or PM; for example, 17:00 is equivalent to 5:00 PM.

To get the current date, use the scripting addition command Current Date. For example,

```
if current date = date "Sunday, January 23, 1992 12:05 PM"
   then print the front window
end if
```

For a complete description of the Current Date command, see the *AppleScript Scripting Additions Guide.*

Integer

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

LITERAL EXPRESSIONS

```
1
2
-1
1000
```

PROPERTY

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

ELEMENTS

None

OPERATORS

The Div operator always returns an integer as its result. The +, –, *, Mod, and ^ operators return integers or real numbers.

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

COERCIONS SUPPORTED

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

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

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

NOTES

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

List

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

LITERAL EXPRESSIONS

A list appears in a script as a series of expressions contained within braces and separated by commas. For example,

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

is a list containing a string, an integer, and a Boolean.

Each list item can be any valid expression. For example,

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

has the same value as the list in the previous example, because each of the expressions it contains has the same value as the corresponding expression in the previous example.

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

```
{}
```

PROPERTIES

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

ELEMENT

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

OPERATORS

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

For detailed explanations and examples of how AppleScript operators treat lists, see "Operators That Handle Operands of Various Classes," which begins on page 168.

COMMANDS HANDLED

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

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

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

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

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

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

REFERENCE FORMS

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

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

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

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

COERCIONS SUPPORTED

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

AppleScript also supports coercion of an entire list to a string if all items in the list can be coerced to a string. The resulting string concatenates all the items:

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

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

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

NOTES

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

```
{"This"} & {"is", "a", "list"}
```

results in

```
{"This", "is", "a", "list"}
```

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

Number

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

LITERAL EXPRESSIONS

```
1
2
-1
1000

10.2579432
1.0
1.
```

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

PROPERTY

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

ELEMENTS

None

OPERATORS

Because values identified as values of class Number are really values of either class Integer or class Real, the operators available are the operators described in the definitions of the Integer or Real value classes, beginning on page 48 and page 54, respectively.

COERCIONS SUPPORTED

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

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

Real

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

LITERAL EXPRESSIONS

```
10.2579432
1.0
1.
```

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

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

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

PROPERTY

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

ELEMENTS

None

OPERATORS

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

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

COERCIONS SUPPORTED

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

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

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

NOTES

Real numbers that are greater than or equal to 10,000.0 or less than or equal to 0.0001 are converted to exponential notation when scripts are compiled. The largest value that can be evaluated (positive or negative) is 1.79769e+308.

Record

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

LITERAL EXPRESSIONS

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

```
{ name:"Mitchell", height:70.5, weight:165 }
```

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

AppleScript evaluates expressions in a record before using the record in other expressions. For example, the record

```
{ name:"Mitchell", height:72 - 1.5, weight:150 + 15 }
```

is equivalent to

```
{ name:"Mitchell", height:70.5, weight:165 }
```

PROPERTIES

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

`Class` The class identifier for the object. For most records, the value of the Class property is `record`. However, the Class property of a record is not read-only. AppleScript and applications use special records for certain data. For example, the Scriptable Text Editor uses special records to specify the styles (such as `bold` and `underline`) of text objects. The value of the Class property for these records is the class identifier Text Style Info, as illustrated in this example:

```
{class:Text Style Info, On Styles:{Bold}, ¬
    Off Styles:{ Italic, Outline, Shadow}}
```

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

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

OPERATORS

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

For detailed explanations and examples of how AppleScript operators treat records, see "Operators That Handle Operands of Various Classes," which begins on page 168.

COMMANDS HANDLED

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

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

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

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

REFERENCE FORMS

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

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

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

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

COERCIONS SUPPORTED

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

NOTES

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

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

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

```
name of writer
```

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

After you define a record, you cannot add additional properties to it. You can, however, concatenate records. For more information, see "Concatenation" on page 177.

Reference

A value of class **Reference** is a reference to an object. You can create a value of class Reference by using the A Reference To operator. In addition, applications can return references in response to commands.

A value of class Reference is different from the value of the object to which a reference refers. For example, the reference

```
word 1 of front window of app "Scriptable Text Editor"
--result: a string
```

refers to a word object, whose value is a string, such as `"Today"`. But a value of class Reference created with the A Reference To operator is a structure within AppleScript that refers (or points) to a specific object.

```
a reference to word 1 of front window of app ¬
   "Scriptable Text Editor"
--result: word 1 of window 1 of application ¬
   "Scriptable Text Editor"
```

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

Values of class Reference are similar to pointers in other programming languages, but unlike pointers, references can refer only to objects. For more information about creating values of class Reference, see "The 'A Reference To' Operator" on page 153.

LITERAL EXPRESSIONS

```
word 1 of document "Report"

window "Graph"
```

PROPERTIES

`Class`	The class identifier for the object. This property is read-only, and its value is always `reference`.
`Contents`	The value of the object to which the reference refers. The class of the value depends on the reference. For information about how to use the Contents property, see "The 'A Reference To' Operator" on page 153.

ELEMENTS

None

OPERATORS

The A Reference To operator returns a reference as its result.

COERCIONS SUPPORTED

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

NOTES

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

For example, the reference in the following example functions as a reference to an object. It identifies the object to which the Copy command is sent.

```
copy word 1 of front document of application "Scriptable Text Editor"
```

On the other hand, the reference in the following example functions as a reference expression:

```
repeat (word 1 of front document of application ¬
   "Scriptable Text Editor") times
      display dialog "Hello"
end repeat
```

When AppleScript executes the statement, it gets the value of the reference `word 1 of front document of application "Scriptable Text Editor"`—a string—and then coerces it to an integer, if possible. (For information about the Repeat statement, see “Repeat Statements,” which begins on page 194. For information about coercions, see “Coercing Values” on page 68.)

String

A value of class **String** is a character string (an ordered series of characters) in AppleScript.

LITERAL EXPRESSIONS

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

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

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

PROPERTIES

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

ELEMENTS

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

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

Character	A single character contained in the string.
Paragraph	A series of characters ending with either (1) a return character or (2) the end of the string and beginning immediately after either (1) the first character after the end of the preceding paragraph or (2) the beginning of the string.
Text	A continuous series of characters, including spaces, tabs, and all other characters, within a string (see "Notes" later in this section).

Word — A continuous series of characters that contains only the following types of characters:

letters (including letters with diacritical marks)

digits

nonbreaking spaces

dollar signs, cent signs, English pound symbols, or yen symbols

percent signs

commas between digits

periods before digits

apostrophes between letters or digits

hyphens (but not minus signs [Option-hyphen] or dashes [Option-Shift-hyphen]).

Here are some examples of words:

```
non-functional
he's
v1.0
$99.99
12c-d
```

Note that this definition applies to English text in the Roman script system. Words in other languages are defined by the script system for each language if the appropriate script system is installed. (For more information about script systems, see page 317.)

OPERATORS

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

For detailed explanations and examples of how AppleScript operators treat strings, see "Operators That Handle Operands of Various Classes," which begins on page 168.

REFERENCE FORMS

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

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

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

SPECIAL CHARACTERS IN STRINGS

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

Backslash character	\\
Double-quote character	\"

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

Tab character	\t
Return character	\r

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

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

is displayed in a dialog box as

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

STRING CONSTANTS

AppleScript defines three constants for string values:

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

COERCIONS SUPPORTED

If a string consists of an appropriate number, AppleScript supports coercion of the string to an integer, a number, or a real number. Similarly, any integer, number, or real number can be coerced to a string. AppleScript also supports coercion of a string to a single-item list and coercion of a list whose items are all strings to a single concatenated string.

NOTES

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

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

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

The result of the same statement without the text element is a list.

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

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

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

Styled Text

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

LITERAL EXPRESSIONS

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

PROPERTIES

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

ELEMENTS

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

OPERATORS

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

For detailed explanations and examples of how AppleScript operators treat strings, see "Operators That Handle Operands of Various Classes," which begins on page 168.

REFERENCE FORMS

You can use the same reference forms with styled text that you can use with strings: Property, Index, Middle, Arbitrary, Every Element, and Range. For details, see page 63.

SPECIAL CHARACTERS AND STRING CONSTANTS

You can use the same special characters, constants, and coercions with styled text that you can use with strings. For details, see page 63. Note that literal string constants do not include style and font information; in other words, they are not styled text.

COERCIONS SUPPORTED

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

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

NOTES

You can't change the style or font information from a script, but you can use styled text to preserve style and font information when copying text objects from applications to scripts. For example, you can use a script like this to obtain styled text, manipulate it, and copy it back into a Scriptable Text Editor document:

```
tell application "Scriptable Text Editor"
   copy (word 1 of front document) to myStyledTitle
   set myModifiedTitle to myStyledTitle & ", alpha release"
   copy myModifiedTitle to word 1 of front document
end tell
```

Because the Scriptable Text Editor returns styled text when it returns the data for text objects, you don't need to coerce the returned text to styled text. The style and font of the first word are preserved both when the word is copied to the variable `myStyledTitle` and when it is concatenated with the string `", alpha release"`. The modified title that is copied back to the document consists of the original title with its original style and font, plus the unstyled text, `", alpha release"`, which appears in the style and font of the character immediately preceding it.

Styled text also contains information about the form in which the text is written. If you copy non-Roman text to a variable in a script as styled text, AppleScript preserves the original text information even though the Script Editor may not be able to display it correctly. If you then copy the text to an application that can handle the text in its original form, the text is displayed correctly.

Text

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

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

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

```
class of "A string" as text
--result: string
```

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

Coercing Values

AppleScript coerces values in two ways:

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

The As operator specifies a particular coercion. You can use the As operator to coerce a value to the correct class before using it as a command parameter or operand. For example,

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

coerces the integer `2` into the string `"2"` before storing it in the variable `myString`. Similarly,

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

coerces the string `"2"` to the integer `2`, so that it can be added to the other operand, `8`.

If you provide a command parameter or operand of the wrong class, AppleScript automatically coerces the operand or parameter to the expected class, if possible. For example, when AppleScript executes this statement,

```
repeat ( word 2 of document "Big" of application ¬
   "Scriptable Text Editor") times
      display dialog "Hello"
end repeat
```

it expects the number of times to be an integer. To coerce `word 2 of document "Big" of application "Scriptable Text Editor"` to an integer, AppleScript gets the value of `word 2 of document "Big" of application "Scriptable Text Editor"`—a string—and then coerces it to an integer, if possible.

Not all values can be coerced to all other classes of values. Figure 3-2 summarizes the coercions that AppleScript supports. To use the figure, find the class of the value to be coerced in the column at the left. Search across the table to the column labeled with the class to which you want to coerce the value. If there is a square at the intersection, then AppleScript supports the coercion.

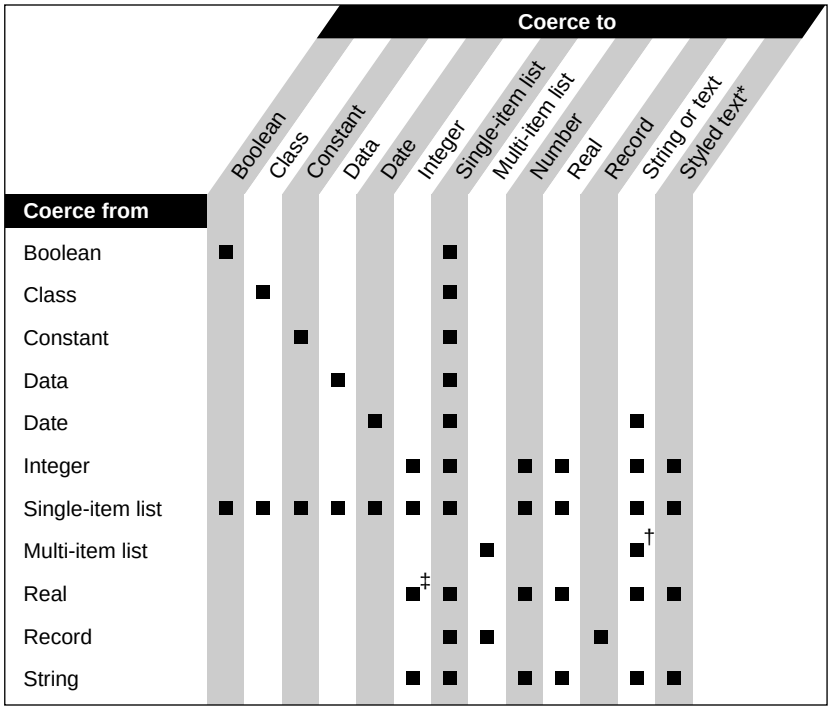
Reference values are not included in the table because applications determine whether the value of an object specified by a reference value can be coerced to a desired class.

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

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

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

Figure 3-2 Coercions supported by AppleScript



* Only values that include style and font information can be coerced to styled text.
† Only a list whose items can all be coerced to strings can be coerced to a string.
‡ Only a real value that has no fractional part can be coerced to an integer.

Three of the identifiers mentioned at the top of Figure 3-2 act only as synonyms for other value classes: “number” is a synonym for either “integer” or “real,” “text” is a synonym for “string,” and “styled text” is a synonym for a string that contains style and font information. You can coerce values using these synonyms, but the class of the resulting value is always the appropriate value class, not the synonym. Here are some examples:

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

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

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

CHAPTER 4

Commands

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

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

This chapter begins by describing types of commands and targets of commands. It summarizes which types of commands work with which types of targets. Next, it describes the details of using commands and command definitions, including specifying parameters and using results of commands. The chapter ends with definitions of standard commands.

Types of Commands

You can use four types of commands in AppleScript to request actions: application commands, AppleScript commands, scripting additions, and user-defined commands.

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

The sections that follow describe the different types of commands and their targets.

Application Commands

Application commands are commands that cause actions in scriptable applications. The target of an application command is an application object or a script object. Different application objects respond to different commands. To determine which commands a particular object can respond to, see the definitions provided by the application's documentation. (For example, each object definition in Appendix B, "Scriptable Text Editor Dictionary," of this book includes a list of commands that can act on that object.)

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

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

For example, in the following statement, the reference `word 1 of front document of app "Scriptable Text Editor"` is the direct parameter of the Delete command:

```
delete word 1 of front document of app "Scriptable Text Editor"
```

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

```
tell word 1 of front document of app "Scriptable Text Editor"
   delete
end tell
```

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

```
tell front document of app "Scriptable Text Editor"
   delete word 1
end tell
```

For information about sending application commands to script objects, see Chapter 9, "Script Objects," which begins on page 265.

AppleScript Commands

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

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

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

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

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

```
count words in paragraph 1 of front document of ¬
   application "Scriptable Text Editor"
```

For the Copy command, if the value of the `to` parameter is a reference to an application object, then the command functions as an application command. Otherwise, the command is an AppleScript command.

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

Scripting Addition Commands

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

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

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

For example, the scripting addition command Display Dialog displays a dialog box that can include text, one or more buttons, an icon, and a field in which the user can type text. In the script that follows, the target of the Display Dialog command is the Scriptable Text Editor application. When the script runs, the Scriptable Text Editor becomes the frontmost application (that is, its menus become visible and its windows become the frontmost windows on the screen) and passes the command to the scripting addition's handler for the Display Dialog command, which displays the dialog box.

```
tell application "Scriptable Text Editor"
   display dialog "What's your name?" default answer ""
end tell
```

In the next example, the Display Dialog command is not enclosed in a Tell statement, nor does it have a direct parameter, so its target is the Script Editor (or whatever application runs the script). When you run the script, the Script Editor passes the command to the scripting addition's handler for the Display

Dialog command, which displays the dialog box in the Script Editor's layer (that is, in front of any other Script Editor windows that may be open), while the Script Editor is still the active application.

```
set theCount to number of words in front document of ¬
   app "Scriptable Text Editor"
if theCount > 500 then
   display dialog "You have exceeded your word limit."
end
```

Each scripting addition that contains command handlers has its own dictionary, which lists the reserved words—including the command names, parameter labels, and in some cases object names—used to invoke the commands supported by the scripting addition. If a scripting addition dictionary includes words that are also part of an application dictionary, then you cannot use those words within Tell statements to that application.

For example, the Offset command provided by the String Commands scripting addition reports the offset, in characters, of a string within another string. Offset is also a property of several Scriptable Text Editor objects and is thus a word in the Scriptable Text Editor dictionary. Therefore, you cannot use Offset as a scripting addition command within Tell statements to the Scriptable Text Editor. If you do, you'll get a syntax error, because AppleScript treats the word Offset as a property of a text object.

```
tell front document of application "Scriptable Text Editor"
   offset of "great" in "To be great"
end tell
--result: syntax error
```

If you specify a script object as the target of a scripting addition command, the script object either handles the command itself (potentially modifying it) or passes the command to the default target application. For more information about scripting additions and script objects, see "Using Continue Statements to Pass Commands to Applications," which begins on page 280.

For information about the scripting additions available for AppleScript English and definitions of the commands they provide, see the *AppleScript Scripting Additions Guide.*

User-Defined Commands

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

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

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

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

```
tell application "Scriptable Text Editor"
   set theCount to my minimumValue(12,105)
   get word theCount of front document
end tell
```

Without the word `my` before the `minimumValue` command, AppleScript sends the `minimumValue` command to the Scriptable Text Editor, resulting in an error.

Chapter 8, "Handlers," describes the syntax for defining and invoking subroutines such as `minimumValue` in more detail.

Note
You can also define subroutines in script objects. The target of a user-defined command whose subroutine is defined in a script object is the script object. For information about defining and invoking subroutines in script objects, see Chapter 9, "Script Objects." ◆

Using Command Definitions

Command definitions contain information about what commands do and how to use them in scripts. Figure 4-1 shows the definition for the Move command, an application command. The definition contains four types of information: syntax, parameters, results, and examples. Some definitions include information about errors as well. The sections following the figure explain the information conveyed by each part of the definition.

Figure 4-1 Command definition for the Move command

Move application command

A Move command is a request to move an object or objects.

SYNTAX

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

PARAMETERS

referenceToObject A reference to the object or objects to move.
Class: Reference

referenceToLocation A reference to the location to which to move the object or objects.
Class: Reference

RESULT

A reference to the object that was moved.
Class: Reference

EXAMPLE

```
tell document 1 of app "Scriptable Text Editor"
  move word 10 to before paragraph 11
end tell
```

Syntax

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

For example, to use the Move command, you must replace *directParameter* with a reference to the object to move and *location* with a reference to the location to which to move it.

Parameters

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

Many commands include a *direct parameter* that specifies the object of the action. If a command includes parameters other than the direct parameter, they are identified by labels. Parameters that are identified by labels are called **labeled parameters.** The direct parameter immediately follows the command; labeled parameters can be listed in any order. The Move command defined in Figure 4-1 has a direct parameter (referred to in the definition as *referenceToObject*) that specifies the object to move and a labeled parameter (whose label is `to`) that specifies where to move the object.

Each parameter value must belong to a particular class, which is listed in its description in the command definition. For the Move command, the direct parameter belongs to the class reference. Its value, a reference, is a phrase that identifies the object to be moved. The `to` parameter also belongs to the class reference. It specifies the location to which to move the object. References are described in Chapter 5, "Objects and References."

Parameters can be required or optional. **Required parameters** must be included with the command; **optional parameters** need not be. Optional parameters are enclosed in brackets in syntax descriptions. For optional

parameters, the description in the "Parameters" section specifies a default value that is used if you don't include the parameter.

For more information about direct parameters, see "Application Commands" on page 72. For more information about using parameters, see "Using Parameters" on page 80.

Result

Many, but not all, commands return results. The **result** of a command is the value generated when the command is executed. The "Result" section of a command definition tells whether a result is returned, and if so, lists its class. For example, the result of the Move command shown in Figure 4-1 is a reference to the object that was moved.

For more information about results, see "Using Results" on page 82.

Examples

Each command definition includes one or more short examples demonstrating how to use the command. The example in Figure 4-1 shows how to use the Move command to move a word in the Scriptable Text Editor.

Errors

Commands can return error messages as well as results. An **error message** is a message that is returned by an application, AppleScript, or the operating system if an error occurs during the handling of a command. The "Errors" section of a command definition, if present, lists errors that are likely to be returned by a particular command. This information can help you decide if you need to write *error handlers* to respond to the error messages that are returned. Error handlers are described in Chapter 8, "Handlers."

Some "errors" are not the result of abnormal conditions but are the normal way you get information about what happened during command execution. For example, you use the Choose File command to ask the user to choose a file. When AppleScript executes this command, it displays a dialog box similar to the one you get when you choose Open from the File menu. If the user presses

the Cancel button in the dialog box, AppleScript returns error number –128 and the error string `"User canceled"`. Your script must handle this error for script execution to continue.

For a complete description of handling errors that occur during script execution, see Chapter 8, "Handlers."

Using Parameters

This section describes how to

- use parameters that specify locations
- coerce parameters
- deal with raw data in parameters

Parameters That Specify Locations

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

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

```
move word 1 to before paragraph 10
```

In the following statement, the value of the `to` parameter is an object, `word 10`. The Move command replaces `word 10` with `word 1`.

```
tell front window of application "Scriptable Text Editor"
   move word 1 to word 10
end tell
```

Phrases such as `before paragraph 10` and `word 1` are called relative references and index references, respectively. These kinds of references specify locations. For more information about these kinds of references, see “Index” on page 131 and “Relative” on page 139.

Coercion of Parameters

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

```
2 as string
```

AppleScript performs some coercions, including the previous one, automatically. For example, in the following statement, the direct parameter of the Copy command should be a string because it is being inserted into a text editor document.

```
tell application "Scriptable Text Editor"
   copy 12 to beginning of document "Introduction"
end tell
```

When AppleScript executes this statement, it automatically coerces the integer 12 to the string `"12"` and inserts the string at the beginning of the document.

The coercions that AppleScript can perform are listed in Chapter 3, “Values.” Applications can also perform additional coercions, such as coercions for classes that are specific to an application. These coercions are listed in the documentation for the application.

Raw Data in Parameters

Some application commands return values that do not belong to any of the normal AppleScript value classes. An example is the Edit Graphic command supported by some graphics applications. The values that are returned belong to the class Data, which is described in Chapter 3, “Values.” Values of class Data cannot be displayed by AppleScript, but they can be stored in variables and sent as parameters in other commands. For example, if it’s necessary to use

two different applications to edit a graphic, the data value returned by one Edit Graphic command can be sent as the direct parameter of another Edit Graphic command.

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

Using Results

The result of a command is the value generated when the command is executed. You can display the result of a command in the Script Editor. For example, if you run the following script,

```
tell front document of application "Scriptable Text Editor"
   move word 1 to end of paragraph 1
end tell
```

and then choose Show Result from the Controls menu in the Script Editor, you'll see a value such as

```
word 32 of front document of application "Scriptable Text Editor"
```

You can use a command that returns a result as a value. For example, the Count command in the following statement returns a value: the number of words in the third paragraph.

```
count words in paragraph 3
```

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

```
set numWords to (count words in paragraph 3)
```

In addition to displaying the result of a command in the result window, AppleScript puts the result into a predefined variable called `result`. The value remains there until the next command is executed. If the next command does not return a result, the value of `result` is undefined. The following two

commands show how to use the `result` variable to set the value of `numWords` to the value returned by the Count command:

```
count words in paragraph 3
set numWords to result
```

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

```
get paragraphs 1 thru 3 of first document
```

The result is a list of strings similar to the following. The first string is the value of the first paragraph, the second string is the value of the second paragraph, and the third string is the value of the third paragraph.

```
{"This is paragraph one.", "This is paragraph two." ¬
   "This is paragraph three."}
```

Double Angle Brackets in Results and Scripts

You may occasionally notice terms like this within double angle brackets in a script or a result:

```
«event sysodlog»
```

In general, AppleScript uses double angle brackets when it can't locate the dictionary it needs to identify a term or can't display a value directly. The first word within the double angle brackets can be any of these: `event`, `property`, `class`, `data`, `preposition`, `keyform`, `constant`, or `script`. The second word varies depending on the context.

If double angle brackets appear unexpectedly in a script when you open it with the Script Editor, it may be because a command used in the script is not present in the Scripting Additions folder (which is located in the Extensions folder of the System folder) for your computer. For example, if you create a script that uses the Display Dialog command, then open the script at a later time or on a

different computer when the Display Dialog scripting addition is not present in the Scripting Additions folder, AppleScript replaces the words `display dialog` in the script with «`event sysodlog`». In this case you should drop the icon for the Display Dialog scripting addition into the Scripting Additions folder before attempting to run the script.

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

```
script Joe
   property theCount : 0
end script

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

(For more information about script objects, see Chapter 9, "Script Objects.")

Similarly, if the value of a variable is of class Data and AppleScript can't represent the data directly, the value of the data is represented within angle brackets by the word `data` followed by some sequence of ASCII characters. Although this may not resemble the original data, the data's original format is preserved. You can treat the data like any other value, except that you can't view it directly in any Script Editor window.

Command Definitions

The sections that follow are in alphabetical order by command name and provide definitions for both AppleScript commands and standard application commands. The general features of these types of commands are described in "Types of Commands,"which begins on page 71. The command type is listed opposite the command name on the first line of each definition.

For definitions of commands provided by the scripting additions that come with AppleScript English, see the *AppleScript Scripting Additions Guide*. For definitions of commands provided by other scripting additions, see the documentation for those scripting additions.

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

Application dictionaries list application commands under two categories, the Required suite and the Standard suite. All applications support the commands in the Required suite. Different applications may support different commands in the Standard suite. Table 4-1 summarizes the standard application commands described in this chapter that belong to each suite.

Many applications also define their own suite of more specialized commands. The application's dictionary provides definitions of all commands supported by the application. Check the appropriate application dictionary before using application commands. You can open an application's dictionary by selecting the application's icon on the desktop, dragging it over the Script Editor's icon, and releasing the mouse button.

Table 4-1 Standard application commands defined in this chapter

Command	Summary
Required suite	
Open	Opens a file.
Print	Prints one or more objects.
Quit	Terminates an application.
Run	Launches an application and invokes its standard startup procedures.
Standard suite	
Close	Closes one or more objects.
Copy	Copies an object or objects to the Clipboard or to a new location.
Count	Counts elements of a particular class in an object.
Data Size	Returns the size, in bytes, of the value of an object.
Delete	Deletes one or more objects.

continued

Table 4-1 Standard application commands defined in this chapter (continued)

Command	Summary
Duplicate	Copies an object or objects to a new location.
Exists	Determines if an object exists.
Get	Returns the value of an object.
Launch*	Launches an application without invoking its standard startup procedures.
Make	Creates a new object.
Move	Moves an object or objects.
Save	Saves an object to a file.
Set	Assigns a value to an object.

* Although the target of a Launch command is always an application, it is actually handled by AppleScript. Unlike the other commands listed in this table, the Launch command doesn't need to be explicitly supported by applications and doesn't appear in any application's dictionary.

Table 4-2 lists the AppleScript commands defined in this chapter.

Table 4-2 AppleScript commands defined in this chapter

Command	Summary
Copy	Assigns a value to a variable.
Count	Counts the elements of a compound value.
Get	Returns the value of an expression.
Run	Executes statements other than handler and property definitions in a script object definition.
Set	Assigns a value to a variable.

Another AppleScript command, the Error command, is described in "Try Statements," which begins on page 204.

Close application command

A Close command is a request to close one or more objects.

SYNTAX

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

PARAMETERS

referenceToObject
A reference to the object or objects to close.
Class: Reference

referenceToFile A reference of the form `file` *nameString* or `alias` *nameString* (see "Notes").
Class: Reference
Default value: The file in which the object was last saved. If the object hasn't been saved before, the application creates a file with the specified name in the current directory.

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

RESULT

None

EXAMPLES

```
tell application "Scriptable Text Editor"
   close window "Tremendous" saving Yes
end tell

tell application "Scriptable Text Editor"
   close saving in file "Macintosh HD:Documents:Report"
end tell
```

NOTES

To specify the name (*nameString*) of a file in which to save the object, use a string of the form "*Disk*:*Folder1*:*Folder2*:...:*Filename*"; for details, see "References to Files," which begins on page 144. You can also specify a string with only a filename ("*Filename*"). In this case, the application attempts to find the file in the current directory. If it can't find the specified file, the application creates a file with the specified name in the current directory.

Copy

AppleScript command, application command

The Copy command can function as an AppleScript command or an application command. The AppleScript command makes a copy of one or more values and stores it in one or more variables.

The application command is a request to copy an object or objects. If the command includes a direct parameter, the Copy command makes a copy of one or more objects specified in the direct parameter and puts them in one or more new locations (if any are specified) or on the Clipboard (if no new location is specified).

If the command does not include a direct parameter, the Copy command makes a copy of the object or objects in the current selection and puts them on the Clipboard. This is the same as choosing Copy from the Edit menu in an application.

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

APPLESCRIPT COMMAND SYNTAX

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

APPLICATION COMMAND SYNTAX

```
( copy | put ) expression ( to | into) referencePattern
```

```
( copy | put ) [ referenceToObject ]
```

PARAMETERS

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

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

referencePattern
A reference to the location to which to copy *expression*, a list of reference patterns, or a record of reference patterns.
Class: Reference, list, or record
Default value: If you do not specify a new location, the object specified in the direct parameter is copied and put on the Clipboard.

referenceToObject
A reference to the object or objects to be copied, or a list of reference patterns, or a record of reference patterns.
Class: Reference, list, or record
Default value: If this parameter is omitted, the object or objects in the current selection are copied and put on the Clipboard.

RESULT

If the Copy command is used to create a variable, the result is the value that was stored in the variable. If the command is used to copy an object, the result

is a reference to the copied object; however, if the command does not include parameters, there is no result.
Class: Varies

EXAMPLES

This example copies a string to the variable `myOccupation`:

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

This example copies the value of a reference to the variable `x`:

```
copy word 1 of front document of app "Scriptable Text Editor" to x
```

This example makes a copy of a word, and then inserts it at the beginning of the fourth paragraph:

```
tell application "Scriptable Text Editor"
   copy word 1 to beginning of paragraph 4
end tell
```

The next example copies a word to the Clipboard and then pastes it from the Clipboard to the insertion point after the tenth paragraph.

```
tell application "Scriptable Text Editor"
   select word 1 of document "Test"
   copy
   select insertion point after paragraph 10 of document "Test"
   paste
end tell
```

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

```
tell application "Scriptable Text Editor"
   copy position of front window to {x, y}
end tell
```

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

Patterns copied with the Copy command can also be more complex. Here's an example:

```
set x to {8, 94133, {firstName:"John", lastName:"Chapman"}}
copy x to {p, q, {lastName:r}}
(* now p, q, and r have these values: p = 8
                                      q = 94133
                                      r = "Chapman" *)
```

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

The use of the Copy command with patterns is similar to the use of the Set command with patterns. For information about the Set command, see page 113.

NOTES

For more information about using the Copy command to create or change the values of variables, see "Variables," which begins on page 150.

If you use the Copy command without parameters and there is no selection to be copied, the application does not change the contents of the Clipboard.

When copying objects between applications via the Clipboard, you must use the Activate command to make the receiving application active before attempting to paste from the Clipboard.

ERRORS

Error number	Error message
–1728	Can't get <reference>.
–10006	Can't set <destination> to <source>.

Count — AppleScript command, application command

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

APPLESCRIPT COMMAND SYNTAX

```
count [ [ each | every ] className ( in | of ) ] compoundValue

number of [ pluralClassName ( in | of ) ] compoundValue
```

APPLICATION COMMAND SYNTAX

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

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

PARAMETERS

className The class name of the elements to be counted. If you use the term `each` or `every`, you can use only the singular form of the class name. The elements of lists, records, and strings are listed in the value class definitions in Chapter 3, "Values." The elements of application objects are listed in their object class definitions in the application dictionary.
Class: Class identifier
Default value: Item for lists, records, and application objects; Character for strings (see "Notes")

compoundValue
An expression that evaluates to a compound value whose elements are to be counted.
Class: List, record, reference, or string

*pluralClassNam*e

The plural class name of the elements to be counted. The elements of lists, records, and strings are listed in the value class definitions in Chapter 3, "Values."
Class: Class identifier
Default value: Item for lists, records, and application objects; Character for strings (see "Notes")

referenceToObject

A reference to the object or objects whose elements are to be counted. If you do not specify this parameter, the application counts the elements in the default target of the Tell statement.
Class: List, record, reference, or string

RESULT

The result of the AppleScript command is an integer that specifies the number of elements of a specified class in a compound value.

The result of the application command is either an integer or a list of integers. See "Notes" for details.

Class: Integer or list of integers

EXAMPLES

In the following example, *compoundValue* is a list. The command does not explicitly specify a class of elements to count, so AppleScript counts all the items in the list.

```
count {"Yes", "No", "Maybe", 4, 5, 6}
--result: 6
```

In this example, *className* is `integers` and *referenceToObject* is a list of strings and integers. AppleScript counts the integers in the list.

```
count integers in {"Yes", "No", "Maybe", 4, 5, 6}
--result: 3
```

This example shows another way to count the integers in the list:

```
count each integer in {"Yes", "No", "Maybe", 4, 5, 6}
--result: 3
```

In the following example, `every word of document "simple"` consists of a list of words. The Scriptable Text Editor counts the words in the list.

```
tell application "Scriptable Text Editor"
   count every word of document "simple"
end tell
--result: 12
```

The following statement is equivalent to the previous example:

```
tell application "Scriptable Text Editor"
   count words of document "simple"
end tell
```

In the following example, *referenceToObject* is `documents of application "Scriptable Text Editor"`, which is a list of documents. The Scriptable Text Editor counts the documents in the list.

```
tell application "Scriptable Text Editor"
   repeat with i from 1 to (count of documents)
      set the style of paragraph 1 of document i to ¬
         {outline, bold}
   end repeat
end tell
```

NOTES

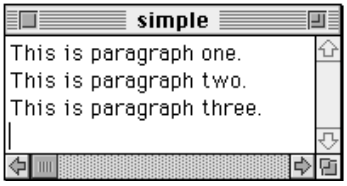
If you use the Count command on a string without specifying the class to be counted, AppleScript counts the characters; for example,

```
count "This is a string"
--result: 16
```

The result of the Count command depends on how you specify the range of objects to be counted. For example, consider the following statement, given the Scriptable Text Editor document in Figure 4-2:

```
tell document "simple" of app "Scriptable Text Editor"
   count words from paragraph 2 to paragraph 3
end tell
--result: 8
```

Figure 4-2 The Scriptable Text Editor document “simple”



The reference `words from paragraph 2 to paragraph 3` specifies a list of the words in the second and third paragraphs:

```
{"This", "is", "paragraph", "two.", ¬
   "This", "is", "paragraph", "three."}
```

Each item in the list is a word. The Scriptable Text Editor counts the items in the list and returns the result 8.

Sometimes the Count command returns a list of integers. Consider the following statement:

```
tell document "simple" of app "Scriptable Text Editor"
   count words of paragraphs 2 thru 3
end tell
--result: {4, 4}
```

The counting in this example requires several steps, beginning with the reference to the outermost container. The reference `paragraphs 2 thru 3` specifies a list of two items, each of which is a paragraph:

```
{"This is paragraph two.", "This is paragraph three."}
```

On the basis of this list, the Scriptable Text Editor evaluates the reference `words of paragraphs 2 thru 3` as a list of two items, each of which is a list of the words in one paragraph:

```
{{"This", "is", "paragraph", "two."}, ¬
   "This, "is", "paragraph", "three."}}
```

Finally, the Scriptable Text Editor counts the items in each list and returns a list of two items, each of which specifies the number of words in one paragraph: `{4, 4}`.

References to nested containers are always evaluated before counting takes place, beginning with the outermost container. Here's another example:

```
tell document "simple" of app "Scriptable Text Editor"
   count characters of words of paragraphs 2 thru 3
end tell
--result: {{4, 2, 9, 3}, {4, 2, 9, 5}}
```

The previous example demonstrated that the reference `words of paragraphs 2 thru 3` specifies a list of two items, each of which is a list of the words in one paragraph:

```
{{"This", "is", "paragraph", "two."}, ¬
   {"This, "is", "paragraph", "three."}}
```

The Scriptable Text Editor counts the items in each list and returns a list of two items, each of which is a list of the number of characters in each of the words in one paragraph:

```
{{4, 2, 9, 3}, {4, 2, 9, 5}}
```

Data Size — application command

A Data Size command is a request for the size, in bytes, of the data of one or more objects. The value returned is the size of the data (a value) that would result from a Get command on the same object or objects.

SYNTAX

```
data size of referenceToObject [ as className ]
```

PARAMETERS

referenceToObject
: A reference to the object or objects whose data size is to be returned.
Class: Reference

className
: The class of data for which to determine the size (see "Notes").
Class: Class identifier
Default value: The default value class for the object

RESULT

The size, in bytes, of the object or objects.

If the *referenceToObject* parameter specifies a single object only (such as `word 1` or `the last word`), the result is a single integer that specifies the size of the object in bytes. If the specified object doesn't exist, for example, if the reference is `word 12` and there are fewer than 12 words in the specified container, the application returns an error.

If the *referenceToObject* parameter refers to more than one object (such as `the words whose first letter is "B"`), the result is a list of integers. The first item in the list is the size of the first object specified, the second item is the size of the second object specified, and so on. If the specified objects don't exist, for example, if the reference is `the words whose first letter is "B"` and there are no words that begin with "B", the result is an empty list.

Class: Integer or list of integers

EXAMPLE

```
set theSize to the data size of paragraph 1 through 10
```

NOTES

As described in the definition of the Get command, the data of some application objects can be returned as values of different classes. Because the size of the data returned as different classes can be different, the Data Size command includes an optional *class* parameter that allows you to specify the class of the data.

Delete

application command

A Delete command is a request to delete one or more objects.

SYNTAX

`delete` *referenceToObject*

PARAMETER

referenceToObject
A reference to the object or objects to be deleted.
Class: Reference

RESULT

None

EXAMPLE

```
tell document "Intro" of app "Scriptable Text Editor"
   delete words 1 through 5
end tell
```

Duplicate

application command

A Duplicate command is a request to make a copy of an object or objects and insert the new copy either at a location specified in the command or at the location following the object that was copied.

SYNTAX

`duplicate` *referenceToObject* `[ to` *newLocation* `]`

PARAMETERS

referenceToObject

A reference to the object or objects to be duplicated.
Class: Reference

newLocation

The new location for the object.
Class: Reference
Default value: If you do not specify a new location, the object is inserted at the location immediately following the object specified in the direct parameter.

RESULT

A reference to the new object.

Class: Reference

Exists

application command

An Exists command is a request to determine whether the object specified by a reference exists.

SYNTAX

referenceToObject `exists`

`exists` *referenceToObject*

PARAMETER

referenceToObject
A reference to the object or objects to find.
Class: Reference

RESULT

If `true`, all of the objects referred to by *referenceToObject* exist. If `false`, one or more of the objects referred to by *referenceToObject* do not exist.
Class: Boolean

EXAMPLES

```
tell document "Tremendous" of app "Scriptable Text Editor"
   if word 7 exists then
      delete word 7
   end if
end tell

tell application "Scriptable Text Editor"
   if exists front document then
      print front document
   end if
end tell
```

Get

AppleScript command, application command

The Get command can function as an AppleScript command or an application command. The AppleScript command returns the value of an expression. The application command returns the value of an object. In both cases, the command assigns the value returned to the predefined variable `result`.

APPLESCRIPT COMMAND SYNTAX

```
[ get ] expression [ as className ]
```

APPLICATION COMMAND SYNTAX

```
[ get ] referenceToObject [ as className ]
```

PARAMETERS

expression An expression whose value is to be returned in the `result` variable.
Class: Any AppleScript expression

className A class identifier that specifies the desired value class for the returned data.
Class: Class
Default value: The default value class for the object or objects

referenceToObject
A reference to an object whose value is to be returned in the `result` variable.
Class: Reference

RESULT

The result is the value of the specified reference or expression.

If the *referenceToObject* parameter specifies a single object only (such as `word 1` or `the last word`), the result is a single value. If the specified object doesn't exist, for example, if the reference is `word 12` and there are fewer than 12 words in the specified container, no result is returned.

If the *referenceToObject* parameter refers to more than one object (such as `the words whose first letter is "B"`), the result is a list of values. The first item in the list is the value of the first object specified, the second item is the value of the second object specified, and so on. If the specified objects don't exist, for example, if the reference is `the words whose first letter is "B"` and there are no words that begin with "B", the result is an empty list.

Class: The class specified by the *className* parameter or a list of values of that class. If the application cannot return data in the value class specified by the *className* parameter, it returns a value or values of the default value class.

EXAMPLE

```
tell application "Scriptable Text Editor"
   get paragraph 3 --gets the value
   copy result to item 2 of x --puts the result
end tell
```

NOTES

The word `get` in the Get command is optional because AppleScript automatically gets the value of expressions and references when they appear in scripts.

For example, the following statements are equivalent:

```
item 1 of {"Hi,", "how", "are", "you?"}

get item 1 of {"Hi,", "how", "are", "you?"}
```

The following statements are also equivalent:

```
tell application "Scriptable Text Editor"
   word 1 of document "Introduction"
end tell

tell application "Scriptable Text Editor"
   get word 1 of document "Introduction"
end tell
```

ERROR

Error number	Error message
–1728	Can't get <reference>.

Launch — application command

If an application is not already running, sending it a Launch command launches it without sending a Run command. (If the application is already running, the Launch command has no effect.) This allows an application to open without performing its usual startup procedures, such as opening a new window or, in the case of a script application, running the script.

SYNTAX

```
launch [ referenceToApplication ]
```

PARAMETER

referenceToApplication
A reference of the form `application` *nameString* (see "Notes"). This parameter is optional if the Launch command is used within an appropriate Tell statement.
Class: Reference

RESULT

None

EXAMPLES

```
launch application "Scriptable Text Editor"

tell application "Scriptable Text Editor"
   launch
end tell
```

NOTES

To specify the name (*nameString*) of an application to launch, use a string of the form "*Disk*:*Folder1*:*Folder2*:...:*ApplicationName*"; for details, see "References to Applications," which begins on page 146. You can also specify a string with only an application name ("*ApplicationName*"). In this case, AppleScript attempts to find the application in the current directory.

AppleScript sends an implicit Run command whenever it begins to execute a Tell statement whose target is an application that is not already open. This can cause problems with applications such as the Scriptable Text Editor that normally perform specific tasks on startup, such as opening a new window. Here's an example:

```
tell application "Scriptable Text Editor"
   open file "Hard Disk:Status Report"
end tell
```

Before AppleScript opens the file Status Report, it sends an implicit Run command to Scriptable Text Editor. If the application is not already open, the Run command causes Scriptable Text Editor not only to launch but also to perform its usual startup tasks, including opening an untitled window. Therefore, running this script opens two windows: an untitled window and a window for the file Status Report.

If you don't want AppleScript to send an implicit Run command when it launches an application as the result of a Tell statement, use the Launch command explicitly at the beginning of the statement:

```
tell application "Scriptable Text Editor"
   launch
   open file "Hard Disk:Status Report"
end tell
```

In this case, AppleScript launches the application without sending it a Run command, and the application opens only a window for the requested document.

For similar reasons, it is sometimes important to use the Launch command before sending the Run command to a script application. For more information, see "Calling a Script Application," which begins on page 251. For information about Run handlers, see "Run Handlers," which begins on page 243.

Although the target of a Launch command is always an application, it is actually handled by the Finder. Unlike the other application commands defined in this chapter, it doesn't need to be explicitly supported by applications and doesn't appear in any application's dictionary.

Make

application command

A Make command is a request to create a new object. The command can include values for properties of the object, for the data of the object, or both.

SYNTAX

```
make [new] className at referenceToLocation                          ¬
   [ with properties                                                 ¬
      { propertyLabel:propertyValue [, propertyLabel:propertyValue ]...}] ¬
   [ with data dataValue ]
```

PARAMETERS

className The class of the object to be created.
Class: Class identifier

referenceToLocation The location at which to create the new object.
Class: Reference

propertyLabel The name of a property whose value is to be set.
Class: String

propertyValue The value to assign to the property.
Class: The value class of the property, as specified in the application dictionary definition of the object class being created, or a value that can be coerced into the class of the property
Default value: The default value of the property, as specified in the application dictionary definition of the object class being created

dataValue The value to assign to the object.
Class: The default value class of the object, or a value that can be coerced into the default value class. Default value classes of objects are listed in the "Default Value Class Returned" sections of the dictionary definitions of the objects.
Default value: None

RESULT

A reference to the newly created object.

Class: Reference

EXAMPLE

```
tell application "Scriptable Text Editor"
   make window at beginning with properties ¬
      {bounds:{400, 300, 500, 600}, contents:"This window is narrow."}
end tell
```

NOTES

If you use the Make command to create a new text object, check the application dictionary to determine how the application handles delimiters. Some applications, such as the Scriptable Text Editor, supply delimiters automatically, so you don't have to include them in the value of the `with data` parameter.

Move

application command

A Move command is a request to move an object or objects.

SYNTAX

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

PARAMETERS

referenceToObject
A reference to the object or objects to move.
Class: Reference

referenceToLocation
A reference to the location to which to move the object or objects.
Class: Reference

RESULT

A reference to the object that was moved.

Class: Reference

EXAMPLE

```
tell front document of app "Scriptable Text Editor"
   move word 10 to before paragraph 11
end tell
```

Open

application command

An Open command is a request to open a file or files.

SYNTAX

`open` *referenceToFile*

PARAMETER

referenceToFile
A reference of the form `file` *nameString* or `alias` *nameString*, or a list of such references (see "Notes").
Class: Reference or list of references

RESULT

None

EXAMPLE

```
tell app "Scriptable Text Editor"
   open file "Macintosh HD:New Products:Mammoth:Product Intro"
end tell

tell app "Scriptable Text Editor"
   open { file "HD:Letters:Offer", file "HD:Letters:Acceptance"}
end tell
```

NOTES

To specify the name (*nameString*) of a file to open, use a string of the form "*Disk*:*Folder1*:*Folder2*:...:*Filename*"; for details, see "References to Files," which begins on page 144. You can also specify a string with only a filename ("*Filename*"). In this case, the application attempts to find the file in the current directory.

If the file or files specified by *referenceToFile* is already open, it remains open.

Print — application command

A Print command is a request to print one or more objects.

SYNTAX

`print` *referenceToObject*

PARAMETER

referenceToObject
A reference to the object or objects to print—typically file(s), document(s), or window(s).
Class: Reference or list of references

RESULT

None

EXAMPLES

```
tell application "Scriptable Text Editor"
   print document "Introduction"
end tell

tell application "Scriptable Text Editor"
   print windows 1 thru 5
end tell

tell application "Scriptable Text Editor"
   print { file "HD:Letters:Zoning Department", ¬
      file "HD:Letters:Mayor"}
end tell
```

NOTES

To specify the name of a file to print, use the term `file` or `alias` followed by a string of the form "*Disk*:*Folder1*:*Folder2*:...:*Filename*"; for details, see "References to Files," which begins on page 144. You can also specify a string with only a filename ("*Filename*"). In this case, the application attempts to find the file in the current directory.

Quit — application command

A Quit command is a request for an application to terminate. If no optional parameters are specified, the Quit command has the same result as choosing the Quit menu item in the application.

SYNTAX

`quit` *referenceToApplication* `[ saving` *saveOption* `]`

PARAMETERS

referenceToApplication
A reference of the form `application` *nameString*, where *nameString* is a string that matches the name of the application you want to quit as it is listed in the Application menu.
Class: Reference

saveOption A constant that specifies whether to save documents that have been modified before quitting. The possible values are `yes`, `no`, and `ask`. The value `yes` specifies to save the documents. The value `no` specifies not to save the documents. The value `ask` specifies to ask the user whether or not to save the documents.
Class: Constant
Default Value: `ask`

RESULT

None

EXAMPLES

```
tell application "Scriptable Text Editor"
   quit saving no
end tell

quit application "Scriptable Text Editor" saving ask
```

Run

AppleScript command, application command

The Run command can function as an AppleScript command or an application command.

The AppleScript Run command acts on script objects; it executes statements other than handler and property definitions in script object definitions.

The application Run command launches an application if it's not already running. The application must be on a local or mounted volume. If the application is already running, then the effect of the Run command depends

on the application. Some applications are not affected; others, such as the Scriptable Text Editor, repeat their startup procedures each time they receive a Run command.

APPLESCRIPT COMMAND SYNTAX

```
run [ scriptObjectVariable ]
```

APPLICATION COMMAND SYNTAX

```
run [ referenceToApplication ]
```

PARAMETERS

scriptObjectVariable
: A variable identifier whose value is a script object. This parameter is optional if the Run command is used within an appropriate Tell statement.
Class: Script

referenceToApplication
: A reference of the form `application` *nameString* (see "Notes"). This parameter is optional if the Run command is used within an appropriate Tell statement.
Class: Reference

RESULT

The AppleScript Run command returns the result, if any, returned by the specified script object's Run handler.

The application Run command doesn't return a result.

EXAMPLES

```
run application "Scriptable Text Editor"

tell application "Scriptable Text Editor"
   run
end tell
```

NOTES

To specify the name (*nameString*) of an application to run, use a string of the form "*Disk*:*Folder1*:*Folder2*:...:*ApplicationName*"; for details, see "References to Applications," which begins on page 146. You can also specify a string with only an application name ("*ApplicationName*"). In this case, if the application is not already running, AppleScript attempts to find the application in the current directory.

AppleScript sends an implicit Run command whenever it begins to execute a Tell statement whose target is an application that is not already open. This can cause problems with applications such as Scriptable Text Editor that normally perform specific tasks on startup, such as opening a new window. To launch an application without invoking its usual startup behavior, use the Launch command as described on page 103. For information about using the Run and Launch commands with script applications, see "Calling a Script Application,"which begins on page 251.

For information about Run handlers, see "Run Handlers," which begins on page 243. For information about using the Run command with script objects, see Chapter 9, "Script Objects."

Save — application command

A Save command is a request to save an object or objects.

SYNTAX

```
save referenceToObject [ in referenceToFile ]
```

PARAMETERS

referenceToObject
A reference to the object or objects to be saved.
Class: Reference

referenceToFile A reference of the form `file` *nameString* or `alias` *nameString* that specifies the file in which to save the objects (see "Notes").
Class: Reference
Default value: The file in which the object was last saved. If the object has not been saved before, the application creates a new file.

RESULT

None

EXAMPLE

```
save document "Stupendous" in file "Elephantine"
```

NOTES

To specify the name (*nameString*) of a file in which to save the specified object or objects, use a string of the form "*Disk*:*Folder1*:*Folder2*:...:*Filename*"; for details, see "References to Files," which begins on page 144. You can also specify a string with only a filename ("*Filename*"). In this case, the application attempts to find the file in the current directory.

If you use the form `file` *nameString* and the specified file is not present in the specified location, the application creates a file with the specified name in that location. If you use the form `alias` *nameString* and the specified file is not present in the specified location, the script won't compile.

Set

AppleScript command, application command

The Set command can function as an AppleScript command or an application command. The AppleScript command assigns one or more values to one or more variables. It can also be used to share data among lists, records, or script objects (see "Notes"). The application command sets the values of one or more objects.

APPLESCRIPT COMMAND SYNTAX

`set` *variablePattern* `to` *expression*

expression `returning` *variablePattern*

APPLICATION COMMAND SYNTAX

`set` *referencePattern* `to` *expression*

expression `returning` *referencePattern*

PARAMETERS

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

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

referencePattern A reference to the location whose value is to be set, or a list of reference patterns, or a record of reference patterns.
Class: Reference, list, or record

RESULT

The value assigned.

EXAMPLES

You can use the Set command to set a variable to any value:

```
set x to 5

set myList to { 1, 2, "four" }

tell application "Scriptable Text Editor"
   set x to word 1 of front document
end tell
```

These two statements are equivalent:

```
set x to 3
3 returning x
```

Similarly, the following examples are equivalent:

```
tell front document of application "Scriptable Text Editor"
   set x to word 1
end tell

tell front document of application "Scriptable Text Editor"
   word 1 returning x
end tell
```

In addition to setting a variable to a single value, you can set patterns of variables to patterns of values. For example, this script sets a list of two variables to the position of the front window.

```
tell application "Scriptable Text Editor"
   set {x, y} to position of front window
end tell
```

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

Patterns set with the Set command can also be more complex. Here are some examples:

```
set x to {8, 94133, {firstName:"John", lastName:"Chapman"}}
set {p, q, r} to x
(* now p, q, and r have these values:
            p = 8
            q = 94133
            r = {firstName:"John", lastName:"Chapman"} *)

tell front document of application "Scriptable Text Editor"
   set {word 1, word 2} to ¬
      {firstName of item 3 of x, lastName of item 3 of x}
end tell
--now word 1 = "John" and word 2 = "Chapman"

set {p, q, {lastName:r}} to x
(* now p, q, and r have these values: p = 8
                                      q = 94133
                                      r = "Chapman" *)
```

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

The use of the Set command with patterns is similar to the use of patterned parameters with subroutines, which is described in "Subroutines With Positional Parameters," beginning on page 235.

NOTES

If you use the Set command to set a variable to a list, record, or script object, the variable shares data with the original list, record, or script object. If you change the data of the original, the value of the variable also changes. Here's an example of how this works:

```
set myList to { 1, 2, 3 }

set yourList to myList

set item 1 of myList to 4
```

The result of these statements is that item 1 of both `myList` and `yourList` is 4.

Data sharing promotes efficiency when using large data structures. Rather than making copies of shared data, the same data can belong to multiple structures. When one structure is updated, the others are automatically updated.

IMPORTANT

To avoid data sharing for lists, records, and script objects, use the Copy command instead of the Set command. ▲

Only data in lists, records, and script objects can be shared; you cannot share other values. Moreover, you can share data only on the same computer, and the shared structures must all be in the same script.

CHAPTER 5

Objects and References

Objects are the things in applications, the Operating System, or AppleScript that can respond to commands. For example, application objects are objects stored in applications and their documents. Usually, they are identifiable items that users can manipulate in applications, such as windows, words, characters, and paragraphs in a text-editing application. Objects can contain data, in the form of values, properties, and elements, that can change over time.

Each object belongs to an object class, which is a category for objects that have similar characteristics and respond to the same commands. To find out what types of data an object contains, or which commands it can respond to, you can check its object class definition. To refer to objects from scripts, you use references, which are compound names, similar to paths or addresses, that identify objects or groups of objects.

This chapter describes how to interpret object class definitions and how to use references to specify objects. For examples of object class definitions provided by an application, see "Scriptable Text Editor Object Class Definitions," which begins on page 318.

Most objects are contained in applications. It is also possible to create another type of object, called a script object, that can be stored in scripts or saved in files. For information about script objects, see Chapter 9, "Script Objects."

Using Object Class Definitions

Object class definitions describe what objects that belong to a particular class have in common. This guide contains object class definitions for system objects (in this chapter) and Scriptable Text Editor objects (in Appendix B). This section describes the information contained in object class definitions.

Figure 5-1 shows a sample object class definition for a paragraph object. The definition contains four types of information: properties, element classes, commands handled, and default value class returned. The sections that follow describe the kinds of information provided by an object class definition.

Properties

A **property** of an object is a characteristic that has a single value, such as the name of a window or the font of a character. Properties are contained by objects in much the same way that elements are. The main difference is that each of an object's properties has only one value, whereas an object may have many different elements of a single class. Properties of an object are distinguished from each other by their unique labels.

The definitions for two properties (from a total of six) are visible in the object class definition shown in Figure 5-1. These properties' labels are Font and Size. The definition also lists the class to which each property belongs. For example, the class of the Font property is String, indicating that the value of the Font property is a character string. Properties can belong to object or value classes.

Element Classes

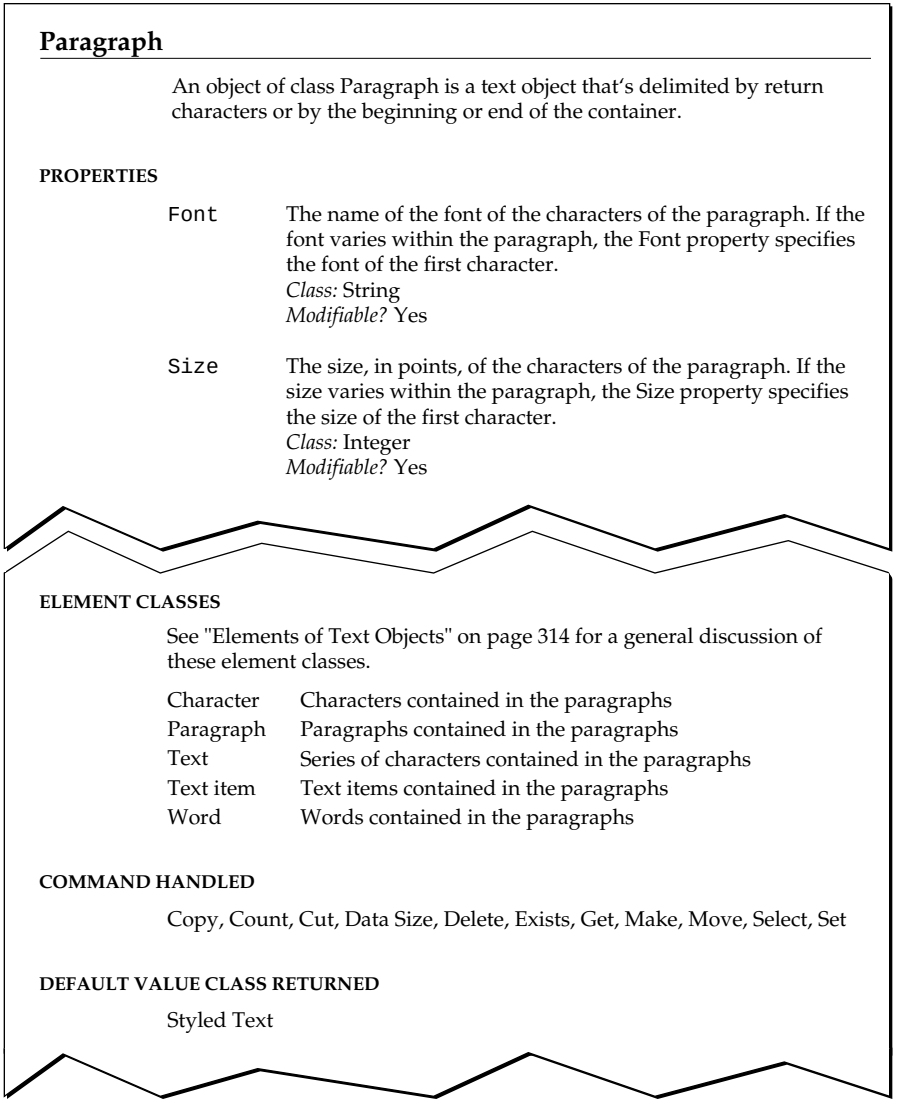
Elements are objects contained within an object. The element classes listed in an object class definition indicate what kinds of elements objects of that object class can contain. An object can contain many elements or none, and the number of elements of a particular class that it contains may change over time. Most application and system objects can contain elements.

The definition in Figure 5-1 shows that a paragraph object can include word elements. It is possible for a paragraph to have no words. At a later time, the same paragraph might have many words.

Commands Handled

Objects that belong to the same class can respond to the same commands. Object class definitions list the commands to which all objects of that class respond.

Figure 5-1 The Scriptable Text Editor's object class definition for paragraph objects



Paragraph

An object of class Paragraph is a text object that's delimited by return characters or by the beginning or end of the container.

PROPERTIES

`Font` The name of the font of the characters of the paragraph. If the font varies within the paragraph, the Font property specifies the font of the first character.
Class: String
Modifiable? Yes

`Size` The size, in points, of the characters of the paragraph. If the size varies within the paragraph, the Size property specifies the size of the first character.
Class: Integer
Modifiable? Yes

ELEMENT CLASSES

See "Elements of Text Objects" on page 314 for a general discussion of these element classes.

Character	Characters contained in the paragraphs
Paragraph	Paragraphs contained in the paragraphs
Text	Series of characters contained in the paragraphs
Text item	Text items contained in the paragraphs
Word	Words contained in the paragraphs

COMMAND HANDLED

Copy, Count, Cut, Data Size, Delete, Exists, Get, Make, Move, Select, Set

DEFAULT VALUE CLASS RETURNED

Styled Text

The definition in Figure 5-1 shows that all paragraph objects respond to the Copy, Count, Cut, Data Size, Delete, Exists, Get, Make, Move, Select, and Set commands.

Default Value Class Returned

Each object has a value. For example, the value of a paragraph object is a string that includes style and font information. You can get the value of a system or application object by sending it a Get command or simply referring to it in a script. If the Get command doesn't specify a value class for the value returned, the default value class is used. For example, according to the definition of a paragraph in Figure 5-1 a Get command that specifies a paragraph without specifying a value class returns styled text.

References

A **reference** is a phrase that specifies one or more objects. You use references to identify objects within applications. An example of a reference is

```
word 5 of paragraph 10 of document "Work in Progress"
```

which specifies a word object in the document named Work in Progress.

A reference describes what type of object you're looking for, where to look for the object, and how to distinguish the object from other objects of the same type. These three types of information—the *class*, or type; the *container*, or location; and the *reference form*, or distinguishing information—allow you to specify any object of an application.

In general, you list the class and distinguishing information at the beginning of a reference, followed by the container. In the previous example, the class of the object is `word`. The container is the phrase `paragraph 10 of document "Work in Progress"`. The distinguishing information (the reference form) is the combination of the class, `word`, and an index value, 5, which together indicate the fifth word.

References allow you to identify objects in a flexible and intuitive way. Just as there might be several ways to identify an object on the desktop, AppleScript has different reference forms that allow you to specify the same object in different ways. For example, here's another way to specify the fifth word of a document:

```
word after word 4 of document "Work in Progress"
```

To write effective scripts, you should be familiar with AppleScript's reference forms and know how to use containers and reference forms to identify the objects you want to manipulate. The sections that follow describe containers and reference forms.

Containers

A **container** is an object that contains one or more objects or properties. In a reference, the container specifies where to find an object or a property. To specify a container, use the word `of` or `in`, as in

```
word 5 of paragraph 10
```

and

```
character 2 in word 12
```

where `word 5` is contained in a paragraph object and `character 2` is contained in a word object.

A container can be an object or a series of objects. In a series, list the smallest object first, followed by the larger objects that contain it. Use the word `of` or `in` to separate each object from its larger, containing object. For example, in

```
word 5 of paragraph 10 of document "Report"
```

`word 5` is contained by the larger object, `paragraph 10`; `paragraph 10` is contained by the larger object `document "Report".`

You can also use the possessive form (`'s`) to specify containers. If you use the possessive form, list the container before the object it contains. For example, in

```
first window's name
```

the container is `first window`. The object it contains is a Name property.

All properties and elements have containers. The previous example specified the Name property of a window, which is contained in a window object. Similarly, the following example specifies the Style property, which is contained in a character object.

```
style of first character
```

Complete and Partial References

A **complete reference** has enough information to identify an object or objects uniquely. For a reference to an application object to be complete, its outermost container must be the application itself, as in

```
paragraph 10 of document "Report" of application ¬
   "Scriptable Text Editor"
```

In contrast, **partial references** do not specify enough information to identify an object or objects uniquely; for example:

```
word 1 of paragraph 10
```

When AppleScript encounters a partial reference, it attempts to use the default target specified in the Tell statement to complete the reference. The default target of a Tell statement is the object that receives commands if no other object is specified. For example, the following Tell statement tells the Scriptable Text Editor to delete the first paragraph of the front document.

```
tell paragraph 1 of front document of application ¬
   "Scriptable Text Editor"
      delete
end tell
```

Similarly, the following Tell statement tells the Scriptable Text Editor to delete the third word of the first paragraph of the front document.

```
tell paragraph 1 of front document of application ¬
   "Scriptable Text Editor"
      delete word 3
end tell
```

Tell statements can contain other Tell statements, called nested Tell statements. When AppleScript encounters a partial reference in a nested Tell statement, it tries to complete the reference starting with the innermost Tell statement. If that does not provide enough information, AppleScript uses the direct object of the next Tell statement, and so on. For example, the following Tell statement is equivalent to the previous example.

```
tell front document of application "Scriptable Text Editor"
   tell paragraph 1
      tell word 3
         delete
      end tell
   end tell
end tell
```

Reference Forms

A **reference form** is the syntax, or rule, for writing a phrase that identifies an object or group of objects. For example, the Index reference form allows you to identify an object by its number, as in

```
word 5 of paragraph 10
```

AppleScript includes other reference forms for identifying objects in applications. Table 5-1 summarizes the reference forms you can use to identify objects. Each section that follows includes a brief explanation of the reference form, a syntax summary, and examples of how to use the reference form to specify application objects. The Filter reference form is described in more detail in “Using the Filter Reference Form,” which begins on page 140.

Table 5-1 Reference forms

Reference form	Purpose
Arbitrary Element	Specifies an arbitrary object in a container
Every Element	Specifies every object of a particular class in a container
Filter	Specifies every object in a particular container that matches conditions defined in a Boolean expression
ID	Specifies an object by its ID property
Index	Specifies the position of an object with respect to the beginning or end of a container
Middle Element	Specifies the middle object in a container
Name	Specifies an object by its Name property
Property	Specifies a property of an application object, a record, a script object, or a date
Range	Specifies a series of objects
Relative	Specifies the position of an object in relation to another object

Arbitrary Element

The **Arbitrary Element** reference form specifies an arbitrary object in a container. If the container is a value (such as a list), AppleScript chooses an object at random (that is, it uses a random-number generator to choose the object). If the container is an application object, it is up to the application to choose an object. It can choose a random object or any object at all.

SYNTAX

`some` *className*

where

className is the class identifier for the desired object.

EXAMPLES

```
some word of paragraph 5
some word whose style contains outline
```

Every Element

The **Every Element** reference form specifies every object of a particular class in a container.

SYNTAX

`every` *className*

pluralClassName

where

className is a singular class name (such as word or paragraph).

pluralClassName is the plural form defined by AppleScript or an application (such as `words` or `paragraphs`). The plural form of an object class name has the same effect as the word `every` before an object class name. Plural forms are listed in application dictionaries.

VALUE

The value of an Every Element reference is a list of the objects in the container. If the container does not contain any objects of the specified class, the list is an empty list. For example, the value of the expression

```
every paragraph of {1, 2, 3}
```

is the empty list:

```
{}
```

EXAMPLES

The following example assigns a string to the variable `myString`, and then uses the Every Element reference form to specify every word contained in the string.

```
set myString to "That's all, folks"
every word of myString
```

The value of the reference `every word of myString` is a list with three items:

```
{"That's", "all", "folks"}
```

The following reference specifies the same list:

```
words of myString
```

The following references specify a list of all the words in the first paragraph of a document.

```
tell front document of application "Scriptable Text Editor"
   every word of paragraph 1
   words of paragraph 1
end tell
```

NOTES

If you specify an Every Element reference as the container for a property or object, the result is a list containing the specified property or object for each object of the container. The number of items in the list is the same as the number of objects in the container. For example, the value of the reference

```
length of every word
```

is a list such as

```
{ 2, 3, 6 }
```

The first item in the list is the length of the first word, the second item is the length of the second word, and so on.

Filter

The **Filter** reference form specifies all objects in a container that match one or more conditions specified in a Boolean expression. The Filter reference form specifies application objects only. It cannot be used to specify values. (For information about how to use the Filter reference form, refer to the next section, "Using the Filter Reference Form," which begins on page 140.)

SYNTAX

referenceToObject `( whose | where )` *Boolean*

where

referenceToObject is a reference that specifies one or more objects.

Boolean is any Boolean expression.

The words `whose` and `where` have the same meaning.

EXAMPLES

The following are some examples of references that use the Filter reference form. For examples with explanations, see the next section, "Using the Filter Reference Form," which begins on page 140.

```
every paragraph whose first word = last word
first word whose style contains italic
paragraph whose first word = last word
words whose style contains italic
every word whose size > 12 and font = "Palatino"
paragraphs where (count of characters) > 10
every word where it contains "ly"
every word where it ≠ "the"
```

NOTES

Except for the Every Element reference form, the application returns an error if no objects pass the test or tests. For the Every Element reference form, the application returns an empty list if no objects pass the test or tests.

To specify a container after a filter, you must enclose the filter and the reference it applies to in parentheses. For example, the parentheses around `words 1 thru 5 whose first character = "M"` in the following reference are required because the container `of paragraph 5` follows the filter.

```
(words 1 thru 5 whose first character = "M") of paragraph 5
```

ID

The **ID** reference form specifies an object by the value of its ID property. You can use this reference form only for objects that have an ID property.

SYNTAX

className `id` *IDvalue*

where

className is the class identifier for the specified object.

IDvalue is the value of the object's ID property.

EXAMPLES

```
document ID 9096
window id 777
```

NOTES

Although ID properties are most often integers, an ID property can belong to any class. An application that includes ID properties must guarantee that the IDs are unique within a container. Some applications may also provide

additional guarantees, such as ensuring the uniqueness of an ID among all objects.

The value of an ID property is not modifiable. It does not change even if the object is moved within the container. This allows you to save an object's ID and use it to refer to the object for as long as the object exists.

Applications are not required to support ID properties. To find out if or how an application uses ID properties, see the documentation for the application.

Index

The **Index** reference form specifies an object or a location by describing its position with respect to the beginning or end of a container.

SYNTAX

className `[ index ]` *integer*

integer`(st | nd | rd | th )` *className*

```
( first | second | third | fourth | fifth | sixth |
    seventh | eighth | ninth | tenth ) className
```

```
( last | front | back ) className
```

where

className is the class identifier of the object being specified.

integer is an integer that describes the position of the object in relation to the beginning of the container (if *integer* is a positive integer) or the end of the container (if *integer* is a negative integer).

The forms `first`, `second`, and so on are equivalent to the corresponding integer forms (for example, `second word` is equivalent to `word 2`). For objects whose index is greater than 10, you can use the forms `12th`, `23rd`, `101st`, etc. (Note that any integer followed by any of the suffixes listed is valid; for example, you can use `11rd` to refer to the eleventh object.)

The `front` form (for example, `front window`) is equivalent to *className* `1` or `first` *className*. The `last` and `back` forms (for example, `last word` and `back window`) refer to the last object in a container. They are equivalent to *className* `-1`.

The following forms refer to insertion points and are used to specify locations:

```
beginning | front

end | back
```

The `beginning` and `front` forms are equivalent and refer to the first insertion point of the container (`insertion point 1`). The `end` and `back` forms are equivalent and refer to the last insertion point of the container (`insertion point -1`).

EXAMPLES

The following references specify the second word from the beginning of the third paragraph.

```
word 2 of paragraph 3
2nd word of paragraph 3
second word of paragraph 3
```

The following references specify the last word in the third paragraph.

```
word –1 of paragraph 3
last word of paragraph 3
```

The following reference specifies the next-to-last word in the third paragraph.

```
word –2 of paragraph 3
```

The following references refer to the first insertion point of the document called Introduction.

```
beginning of document "Introduction"
front of document "Introduction"
```

The following example contains two references. The first is a reference to the tenth word of the document called Introduction. The second is a reference to the last insertion point in the same document.

```
tell application "Scriptable Text Editor"
   move word 10 of document "Introduction" to ¬
      end of document "Introduction"
end tell
```

Middle Element

The **Middle Element** reference form specifies the middle object of a particular class in a container.

SYNTAX

`middle` *className*

where

className is the class identifier for the specified object.

EXAMPLES

```
middle word of paragraph 1
middle item of {1, "doughnut", 33}
```

NOTES

AppleScript calculates the middle object with the expression `((n + 1) div 2)`, where `n` is the number of objects and `div` is the integer division operator. If there is an even number of objects in the container, the result is rounded down. For example, the middle word of a paragraph containing twenty words is the tenth word.

Name

The **Name** reference form specifies an object by name. Most applications determine this by examining an object's Name property.

SYNTAX

className `[ named ]` *nameString*

where

className is the class identifier for the specified object.

nameString is the value of the object's Name property (see "Notes").

EXAMPLES

```
document "Report"

window named "Help"

application "Macintosh HD:Applications:Scriptable Text Editor"
```

NOTES

In some applications, it is possible to have multiple objects of the same class in the same container with the same name. In such cases, it is up to the application to determine which object is specified by a Name reference.

For applications and files, the *nameString* parameter can be a string of the form "*Disk*:*Folder1*:*Folder2*:...*FileName*"; for details, see "References to Files and Applications," which begins on page 143.

For more information about Name properties of specific types of objects, see the definitions for object classes provided by the AppleScript documentation or the application's documentation.

Property

The **Property** reference form specifies a property of an application object, a script object, a record, or a date.

SYNTAX

propertyLabel

where

propertyLabel is the label for the property.

EXAMPLES

The following example is a reference to the Name property of the front window. It lists the label for the property (`name`) and its container (`front window`).

```
name of front window
```

The following example is a reference to the UnitPrice property of a record. (A record is an AppleScript value that consists of a collection of properties. For more information about records, see Chapter 3, "Values.") The label of the property is UnitPrice and the container is the record.

```
UnitPrice of {Product:"Super Snack", UnitPrice:0.85, Quantity:10}
```

NOTES

Property labels are listed in object class definitions in application dictionaries. Because a property's label is unique among the properties of an object, the label is all you need to distinguish a property from all the other properties of the object. Unlike other reference forms, there is no need to specify the class of the object.

Range

The **Range** reference form specifies a series of objects of the same class in the same container. You can specify the objects with a pair of indexes (such as `words 12 thru 24`) or with a pair of boundary objects (such as `words from paragraph 3 to paragraph 5`).

SYNTAX

`every` *className* `from` *boundaryReference1* `to` *boundaryReference2*

pluralClassName `from` *boundaryReference1* `to` *boundaryReference2*

className *startIndex* `( thru | through )` *stopIndex*

pluralclassName *startIndex* `( thru | through )` *stopIndex*

where

className is a singular class ID (such as `word` or `paragraph`).

pluralclassName is the plural class identifier defined by AppleScript or an application (such as `words` or `paragraphs`).

boundaryReference1 and *boundaryReference2* are references to objects that bound the range. The range includes the boundary objects. You can use the reserved word `beginning` in place of *boundaryReference1* to indicate the position before the first object of the container. Similarly, you can use the reserved word `end` in place of *boundaryReference2* to indicate the position after the last object in the container.

startIndex and *stopIndex* are the indexes of the first and last object of the range (such as `1` and `10` in `words 1 thru 10`).

VALUE

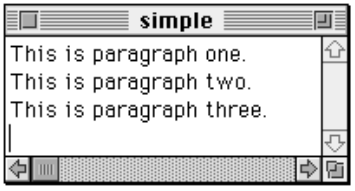
The value of a Range reference is a list of the objects in the range. If the specified container does not contain all of the objects specified in the range, an error is returned. For example, the following reference results in an error.

```
paragraphs 1 thru 3 of {1, 2, 3}
--results in an error
```

EXAMPLES

The following examples and results use the Scriptable Text Editor document shown in Figure 5-2.

Figure 5-2 The Scriptable Text Editor document “simple”



In the following example, the phrase `words from paragraph 1 to paragraph 2` is a range reference that specifies the list of the words in the first and second paragraphs.

```
tell document "simple" of application "Scriptable Text Editor"
   get words from paragraph 1 to paragraph 2
end tell
--result: {"This", "is", "paragraph", "one", ¬
               "This", "is", "paragraph", "two"}
```

In the following example, the phrase `words of paragraphs 1 thru 2` is a reference that consists of the reference `words` (a synonym for `every word`) and the container `paragraphs 1 thru 2` (a range reference).

```
tell document "simple" of application "Scriptable Text Editor"
   get words of paragraphs 1 thru 2
end tell
--result: {{"This", "is", "paragraph", "one"}, ¬
               {"This", "is", "paragraph", "two"}}
```

To get the result, AppleScript first gets the value of the container, which is a list of two paragraphs:

```
{"This is paragraph one.", "This is paragraph two."}
```

AppleScript then gets every word of the resulting list, which results in a list of two lists:

```
{{"This", "is", "paragraph", "one"}, ¬
      {"This", "is","paragraph", "two"}}
```

NOTES

If you specify a Range reference as the container for a property or object, as in

```
font of words 4 thru 6 of document "Mail Form"
```

the result is a list containing the specified property or object for each object of the container. The number of items in the list is the same as the number of objects in the container. For example, the value of the reference in this example might be

```
{helvetica, palatino, geneva}
```

The first item in the list is the font of the fourth word, the second item is the font of the fifth word, and the third item is the font of the sixth word.

To refer to a contiguous series of characters—instead of a list—when specifying a range of text objects, use the text element. Text is an element of most text objects, including all Scriptable Text Editor text objects. Text is also an element of AppleScript strings.

For example, compare the values of the following references.

```
words 1 thru 4 of "We're all in this together"
--result: {"We're", "all", "in", "this"}

text from word 1 to word 4 of "We're all in this together"
--result: "We're all in this"
```

```
text of words 1 thru 4 of "We're all in this together"
--result: "We're all in this"
```

Relative

The **Relative** reference form specifies an object or a location by describing its position in relation to another object, known as the base, in the same container.

SYNTAX

[*className*] (before | [in] front of) *baseReference*

[*className*] (after | [in] back of | behind) *baseReference*

where

className is the class identifier of the specified object. If you leave out this parameter, AppleScript assumes you want an insertion point.

baseReference is a reference to the base object.

The `before` and `in front of` forms, which are equivalent, refer to the object immediately preceding the base object. The `after`, `in back of`, and `behind` forms are equivalent and refer to the object immediately after the base.

EXAMPLES

The following references specify the word immediately before the first figure.

```
word before figure 1
word in front of figure 1
```

The following examples specify the insertion point immediately before the tenth paragraph.

```
before paragraph 10
in front of paragraph 10
```

The following example contains three references. The first two are Index references that specify the front document and the first word. The third is a Relative reference that specifies the insertion point before the tenth paragraph. The command moves the first word to the insertion point before the tenth paragraph.

```
tell front document of application "Scriptable Text Editor"
   move word 1 to before paragraph 10
end tell
```

NOTES

You can specify only a single object with the Relative form. You can use the form to specify an object that is either before or after the base object.

If it is possible for the specified object to contain the base object (as in the expression `paragraph before word 99`), the reference does not specify the container but instead specifies the object immediately before or after the container of the base object. For example, the expression `paragraph before word 99` specifies the paragraph immediately before the paragraph containing the ninety-ninth word.

All applications allow you to specify a base object belonging to the same class as the desired object (such as `window in back of window "Big"`). Not all allow you to specify a base of a different object class (such as `word before figure 1`). The possible base classes for a particular class are up to each application.

Using the Filter Reference Form

When specifying one or more objects contained in an application object, you can use the Filter reference form to include an optional filter. A **filter** restricts the objects you specify to those that match one or more conditions.

For example, compare this reference without a filter

```
every word of paragraph 5
```

to the same reference with a filter:

```
every word of paragraph 5 where character 1 = "M"
```

The first reference specifies all the words in the fifth paragraph. The second reference, which includes the filter `where character 1 = "M"`, specifies all the words in the same container whose first character is “M”. Words that do not pass this test are filtered out.

In effect, a filter reduces the number of objects in the container. Instead of specifying every word of the fifth paragraph, the reference

```
every word of paragraph 5 whose first character = "M"
```

specifies every word of a smaller container, the words of the fifth paragraph whose first characters are “M”. Similarly,

```
words 1 thru 5 of paragraph 5 whose first character = "M"
```

specifies the first five words of the same smaller container.

To determine the objects in the smaller container, the application applies the filter to all of the objects of the specified class in the specified container—in this case, the words in the fifth paragraph. The application uses the filter to test each object in turn, starting with the first.

Within a filter, the predefined variable `it` refers to the object currently being tested. For example, in the reference

```
second paragraph of document "Product Intro"  ¬
   where it contains "dynamo"
```

the word `it` refers to each paragraph in the document Product Intro. The filter, `contains "dynamo"`, is applied to each paragraph in the document, resulting in a smaller container whose paragraphs all contain the string `"dynamo"`. The reference specifies the second paragraph of that smaller container.

A Filter reference includes one or more tests. Each **test** is a Boolean expression that compares a property or element of each object being tested, or the objects themselves, with another object or value. Table 5-2 shows some Filter references, the Boolean expressions they contain, and what is being tested in each reference.

Table 5-2 Boolean expressions and tests in Filter references

Filter reference	Boolean expression	What is being tested
`words whose length > 10`	`length > 10`	The length property of each word
`words whose first character = "M"`	`first character = "M"`	The first character of each word
`words where it contains "el"`	`it contains "el"`	The words themselves

Note
A test can be any Boolean expression (such as `words where 1 < 2`), but only those that actually test objects or their contents are useful for filtering objects. ◆

To include more than one test in a filter, link the tests with Boolean operators, as in

```
words whose length > 10 and tenth character = "M"
```

The Boolean operator And indicates that each word must pass both tests to be included in the smaller container. Another example is

```
words where it contains "M" or it contains "G"
```

The Boolean operator Or indicates that the words can pass either test to be included in the smaller container.

Because each test is a Boolean expression, it can also include the Boolean operator Not. For example, the reference

```
words whose length > 10 and not it contains "M"
```

refers to only those words containing more than ten characters and not containing the letter "M." The expression `it contains "M"` is a valid Boolean expression, and applying the Boolean Not operator to it, as in

```
not (it contains "M")
```

inverts the value of the expression, so that a `true` value becomes `false`, and a `false` value becomes `true`.

A more elegant way to apply the Boolean Not operator to the expression `it contains "M"` is

```
it doesn't contain "M"
```

The expression `it doesn't contain "M"` is a **synonym** for the expression `not (it contains "M")`. AppleScript supports synonyms for many of its operators. Using a synonym doesn't change the meaning of an expression, but it can make the expression easier to read. Operators and synonyms are listed in Chapter 6, "Expressions."

References to Files and Applications

Several application commands and scripting addition commands allow you to use the Name reference form to identify a file or an application as a parameter, including applications on remote machines connected to an AppleTalk network. AppleScript treats references to file, alias, application, machine, and zone objects differently from other references that use the Name reference form. AppleScript takes care of locating these objects, but uses the equivalent definitions in an application's dictionary (if any) to determine their characteristics.

References to Files

You can use either of these forms to refer to any file:

`file` *nameString*

`alias` *nameString*

where

nameString is a string of the form "*Disk*`:`*Folder1*`:`*Folder2*`:...:`*Filename*" that specifies exactly where the file is stored or a string that consists of the file's name only. *Disk* specifies the disk on the local computer on which the application is stored, *Folder1*`:`*Folder2*`:...` specifies the sequence of folders that you would have to open to find the application on the local computer, and *fileName* specifies the name of the file. AppleScript doesn't distinguish uppercase letters from lowercase letters in filenames.

If *nameString* consists of the file's name only, AppleScript attempts to locate the file in the current directory for the application from which the script is being run (for example, Script Editor). The **current directory** is the folder or volume whose contents you can see when you choose Open or the equivalent command from the application's File menu. By default, the current directory for any application is the folder or volume in which the application is stored; but the current directory may change as you open and close files and folders from within the application. To be sure that a command acts on the correct file, specify the entire pathname, including the names of the volume and the entire sequence of folders that you would have to open to find the file.

If you use a reference of the form `file` *nameString*, AppleScript doesn't attempt to locate the file until the script is actually run. When the script is run, the file must be located in the specified folder (or, if only a filename was provided, in the current directory) for AppleScript to locate it successfully. Some commands, such as the Save command, create a file with the specified name in the specified location if it doesn't already exist.

To save a reference of the form `file` *nameString* in a variable, you must use the A Reference To operator as shown in the example that follows. (To use this script successfully, substitute a pathname that corresponds to a volume, folder or folders, and file that actually exist on your computer.)

```
set fileRef to a reference to file "Hard Disk:June Sales"
tell application "Scriptable Text Editor"
   open fileRef
end
```

If you use a reference of the form `alias` *nameString*, AppleScript creates an **alias** for the file—that is, a representation of the file, much like an alias icon on the desktop, that identifies the file no matter where it is located. AppleScript attempts to locate the file whenever you compile the script—that is, whenever you modify the script and then attempt to check its syntax, save it, or run it.

AppleScript treats an alias like a value that can be stored in a variable and passed around within a script. You don't need to use the A Reference To operator. For example, this script first saves an alias in the variable `fileRef`, then uses the variable in a Tell statement that opens the file.

```
set fileRef to alias "Hard Disk:June Sales"
tell application "Scriptable Text Editor"
   open fileRef
end
```

If you save this script as a script application or compiled script, move the file June Sales to another location, then open the script again, the statement `alias "Hard Disk:June Sales"` or its equivalent changes to reflect the file's new location, and the script still works correctly.

The difference between the forms `file` *nameString* and `alias` *nameString* is also apparent when the file in question is located on a remote computer. If you use the form `file` *nameString*, AppleScript doesn't attempt to locate the file until you actually run the script. If you use the form `alias` *nameString*, AppleScript also attempts to locate the file whenever you compile the script, requiring appropriate access privileges and possibly a password each time.

The actions you can perform on a specific file depend on the way the application that created the file defines a file object. If an application provides its own definition for a file object, AppleScript locates the file as described in this section, but uses the definition in the application's dictionary to determine the characteristics of the object, such as its properties and the commands it can handle. For the Scriptable Text Editor's definition of a file, see page 328.

References to Applications

You can use this form to refer to any application:

```
application applicationNameString                                    ¬
   [ of machine computerName [ of zone AppleTalkZoneName ] ]
```

where

applicationNameString is either a string of the form "*Disk*:*Folder1*:*Folder2*:...:*ApplicationName*" that specifies where the application is stored on the local computer or a string that consists of the name of the application. *Disk* specifies the disk on the local computer on which the application is stored, *Folder1*:*Folder2*:... specifies the sequence of folders that you would have to open to find the application on the local computer, and *ApplicationName* specifies the name of the application. If it is located on a remote computer, the application must be running and *applicationNameString* must be the name of the application as listed in the Application menu on that computer. AppleScript doesn't distinguish uppercase letters from lowercase letters in application names.

computerName (a string) is the Macintosh Name assigned in the Sharing Setup control panel of the computer on which the specified application is running. This portion of the reference is required if the application is located on a remote computer.

AppleTalkZoneName (a string) is the name of the zone, if any, in which the specified remote computer is located. The name must appear in the list of AppleTalk Zones displayed in the Chooser.

After a script is compiled, a reference to an application on the local computer identifies the application no matter where it is located on that computer. This behavior resembles the behavior of an alias. However, a reference to an application on a remote computer won't compile unless the application is running and several other conditions are met; see "References to Remote Applications" on page 148 for details.

The actions you can perform on a specific application depend on the way the application that created the file defines an application object. AppleScript always locates the application as described in the sections that follow, but uses

the definition in the application's dictionary to determine the characteristics of the object, such as its properties and the commands it can handle. For the Scriptable Text Editor's definition of an application, see page 318.

References to Local Applications

You can specify an application on the local computer with a string of the form "*Disk*:*Folder1*:*Folder2*:...:*ApplicationName*" that specifies the application's exact location. If AppleScript can't find the application in that location, it displays a directory dialog box asking where the application is located.

You can also specify an application on the local computer with only the application's name ("*ApplicationName*"). In this case, AppleScript attempts to find an application of that name among currently running applications. If the application isn't running, AppleScript attempts to locate it in the current directory. If the application isn't in the current directory, AppleScript displays a directory dialog box asking where the application is located. If the name of the application you select is different from the name specified in the script, the name in the script changes to match the name of the application you select.

When you run a script on the same computer on which it was compiled (that is, on which it was last run or saved, or had its syntax checked), AppleScript finds the application you specified in the original script even if you have moved it or changed its name. If the application has been removed, AppleScript searches for another version of the same application.

As with aliases, it is often convenient to store a reference to an application in a variable:

```
set x to application "Scriptable Text Editor"
x
tell x to quit
```

If you save this script as a script application or compiled script, move the Scriptable Text Editor application to another location, change its name, then open the script again, the name "Scriptable Text Editor" in the script changes to reflect the application's new name, and the script still works correctly.

References to Remote Applications

If the application is on a remote computer, you must specify its name as it would be listed in the Application menu, the name of the computer, and if necessary the name of the zone in which the computer is located:

```
quit application "Scriptable Text Editor" ¬
   of machine "Mr.Science" of zone "Far Side"
```

The specified remote application must be running. AppleScript doesn't open applications on remote computers. In addition, the computer that contains the application and the computer on which the script is run must be connected to an AppleTalk network, program linking (set with the Sharing Setup control panel) must be enabled, access for the user (set with the Users & Groups control panel) must be provided, and the application must allow remote program linking (set by selecting the application, choosing Sharing from the File menu, and selecting the checkbox labeled "Allow remote program linking"). For information about these menus and control panels, see the user's guide for your Macintosh computer.

This script sends several commands to an application on a remote computer:

```
tell application "Scriptable Text Editor" of ¬
   machine "Pegi's Mac" of zone "Publications"
      open file "HD:Reports:Status Report"
      set pegisReport to text from paragraph 1 to ¬
         paragraph 4 of document "Status Report"
      close document "Status Report"
end tell
tell application "Scriptable Text Editor"
   open file "Department Status"
   copy pegisReport to end of document "Department Status"
tell app "Scriptable Text Editor" to quit
```

CHAPTER 6

Expressions

An expression is any series of AppleScript words that has a value. You use expressions to represent or derive values in scripts. When AppleScript encounters an expression, it converts it into an equivalent value. This is known as evaluation.

Chapter 3 describes and gives examples of the simplest kinds of expressions, called literal expressions, which are representations of values in scripts. This chapter begins by describing how to evaluate expressions. It then describes five additional types of expressions:

- variables and script properties, which are named containers for values
- the AppleScript property Text Item Delimiters, which determines the text item delimiters used by AppleScript in all scripts
- reference expressions, which are expressions that derive the value of an object
- operations, which are expressions that derive values from other values

Results of Expressions

The result of any expression is its value.You can use the Script Editor to display the result of an expression by typing an expression on a line by itself and running the script. AppleScript returns the value of the expression. Here's an example:

1. **Open the Script Editor if it is not already open.**
2. **Type the following expression in the editor subwindow:**

```
3 + 4
```

3. **Click the Run button in the Script Editor window.**
   This causes AppleScript to evaluate the expression.
4. **Choose Show Result from the Controls menu.**
   The result window displays the result of the evaluation, 7.

Variables

A **variable** is a named container in which to store a value. When AppleScript encounters a variable in a statement, it evaluates the variable by getting its value. Variables are contained in a script, not in an application, and their values are normally lost when you close the script that contains them. If you need to keep track of variable values that are persistent even after you close a script or shut down your computer, use properties instead of variables. See "Script Properties," which begins on page 156, for more information.

Unlike variables in many other programming languages, AppleScript variables can hold values of any class. For example, you can use the following sequence of assignment statements to set `x` to a string value, an integer value, and finally a Boolean value:

```
set x to "Title"
set x to 12
set x to True
```

The name of a variable is a series of characters, called an identifier, that you specify when you create the variable.

Creating Variables

To create a variable in AppleScript, assign it a value. There are two commands for doing this:

- Set
- Copy

With the Set command, list the variable name first, followed by the value you want to assign:

```
set myName to "Pegi"
```

With the Copy command, list the value first, followed by the variable name:

```
copy "Pegi" to myName
```

Statements like these that assign values to variables are called assignment statements.

The variable name is a series of characters called an identifier. AppleScript identifiers are not case sensitive—for example, the variables `myname`, `myName`, and `MYNAME` all represent the same value. The rules for specifying identifiers are listed in "Identifiers" on page 27.

You can list an expression in place of a value in an assignment statement. AppleScript evaluates the expression and assigns the resulting value to the variable. For example, the following statement creates a variable called `myNumber` whose value is the integer 17.

```
set myNumber to 5 + 12
```

You can also assign a reference as the value of a variable. In this case, AppleScript gets the value of the object specified in the reference and assigns it to the variable. For example, the following statement gets the value of the first word of the document called Report—a string—and stores it in a variable called `myWord`.

```
set myWord to word 1 of document "Report" of application ¬
   "Scriptable Text Editor"
```

You can do the same thing with the Copy command:

```
copy word 1 of document "Report" of application  ¬
   "Scriptable Text Editor" to myWord
```

The results of the two types of assignment statements are the same in all cases except when the value being assigned is a list, record, or script object. The Copy command makes a new copy of the list, record, or script object, and the Set command creates a variable that shares data with the original list, record, or script object. For more information, refer to “Data Sharing” on page 154.

Using Variables

To use the value of a variable in a script, include the variable in a command or expression. For example, the first statement in the following example creates a variable, called `myName`, whose value is the string `"Mitch"`. The second statement uses the variable `myName` in place of a string as the `default answer` parameter of the Display Dialog command.

```
set myName to "Mitch"
display dialog "What is your name?" default answer myName
```

If you assign a new value to a variable, it replaces the old value. The following script shows what happens when you assign a new value. It uses the Display Dialog command to display the values. Try running this script:

```
set myName to "Mitch"
display dialog ("The value of myName is now " & myName) ¬
   buttons "Sure Is" default button 1
set myName to "Warren"
display dialog ("The value of myName is now " & myName) ¬
   buttons "You Betcha" default button 1
```

The first Display Dialog command displays the value stored by the first assignment statement (the string `"Mitch"`). The next Display Dialog command displays the value after the second assignment statement (the string `"Warren"`).

The "A Reference To" Operator

To create a variable whose value is a reference instead of the value of the object specified by a reference, use the A Reference To operator. Here's an example:

```
set myDoc to a reference to document "Report" ¬
   of application "Scriptable Text Editor"
```

The value of the variable `myDoc` is the reference

```
document "Report" of application "Scriptable Text Editor"
```

After you create a variable whose value is a reference, you can use it in a script anywhere a reference is required. When AppleScript executes the statement containing the variable, it replaces the variable with the reference. For example, when AppleScript executes the statement

```
tell myDoc
   get word 1
end tell
```

it replaces the variable `myDoc` with the reference `document "Report" of application "Scriptable Text Editor"`.

The syntax for using the A Reference To operator is

`[a] ( ref [to] | reference to )` *reference*

where *reference* is a reference to an object. As indicated in the syntax description, there are many ways to shorten expressions containing A Reference To. For example, all of these expressions are equivalent:

```
set myDoc to a reference to document "Report" of ¬
   application "Scriptable Text Editor"
set myDoc to reference to document "Report" of ¬
   application "Scriptable Text Editor"
set myDoc to a ref to document "Report" of application ¬
   "Scriptable Text Editor"
```

```
set myDoc to ref to document "Report" of application ¬
   "Scriptable Text Editor"
set myDoc to ref document "Report" of application ¬
   "Scriptable Text Editor"
```

After you create a reference with the A Reference To operator, you can use the Contents property to get the value of the object that it refers to. The Contents property is the value of the object specified by a reference. For example, the result of the following expression is a string containing the text of document Report of the Scriptable Text Editor.

```
contents of myDoc
```

Data Sharing

Data sharing allows you to create two or more variables that share the same list, record, or script object data; it can be used to promote efficiency when working with large data structures. Only data in lists, records, and script objects can be shared; you cannot share other values. In addition, the shared structures must all be on the same computer.

To create a variable that shares data with another variable whose value is a list, record, or script object, use the Set command. For example, the second Set command in the following example creates the variable `yourList`, which shares data with the previously defined variable `myList`.

```
set myList to { 1, 2, 3 }
set yourList to myList  --this command creates yourList,
                        --which shares data with myList
set item 1 of myList to 4
get yourList --result:{ 4, 2, 3}
```

If you update `myList` by setting the value of its first item to 4, then the value of both `myList` and `yourList` is `{4, 2, 3}`. Rather than having multiple copies of shared data, the same data belongs to multiple structures. When one structure is updated, the other is automatically updated.

To avoid data sharing for lists, records, and script objects, use the Copy command instead of the Set command. The Copy command makes a copy of the list, record, or script object. Changing the value of the original changes does not change the value of the variable. Here's an example of using Copy instead of Set to create the variable `yourList`.

```
set myList to { 1, 2, 3 }
copy myList to yourList --this command makes a copy of
                        --mylist
set item 1 of myList to 4
get yourList --result: { 1, 2, 3 }
```

If you update `myList`, the value of `yourList` is still `{1, 2, 3}`.

Scope of Variables

The **scope** of a variable determines where else in a script you may refer to the same variable. The scope of a variable in turn depends on where you declare it and whether you declare it as global or local.

After you define a **global variable** in a script, you can make subsequent references to the same variable either at the top level of the script or in any of the script's subroutines. After you define a **local variable,** you can make subsequent references to the same variable only at the same level of the script at which you defined the variable.

AppleScript assumes that all variables defined at the top level of a script or within its subroutines are local unless you explicitly declare them as global. For more detailed information and examples of the use of variables in subroutines, see "Recursive Subroutines," which begins on page 225.

You can also declare variables within script objects. The scope of variables in a script object is limited to that script object. For more information, see "Scope of Script Variables and Properties," which begins on page 252.

Predefined Variables

Predefined variables are variables whose values are supplied by AppleScript. You can use them in scripts without setting their values. Predefined variables are global—that is, you can use them anywhere in a script.

For a summary of the predefined variables in the AppleScript English language dialect, refer to Appendix A, "The Language at a Glance."

Note
Although AppleScript does not prevent you from setting the values of predefined variables, you should treat predefined variables as constants—that is, you should never change their values. ◆

Script Properties

Script properties are labeled containers for values that you can use in much the same way you use variables. The value of a script property persists until you recompile the script that contains it, and you can easily set the property's initial value without resetting it each time the script is run. You can accomplish the same thing with a global variable, but it is usually more convenient to use a property for this purpose.

This section describes how to define script properties.

Note
The description of script properties in this section assumes that you are using the Script Editor application supplied with AppleScript. Other script editors might not support persistence of script properties. If you are using a different script editor, check its documentation to see how it handles script properties. ◆

Defining Script Properties

The syntax for defining a script property is

`( prop | property )` *propertyLabel* `:` *initialValue*

where

propertyLabel is an identifier. The rules for specifying identifiers are listed in "Identifiers" on page 27.

initialValue is the value that is assigned to the property when you first run the script that contains the property or when you save it or check its syntax.

After you define a script property, you change its value the same way you change variable values: with the Set or Copy command. You can get a script property value using the Get command or by using it in an expression.

Using Script Properties

To see how script properties work, try running the following script, which contains a script property called `theCount`.

```
property theCount : 0
set theCount to theCount+1
display dialog "The value of theCount is: " & theCount ¬
   as string
```

The first time you run the script, the value of `theCount` is set to 0. The Set command adds one to `theCount`, and the Display Dialog command displays the value of `theCount`, which is 1.

Now run the script again. The Set command adds 1 to the value of `theCount` (which is still one because it has not been reset), and the Display Dialog command reports a value of 2. If you run the script a third time, the value of `theCount` is 3, and so on.

Now save the script as a compiled script. Close the script, and then open and run it without making any changes. The value of `theCount` is one more than it was before you closed the script.

Finally, recompile the script. (You can do this by making an insignificant change, such as adding a space at the end of a line, and clicking the Check Syntax button.) The value of `theCount` is set to the initial value in the property definition. The Display Dialog command reports a value of 1.

Scope of Script Properties

Like the scope of a variable, the scope of a script property determines where else in a script you may refer to the same property ID. The scope of a property in turn depends on where you declare it.

You can declare a property at the top level of a script or at the top level of a script object. If you declare it at the top level of a script, a property identifier is visible throughout the script. If you declare it at the top level of a script object, a property identifier is visible only within that script object. After declaring a property, you can use the same identifier as a separate variable only if you first declare it as a local variable.

For detailed information and examples of the use of properties in subroutines, see "Scope of Script Variables and Properties," which begins on page 252.

AppleScript Properties

You can use the global variable `AppleScript` to get properties of AppleScript itself rather than properties of the current target. You can refer to this global variable from any part of any script. Currently, the Text Item Delimiters property is the only AppleScript property available.

Text Item Delimiters

The Text Item Delimiters property consists of a list of strings used as delimiters by AppleScript when it coerces lists to strings or gets text items from strings.

You can get and set the current value of AppleScript's Text Item Delimiters. Normally, AppleScript doesn't use any delimiters. For example, the script

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

returns this result if AppleScript's text delimiters have not been explicitly changed:

```
"breadmilkbutter10.45"
```

For printing or display purposes, it is usually preferable to set the text delimiters to something that's easier to read. For example, the script

```
set AppleScript's text item delimiters to {", "}
{"bread", "milk", "butter", 10.45} as string
```

returns this result:

```
"bread, milk, butter, 10.45"
```

The Text Item Delimiters property also allows you to extract individual names from a pathname. For example, the script

```
set AppleScript's text item delimiters to {":"}
get last text item of "Hard Disk:CD Contents:Release Notes"
```

returns the result `"Release Notes"`.

Once you change the Text Items Delimiters property, it remains set until you restart your computer. Currently, AppleScript uses only the first delimiter in the list.

You may want to use an error handler to reset the Text Item Delimiters property to its former value if an error occurs:

```
set savedTextItemDelimiters to AppleScript's text item¬
   delimiters
try
   set AppleScript's text item delimiters to {"**"}
   --rest of script...

   --finally, reset the text item delimiters:
   set AppleScript's text item delimiters to¬
      savedTextItemDelimiters
```

```
on error m number n from f to t partial result p
   --also reset text item delimiters in case of an error:
   set AppleScript's text item delimiters to ¬
      savedTextItemDelimiters
   --and resignal the error:
   error m number n from f to t partial result p
end try
```

Reference Expressions

References are compound names that refer to objects in applications, the system, or AppleScript. Because each object has a value, a reference can be used to represent a value in a script. A reference expression is a reference that AppleScript interprets as a value.

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

For example, the reference in the following example is a reference to an object. It identifies the object to which the Copy command is sent.

```
copy word 1 of front document of application "Scriptable Text Editor"
```

On the other hand, the reference in the following example is a reference expression:

```
repeat (word 1 of front document of application ¬
   "Scriptable Text Editor") times
      display dialog "Hello"
end repeat
```

When AppleScript executes the statement, it gets the value of the reference `word 1 of front document of application "Scriptable Text Editor"`—a string—and then coerces it to an integer, if possible. (For information about the Repeat statement, refer to Chapter 7, "Control Statements." For information about coercions, refer to "Coercing Values" on page 68.)

Operations

Operations are expressions that use operators to derive values from other values. AppleScript includes operators for performing arithmetic operations, comparing values, performing Boolean evaluations, and coercing values.

The values from which operators derive values are called **operands.** Each operator can handle operands of specific classes, which are defined in the definition of the operator. For example, the operands for the addition (+) operator must belong to the class Integer or Real, while the operand for the Not operator must belong to class Boolean. Certain operators work with operands from a variety of classes. For example, you can use the concatenation operator (&) to join two strings, two lists, or two records.

The result of each operation is a value of a particular class. For many operators, such as the equality operator (=) and the greater than operator (>), the class of the result is always the same—in these cases, Boolean. For other operators, such as the concatenation operator (&), the class of the result depends on the class of the operands. For example, the result of concatenating two strings is a string, but the result of concatenating two integers is a list of integers.

If you use an operator with operands of the wrong classes, AppleScript attempts to coerce the operands to the correct class, if possible. For example, the concatenation operator (&) works with strings, lists, or records. When AppleScript evaluates the following expression, it coerces the integer `66` to a string before concatenating it with the string `"Route"`.

```
"Route " & 66
--result: "Route 66"
```

When evaluating expressions containing operators, AppleScript checks the leftmost operand first. If the operand does not belong to one of the legal classes for the operator, AppleScript coerces it if possible. After coercing the leftmost operand or verifying that it belongs to a legal class, AppleScript checks the rightmost operand and coerces it (if necessary and possible) to be compatible with the leftmost operand. The exceptions to this rule are expressions with the Is Contained By, Equal, and Is Not Equal operators. AppleScript checks the rightmost operand first in expressions with the Is Contained By operator. AppleScript never coerces operands of the Equal and Is Not Equal operators.

If AppleScript cannot coerce the operands, it returns an error. For example, the addition operator (+) works with numbers (integers and real numbers) only. If you attempt to evaluate an expression such as `3 + "cat"`, you'll get an error, because AppleScript cannot coerce `"cat"` to a number.

Operations can be performed either by AppleScript or by an application. The rule is that if the leftmost operand is a value, AppleScript performs the operation, and if the leftmost operand is a reference to an application object, the application performs the operation. For example, the comparison

```
"Hello" contains word 1 of document "Report"
```

is performed by AppleScript, because the first operand is a string. Before performing the comparison, AppleScript must get the value of the first word. In contrast, the comparison

```
word 1 of document "Report" contains "Hello"
```

is performed by the application containing the document named Report.

The Is Contained By operator is an exception to this rule. In expressions with the Is Contained By operator, AppleScript performs the operation if the rightmost operand is a value and the application performs the operation if the rightmost operand is a reference to an application object.

Table 6-1 summarizes the AppleScript operators. For each operator, it includes a brief description of the operation and lists the class (or classes) of the operands and the class (or classes) of the result. A few of the operators are characters that you type with modifier keys. For these operators, the keystrokes are shown in parentheses. The section following the table provides more information about how operators treat different classes of operands.

The sections following the table contain more detailed explanations and examples of operations.

Table 6-1 AppleScript operators

Operator	Description
`and`	And. Binary logical operator that results in `true` if both the operand to its left and the operand to its right are `true`. Both of the operands must evaluate to Boolean values. When evaluating expressions containing the And operator, AppleScript checks the leftmost operand first. If its value is `false`, AppleScript does not evaluate the rightmost operand, because it already knows the expression is `false`. (This behavior is sometimes called short-circuiting.) *Class of operands*: Boolean *Class of result*: Boolean
`or`	Or. Binary logical operator that results in `true` if either the operand to its left or the operand to its right is `true`. At least one of the operands must evaluate to a Boolean value. When evaluating expressions containing the Or operator, AppleScript checks the leftmost operand first. If its value is `true`, AppleScript does not evaluate the rightmost operand, because it already knows the expression is `true`. (This behavior is sometimes called short-circuiting.) *Class of operands*: Boolean *Class of result*: Boolean
`&`	Concatenation. Binary operator that joins two values. If the operand to the left of the operator is a string, the result is a string. If the operand to the left of the operator is a record, the result is a record. If the operand to the left of the operator belongs to any other class, the result is a list. *Class of operands*: Boolean, Class Identifier, Constant, Data, Date, Integer, List, Real, Record, Reference, String *Class of result*: List, Record, String

`=`
`is`
`equal`
`equals`
`[is] equal to`
	Equal. Binary comparison operator that results in `true` if the operand to its left and the operand to its right have the same value. The operands can be of any class. The method AppleScript uses to determine equality depends on the class of the operands.
	Class of operands: Boolean, Class Identifier, Constant, Data, Date, Integer, List, Real, Record, Reference, String
	Class of result: Boolean

continued

Table 6-1 AppleScript operators (continued)

Operator	Description

≠ (Option–equal sign)
`is not`
`isn't`
`isn't equal [to]`
`is not equal [to]`
`doesn't equal`
`does not equal`
	Not equal. Binary comparison operator that results in `true` if the operand to its left and the operand to its right have different values. The operands can be of any class. The method AppleScript uses to determine equality depends on the class of the operands.
	Class of operands: Boolean, Class Identifier, Constant, Data, Date, Integer, List, Real, Record, Reference, String
	Class of result: Boolean

`>`
`[is] greater than`
`comes after`
`is not less than or equal [to]`
`isn't less than or equal [to]`
	Greater than. Binary comparison operator that results in `true` if the value of the operand to its left is greater than the value of the operand to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. The method AppleScript uses to determine which value is greater depends on the class of the operands.
	Class of operands: Date, Integer, Real, String
	Class of result: Boolean

`<`
`[is] less than`
`comes before`
`is not greater than or equal [to]`
`isn't greater than or equal [to]`
	Less than. Binary comparison operator that results in `true` if the value of the operand to its left is less than the value of the operand to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. The method AppleScript uses to determine which value is greater depends on the class of the operands.
	Class of operands: Date, Integer, Real, String
	Class of result: Boolean

≥ (Option–greater-than sign)
`>=`
`[is] greater than or equal [to]`
`is not less than`
`isn't less than`
`does not come before`
`doesn't come before`
	Greater than or equal to. Binary comparison operator that results in `true` if the value of the operand to its left is greater than or equal to the value of the operand to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. The method AppleScript uses to determine which value is greater depends on the class of the operands.
	Class of operands: Date, Integer, Real, String
	Class of result: Boolean

continued

Table 6-1 AppleScript operators (continued)

Operator	Description

≤ (Option–less-than sign)
<=
`[is] less than or equal [to]`
`is not greater than`
`isn't greater than`
`does not come after`
`doesn't come after`
	Less than or equal to. Binary comparison operator that results in `true` if the value of the operand to its left is less than or equal to the value of the operand to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. The method AppleScript uses to determine which value is greater depends on the class of the operands.
	Class of operands: Date, Integer, Real, String
	Class of result: Boolean

`start[s] with` `begin[s] with`	Starts with. Binary containment operator that results in `true` if the list or string to its right matches the beginning of the list or string to its left. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. *Class of operands*: List, String *Class of result*: Boolean
`end[s] with`	Ends with. Binary containment operator that results in `true` if the list or string to its right matches the end of the list or string to its left. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. *Class of operands*: List, String *Class of result*: Boolean
`contain[s]`	Contains. Binary containment operator that results in `true` if the list, record, or string to its right matches any part of the list, record, or string to its left. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. *Class of operands*: List, Record, String *Class of result*: Boolean

continued

Table 6-1 AppleScript operators (continued)

Operator	Description
`does not contain` `doesn't contain`	Does not contain. Binary containment operator that results in `true` if the list, record, or string to its right does not match any part of the list, record, or string to its left. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the right of the operator to the class of the operand to the left. *Class of operands*: List, Record, String *Class of result*: Boolean
`is in` `is contained by`	Is contained by. Binary containment operator that results in `true` if the list, record, or string to its left matches any part of the list, record, or string to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the left of the operator to the class of the operand to the right. *Class of operands*: List, Record, String *Class of result*: Boolean

`is not in`
`is not contained by`
`isn't contained by`
	Is not contained by. Binary containment operator that results in `true` if the list, record, or string to its left does not match any part of the list, record, or string to its right. Both operands must evaluate to values of the same class. If they don't, AppleScript attempts to coerce the operand to the left of the operator to the class of the operand to the right.
	Class of operands: List, Record, String
	Class of result: Boolean

`*`	Multiply. Binary arithmetic operator that multiplies the number to its left and the number to its right. *Class of operands*: Integer, Real *Class of result*: Integer, Real
`+`	Plus. Binary arithmetic operator that adds the number or date to its left and the number or date to its right. Only integers can be added to dates. AppleScript interprets such an integer as a number of seconds. *Class of operands*: Date, Integer, Real *Class of result*: Date, Integer, Real

continued

Table 6-1 AppleScript operators (continued)

Operator	Description
`-`	Minus. Binary or unary arithmetic operator. The binary operator subtracts the number to its right from the number or date to its left. The unary operator makes the number to its right negative. Only integers can be subtracted from dates. AppleScript interprets such an integer as a number of seconds. *Class of operands*: Date, Integer, Real *Class of result*: Date, Integer, Real
`÷` (Option-slash) `/`	Division. Binary arithmetic operator that divides the number to its left by the number to its right. *Class of operands*: Integer, Real *Class of result*: Real
`div`	Integral division. Binary arithmetic operator that divides the number to its left by the number to its right and returns the integral part of the answer as its result. *Class of operands*: Integer, Real *Class of result*: Integer
`mod`	Remainder. Binary arithmetic operator that divides the number to its left by the number to its right and returns the remainder as its result. *Class of operands*: Integer, Real *Class of result*: Integer, Real
`^`	Exponent. Binary arithmetic operator that raises the number to its left to the power of the number to its right. *Class of operands*: Integer, Real *Class of result*: Real
`as`	Coercion. Binary operator that converts the operand to its left to the class listed to its right. Not all values can be coerced to all classes. The coercions that AppleScript can perform are listed in "Coercing Values" on page 68. The additional coercions, if any, that applications can perform are listed in application dictionaries. *Class of operands*: the operand to the right of the operator must be a class identifier; the operand to the left must be a value that can be converted to that class *Class of result*: the class specified by the class identifier to the right of the operator

continued

Table 6-1 AppleScript operators (continued)

Operator	Description
`not`	Not. Unary logical operator that results in `true` if the operand to its right is `false`, and `false` if the operand to its right is `true`. *Class of operand*: Boolean *Class of result*: Boolean
`[a] ( ref [to] | reference to )`	A Reference To. Unary operator that causes AppleScript to interpret the value to its right as a reference instead of getting its value. For more information about the A Reference To operator, see "The 'A Reference To' Operator" on page 153. *Class of operand*: Reference *Class of result*: Reference

Operators That Handle Operands of Various Classes

Many of the operators can handle operands of a variety of classes. The following sections describe how the Equal, Is Not Equal To, Greater Than, Less Than, Starts With, Ends With, Contains, Is Contained By, and concatenation (&) operators behave with different classes of operands.

Equal, Is Not Equal To

The Equal and Is Not Equal To operators can handle operands of any class.

OPERANDS OF DIFFERENT CLASSES

Two expressions of different classes are not equal.

BOOLEAN EXPRESSION

Two Boolean expressions are equal if both of them evaluate to `true` or if both evaluate to `false`. They are not equal if one evaluates to `true` and the other to `false`.

CLASS IDENTIFIER

Two class identifiers are equal if they are the same identifier. They are not equal if they are different identifiers.

CONSTANT

Two constants are equal if they are the same. They are not equal if they are different.

DATA

Two data values are equal if they are the same length in bytes and their bytes are the same (AppleScript does a byte-wise comparison).

DATE

Two dates are equal if they both represent the same date, even if they are expressed in different formats. For example, the following expression is `true`, because `date "12/5/92"` and `date "December 5th, 1992"` represent the same date.

```
date "12/5/92" = date "December 5th, 1992"
```

INTEGER

Two integers are equal if they are the same. They are not equal if they are different.

LIST

Two lists are equal if each item in the list to the left of the operator is equal to the item in the same position in the list to the right of the operator. They are not equal if items in the same positions in the lists are not equal or if the lists have different numbers of items. For example,

```
{ (1 + 1), (4 > 3) } = {2, true}
```

is `true`, because `(1 + 1)` evaluates to `2`, and `(4 > 3)` evaluates to `true`.

REAL

Two real numbers are equal if they both represent the same real number, even if the formats in which they are expressed are different. For example, the following expression is `true`.

```
0.01 is equal to 1e10-2
```

Two real numbers are not equal if they represent different real numbers.

RECORDS

Two records are equal if they both contain the same collection of properties and if the values of properties with the same labels are equal. They are not equal if the records contain different collections of properties, or if the values of properties with the same labels are not equal. The order in which properties are listed does not affect equality. For example, the following expression is `true`.

```
{ name:"Eric", mileage:"8000" } = { mileage:"8000",¬
   name:"Eric"}
```

REFERENCE

Two references are equal if their classes, reference forms, and containers are identical. They are not equal if their classes, reference forms, and containers are not identical, even if they refer to the same object.

For example, the expression `x = y` in the following Tell statement is `true`, because the classes (`word`), reference forms (Index), and containers (`paragraph 1 of document "Intro" of application "Scriptable Text Editor"`) of the two references are identical.

```
tell document "Intro" of application  ¬
   "Scriptable Text Editor"
      set x to a reference to word 1 of paragraph 1
      set y to a reference to word 1 of paragraph 1
      x = y
end tell
--result:true
```

The expression `x = y` in the following statement is `false`, because the containers are different.

```
tell document "Intro" of application ¬
   "Scriptable Text Editor"
      set x to a reference to word 1 of paragraph 1
      set y to a reference to word 1
      x = y
end tell
--result:false
```

When you use references in expressions without the A Reference To operator, the values of the objects specified in the references are used to evaluate the expressions. For example, the result of the following expression is `true` if both documents begin with the same word.

```
word 1 of document "Report" = word 1 document "Intro"
```

STRING

Two strings are equal if they are both the same series of characters. They are not equal if they are different series of characters. AppleScript compares strings character by character. It does not distinguish uppercase from lowercase letters unless you use a Considering statement to consider the `case` attribute. For example, the following expression is `true`.

```
"DUMPtruck" is equal to "dumptruck"
```

AppleScript considers all characters and punctuation, including spaces, tabs, return characters, diacritical marks, hyphens, periods, commas, question marks, semicolons, colons, exclamation points, backslash characters, and single and double quotation marks in string comparisons. AppleScript ignores style in string comparisons.

Note
All string comparisons can be affected by Considering and Ignoring statements, which allow you to selectively consider or ignore the case of characters, as well as specific types of characters. For more information, see "Considering and Ignoring Statements" on page 213. ◆

Greater Than, Less Than

The Greater Than and Less Than operators work with dates, integers, real numbers, and strings.

DATE

A date is greater than another date if it represents a later time. A date is less than another date if it represents an earlier time.

INTEGER

An integer is greater than a real number or another integer if it represents a larger number. An integer is less than a real number or another integer if it represents a smaller number.

REAL

A real number is greater than an integer or another real number if it represents a larger number. A real number is less than an integer or another real number if it represents a smaller number.

STRING

A string is greater than (comes after) another string if it would appear after the other string in an English-language dictionary. For example,

```
"zebra" comes after "aardvark"
```

and

```
"zebra" > "aardvark"
```

are `true`. A string is less than (comes before) another string if it would appear in a dictionary before the other string. For example,

```
"aardvark" comes before "zebra"
```

and

```
"aardvark" < "zebra"
```

are `true`.

AppleScript uses the ASCII collating sequence to determine a word's position in an English-language dictionary. The order of the ASCII collating sequence is

space`!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~`

AppleScript compares strings character by character. When the corresponding characters in two strings are not the same, the string containing the character closest to the beginning of the ASCII collating sequence is less than the other string. If two strings have identical characters but one is shorter than the other, the shorter string is less than the longer string. AppleScript treats all letters as lowercase letters, unless you use a Considering statement to consider the `case` attribute. For more information about Considering statements, refer to "Considering and Ignoring Statements" on page 213.

Starts With, Ends With

The Starts With and Ends With operators work with lists and strings.

LIST

A list starts with another list if the values of the items in the list to the right of the operator are equal to the values of the items at the beginning of the list to the left. A list ends with another list if the values of the items in the list to the right of the operator are equal to the values of the items at the end of the list to

the left. In both cases, the items in the two lists must be in the same order. Both Starts With and Ends With work if the operand to the right of the operator is a single value. For example,

```
{ "this", "is", 2, "cool" } ends with "cool"
```

and

```
{ "this", "is", 2, "cool" } starts with "this"
```

and

```
{ "this", "is", 2, "cool" } starts with { "this", "is" }
```

are all `true`.

STRING

A string starts with another string if the characters in the string to the right of the operator are the same as the characters at the beginning of the string to the left. For example,

```
"operand" starts with "opera"
```

is `true`.

A string ends with another string if the characters in the string to the right of the operator are the same as the characters at the end of the string to the left. For example,

```
"operand" ends with "and"
```

is `true`.

AppleScript compares strings character by character according to the rules for the Equal operator.

Contains, Is Contained By

The Contains and Is Contained By operators work with lists, records, and strings.

LIST

A list contains another list if the list to the right of the operator is a sublist of the list to the left of the operator. A sublist is a list whose items appear in the same order and have the same values as any series of items in the other list. For example,

```
{ "this", "is", 1 + 1, "cool" } contains { "is", 2 }
```

is `true`, but

```
{ "this", "is", 2, "cool" } contains { 2, "is" }
```

is `false`.

A list is contained by another list if the list to the left of the operator is a sublist of the list to the right of the operator. For example,

```
{ "is", 2} is contained by { "this", "is", 2, "cool" }
```

is `true`. Both Contains and Is Contained By work if the sublist is a single value. For example,

```
{ "this", "is", 2, "cool" } contains 2
```

and

```
2 is contained by { "this", "is", 2, "cool" }
```

are `true`.

RECORD

A record contains another record if all the properties in the record to the right of the operator are included in the record to the left, and the values of properties in the record to the right are equal to the values of the corresponding properties in the record to the left. A record is contained by another record if all the properties in the record to the left of the operator are included in the record to the right, and the values of the properties in the record to the left are equal to the values of the corresponding properties in the record to the right. The order in which the properties appear does not matter. For example,

```
{ name:"Eric", mileage:"8000", description:"fast"} ¬
   contains { description:"fast", name:"Eric" }
```

is `true`.

STRING

A string contains another string if the characters in the string to the right of the operator are equal to any contiguous series of characters in the string to the left of the operator. For example,

```
"operand" contains "era"
```

is `true`, but

```
"operand" contains "dna"
```

is `false`.

A string is contained by another string if the characters in the string to the left of the operator are equal to any series of characters in the string to the right of the operator. For example, this statement is `true`:

```
"era" is contained by "operand"
```

Concatenation

The concatenation operator (&) can handle operands of any class.

STRING

The concatenation of two strings is a string that begins with the characters in the string to the left of the operator, followed immediately by the characters in the string to the right of the operator. AppleScript does not add spaces or other characters between the two strings. For example,

```
"dump" & "truck"
```

returns the string `"dumptruck"`.

If the operand to the left of the operator is a string, but the operand to the right is not, AppleScript attempts to coerce the operand to the right to a string. For example, when AppleScript evaluates the expression

```
"Route " & 66
```

it coerces the integer `66` to the string `"66"`, and the result is

```
"Route 66"
```

RECORD

The concatenation of two records is a record that begins with the properties of the record to the left of the operator, followed by the properties of the record to the right of the operator. If both records contain properties with the same name, the value of the property from the record to the left of the operator appears in the result. For example, the result of the expression

```
{ name:"Eric", mileage:"8000" } & ¬
   { name:"Mitch", framesize:58 }
```

is

```
{ name:"Eric", mileage:"8000", frameSize:58 }
```

ALL OTHER CLASSES

The concatenation of two operands that are not strings or records is a list whose first item is the value of the operand to the left of the operator, and whose second item is the value of the operand to the right of the operator. If the operands to be concatenated are lists, then the result is a list containing all the items in the list to the left of the operator, followed by all the items in the list to the right of the operator. For example,

```
{ "This" } & { "and", "that" }
```

returns a list containing three items:

```
{ "This", "and", "that" }
```

Operator Precedence

AppleScript allows you to combine expressions into larger, more complex expressions. When evaluating expressions, AppleScript uses operator precedence to determine which operations are performed first. Table 6-2 shows the order in which AppleScript performs operations.

To see how operator precedence works, consider the following expression.

```
2 * 5 + 12
--result: 22
```

To evaluate the expression, AppleScript performs the multiplication operation `2 * 5` first, because as shown in Table 6-2, multiplication has higher precedence than addition.

The column labeled “Associativity” in Table 6-2 indicates the order in which AppleScript performs operations if there are two or more operations of the same precedence in an expression. The word “none” in the Associativity column indicates that you cannot have multiple consecutive occurrences of the operation in an expression. For example, the expression `3 = 3 = 3` is not legal because the associativity for the equal operator (=) is “none.” The word “unary” indicates that the operator is a unary operator. To evaluate expressions with multiple unary operators of the same order, AppleScript applies the operator closest to the operand first, then applies the next closest operator, and so on. For example, the expression `not not not true` is evaluated as `not (not (not true))`.

Table 6-2 Operator precedence

Order	Operators	Associativity	Type of operator
1	`( )`	Innermost to outermost	Grouping
2	`+` `-`	Unary	Plus or minus sign for numbers
3	`^`	Right to left	Exponentiation

4	`*`
	`/`
	`÷`
	`div`
	`mod`
		Left to right	Multiplication and division

5	`+` `-`	Left to right	Addition and subtraction
6	`as`	Left to right	Coercion

7	`<`
	`≤`
	`>`
	`≥`
		None	Comparison

8	`=` `≠`	None	Equality and inequality
9	`not`	Unary	Logical negation
10	`and`	Left to right	Logical for Boolean values
11	`or`	Left to right	Logical for Boolean values

You can change the order in which AppleScript performs operations by grouping expressions in parentheses. As shown in Table 6-2, AppleScript evaluates expressions in parentheses first. For example, adding parentheses around `5 + 12` in the following expression causes AppleScript to perform the addition operation first and changes the result.

```
2 * ( 5 + 12 )
--result:34
```

Date-Time Arithmetic

AppleScript supports these operations with the + and - operators on date and time difference values:

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

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

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

where *date* is a date value and *timeDifference* is an integer value specifying a time difference in seconds.

To simplify the notation of time differences, you can also use one or more of these constants:

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

Here's an example:

```
date "Apr 15, 1992" + 4 * days + 3 * hours + 2 * minutes
```

It is often useful to be able to specify a time difference between two dates; for example:

```
set timeInvestment to current date - "May 16, 1992"
```

After running this script, the value of the `timeInvestment` variable is an integer that specifies the number of seconds between the two dates. If you then add this time difference to the starting date (May 16, 1992), AppleScript returns a date value equal to the current date when the `timeInvestment` variable was set.

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

```
31449600 / years
--result: 1

151200 / days
--result: 1.75
```

To get an integral number of hours, days, and so on, use the `div` operator:

```
151200 div days
--result: 1
```

To get the difference, in seconds, between the current time and Greenwich mean time, use the scripting addition command Time to GMT. For example, if you are in Cupertino, California, and your computer is set to Pacific Standard Time, Time to GMT produces this result:

```
time to GMT
--result: -28800
```

For more information about the Time to GMT command, see the *AppleScript Scripting Additions Guide.*

Control Statements

Control statements are statements that control when and how other statements are executed. Most control statements are compound statements—that is, statements that contain other statements.

By default, AppleScript executes the statements in a script in sequence, one after the other. Control statements can change the order in which AppleScript executes statements by causing AppleScript to repeat or skip statements or go to a different statement.

This chapter describes the following control statements:

- Tell, which defines the default target to which commands are sent if no direct object is specified
- If, which allows you to execute or skip statements based on the outcome of one or more tests
- Repeat, which allows you to repeat a series of statements
- Try, which allows you to handle error messages
- Considering and Ignoring, which allow you to consider or ignore certain attributes, such as case, punctuation, and white space, in string comparisons
- With Timeout, which allows you to specify how long AppleScript waits for an application command or scripting addition to complete before stopping execution of the script and returning an error
- With Transaction, which 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

Characteristics of Control Statements

Most control statements are compound statements that contain other statements. For example, the If statement

```
if today = last day of theMonth
   set MonthlyReport to prepareReport(currentMonth)
   print MonthlyReport
end if
```

is a compound statement that contains a Set command and a Print command. Compound statements begin with one or more reserved words, such as `if` in the example above, that identify the type of compound statement. The last line of a compound statement is always `end`, which can optionally include the word that begins the control statement.

Control statements can contain other control statements. For example, this Tell statement contains the If statement of the previous example.

```
tell application "ReportWizard"
   if today = last day of theMonth
      set MonthlyReport to prepareReport(currentMonth)
      print MonthlyReport
   end if
end tell
```

Control statements that are contained within other control statements are sometimes called **nested control statements.**

All control statements can be compound statements. In addition, some control statements can be written as single statements. For example, the statement

```
if (x > y) then return x
```

is equivalent to

```
if (x > y) then
   return x
end if
```

You can use a simple statement only when you're controlling the execution of a single statement (such as `return x` in the previous example).

Tell Statements

Tell statements specify the default target, the object to which commands are sent if they do not include a direct parameter. For example, in the following Tell statement, the Close command does not include a direct parameter.

```
tell front window
   close
end tell
```

As a result, the Close command is sent to the front window, the default target specified in the Tell statement.

When AppleScript encounters a partial reference (a reference that does not specify every container of an object), it uses the default target to complete it. For example, in the following Tell statement, the reference `word 3` does not specify all of the containers of the word object, so AppleScript completes it with the default target.

```
tell front document of application "Scriptable Text Editor"
   delete word 3
end tell
```

The result is that the Delete command is sent to the third word of the front document of the Scriptable Text Editor.

A Tell statement also indicates which dictionary AppleScript should use to interpret words contained in the statement. For example, the previous Tell statement tells AppleScript to use the Scriptable Text Editor dictionary, which contains the definitions for the Delete command and the word object. If the Tell statement had not specified the application, AppleScript would not have understood the Delete command.

If you refer to another application within a Tell statement to an application, AppleScript uses the dictionaries of both applications to interpret the words in the statement. For example, in response to the following Tell statement,

AppleScript uses the Microsoft Excel dictionary for definitions of the Copy command and cell object, and the Scriptable Text Editor dictionary for the definition of the word object.

```
tell application "Microsoft Excel"
   copy word 5 of document "TestDocument" of application ¬
      "Scriptable Text Editor" to Cell "R1C1" of Document ¬
      "spreadsheet"
end tell
```

AppleScript defines two variables, `it` and `me`, that you can use in Tell statements.

The variable `it` is the default target. The value of `it` is a reference, as in

```
tell document "Introduction" of application  ¬
   "Scriptable Text Editor"
      get name of it
end tell
```

The value of the variable `it` is `document "Introduction" of application "Scriptable Text Editor"`. The result of the Get command is the string `"Introduction"`.

The variable `me` refers to the current script, as in

```
property name : "Script"
tell document "Introduction" of application  ¬
   "Scriptable Text Editor"
      get name of me
end tell
--result: "Script"
```

The reference `name of me` refers to the name property of the current script. The result of the Get command is the string `"Script"`.

AppleScript defines another word, `my`, that you can use instead of the phrase `of me`. For example, the following script is equivalent to the previous example:

```
property name : "Script"
tell document "Introduction" of application  ¬
   "Scriptable Text Editor"
      get my name
end tell
--result: "Script"
```

If you refer to a property in a Tell statement without using either `it` or `me`, AppleScript assumes that you want the property of the default target of the Tell statement. For example, the result of the Get command in the following Tell statement is `"Introduction"`.

```
property name : "Script"
tell document "Introduction" of application  ¬
   "Scriptable Text Editor"
      get name
end tell
--result: "Introduction"
```

If AppleScript cannot find the property in the dictionary of the default target of the Tell statement, then it assumes you want the property of the current script. For example, the result of the Get command in the following Tell statement is `1000000`.

```
property x : 1000000
tell document "Introduction" of application  ¬
   "Scriptable Text Editor"
      get x
end tell
--result: 1000000
```

In addition to distinguishing script properties from object properties, `me` and `my` are used to distinguish user-defined commands (subroutines) from application commands in Tell statements. For more information, see Chapter 8, “Handlers.”

Note
Within tests in Filter references, the direct object is the object being tested, so the variable `it` refers to the object currently being tested. See "Using the Filter Reference Form" on page 140 for information about the use of `it` in tests. ◆

Tell (Simple Statement)

A simple Tell statement specifies the object to which to send a command.

SYNTAX

`tell` *referenceToObject* `to` *statement*

where

referenceToObject is a reference to an application object, system object, or script object.

statement is any AppleScript statement.

EXAMPLE

```
tell front window of application "Scriptable Text Editor" to close
```

NOTES

If *referenceToObject* specifies an application on a remote computer, additional conditions must be met. These conditions are described in "References to Applications," which begins on page 146.

If *referenceToObject* specifies an application on the same computer that is not running, AppleScript launches the application.

Tell (Compound Statement)

A compound Tell statement specifies the default target of the commands it contains.

SYNTAX

```
tell referenceToObject
   [ statement ]...
end [ tell ]
```

where

referenceToObject is a reference to an application object, system object, or script object.

statement is any AppleScript statement.

EXAMPLES

```
tell application "Scriptable Text Editor"
   tell front window
      close
   end tell
end tell

tell front window of application "Scriptable Text Editor"
   close
end tell

tell application "Scriptable Text Editor" of machine ¬
   "Mitch's PowerBook" of zone "Apple Berkeley"
      tell front window
         close
      end tell
end tell
```

NOTES

If *referenceToObject* specifies an application on a remote computer, additional conditions must be met. These conditions are described in "References to Applications," which begins on page 146.

If *referenceToObject* specifies an application on the same computer that is not running, AppleScript launches the application.

If Statements

If statements allow you to define statements or groups of statements that are executed only in specific circumstances. Each If statement contains one or more **Boolean expressions** whose values can be either `true` or `false`. AppleScript executes the statements contained in the If statement only if the value of the Boolean expression is `true`.

If statements are also called **conditional statements.** Boolean expressions in If statements are also called **tests.**

The following example uses an If statement to control whether or not a particular dialog box is displayed:

```
if dependents > 2 then
   display dialog "You might need to file an extra form"
end if
```

The If statement contains the Boolean expression `dependents > 2`. If the value of the Boolean expression is `true`, the Display Dialog command is executed. If the value of the Boolean expression is `false`, the Display Dialog command is not executed. (Display Dialog is a scripting addition command. For more information about the way it works, see the *AppleScript Scripting Additions Guide*.)

If statements can contain multiple tests. For example, the following statement contains three tests.

```
if ( x > y ) then
   set myMessage to " is greater than "
else if ( x < y ) then
   set myMessage to " is less than "
```

```
else
   set myMessage to " is equal to "
end if
set myResult to (x as string) & myMessage & (y as string)
```

If the expression `x > y` is `true`, the value of the variable `myMessage` is set to `" is greater than "` and the If statement is finished. Control passes to the Set statement, which uses the value of the variable `myMessage` to set the value of another variable, called `myResult`. The value of `myResult` is a string such as `"7 is greater than 5"`. If the first Boolean expression is `false`, the next expression, `x < y`, is evaluated with similar results.

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. For example, suppose the values of `x` and `y` in the previous example are both 112. The first two tests, `x > y` and `x < y`, fail. The value of the variable `myMessage` is set to `" is equal to "`, and the value of `myResult` is `"112 is equal to 112"`.

If statements can be more elaborate, as in this example:

```
display dialog "How many dependents?" default answer ""
set dependents to (text returned of result) as integer
display dialog "Have you ever been audited?" buttons  ¬
   {"No", "Yes"}

if button returned of result = "Yes" then
   set audit to true
else
   set audit to false
end if

if dependents < 9 and audit = false then
   display dialog "No extra forms are required."
else if dependents < 9 and audit = true then
   display dialog "You might need to file an extra form."
```

```
else --anything greater than 9
   display dialog "You will need to file an extra form."
end if
```

The example shows how you can create a more complex Boolean expression with the help of Boolean operators, such as the And operator. The expression

```
dependents < 9 and audit = false
```

has two Boolean expressions as operands (`dependents < 9`, `audit = false`). If both expressions are `true`, the value of the entire expression is `true`. Other Boolean operators are Or (another binary operator; if either of its operands is `true`, the entire expression is `true`), and Not (a unary operator; if its operand is `true`, the expression is `false`, and vice versa). For more information about operators, see Chapter 6, "Expressions."

If (Simple Statement)

A simple If statement contains one Boolean expression and a statement to be executed if the value of the Boolean expression is `true`.

SYNTAX

`if` *Boolean* `then` *statement*

where

Boolean is an expression whose value is `true` or `false`.

statement is any AppleScript statement.

EXAMPLES

In the following If statement

```
if result > 3 then display dialog "The result is " & ¬
   result as string
```

the Display Dialog command is executed only if the value of the Boolean expression `result > 3` is `true`.

If (Compound Statement)

A compound If statement contains one or more Boolean expressions and groups of statements to be executed if the value of the corresponding Boolean expression is `true`.

SYNTAX

```
if Boolean [ then ]
   [ statement ]...
[ else if Boolean [ then ]
   [ statement ]...]...
[ else
   [ statement ]...]
end [ if ]
```

where

Boolean is an expression whose value is `true` or `false`.

statement is any AppleScript statement.

EXAMPLE

In the following If statement, the statements that copy an individual's status report to the end of a department status report are executed only if the date is March 1, 1993.

```
if Current Date = "March 1, 1993"
   tell application "Scriptable Text Editor"
      open file "Status Report"
      set myStatus to text from paragraph 1 to ¬
         paragraph 10 of document "Status Report"
      close document "Status Report"
      open file "Department Status"
      copy myStatus to end of document "Department Status"
      close document "Department Status"
   end tell
end if
```

Repeat Statements

Repeat statements are used to create **loops,** or groups of repeated statements, in scripts. There are several types of Repeat statements, which differ in the way they specify when the repetition stops.

For example, the following Repeat statement performs the same action a specified number of times:

```
repeat 2 times
   beep
end repeat
```

The following Repeat statement performs the same actions while a specific condition is `true`:

```
tell application "Scriptable Text Editor"
   set numberOfWindows to (count windows)
   repeat while numberOfWindows > 0
      close front window
      set numberOfWindows to (count windows)
   end repeat
end tell
```

You can also specify an **infinite loop,** which is a Repeat statement that does not specify when the repetition stops. You can use an Exit statement within an infinite loop or any other Repeat statement to immediately exit the Repeat statement.

Here's an example of a Repeat statement with an Exit statement:

```
tell application "Scriptable Text Editor"
   set numberOfWindows to (count windows)
   set i to 1
   repeat
      if i > numberOfWindows then
         exit
      end if
      print window i
      set i to i + 1
   end repeat
end tell
```

More elaborate forms of the Repeat statement use looping variables that you can refer to in the body of the loop. Here's an example:

```
tell application "Scriptable Text Editor"
   set contents of front window to ""
   set selection to "David Numberman's Top Ten Numbers for Lists
"
   repeat with n from 1 to 10
      copy (n as string & ". " & n as string & "
" ) to n
   select end of front window
   set selection to n
   end repeat
end tell
```

Note that two of the strings in the preceding statement include a return character. These are valid strings even though the surrounding quotation marks are on different lines within the statement. Running the preceding statement results in the following text:

```
David Numberman's Top 10 Numbers for Lists

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
```

The line

```
repeat with n from 1 to 10
```

specifies `n` as the **looping variable,** a variable that controls the number of iterations.

At the beginning of each iteration, AppleScript adds 1 to the value of `n`. When the value of the looping variable reaches `10`, AppleScript exits the loop. The expression `n as string` coerces an integer into a string, while the & (concatenation) operator joins two strings to make a single string. For more information about operators and coercing values, see Chapter 6, “Expressions.”

Repeat (forever)

The *Repeat (forever)* form of the Repeat statement is an infinite loop. The only way to exit the loop is by using an Exit statement.

SYNTAX

```
repeat
   [ statement ]...
end [ repeat ]
```

where

statement is any AppleScript statement.

This is an infinite loop; you must use an Exit statement to exit the loop (see page 204).

EXAMPLE

The following example numbers the paragraphs of a document. It uses the Exit statement

```
if paragraphNum > numParagraphs then exit
```

to exit the loop.

```
tell document "List"
   set numParagraphs to (count paragraphs)
   set paragraphNum to 1
   repeat
      if paragraphNum > numParagraphs then exit
      set paragraph paragraphNum to (paragraphNum as string) & " " ¬
         & paragraph paragraphNum
      set paragraphNum to paragraphNum + 1
   end repeat
end tell
```

Repeat (number) Times

The *Repeat (number) Times* form of the Repeat statement repeats a group of statements a specified number of times.

SYNTAX

```
repeat integer [ times ]
   [ statement ]...
end [ repeat ]
```

where

integer is an integer that specifies the number of times to repeat the statements in the body of the loop. The word `times` after *integer* is optional.

statement is any AppleScript statement.

EXAMPLE

The following example numbers the paragraphs of a document with the *Repeat (number) Times* form of the Repeat statement.

```
tell document "List"
   set numParagraphs to (count paragraphs)
   set paragraphNum to 1
   repeat numParagraphs times
      set paragraph paragraphNum to (paragraphNum as string) & " " ¬
         & paragraph paragraphNum
      set paragraphNum to paragraphNum + 1
   end repeat
end tell
```

Repeat While

The *Repeat While* form of the Repeat statement repeats a group of statements as long as a particular condition, specified in a Boolean expression, is met.

SYNTAX

```
repeat while Boolean
   [ statement ]...
end [ repeat ]
```

where

Boolean is an expression whose value is `true` or `false`. The statements in the loop are repeated until *Boolean* becomes `false`. If *Boolean* is `false` when entering the loop, the statements in the loop are not executed.

statement is any AppleScript statement.

EXAMPLE

The following example numbers the paragraphs of a document with the *Repeat While* form of the Repeat statement.

```
tell document "List"
   set numParagraphs to (count paragraphs)
   set paragraphNum to 1
   repeat while paragraphNum ≤ numParagraphs
      set paragraph paragraphNum to (paragraphNum as string) & " " ¬
         & paragraph paragraphNum
      set paragraphNum to paragraphNum + 1
   end repeat
end tell
```

Repeat Until

The *Repeat Until* form of the Repeat statement repeats a group of statements until a particular condition, specified in a Boolean expression, is met.

SYNTAX

```
repeat until Boolean
   [ statement ]...
end [ repeat ]
```

where

Boolean is an expression whose value is `true` or `false`. The statements in the loop are repeated until *Boolean* becomes `true`. If *Boolean* is `true` when entering the loop, the statements in the loop are not executed.

statement is any AppleScript statement.

EXAMPLE

This example numbers the paragraphs of a document with the *Repeat Until* form of the Repeat statement.

```
tell document "List"
   set numParagraphs to (count paragraphs)
   set paragraphNum to 1
   repeat until paragraphNum > numParagraphs
      set paragraph paragraphNum to  (paragraphNum as string) & " " ¬
         & paragraph paragraphNum
      set paragraphNum to paragraphNum + 1
   end repeat
end tell
```

Repeat With (loopVariable) From (startValue) To (stopValue)

In the *Repeat With (loopVariable) From (startValue) To (stopValue)* form of the Repeat statement, the looping variable is an integer that is increased by a specified value after each iteration of the loop. The loop terminates when the value of the variable is greater than a predefined stop value.

SYNTAX

```
repeat with loopVariable from startValue to stopValue [ by stepValue ]
   [ statement ]...
end [ repeat ]
```

where

loopVariable is used to control the number of iterations. It can be any previously defined variable or a new variable you define in the Repeat statement (see "Notes").

startValue (an integer) is the value assigned to *loopVariable* when the loop is entered.

stopValue (an integer) is the value of *loopVariable* at which iteration ends. Iteration continues until the value of *loopVariable* is greater than the value of *stopValue*.

stepValue (an integer) is the value added to *loopVariable* after each iteration of the loop. The default value of *stepValue* is 1.

statement is any AppleScript statement.

EXAMPLE

The following example numbers the paragraphs of a document with the *Repeat With (loopVariable) From (startValue) To (stopValue)* form of the Repeat statement.

```
tell document "List"
   repeat with n from 1 to (count paragraphs)
      set paragraph n to (n as string) & " " & paragraph n
   end repeat
end tell
```

NOTES

You can use an existing variable as the looping variable in a Repeat statement or define a new one in the Repeat statement. You cannot change the value of the looping variable in the loop body. The variable is undefined after the loop has been executed, but you can redefine it outside the loop.

AppleScript evaluates *startValue*, *stopValue*, and *stepValue* when it begins executing the loop and stores the values internally. If you change the values in the body of the loop, it has no effect on the execution of the loop.

Repeat With (loopVariable) In (list)

In the *Repeat With (loopVariable) In (list)* form of the Repeat statement, the looping variable is a reference to an item in a list. The number of iterations is equal to the number of items in the list. In the first iteration, the value of the variable is `item 1 of` *list* (where *list* is the list you specified in the first line of the statement), in the second iteration, its value is `item 2 of` *list*, and so on.

SYNTAX

```
repeat with loopVariable in list
   [ statement ]...
end [ repeat ]
```

where

loopVariable is any previously defined variable or a new variable you define in the Repeat statement (see "Notes").

list is a list or a reference (such as `words 1 thru 5`) whose value is a list. *list* can also be a record; AppleScript coerces the record to a list (see "Notes").

statement is any AppleScript statement.

EXAMPLE

The following example numbers the paragraphs of a document with the *Repeat With (loopVariable) In (list)* form of the Repeat statement. The value of the

reference `paragraphs (the paragraphs of document "List")` is a list of the paragraphs in the document.

```
tell document "List"
   set paragraphNum to 1
   repeat with n in paragraphs
      set paragraph paragraphNum to ¬
         (paragraphNum as string) & " " & contents of n
      set paragraphNum to paragraphNum + 1
   end repeat
end tell
```

NOTES

You can use an existing variable as the looping variable in a Repeat statement or define a new one in the Repeat statement. You cannot change the value of the looping variable in the loop body. The variable is undefined after the loop has been executed, but you can redefine it outside the loop.

AppleScript evaluates *loopVariable* `in` *list* as `item 1 of` *list*, `item 2 of` *list*, `item 3 of` *list*, and so on until it reaches the last item in the list:

```
repeat with i in {1, 2, 3, 4}
   set x to i
end repeat

--result: item 4 of {1, 2, 3, 4}
```

To get the value of an item in the list, you must use the `contents of` operator:

```
repeat with i in {1, 2, 3, 4}
   set x to contents of i
end repeat

--result: 4
```

If the value of *list* is a record, AppleScript coerces the record to a list by stripping the property labels. For example, `{a:1, b:2, c:3}` becomes `{1, 2, 3}`.

Exit

An **Exit statement** is used in a Repeat statement to exit the Repeat statement. When AppleScript executes an Exit statement, it terminates loop execution and resumes execution with the next statement following the Repeat statement. You cannot use Exit statements outside of Repeat statements.

SYNTAX

```
exit
```

EXAMPLE

```
set i to 1
tell application "Scriptable Text Editor"
   repeat
      if i > (count windows)
         exit
      end
      print window i
      set i to i + 1
   end repeat
end tell
```

Try Statements

Scripts don't always work perfectly. When a script is executed, errors can occur in the Operating System (for example, when a specified file isn't found), in an application (for example, when you specify an object that doesn't exist), and in the script itself. When an error occurs, AppleScript sends a special message known as an error message. An **error message** is a message that is returned by an application, AppleScript, or the Operating System if an error occurs during the handling of a command. An error message can include an **error number,**

which is an integer that identifies the error, an **error expression,** which is an expression, usually a string, that describes the error, and other information.

To handle error messages, scripts can include **error handlers.** Error handlers are contained in compound statements, called Try statements, that define the scope of the error handlers they contain. If an error message occurs and there is no handler for it, script execution stops.

Kinds of Errors

Every script error falls into one of the following categories:

- Operating System errors are errors that occur when AppleScript or an application requests services from the Operating System. They are rare, and, more importantly, there's usually nothing you can do about them in a script. A few, such as `"File <name> wasn't found"` and `"Application isn't running"`, make sense for scripts to handle. These errors are listed in Appendix C, "Error Messages."
- Apple event errors are Operating System errors that occur when the underlying message system for AppleScript—known as Apple events—fails. Many of these errors, such as `"No user interaction allowed"`, are of interest to users. Also of interest to users are errors that have to do with reference forms, as well as errors like `"No such object"`. These errors are listed in Appendix C, "Error Messages."
- *Apple Event Registry* errors are errors returned by applications when handling standard AppleScript commands (commands that apply to all applications). Many of these, such as `"The specified object is a property, not an element"`, are of interest to users and should be handled. These errors are listed in Appendix C, "Error Messages."
- AppleScript errors are errors that occur when AppleScript processes script statements. Nearly all of these are of interest to users. These errors are listed in Appendix C, "Error Messages."
- Application errors are any additional errors returned by applications. Applications can define any number of additional error messages for the AppleScript commands they handle. These errors are listed in the documentation for applications that define them.

- Script errors are error messages sent by a script using the Error command. Scripts that define additional errors will often include descriptions of the errors in their documentation.

Note
Many "errors" are the result of the normal operation of a command. For example, the Choose File command returns error –128 if the user presses the Cancel button in the resulting dialog box. Scripts must routinely handle such errors to ensure normal operation. ◆

How Errors Are Handled

When an error occurs, AppleScript checks to see if the statement that caused the error is contained in a Try statement. A **Try statement** is a two-part compound statement that contains a series of AppleScript statements, followed by an error handler to be invoked if any of those statements causes an error. If the statement that caused the error is included in a Try statement, then AppleScript passes control to the error handler in the Try statement. After the error handler completes, control passes to the statement immediately following the end of the Try statement.

If the error occurred within a subroutine and AppleScript does not find a Try statement in that subroutine, AppleScript checks to see if the statement that invoked the current subroutine is contained in a Try statement. If that statement is not contained in a Try statement, AppleScript continues up the call chain, going to the statement that invoked that subroutine, if any, and so on. If none of the calls in the call chain is contained in a Try statement, AppleScript stops execution of the script.

Writing a Try Statement

A Try statement is two-part compound statement. The first part, which begins with the word `try`, is a collection of AppleScript statements. The second part, which begins with the words `on error`, is an error handler—a series of statements that is executed if any of the statements in the first part causes an error message. The Try statement ends with the word `end` (followed optionally by `error` or `try`).

The error handler can include up to five **parameter variables** (also called **formal parameters**) that represent the actual information sent in the error message when the error occurs. When the error handler is called, the parameter variables become local variables in the error handler.

Try

A Try statement is a compound statement consisting of a list of AppleScript statements followed by an error handler to be executed if any of the statements cause an error message.

SYNTAX

```
try
   [ statement ]...
on error                                        ¬
      [ errorMessageVariable ]                  ¬
      [ number errorNumberVariable ]            ¬
      [ from offendingObjectVariable ]          ¬
      [ partial result resultListVariable ]     ¬
      [ to expectedTypeVariable ]
   [ global variable [, variable ]...]
   [ local variable [, variable ]...]
   [ statement ]...
end [ error | try ]
```

where

statement is any AppleScript statement.

errorMessageVariable (an identifier) is a parameter variable for the expression, usually a string, that describes the error. You use this parameter variable to refer to the error expression within the error handler.

errorNumberVariable (an identifier) is a parameter variable for the error number (an integer). You use this parameter variable to refer to the error number within the error handler.

offendingObjectVariable (an identifier) is a parameter variable for the reference to the application or object that caused the error (a reference). You use this parameter variable to refer to the object that caused the error within the error handler.

resultListVariable (an identifier) is a parameter variable for the results for objects that were handled before the error occurred. Its value is a list that can contain values of any class. You use this parameter variable to refer to the partial results within the error handler. This parameter applies only to commands that return results for multiple objects. For example, if an application handles the command `get words 1 thru 5` and an error occurs when handling word 4, the `partial result` parameter contains the results for the first three words.

expectedTypeVariable (an identifier) is a parameter variable for the expected value class (a class identifier)—that is, the value class to which AppleScript was attempting to coerce the value of *offendingObjectVariable*. If an application receives data of the wrong class and cannot coerce it to the correct class, the value of this parameter variable is the class of the coercion that failed. (The example at the end of this definition demonstrates how this works.)

variable is an identifier for either a global variable or a local variable that can be used in the handler. The scope of a local variable is the handler. You cannot refer to a local variable outside the handler. The scope of a global variable can extend to any other part of the script, including other handlers and script objects. For detailed information about the scope of local and global variables, see "Scope of Script Variables and Properties," which begins on page 252.

EXAMPLES

The following Try statement provides an error handler for the Choose File command. (For a complete description of the Choose File command, see the *AppleScript Scripting Additions Guide*.) The Choose File command returns an error if the user clicks the Cancel button in the Choose File dialog box. The error handler gives the user a chance to continue if an error occurs.

```
try
   choose file
   set fileName to result
```

```
on error errText number errNum
   display dialog "An error has occurred: " &  ¬
      errText & "\rDo you want to continue " &  ¬
      "using the default file?"  ¬
      buttons {"Cancel", "Continue"} default button 1
   if button returned of result = "Cancel"
      error number -128 --quit silently
   else
      display dialog "The script will continue " & ¬
         "using the default file."
      set fileName to defaultFileName
   end if
end try
```

For the preceding example to work correctly, `defaultFileName` must have been set to a filename earlier in the same script.

The next example demonstrates the use of the To keyword to capture additional information about an error that occurs during a coercion failure.

```
tell application "Scriptable Text Editor"
   try
      repeat with i from 1 to "Toronto"
         i
      end repeat
   on error from obj to newClass
      {obj, newClass}
   end try
end tell

--result: {"Toronto", integer}
```

The Repeat statement fails because the string `"Toronto"` is the wrong class. The error handler simply returns the values of `obj` (the offending value, `"Toronto"`) and `newClass` (the class of the coercion that failed, `integer`) in the result window.

Signaling Errors in Scripts

A script can signal an error—which can then be handled by an error handler—with the Error command. This allows scripts to define their own messages for errors that occur within the script.

Error

The Error command signals an error in a script.

SYNTAX

```
error                                ¬
   [ errorMessage ]                  ¬
   [ number errorNumber ]            ¬
   [ from offendingObject ]          ¬
   [ partial result resultList ]     ¬
   [ to expectedType ]
```

where

errorMessage is an expression, usually a string, describing the error. Although this parameter is not required, you should provide descriptive expressions for errors wherever possible (a string is the best way to inform the user of an error), and you should always provide an expression if you do not include a `number` parameter. If you do not include an error expression, an empty string (`""`) is passed to the error handler.

errorNumber is the error number for the error. You do not have to include an error number, but if you do, the number must not be any of the error numbers listed in Appendix C, "Error Messages." In general, positive numbers from 500 to 10,000 do not conflict with error numbers for AppleScript, the Operating System, or Apple events. If you do not include a `number` parameter, the value `-2700` is passed to the error handler.

offendingObject is a reference to the object, if any, that caused the error. If you provide a partial reference, AppleScript completes it using the value of the default object.

resultList applies only to commands that return results for multiple objects. If results for some, but not all, of the objects specified in the command are available, you can include them in the `partial result` parameter. If you do not include a `partial result` parameter, an empty list (`{}`) is passed to the error handler.

expectedType is a class identifier. If a parameter specified in the command was not of the expected class, and AppleScript was unable to coerce it to the expected class, then you can include the expected class in the `to` parameter.

EXAMPLES

The following example shows how to signal and provide a handler for an error. The `CentimeterConversion` subroutine signals error number 750 if its parameter is not a number. The error handler tests the error number, and if it is equal to 750, returns a string indicating that the parameter must be a real number or integer.

```
on CentimeterConversion from x
   --make sure the parameter is a real number or an integer
   try
      if {integer, real} contains class of x then
         return x * 2.54
      else
         error number 750
      end if
   on error number errorNumber
      if errorNumber = 750 then
         return "The parameter must be a real number or integer."
      else
         error errorNumber --unknown error, resignal
      end if
   end try
end CentimeterConversion

CentimeterConversion from "Cupertino"
--result: "The parameter must be a real number or integer."
```

You can use the Error command to resignal an error. For example, in the following Try statement, the Error command in the error handler resignals the error exactly as it was received.

```
try
   word 5 of "one two three"
on error number errNum from badObj
   --statements that handle the error
   error number errNum from badObj
end try
```

In the following Try statement, the Error command in the error handler resignals the error, but changes the error message and error number. The new error number is `600`.

```
try
   word 5 of "one two three"
on error
   --statements that determine the cause of the error
   error "There are not enough words." number 600
end try
```

Considering and Ignoring Statements

Considering statements allow you to control the way AppleScript executes operations and commands by listing specific characteristics, called *attributes*, to be taken into account as the operations and commands are executed. **Ignoring statements** work the same way, except that you list specific attributes to be ignored.

The attributes you can use include

- case, white space, and others that affect string comparisons
- an attribute called `application responses` that controls whether or not AppleScript waits for responses from commands sent to applications

Here's an example of a string comparison without a Considering statement:

```
"This" = "this"
--result: true
```

The value of the string comparison is `true`, because by default, AppleScript does not distinguish uppercase from lowercase letters.

Here's an example of the same comparison within a Considering statement:

```
considering case
   "This" = "this"
end considering
--result: false
```

The Considering statement specifies that a particular attribute of strings—their case—is to be used in comparisons. As a result the comparison `"This" = "this"` is now `false`, because the uppercase "T" in `"This"` does not match the lowercase "t" in `"this"`.

Considering/Ignoring

Considering and Ignoring statements cause AppleScript to consider or ignore specific characteristics, called attributes, as it executes groups of statements.

SYNTAX

```
considering attribute [, attribute ... and attribute ]                ¬
      [ but ignoring attribute [, attribute ... and attribute ] ]
   [ statement ]...
end considering

ignoring attribute [, attribute ... and attribute ]                   ¬
      [ but considering attribute [, attribute ... and attribute ] ]
   [ statement ]...
end ignoring
```

where

statement is any AppleScript statement.

attribute is an attribute to be considered or ignored. Attributes are listed next under "Attributes".

ATTRIBUTES

An **attribute** is a characteristic that can be considered or ignored in a Considering or Ignoring statement. A Considering or Ignoring statement can include any of the following attributes:

`case`: In string comparisons, uppercase letters are not distinguished from lowercase letters (all letters are treated as lowercase letters). If this attribute is considered, uppercase letters are distinguished from lowercase letters.

`white space`: Spaces, tab characters, and return characters are considered in string comparisons. If this attribute is ignored, the strings are compared as if these characters were not present; for example `"Brick house"` would be considered equal to `"Brickhouse"`.

`diacriticals`: Diacritical marks (such as ´, `, ^, ¨, and ˜) are considered in string comparisons. If this attribute is ignored, `"résumé"` is considered equal to `"resume"`, and so on.

`hyphens`: In string comparisons, hyphenated words are considered different from their nonhyphenated counterparts. If this attribute is ignored, the strings are compared as if any hyphens were not present; for example `"anti-war"` would be considered equal to `"antiwar"`.

`expansion`: In string comparisons, AppleScript normally treats the characters æ, Æ, œ, and Œ as identical to the character pairs ae, AE, oe, and OE, respectively. If this attribute is ignored, AppleScript treats these characters like single characters; for example æ would be considered not equal to the character pair ae.

`punctuation`: The punctuation marks (. , ? : ; ! \ ' " `) are considered in string comparisons. If this attribute is ignored, the strings are compared as if these punctuation marks were not present; for example `"This!"` would be considered equal to `"This"`.

`application responses`: Normally, AppleScript waits for a response from each application command before proceeding to the next statement or operation. The response indicates if the command completed successfully, and also returns results and error messages, if there are any. If this attribute is ignored, AppleScript does not wait for responses from application commands before proceeding to the next statement, and ignores any results or error messages that are returned. Results and error messages from AppleScript commands, scripting additions, and expressions are not affected by the `application responses` attribute.

EXAMPLES

```
considering case
   "a" comes before "b"
end considering

considering case and white space but ignoring diacriticals
   "a" comes after "b"
end considering
```

```
ignoring punctuation
   if "this !,:book" = "this book" then
      (* additional statements *)
      end if
end ignoring
```

NOTES

The `case`, `white space`, `diacriticals`, `hyphens`, `expansion`, and `punctuation` considerations apply only to comparisons performed by AppleScript. Comparisons are performed by AppleScript if the first operand in the comparison is a value in a script; if the first operand is a reference to an application or system object, the comparison is performed by the application or operating system.

In contrast, the `application responses` consideration applies only to application commands. AppleScript commands, scripting additions, and AppleScript expressions are not affected.

As with all other control statements, you can nest Considering and Ignoring statements. If the same attribute appears in both an outer and inner statement, the attribute specified in the inner statement takes precedence. For example, in the following statement, the first comparison is `true`, because case attribute is ignored (as specified in the Ignoring statement), while the second comparison is `false`, because the case attribute is once again considered (as specified in the inner Considering statement).

```
ignoring case and punctuation
   if "This" = "this" then beep 1 --true
   considering case
      if "This" = "this" then beep 2 --false
   end considering
end considering
```

When attributes in an inner Considering or Ignoring statement are different from those in outer statements, they are added to the attributes to be considered and ignored. For example, in the following statement, the first comparison is `false`, because only case is ignored, while the second comparison is `true`, because both case and white space are ignored.

```
ignoring case
   if "This or that" = "thisorthat" then beep 2 --false
   ignoring white space
      if "This or that" = "thisorthat" then beep 1 --true
   end ignoring
end ignoring
```

With Timeout Statements

When AppleScript sends a command to an application, it normally waits for the command to complete execution before continuing with the rest of the script. If the command takes longer than one minute to complete, AppleScript stops running the script and returns the error `"event timed out"`.

With Timeout statements let you change how long AppleScript waits before stopping execution of a script. The amount of time you specify in a With Timeout statement applies to some types of commands within the statement that are sent to other applications, but not to any commands sent to the application that's running the script.

The specified time applies to all application commands and to any scripting addition commands whose targets are application objects—that is, scripting addition commands within Tell statements to application objects or scripting addition commands whose direct parameters are application objects. The time specified by a With Timeout statement does not apply to AppleScript commands, AppleScript operations, or scripting addition commands whose targets are not application objects.

Note
If you want AppleScript to proceed to the next statement without waiting for application commands to complete, use an Ignoring statement to ignore the `application responses` attribute. For more information, see "Considering and Ignoring Statements" on page 213. ◆

With Timeout

With Timeout statements let you change how long AppleScript can wait before stopping execution of application and scripting addition commands that are sent to other applications.

SYNTAX

```
with timeout [ of ] integer second[s]
   [ statement ]...
end [ timeout ]
```

where

integer is an integer that specifies the amount of time, in seconds, AppleScript allows for each application command or command addition contained in the With Timeout statement that is sent to any application other than the current one.

statement is any AppleScript statement.

EXAMPLE

The following With Timeout statement gives the user five minutes, instead of the usual one minute, to type in his or her name.

```
try
   with timeout of 300 seconds
      tell application "Scriptable Text Editor"
         display dialog "What is your name?" ¬
            default answer ""
      end tell
   end timeout
on error
   beep
end
```

With Transaction Statements

Some applications, such as databases, support the notion of a transaction—that is, a sequence of related events that should be performed as if they were a single operation. The **With Transaction statement** allows you to specify transactions for such applications.

At the beginning of a With Transaction statement, AppleScript requests a transaction ID from the target application (established by an enclosing Tell statement) and attaches that transaction ID to every Apple event it sends to the target application as a result of executing commands in the body of the With Transaction statement.

Whenever AppleScript exits a With Transaction statement, it informs the application that the transaction is over, even if the exit occurs before the end of the statement because of an error. Thus, if an error occurs within the body of the With Transaction statement but is not handled within the statement, AppleScript exits the statement, the application is informed that the transaction is over, and the error continues through subsequent statements until it is handled.

With Transaction

With Transaction statements cause AppleScript to associate a single transaction ID with any events it sends to a target application as a result of executing commands in the body of the With Transaction statement.

SYNTAX

```
with transaction [ session ]
   [ statement ]...
end [ transaction ]
```

where

session is an object that specifies a specific session.

statement is any AppleScript statement.

EXAMPLES

This example uses a With Transaction statement to ensure that a record can be modified by one user without being modified by another user at the same time.

```
tell application "Small DB"
   with transaction
      set oldName to Field "Name"
      set oldAddress to Field "Address"
      set newName to display dialog ¬
         "Please type a new name" ¬
         default answer oldName
      set newAddress to display dialog ¬
         "Please type the new address" ¬
         default answer oldAddress
      set Field "Name" to newName
      set Field "Address" to newAddress
   end transaction
end tell
```

The Set statements obtain the current values of the Name and Address fields and invite the user to change them. Enclosing these Set statements in a single With Transaction statement informs the application that other users should not be allowed to access the same record at the same time.

With Transaction statements only work with applications that explicitly support them. Some applications only support With Transaction statements (like the one in the previous example) that do not take a session object as a parameter. Other applications support both With Transaction statements that have no parameter and With Transaction statements that take a session parameter.

The following example demonstrates how to specify a session for a With Transaction statement.

```
tell application "Super DB"
   set mySession to make session with ¬
      data {user: "Bob", password: "Secret"}
   with transaction mySession
      ...
   end transaction
end tell
```

CHAPTER 8

Handlers

Handlers are collections of statements that are executed in response to commands or error messages.

This chapter describes

- subroutines, which are handlers for user-defined commands
- command handlers for system or application commands
- command handlers for script applications
- the scope of script variables and properties declared in handlers and elsewhere in scripts

Using Subroutines

Subroutines are collections of statements that AppleScript runs in response to user-defined commands. They are similar to functions, methods, and procedures in other programming languages. This section explains how to write and call subroutines.

Subroutines are useful in scripts that perform the same action in more than one place. For example, if you have a series of statements for comparing values and you need to use those statements at several places in a script, you can package the statements as a subroutine and call it from anywhere in the script. Your script becomes shorter and easier to maintain. In addition, you can give subroutines descriptive names that make their purposes clear and make scripts easy to read.

Here's a subroutine, called `minimumValue`, that returns the smaller of two values:

```
--minimumValue subroutine:
on minimumValue(x, y)
   if x ≤ y then
      return x
   else
      return y
   end if
end minimumValue

--how to call minimumValue:
minimumValue(5, 105)
```

The first line of the `minimumValue` subroutine specifies the parameters of the subroutine. These can be positional parameters—like `x` and `y` in the example—where the order of the parameters is significant, or labeled parameters—like those for AppleScript and application commands—where the order of parameters other than the direct parameter doesn't matter.

The `minimumValue` subroutine includes two Return statements. A Return statement is one of the ways a subroutine can return a result. When AppleScript executes a Return statement, it returns the value (if any) listed in the statement and immediately exits the subroutine. If AppleScript executes a Return statement without a value, it exits the subroutine immediately and does not return a value.

If a subroutine does not include any Return statement, AppleScript executes the statements in the subroutine and, after handling the last statement, returns the value of the last statement in the subroutine. If the last statement does not return a value, then the subroutine does not return a value.

When AppleScript has finished executing a subroutine, it passes control to the place in the script immediately after the place where the subroutine was called. If a subroutine call is part of an expression, AppleScript uses the value returned by the subroutine to evaluate the expression. For example, to evaluate the following expression, AppleScript calls the subroutine for `minimumValue`.

```
minimumValue(5, 105) + 100
```

After the value of `minimumValue` is returned, AppleScript evaluates the rest of the expression.

Types of Subroutines

There are two types of subroutines: those with labeled parameters and those with positional parameters.

- **Labeled parameters** are identified by their labels and can be listed in any order. Subroutines with labeled parameters can also have a direct parameter. The direct parameter, if present, must be listed first.
- **Positional parameters** must be listed in a specific order, which is defined in the subroutine definition.

For example, the following statement calls a subroutine with positional parameters.

```
minimumValue(150, 4000)
```

The following statement calls a subroutine with labeled parameters. The direct parameter is the list of filenames. The labeled parameters are identified by the labels `stringToFind` and `checkCase`.

```
findFiles of {"March Expenses", "April Expenses", ¬
   "May Expenses", "June Expenses"} given ¬
   stringToFind:"LeChateau", checkCase:false
```

The definition for a subroutine determines what kind of parameters the subroutine requires. When you call a subroutine, you must list its parameters in the same way they are specified in the subroutine definition.

You can also have subroutines with no parameters. To indicate that a subroutine has no parameters, you must include a pair of empty parentheses after the subroutine name in both the subroutine definition and the subroutine call. For example, the following script shows the definition and subroutine call for a subroutine called `helloWorld` that has no parameters.

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

helloWorld()
```

Scope of Subroutine Calls in Tell Statements

If you need to call a subroutine from within a Tell statement, you must use the reserved words `of me` or `my` to indicate that the subroutine is part of the script—not a command that should be sent to the object of the Tell statement.

For example, the `minimumValue` subroutine call in the following Tell statement is unsuccessful, because AppleScript sends the `minimumValue` command to the Scriptable Text Editor. (You get an error message saying that the Scriptable Text Editor does not understand the `minimumValue` command.)

```
tell application "Scriptable Text Editor"
   minimumValue(12, 400)
   copy result as string to word 15 of front document
end tell
(* result: the subroutine call is unsuccessful because
AppleScript sends the minimumValue command to the
Scriptable Text Editor *)
```

If you use the words `of me` in the subroutine call, as shown in the following Tell statement, the subroutine call is successful, because AppleScript knows that the subroutine is part of the script.

```
tell application "Scriptable Text Editor"
   minimumValue(12, 400) of me
   copy result as string to word 15 of front document
end tell
(* result: the subroutine call is successful because the
words "of me" tell AppleScript that the minimumValue
command is part of the script *)
```

The word `my` before the subroutine call is a synonym for the words `of me` after the subroutine call. For example, the following two subroutine calls are equivalent:

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

Checking the Classes of Subroutine Parameters

You cannot specify the class of a parameter in a subroutine definition. You can, however, get the value of the Class property of a parameter and check it to see if the parameter belongs to the correct class. If it doesn't, you may be able to coerce it with the As operator, or failing that, you can return an error. (For information about coercing values, see Chapter 6, "Expressions." For information about returning errors, see "Try Statements," which begins on page 204.)

Here's an example of a subroutine that checks to see if its parameter is a real number or an integer:

```
on CentimeterConversion from x
   --make sure the parameter is a real number or an integer
   if class of x is contained by {integer, real}
      return x * 2.54
   else
      error "The parameter must be a real number or an integer"
   end if
end CentimeterConversion
```

Recursive Subroutines

A **recursive subroutine** is a subroutine that calls itself. Recursive subroutines are legal in AppleScript. You can use them to perform repetitive actions. For example, this recursive subroutine generates a factorial.

```
on factorial(x)
   if x > 0 then
      return x * (factorial(x - 1))
   else
      return 1
   end if
end factorial

factorial(10)
```

To generate 10 factorial, the subroutine `factorial` is called once from the top level of the script, and then calls itself ten more times, until the value of `x` is 0. When `x` is equal to 0, AppleScript skips to the Else clause and finishes executing all the partially executed subroutines, including the original `factorial` subroutine call.

When you call a recursive subroutine, AppleScript keeps track of the variables and pending statements in the original (partially executed) subroutine until the recursive subroutine has completed. The limit on the number of pending subroutines depends on the amount of memory available.

Saving and Loading Libraries of Subroutines

So far, you've seen examples of defining and calling subroutines in the same script. This is useful for functions that are repeated more than once in the same script. But you can also write subroutines for generic functions, such as numeric operations, that are useful in many different scripts. To make a subroutine available in any script, save it as a compiled script, and then use the scripting addition command Load Script to make it available in a particular script. You can use this technique to create libraries of subroutines for use in many scripts.

For example, the following script contains three subroutines: `centimeterConversion`, which converts inches to centimeters; `factorial`, which returns the factorial of a number; and `min`, which returns the smallest number in a list of numbers.

```
--the centimeterConversion subroutine converts inches to centimeters
on centimeterConversion from x
   if class of x is contained by {integer, real} then
      return x * 2.54
   else
      error "The parameter must be a real number or an integer."
   end if
end centimeterConversion
```

```
--the factorial() subroutine returns the factorial of a number
on factorial(x)
   if x > 0 then
      return x * (factorial(x - 1))
   else
      return 1
   end if
end factorial

--the min() subroutine returns the smallest number in a list
on min(numberList)
   if class of numberList ≠ list or numberList = {} then ¬
      return numberList
   if length of numberList = 1 then return item 1 of numberList
   copy item 1 of numberList to frontNumber
   copy length of numberList to listLength
   copy min(items 2 thru listLength of numberList) to tailNumber
   if frontNumber > tailNumber then
      return tailNumber
   else
      return frontNumber
   end if
end min
```

To save this script as a compiled script, choose Save As from the Script Editor's File menu and choose Compiled Script from the Kind pop-up menu. Then save the script as a file called Numeric Operations. (If you are using a different script editor, see the documentation that came with it.)

After you save the script as a compiled script, use the Load Script scripting addition command to make the subroutines it contains available in the current script. For example, the Load Script command in the following script assigns the compiled script Numeric Operations to the variable `NumberLib`. To call the subroutines in Numeric Operations, use a Tell statement. The Tell statement in

the example calls the `factorial` subroutine. (You must have a compiled script called Numeric Operations in the specified location for this script to work correctly.)

```
set NumberLib to (load script file "MacHD:Scripts:Numeric Operations")

tell NumberLib
   factorial(10)
end tell
```

Note
The Load Script scripting addition command loads the compiled script as a script object. For a definition of Load Script, see the *AppleScript Scripting Additions Guide.*

Script objects are user-defined objects that are treated as values by AppleScript; for more information about them, see Chapter 9, "Script Objects." ◆

Subroutine Definitions and Calls

A subroutine definition contains

- a template for calls to the subroutine
- optional variable declarations
- statements; among these can be a Return statement that when executed returns a value and exits the subroutine

You cannot nest subroutine definitions; that is, you cannot define a subroutine within a subroutine definition.

The way you call a subroutine is determined by the way the subroutine was defined:

- You must provide all the parameters specified in the definition.
- You must provide either labeled parameters or positional parameters, as specified in the definition.

The sections that follow describe how to define and call subroutines.

Subroutines With Labeled Parameters

This section describes the syntax for defining and calling subroutines with labeled parameters. Examples of subroutines that use this syntax begin on page 232.

Subroutine Definition, Labeled Parameters

The definition for a subroutine with labeled parameters lists the labels to use when calling the subroutine and the statements to be executed when it is called.

SYNTAX

```
( on | to ) subroutineName                                          ¬
          [ [ of | in ] directParameterVariable ]                   ¬
          [ subroutineParamLabel paramVariable ]...                 ¬
          [ given label:paramVariable [, label:paramVariable ]...]
   [ global variable [, variable ]...]
   [ local variable [, variable ]...]
   [ statement ]...
end [ subroutineName ]
```

where

subroutineName (an identifier) is the subroutine name.

directParameterVariable (an identifier) is a **parameter variable** (also called a **formal parameter**) that represents the actual value of the direct parameter. You use this identifier to refer to the direct parameter in the body of the subroutine definition. As with application commands, the direct parameter must be first.

Note
If a subroutine includes a direct parameter, the subroutine must also include either the *subroutineParamLabel* parameter or the `given` *label*`:`*paramVariable* parameter. ◆

subroutineParamLabel is one of the following labels: `above`, `against`, `apart from`, `around`, `aside from`, `at`, `below`, `beneath`, `beside`, `between`, `by`, `for`, `from`, `instead of`, `into`, `on`, `onto`, `out of`, `over`, `thru` (or `through`), `under`. These labels are the only labels that can be used without the special label `given`. As in other commands, each label must be unique among the labels for the subroutine (that is, you cannot use the same label for more than one parameter).

paramVariable (an identifier) is a parameter variable for the actual value of a parameter. You use this identifier to refer to the parameter in the body of the subroutine.

label is any parameter label. This can be any valid AppleScript identifier. You must use the special label `given` to specify parameters whose labels are not among the labels for *subroutineParamLabel*.

variable is an identifier for either a global or local variable that can be used in the handler. The scope of a local variable is the handler. You cannot refer to a local variable outside the handler. The scope of a global variable can extend to any other part of the script, including other handlers and script objects. For detailed information about the scope of local and global variables, see "Scope of Script Variables and Properties," which begins on page 252.

statement is any AppleScript statement.

NOTES

For examples of subroutines with labeled parameters, see page 232.

Subroutine Call, Labeled Parameters

A subroutine call for a subroutine with labeled parameters lists parameters other than the direct parameter in any order, using the labels in the subroutine definition to identify the parameter values.

SYNTAX

```
subroutineName                                                    ¬
   [ [ of | in ] directParameter ]                                ¬
   [ [ subroutineParamLabel parameterValue ]                      ¬
      | [ with labelForTrueParam [, labelForTrueParam ]...        ¬
         [ ( and | or | , ) labelForTrueParam ] ]                 ¬
      | [ without labelForFalseParam [, labelForFalseParam ]...]  ¬
         [ ( and | or | , ) labelForFalseParam ] ]                ¬
      | [ given label:parameterValue                              ¬
         [, label:parameterValue ]...] ]...
```

where

subroutineName (an identifier) is the name of the subroutine.

directParameter is the direct parameter, if one is included in the subroutine definition. It can be any valid expression. As in application commands, the direct parameter must be first if it is included at all.

subroutineParamLabel is one of the following labels used in the definition of the subroutine: `above`, `against`, `apart from`, `around`, `aside from`, `at`, `below`, `beneath`, `beside`, `between`, `by`, `for`, `from`, `instead of`, `into`, `on`, `onto`, `out of`, `over`, `thru` (or `through`), `under`.

parameterValue is the value of a parameter, which can be any valid expression.

labelForTrueParam is the label for a Boolean parameter whose value is `true`. You use this form in With clauses; because the value `true` is implied by the word With, you provide only the label, not the value. (For an example of how to use a With clause, see page 233.) If you use `or` or a comma instead of `and` with the last parameter of a `with` clause, AppleScript changes the `of` or the comma to `and` during compilation.

labelForFalseParam is the label for a Boolean parameter whose value is `false`. You use this form in Without clauses; because the value `false` is implied by the word Without, you provide only the label, not the value. If you use `or` or a comma instead of `and` with the last parameter of a `without` clause, AppleScript changes the `or` or the comma to `and` during compilation.

label is any parameter label used in the definition of the subroutine that is not among the labels for *subroutineParamLabel*. You must use the special label `given` to specify these parameters. (For an example, see "Examples" later in this section.)

If you use `or` or a comma instead of `and` with the last parameter of a `with` clause, AppleScript changes the `or` or the comma to `and` during compiling.

NOTES

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

When calling a subroutine, you can list any parameter-value pairs except the direct parameter after the label `given`, not just the parameters that were specified that way in the subroutine definition. For example, the following two calls to the `searchFiles` subroutine described in the next section are interchangeable.

```
searchFiles of {"March Expenses", "April Expenses"} for ¬
   "Le Chateau"

searchFiles of {"March Expenses", "April Expenses"} ¬
   given for:"Le Chateau"
```

With the exception of the direct parameter, which must directly follow the subroutine name, labeled parameters can appear in any order. This includes parameters listed in Given, With, and Without clauses. Furthermore, you can include any number of Given, With, and Without clauses in a subroutine call.

Examples of Subroutines With Labeled Parameters

This section provides examples of subroutine definitions with labeled parameters and of calls to those subroutines.

The following subroutine converts inches to centimeters:

```
on CentimeterConversion from x
   --make sure the parameter is a real number or an integer
   if class of x is contained by {integer, real}
      return x * 2.54
   else
      error "The parameter must be a real number or an integer"
   end if
end CentimeterConversion
--to call CentimeterConversion:
CentimeterConversion of 36
```

The following subroutine searches for a specific string in a list of files.

```
to searchFiles of filesToSearch for theString
   --filesToSearch: list of Scriptable Text Editor files
   --theString: the string to be searched for
   set hits to {}
   tell application "Scriptable Text Editor"
      repeat with i from 1 to (count items of filesToSearch)
         set currentFile to item i of filesToSearch
         if contents of document currentFile contains theString
            --append currentFile to list of hits
            set hits to hits & currentFile
         end if
      end repeat
      return hits
   end tell
end searchFiles

--to call searchFiles:
searchFiles of {"March Expenses", "April Expenses", ¬
   "May Expenses", "June Expenses"} for "LeChateau"
```

The specified files must be open for the `searchFiles` handler to work.

The following subroutine uses the special label `given` to define a parameter with the label `rounding`. By using verb forms ending with "ing" as labels, you can often make subroutine calls easier to read.

```
to findNumbers of numberList above minLimit ¬
   given rounding:roundBoolean
      set resultList to {}
      repeat with i from 1 to (count items of numberList)
         set x to item i of numberList
         if roundBoolean = true then
            copy (x + 0.5) div 1 to x
         end if
         if x > minLimit then
            copy resultList & x to resultList
         end if
      end repeat
      return resultList
end findNumbers

--to call findNumbers:
findNumbers of myList above 3 given rounding:true
```

Another way to call the `findNumbers` subroutine is to use a With or Without clause to specify the value of the `rounding` parameter. You can use With or Without clauses to specify parameters whose values are `true` or `false`.

```
--this call is equivalent to the previous example
findNumbers of myList above 3 with rounding
```

The subroutine parameter labels that can be used without the special label `given` allow you considerable flexibility in defining handlers that sound English-like. For example, here's a routine that takes any parameter that can be displayed as a string and displays it in a dialog box:

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

The statement

```
rock around the current date
```

later in the same script displays the current date in a dialog box.

Here's another example of the use of subroutine parameter labels:

```
to check for yourNumber from bottom thru top
   if bottom ≤ yourNumber and yourNumber ≤ top then
      display dialog "Congratulations! You scored."
   end if
end check
```

The statement

```
check for 8 from 7 thru 10
```

later in the same script displays the specified dialog box.

Subroutines With Positional Parameters

The sections that follow describe the syntax for defining and calling subroutines with positional parameters. Examples of subroutines that use this syntax begin on page 238.

Subroutine Definition, Positional Parameters

The definition for a subroutine with positional parameters lists the order in which to list parameters when calling the subroutine and the statements to be executed when the subroutine is called.

SYNTAX

```
( on | to ) subroutineName ( [ paramVariable [, paramVariable ]...] )
   [ global variable [, variable ]...]
   [ local variable [, variable ]...]
   [ statement ]...
end [ subroutineName ]
```

where

subroutineName (an identifier) is the name of the subroutine.

paramVariable (an identifier) is a parameter variable for the actual value of the parameter. You use this identifier to specify the parameter in the body of the subroutine.

variable is an identifier for either a global or local variable that can be used in the handler. The scope of a local variable is the handler. You cannot refer to a local variable outside the handler. The scope of a global variable can extend to any other part of the script, including other handlers and script objects. For detailed information about the scope of local and global variables, see "Scope of Script Variables and Properties," which begins on page 252.

statement is any AppleScript statement.

The parentheses that enclose the series of positional parameters in the syntax definition are a required part of the language. They are shown in bold to distinguish them from parentheses that show grouping but are not part of the language. The parentheses must be included even if the subroutine definition doesn't include any parameters.

For examples of subroutines with positional parameters, see page 238.

Subroutine Call, Positional Parameters

A subroutine call for a subroutine with positional parameters lists the parameters in the same order as they are specified in the subroutine definition.

SYNTAX

subroutineName **(** [*parameterValue* [, *parameterValue*]...] **)**

where

subroutineName (an identifier) is the name of the subroutine.

parameterValue is the value of a parameter, which can be any valid expression. If there are two or more parameters, they must be listed in the same order in which they were specified in the subroutine definition.

The parentheses that enclose the series of positional parameters are a required part of the language. They are shown in bold to distinguish them from parentheses that show grouping but are not part of the language. The parentheses must be included even if the subroutine definition doesn't include any parameters.

NOTES

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

You can use a subroutine call as a parameter of another subroutine call. Here's an example.

```
minimumValue(2, maximumValue(x, y))
```

The second parameter of the call to `minimumValue` is the value from the subroutine call to `maximumValue`. (The `minimumValue` subroutine is defined in the next section.)

A call to a subroutine with positional parameters can include parameters that aren't literals as long as they evaluate to a pattern defined for the subroutine. Similarly, the properties of a record passed to a subroutine don't have to be given in the same order they are given in the subroutine's declaration, as long as all the properties required to fit the defined pattern are present. The examples that follow include subroutines with positional parameters that define a pattern.

Examples of Subroutines With Positional Parameters

Here is a subroutine that returns the minimum value of a pair of values followed by an example of how to call the subroutine.

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

minimumValue(21, 40000)
```

You can also define a subroutine whose positional parameters define a pattern to match when calling the subroutine. For example, the subroutine that follows takes a single parameter whose pattern consists of a list of two items in a list.

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

set mypoint to {3, 8}
point(mypoint)
```

A parameter pattern can be much more complex than a single list. The handler in the next example takes two numbers and a record whose properties include a list of bounds and displays a dialog box summarizing some of that 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)
```

As you can see from this example, a call to a subroutine with patterned parameters can include parameters that aren't literals, as long as they evaluate to the appropriate pattern. Similarly, the properties of a record passed to a subroutine with patterned parameters don't have to be given in the same order in which they are given in the subroutine's definition, as long as all the properties required to fit the pattern are present.

The Return Statement

A Return statement allows you to stop execution of a handler before all its statements are executed and to obtain a value. Many of the preceding examples in this chapter use Return statements.

Return

A Return statement exits a handler and returns a value. When AppleScript executes a Return statement, it stops handler execution and resumes execution

at the place in the script where the handler was called, using the value returned as the value of the handler.

SYNTAX

```
return expression
```

where

expression is an AppleScript expression. When AppleScript executes a Return statement, it returns the value of the expression. Expressions are described in Chapter 6, "Expressions."

EXAMPLE

To return a value and exit a subroutine, include a Return statement in the body of the subroutine. For example, the following statement returns the integer 2:

```
return 2
```

If you include a Return statement without an expression, AppleScript exits the subroutine immediately and no value is returned.

NOTES

If a subroutine does not include a Return statement, AppleScript executes the statements in the subroutine and, after handling the last statement, returns the value of the last statement in the subroutine. If the last statement does not return a value, then no value is returned.

When AppleScript has finished executing a subroutine (that is, when it executes a Return statement or the last statement in the subroutine), it passes control to the place in the script immediately after the place where the subroutine was called.

Command Handlers

Command handlers are handlers for application commands. They are similar to subroutine handlers, but instead of defining responses to user-defined commands, they define responses to application commands, such as Open, Print, or Move, sent to application objects.

You define command handlers in scripts, but they handle commands that are sent to application objects. To be useful, scripts that contain command handlers must be associated with the application objects that receive those commands. This is called **attaching a script to an application object.**

Scripts that are attached to objects can change the way those objects respond to particular commands. Each application determines which, if any, of its objects can have attached scripts, and how you attach the scripts.

This section describes the syntax for command handler definitions. For information about recursion in command handlers, see the"Recursive Subroutines," which begins on page 225. For information about the scope of variables and properties in handlers, see "Scope of Script Variables and Properties," which begins on page 252.

Command Handler Definition

A command handler definition is a set of statements that is executed in response to an application command. Command handler definitions need not include all of the possible parameters of the commands they respond to. If a command handler receives more parameters than are specified in the command handler definition, it ignores the extra parameters.

SYNTAX

The syntax for a command handler definition is

```
( on | to ) commandName                                          ¬
      [ [ of ] directParameterVariable ]                         ¬
      [ given label:paramVariable [, label:paramVariable ]...]
   [ global variable [, variable ]...]
   [ local variable [, variable ]...]
   [ statement ]...
end [ commandName ]
```

where

commandName (an identifier) is a command name.

directParameterVariable (an identifier) is a parameter variable for the actual value of the direct parameter. You use this parameter variable to refer to the direct parameter in the body of the subroutine. If it is included, *directParameter* must be listed immediately after the command name. The word `of` before *directParameter* is optional.

label is the parameter label for one of the parameters of the command being handled. The label `given` is optional.

paramVariable (an identifier) is a parameter variable for the actual value of the parameter. You use this identifier to refer to the parameter in the body of the handler.

variable is an identifier for either a global or local variable that can be used in the handler. The scope of a local variable is the handler. You cannot refer to a local variable outside the handler. The scope of a global variable can extend to any other part of the script, including other handlers and script objects. For detailed information about the scope of local and global variables, see "Scope of Script Variables and Properties," which begins on page 252.

statement is any AppleScript statement.

NOTES

The statements in a command handler can include a Continue statement, which passes the command to the application's default handler for that command. This allows you to invoke an application's default behavior for a command from within a command handler. For more information, see "The Continue Statement," which begins on page 277.

Command Handlers for Script Applications

Getting Started With AppleScript describes how you can use the Script Editor to save a script as a script application. A **script application** is a script that you can run from the Finder much like any other application. If you save a script as a stay-open application, it stays open after it runs; if you don't, it quits right after it runs.

Every script application can respond to at least two commands: the Run command and the Open command. Like any other application, a script application receives a Run command whenever it is launched, and an Open command whenever another icon is dragged and dropped over its icon.

Stay-open script applications can also receive and handle any other commands. All stay-open applications receive periodic Idle commands whenever they're not responding to other events and Quit commands whenever the user quits the application.

This section describes handlers for the Run, Open, Idle, and Quit commands. It also describes how you can call a script application from another script.

Run Handlers

All applications that are compatible with System 7 can respond to the Run command, even if they aren't scriptable. The Finder sends a Run command to an application whenever that application is not already running and one of the following actions occurs:

- The user double-clicks the application's icon.
- The user selects the application's icon and chooses Open from the File menu.
- The application's icon is in the Apple Menu Items folder and the user chooses it from the Apple menu.
- The application's icon is in the Startup Items folder and the user restarts the computer.

If the application is already running when one of these actions occurs, the application is activated but no commands are sent to it. If the application isn't

running, the Finder launches the application and sends it a Run command. The application responds by performing the actions the user expects when the application first opens, such as opening an untitled document.

Like any other application, a script application receives a Run command whenever one of the actions just listed occurs. You can provide a handler for the Run command in a couple of ways. An **implicit Run handler** consists of all statements at the top level of a script except for property declarations, script object definitions, and other command handlers. An **explicit Run handler,** like any other handler, is enclosed within an `on...end` statement.

For example, the script that follows consists a property declaration, an `increment` command, a handler for the `increment` command, and a Tell statement. For the Tell statement to work, you have a Scriptable Text Editor document named Count Log open before you run the script. Each time you run the script, the value of the property x increases by 1 and the increase is recorded in the Count Log.

```
property x : 0

increment()

on increment()
   set x to x + 1
   display dialog "Count is now " & x & "."
end increment

tell document ¬
   "Count Log" of application "Scriptable Text Editor"
   set selection to "Count is now " & x & "." & return
end tell
```

The implicit Run handler for this script consists of the statement `increment()` and the Tell statement. If you store this script in a script application and then double-click the script application's icon, the Finder sends a Run command to the script, and the Run command invokes the two statements in the implicit Run handler.

The script in the preceding example behaves exactly the same way if you rewrite it with an explicit Run handler, like this:

```
property x : 0

on run
   increment()
   tell document ¬
      "Count Log" of application "Scriptable Text Editor"
      set selection to "Count is now " & x & "." & return
   end tell
end run

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

The Run handlers in the preceding examples respond the same way to a Run command whether the script is saved as a script application or as a compiled script. If the script is saved as a compiled script, you can invoke its Run handler by clicking the Run button in the Script Editor.

Note
A script can't include both a implicit and an explicit Run handler. If a script includes both an explicit `on run` handler and top level commands that constitute an implicit Run handler, AppleScript returns an error when you try to compile the script—that is, when you try to run it, check its syntax, or attempt to save it. ◆

The implicit Run handler allows a user to execute top-level statements in a script application by launching it from the Finder. For example, if a script application whose script consists only of the word

```
beep
```

is not already open and a user double-clicks its icon, the script application launches and (after optionally displaying a startup screen) beeps.

By default, a startup screen appears before the script runs. The user must click the startup screen's Run button or press the Return key before the Finder actually sends the Run command. This allows the user to read the description of the script before running it. If the Never Show Startup Screen checkbox is selected in the Script Editor's Save As dialog box when the script application is created, the script runs immediately without displaying the startup screen.

You can also send a Run command to a script application from within another script. For information about how to do this, see "Calling a Script Application" on page 251.

Open Handlers

All applications that are compatible with System 7 can respond to the Open command, even if they aren't scriptable. The Finder sends an Open command to an application whenever the user drags file, folder, or disk icons over the application's icon and releases the mouse button. The Open command is sent even if the application is already running.

Like any other application, a script application receives an Open command whenever the user drags file, folder, or disk icons over the application's icon. If the script in the script application includes an Open handler, the statements within the handler run when the application receives the Open command. The Open handler takes a single parameter; when the handler is called, the value of that parameter is a list of all the items whose icons were dropped on the script application's icon. (Each item in the list is an alias; you can convert it to a pathname by using `as string`.)

For example, this Open handler makes a list of the pathnames for all items dropped on the script application's icon:

```
on open names
   tell application "Scriptable Text Editor"
      make new window
      repeat with i in names
         set iPath to (i as string)
         set selection to iPath & return
      end repeat
      save front window in file "List of Files"
   end tell
end open
```

Files, folders, or disks are not moved, copied, or affected in any way when their icons are dragged and dropped over a script application's icon. The Finder just gets a list of their identities and sends that list to the script application as the direct parameter of the Open event. Of course, the script in the script application could easily tell the Finder to move, copy, or otherwise manipulate them.

Note
Due to a known limitation of system software, you can't drop icons on an icon for a script application that's stored on a floppy disk. ◆

You can also run an Open handler by sending a script application the Open command. For details, see "Calling a Script Application" on page 251.

Handlers for Stay-Open Script Applications

By default, a script application that receives a Run or Open command handles that single command and then quits. This allows it to perform a single task and get out of your way. In contrast, a stay-open script application (one saved with the Stay Open checkbox selected in the Script Editor's Save As dialog box) stays open after it's launched.

A stay-open script application can be used for any of these purposes:

- If you run a script frequently, it runs faster as a stay-open application than it does if it has to be launched each time.
- Stay-open script applications can receive and handle other commands in addition to Run and Open. This allows you to use a script application as a script server that, when its running, provides a collection of handlers that can be invoked by any other script.
- Stay-open script applications can perform periodic actions, even in the background, as long as the script application is running.

All stay-open applications receive periodic Idle events. If a stay-open script application includes a handler for the Idle event, it can perform periodic actions whenever it is not responding to other events. If a stay-open script application includes a handler for the Quit event, it can perform some action, such as checking with the user, before quitting.

Idle Handlers

If a stay-open script application includes an Idle handler, AppleScript sends the script application periodic Idle commands whenever it's not responding to incoming events. The statements in the handler run periodically (every 30 seconds, by default).

For example, this handler causes a stay-open script application to beep every 30 seconds after it has been launched:

```
on idle
   beep
end idle
```

To change the rate, return the number of seconds to wait as the result of the script. For example, this script beeps every 5 seconds:

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

If an Idle handler returns a positive number, that number becomes the rate (in seconds) at which the handler is called. If the handler returns a non-numeric value, the rate is not changed.

Remember that the result returned from a handler is just the result of the last statement, even if it doesn't include the word `return` explicitly. For example, this handler only gets called every 15 minutes.:

```
on idle
   set x to 30
   beep
   set x to x * x --the result (900) is returned from
                  --the handler
end idle
```

To make sure you're not changing the idle rate, return 0 at the end of the handler.

Quit Handlers

AppleScript sends a stay-open script application a Quit command whenever the user chooses the Quit menu command or presses Command-Q while the application is active. If the script includes a Quit handler, the statements in the handler are run before the application quits.

A Quit handler can be used to set script properties, tell another application to do something, display a dialog box, or perform almost any other task. If the handler includes a `continue quit` statement, the script application's default quit behavior is invoked and it quits. If the Quit handler returns before it encounters a `continue quit` statement, the application doesn't quit.

For example, this handler checks with the user before allowing the application to quit:

```
on quit
   display dialog "Really quit?" ¬
      buttons {"No", "Quit"} default button "Quit"
   if the button returned of the result is "Quit" then
      continue quit
   end if
   --if the continue statement isn't encountered, the
   --script application doesn't quit.
end quit
```

▲ **WARNING**

If AppleScript doesn't encounter a `continue quit` statement while executing an `on quit` handler, it may seem impossible to quit the application. For example, if the handler gets an error before the `continue quit` statement, attempting to quit the application just produces an error alert. As a last resort, use the emergency Quit command: press Command-Shift-Q or hold down the Shift key and choose Quit from the File menu. This saves changes to script properties and quits immediately, bypassing the Quit handler. ▲

Interrupting a Script Application's Handlers

A stay-open script application handles incoming commands even if it is already running a handler in response to a previous command. This means that execution of a handler can be interrupted while another handler is run. Because script applications are not multitasking, execution of the first handler halts until the second one finishes.

This can cause problems if both handlers modify the same script property or global variable or if both attempt to modify an application's data. For example, suppose that running a script application named Increment causes it to increment the property `p` for several minutes:

```
property p : 0

on close
   set temp to p
   set p to 0
   return temp
end close

set p to 0
repeat 1000000 times
   set p to p + 1
end repeat
```

If while this script application is running it receives a Close command, the property `p` is reset to 0 and the script application begins incrementing `p` all over again:

```
tell application "Increment" to close
```

AppleScript can't deal with such interruptions automatically.

Calling a Script Application

As previously mentioned, any script can send commands to a script application just as it can to any other application. However, script applications, like other applications, sometimes respond to the Run command in ways that you might not expect.

As explained in the description of the Launch command on page 103, AppleScript sends an implicit Run command whenever it begins to execute a Tell statement whose target is an application that is not already open. This creates problems for a script application that doesn't stay open.

For example, a script like this won't run correctly if the target application is a script application that doesn't stay open:

```
tell application "NonStayOpen" to run
```

Instead, the Tell statement launches the script application and sends it an implicit Run command. The application handles that Run command. AppleScript then gets to the explicit Run command in the calling script and tries to send another run event to the script application. Unfortunately, the application has already handled its one event and quits without responding to the second Run command. The calling script waits in vain until it times out, and then receives an error.

The culprit is the implicit Run command sent by the Tell statement when it launches the application. To launch a non-stay-open application and run its script, use a Launch command followed by a Run command, like this:

```
launch application "NonStayOpen"
run application "NonStayOpen"
```

The Launch command launches the script application without sending it an implicit Run command. When the Run command is sent to the script application, it processes the event, sends back a reply if necessary, and quits.

Similarly, to launch a non-stay-open application and run its Open Handler, use a Launch command followed by an Open command, like this:

```
tell application "NonStayOpen"
   launch
   open {alias "HardDisk:MyFile", ¬
      alias "HardDisk:MyOtherFile"}
end tell
```

For example, if the Open handler on page 246 were saved as a script application called "NonStayOpen," the script in the preceding example would cause the handler to create a list of the two specified pathnames.

Scope of Script Variables and Properties

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

The **scope** of a variable or property declaration is the extent to which AppleScript recognizes the declared identifier within a script. It is often convenient to limit the scope of a particular identifier to a single handler—that is, to treat the identifier as a **local variable.** After a local variable has served its purpose, its identifier no longer has any value associated with it and can be used again for other purposes elsewhere in the script.

If you want the value of a script to persist after a script is run, or if you wish to use the same identifier in several different places in a script, you can declare it as either a script property or a **global variable.** AppleScript keeps track of properties and global variables across multiple handlers and script objects within a single script.

This section describes how AppleScript interprets various forms of declarations within handlers, within script objects, and at the top level of a script. You should be familiar with the section "Run Handlers," which begins on page 243, before you read this section.

You can declare a property and set its initial value using a statement like this:

```
property x: 3
```

The scope of a property declaration can be either a script object or an entire script. The value set by a property declaration is not reset each time the script is run; instead, it persists until the script is recompiled.

A global declaration is much the same as a property declaration except that it doesn't set an initial value:

```
global x
```

The scope of a global variable declaration can be limited to specific handlers or script objects or can extend throughout an entire script. Like the value of a property, the value of a global variable is not reset each time a script is run. However, the value of a global variable must be set by other statements in the script.

To set the value of any property or variable, use the Set command. (You can also use the Copy command for this purpose.)

```
set x to 3
```

If the variable has not previously been declared, the Set or Copy command declares it as a local variable. But in some cases it is also necessary to declare a local variable explicitly.

```
local x
```

Like a global declaration, an explicit local declaration doesn't set an initial value.

The preceding examples represent the four basic forms for declaring variables and properties in AppleScript. The sections that follow describe how AppleScript interprets these four forms of declarations within handlers, within script objects, and at the top level of a script.

Scope of Properties and Variables Declared at the Top Level of a Script

Figure 8-1 summarizes the scope of properties and variables declared at the top level of a script. Sample scripts using each form of declaration follow.

Figure 8-1 Scope of property and variable declarations at the top level of a script

Form of declaration	Scope of declaration	Where AppleScript looks for x
`property x: 3`	Everywhere in script	To top level of script
`global x`		
`set x to 3`	Within Run handler only	
`local x`		Within Run handler only

The scope of a property declaration at the top level of a script extends to any subsequent statements anywhere in the script. Here's an example:

```
property theCount : 0
increment()

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

When it encounters the identifier `theCount` at any level of this script, AppleScript associates it with the `theCount` property declared at the top level.

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

Similarly, the scope of a global variable declaration at the top level of a script extends to any subsequent statements anywhere in the script. The next

example accomplishes the same thing as the previous example, except that it uses a global variable instead of a property to keep track of the count.

```
global theCount
increment()

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

When it encounters the identifier `theCount` at any level of this script, AppleScript associates it with the `theCount` variable declared as a global at the top level of the script. However, because a global variable declaration doesn't set the initial value of a property, the script must use a Try statement to determine whether the value has been previously set. Thus, if you want the value associated with an identifier to persist, it is often easier to declare it as a property so that you can declare its initial value at the same time.

If you don't want the value associated with an identifier to persist after a script is run but you want to use the same identifier throughout a script, declare a global variable and use the Set command to set its value each time the script is run. Here's an example:

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

increment() --result: 1
increment() --result: 2
```

Each time the `on increment` handler is called within the script, the global variable `theCount` increases by 1. However, when you run the entire script again, `theCount` is reset to 1.

In the absence of a global variable declaration at the top level of a script, the scope of a variable declaration using the Set command at the top level of a script is normally restricted to the Run handler for the script. For example, this script declares two separate `theCount` variables:

```
set theCount to 10
on increment()
   set theCount to 5
end increment

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

The scope of the first `theCount` variable's declaration, at the top level of the script, is limited to the Run handler for the script. The scope of the second `theCount` declaration, within the `on increment` handler, is limited to that handler. AppleScript keeps track of each variable independently.

To associate a variable in a handler or a script object with the same variable declared at the top level of a script with the Set command, you can use a global declaration in the handler, as shown in the next example.

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

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

In this case, when AppleScript encounters the `theCount` variable within the `on increment` handler, it looks for a previous mention of `theCount` not only within the handler, but also at the top level of the script. However, references

to `theCount` in any other handler in the script are local to that handler unless the handler also explicitly declares `theCount` as a global. This kind of global declaration is discussed in more detail in the sections that follow.

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 theCount
set theCount to 10
on increment()
   global theCount
   set theCount to theCount + 2
end increment

increment() --error: "The variable theCount is not defined"
theCount --result: 10
```

Because the `theCount` variable in this example is declared as local to the Run handler, any subsequent attempt to use the same variable as a global results in an error.

Note
If you declare a variable with the Set command at the top level of a script or script object and then declare the same identifier as a property, the declaration with the Set command overrides the property declaration. For example, the script

```
set x to 10
property x: 5
return x
```

returns 10, not 5. This occurs because AppleScript always evaluates property declarations at the top level of a script before it evaluates Set command declarations. ◆

Scope of Properties and Variables Declared in a Script Object

You should be familiar with Chapter 9, "Script Objects," before you read this section.

Figure 8-2 summarizes the scope of properties and variables declared at the top level of a script object. Sample scripts using each form of declaration follow.

Figure 8-2 Scope of property and variable declarations at the top level of a script object

Form of declaration	Scope of declaration	Where AppleScript looks for x
`property x: 3`	Everywhere in script object	To top level of script object
`global x`		To top level of script
`set x to 3`	Within script object's Run handler only	Within script object's Run handler only
`local x`		

The scope of a property declaration at the top level of a script object extends to any subsequent statements in that script object. Here's an example.

```
script Joe
property theCount : 0
   on increment()
      set theCount to theCount + 1
      return theCount
   end increment
end script

tell Joe to increment() --result: 1
tell Joe to increment() --result: 2
```

When it encounters the identifier `theCount` at any level of the script object `Joe`, AppleScript associates it with the same identifier declared at the top level of the script object. The value of the property `theCount` persists until you reinitialize the script object by running the script again.

The scope of a property declaration at the top level of a script object doesn't extend beyond the script object. Thus, it is possible to use the same identifier in different parts of a script to refer to different properties, as this example demonstrates:.

```
property theCount : 0
   script Joe
      property theCount : 0
      on increment()
         set theCount to theCount + 1
         return theCount
      end increment
   end script

tell Joe to increment() --result: 1
tell Joe to increment() --result: 2
theCount --result: 0
```

AppleScript keeps track of the property `theCount` declared at the top level of the script separately from the property `theCount` declared within the script object `Joe`. Thus, the `theCount` property declared at the top level of the script `Joe` is increased by 1 each time `Joe` is told to increment, but the `theCount` property declared at the top level of the script is not affected.

Like the scope of a property declaration, the scope of a global variable declaration at the top level of a script object extends to any subsequent statements in that script object. However, as the next example demonstrates, AppleScript also associates a global variable with the same variable declared at the top level of the entire script.

```
set theCount to 0
script Joe
global theCount
   on increment()
      set theCount to theCount + 1
      return theCount
   end increment
end script

tell Joe to increment() --result: 1
tell Joe to increment() --result: 2
```

The preceding example first sets the value of `theCount` at the top level of the script. When AppleScript encounters the `theCount` variable within the `on increment` handler, it first looks for an earlier occurrence within the handler, then at the top level of the script `Joe`. When AppleScript encounters the global declaration for `theCount` at the top level of script object `Joe`, it continues looking at the top level of the script until it finds the original declaration for `theCount`. This can't be done with a property of a script object, because AppleScript looks no further than the top level of a script object for that script object's properties.

Like the value of a script object's property, the value of a script object's global variable persists after the script object has been run, but not after the script itself has been run. Thus, telling `Joe` to increment repeatedly in the preceding example continues to increment the value of `theCount`, but running the whole script again sets `theCount` to 0 again before incrementing it.

The next example demonstrates how you can use a global variable declaration in a script object to associate a global variable with a property declared at the top level of a script.

```
property theCount : 0
script Norah
   property theCount : 20
   script Joe
      global theCount
      on increment()
         set theCount to theCount + 1
         return theCount
      end increment
   end script
   tell Joe to increment()
end script
```

```
run Norah --result: 1
run Norah --result: 2
theCount --result: 2
theCount of Norah --result: 20
```

This script declares two separate `theCount` properties: one at the top level of the script and one at the top level of the script object `Norah`. Because the script `Joe` declares the global variable `theCount`, AppleScript looks for `theCount` at the top level of the script, thus treating Joe's `theCount` and `theCount` at the top level of the script as the same variable.

If the script object `Joe` in the preceding example doesn't declare `theCount` as a global variable, AppleScript treats Joe's `theCount` and the `theCount` at the top level of the script object `Norah` as the same variable. This leads to quite different results, as shown in the next example.

```
property theCount : 0
script Norah
   property theCount : 20
   script Joe
      on increment()
         set theCount to theCount + 1
         return theCount
      end increment
   end script
   tell Joe to increment()
end script
```

```
run Norah --result: 21
run Norah --result: 22
theCount --result: 0
theCount of Norah -- result:22
```

The scope of a variable declaration using the Set command at the top level of a script object is limited to the Run handler:

```
script Joe
   set theCount to 10
   on increment()
      global theCount
      set theCount to theCount + 2
   end increment
   return theCount
end script

tell Joe to increment()
--error: "The variable theCount is not defined."

run Joe--result: 10
```

In contrast to the way it treats such a declaration at the top level of a script, AppleScript treats the `theCount` variable declared at the top level of the script object `Joe` in the preceding example as local to the script object's Run handler. Any subsequent attempt to use the same variable as a global results in an error.

Similarly, the scope of an explicit local variable declaration at the top level of a script object is limited to that script object's Run handler, even if the same identifier has been declared as a property at a higher level in the script:

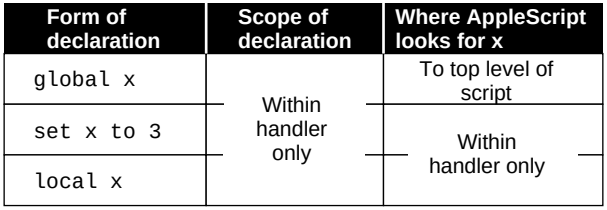
```
property theCount : 0
script Joe
   local theCount
   set theCount to 5
   on increment()
      set theCount to theCount + 1
   end increment
end script

run Joe --result: 5
tell Joe to increment() --result: 1
```

Scope of Variables Declared in a Handler

You can't declare a property in a handler, although you can refer to a property declared at the top level of the script or script object to which the handler belongs.

Figure 8-3 summarizes the scope of variables declared in a handler. Examples of each form of declaration follow.

Figure 8-3 Scope of variable declarations within a handler



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 theCount to 10
on increment()
   global theCount
   set theCount to theCount + 2
end increment

increment() --result: 12
theCount --result: 12
```

When AppleScript encounters the `theCount` variable within the `on increment` handler, it doesn't restrict its search for a previous occurrence to that handler but keeps looking until it finds the declaration at the top level of the script. However, references to `theCount` in any subsequent handler in the script are local to that handler unless the handler also explicitly declares `theCount` as a global variable.

The scope of a variable declaration using the Set command within a handler is limited to that handler:

```
script Henry
   set theCount to 10
   on increment()
      set theCount to 5
   end increment
   return theCount
end script

tell Henry to increment() --result: 5
run Henry --result: 10
```

The scope of the first declaration of the first `theCount` variable, at the top level of the script object `Henry`, is limited to the Run handler for the script object. The scope of the second `theCount` declaration, within the `on increment` handler, is limited to that handler. AppleScript keeps track of each variable independently.

The scope of a local variable declaration in a handler is limited to that handler, even if the same identifier has been declared as a property at a higher level in the script:

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

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

CHAPTER 9

Script Objects

Script objects are objects that you define and use in scripts. Like the application and system objects described earlier in this manual, script objects have properties and can respond to commands. Unlike application or system objects, script objects are defined within scripts.

This chapter describes how to define and use script objects. It begins by describing a simple script object definition and demonstrating how you would send a command to the resulting script object. Subsequent sections describe in more detail how to define, send commands to, and initialize script objects.

You can define groups of script objects that share properties and handlers, and you can extend the behavior of a handler in one script object by calling it from another script object. The section "Inheritance and Delegation" describes how this works. If you are familiar with object-oriented design, you may recognize the techniques described in this section.

The last section, "Using the Copy and Set Commands With Script Objects," describes what to expect when you set a variable to a script object or copy a script object to a variable and how to write a handler that creates copies of script objects.

About Script Objects

Script objects are user-defined objects that combine data (in the form of properties) and potential actions (in the form of handlers). **Script object definitions** are compound statements that can contain collections of properties, handlers, and other AppleScript statements.

Here is a simple script object definition:

```
script John
   property HowManyTimes : 0
   to sayHello to someone
      set HowManyTimes to HowManyTimes + 1
      return "Hello " & someone
   end sayHello
end script
```

It defines a script object that can handle the `sayHello` command. It assigns the script object to the variable `John`. The definition includes a handler for the `sayHello` command. It also includes a property, called HowManyTimes, that indicates how many times the `sayHello` command has been called.

A handler within a script object definition follows the same syntax rules as a subroutine definition. Unlike a subroutine definition, however, you can group a handler within a script object definition with properties whose values are related to the handler's actions.

After you define a script object, you initialize it by running the script that contains the script object definition. You can then use a Tell statement to send commands to the script object. For example, the following statement sends the `sayHello` command the script object defined above.

```
tell John to sayHello to "Herb"
```

The result is `"Hello Herb"`.

You can manipulate the properties of script objects in the same way you manipulate the properties of system and application objects. Use the Get command to get the value of a property and the Set or Copy command to change the value of a property.

The following statement uses a Get command to get the value of the HowManyTimes property of script object `John`.

```
get HowManyTimes of John
if the result > 10
   return "John, aren't you tired of saying hello?"
end if
```

Defining Script Objects

Each script object definition begins with the keyword `script`, followed by an optional variable name, and ends with the keyword `end` (or `end script`). The statements in between can be any combination of property definitions, handler definitions, and other AppleScript statements.

The syntax of a script object definition is

```
script [ scriptObjectVariable ]
   [( property | prop ) propertyLabel : initialValue ]...
   [ handlerDefinition ]...
   [ statement ]...
end [script]
```

where

scriptObjectVariable is a variable identifier. If you include *scriptObjectVariable*, AppleScript stores the script object in a variable. You can use the variable identifier to refer to the script object elsewhere in the script.

propertyLabel is an identifier for a property. Properties are characteristics that are identified by unique labels. They are similar to instance variables in object-oriented programming.

initialValue is the value that is assigned to the property each time the script object is initialized. Script objects are initialized when the scripts or handlers that contain them are run. *initialValue* is required in property definitions.

handlerDefinition is a handler for a user-defined or system command. The handlers within a script object definition determine which commands the script object can respond to. Script object definitions can include handlers for user-defined commands (subroutines) or for system or application commands. Handlers in script objects are similar to methods in object-oriented programming. For a detailed description of the syntax of handler definitions, refer to Chapter 8, “Handlers.”

statement is any AppleScript statement. Statements other than handler and property definitions are treated as if they were part of a handler definition for the Run command; they are executed when a script object receives the Run command.

Sending Commands to Script Objects

You use Tell statements to send commands to script objects. A Tell statement sent to a script object is similar to a Tell statement sent to an application, except that it uses a variable name, instead of a reference, to identify the script object. For example,

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

sends two `sayHello` commands to the script object `John`. The parameters of the commands in the Tell statement, if any, must match the parameters defined in the handler definitions in the script object definition. For example, the statement

```
tell John
   sayHello ("Herb")
end tell
--results in an error
```

results in an error message because the handler definition for the `sayHello` command (shown earlier in this chapter) defines a labeled parameter, not a positional parameter.

For a script object to respond to a command within a Tell statement, either the script object or its parent script object must have a handler for the command. A parent script object is a script object from which a script object inherits handlers and properties. (For more information about parent script objects, see “Inheritance and Delegation” on page 271.)

The one command that any script object can handle, even without an explicitly defined handler, is the Run command. A handler for the Run command can consist of all statements at the top level of a script object definition other than property and handler definitions. If the script object definition contains only

handler and property definitions, and does not include any additional top-level statements, the definition may include an explicit Run handler that begins with `on run`. If a script object definition includes neither an implicit Run handler (in the form of top-level statements) nor an explicit Run handler, the Run command doesn't do anything. (For more information about Run handlers, see "Command Handlers for Script Applications," which begins on page 243.)

For example, the Display Dialog command in the following script object definition is executed only if you send a Run command to script object `John`.

```
script John
   property HowManyTimes : 0
   to sayHello to someone
      set HowManyTimes to HowManyTimes + 1
      return "Hello " & someone
   end sayHello
   display dialog "John received the Run command"
end script
```

Initializing Script Objects

When you define a script object, you define a collection of handlers and properties. When you run a script containing a script object definition, AppleScript creates a script object with the properties and handlers listed in the definition. This is called **initializing a script object.** A script object must be initialized before it can respond to commands.

If you include a script object definition at the top level of a script—that is, as part of the script's Run handler—AppleScript initializes the script object each time the script's Run handler is executed. (For more information about Run handlers, see page 243.)

Similarly, if you include a script definition in another handler within a script, AppleScript initializes a script object each time the handler is called. The parameter variables in the handler definition become local variables of the

script object. For example, the `makePoint` handler in the following script contains a script object definition for the script object `point`:

```
on makePoint(x, y)
   script point
      property xCoordinate:x
      property yCoordinate:y
   end script
   return point
end makePoint

set myPoint to makePoint(10,20)
get xCoordinate of myPoint
get yCoordinate of myPoint
```

AppleScript initializes the script object `point` when it executes the `makePoint` command. The parameter variables in the `makePoint` handler, in this case, `x` and `y`, become local variables of the script object `point`. The initial value of `x` is 10, and the initial value of `y` is 20, because those are the parameters of the `makePoint` command that initialized the script object.

One way to use script object definitions in handlers is to define constructor functions, that is, handlers that create script objects. The following script uses a constructor function to create three script objects.

```
on makePoint(x, y)
   script
      property xCoordinate:x
      property yCoordinate:y
   end script
end makePoint

set PointA to makePoint(10,20)
set PointB to makePoint(100,200)
set PointC to makePoint(1,1)
```

As in the previous example, you can retrieve the coordinates of the three script objects using the Get command.

Note
The distinction between defining a script object and initializing a script object is similar to the distinction between a class and an instance in object-oriented design. When you define a script object, you define a class of objects that respond to the same commands. When AppleScript initializes a script object, it creates an instance of the class. The script object gets its initial context (property values and handlers) from the script object definition, but its context can change as it responds to commands. ◆

Inheritance and Delegation

You can use AppleScript's inheritance mechanism to define related script objects in terms of one another. This allows you to share property and handler definitions among many script objects without repeating the shared definitions.

This section describes

- how to a define a script object that inherits properties and handlers from another script object
- how inheritance works
- how to use the Continue statement to extend the behavior of an inherited handler without completely replacing it

Defining Inheritance

You define **inheritance** with the Parent property. A script object that includes a Parent property inherits the properties and handlers of the script object listed in the Parent property.

The script object listed in a Parent property definition is called the **parent script object,** or parent. A script object that includes a Parent property is referred to as a **child script object,** or child. The Parent property is not required. A script object can have many children, but a child script object can have only one parent.

The syntax for defining a parent script object is

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

where

variable is a variable that contains the parent script object.

A script object must be initialized before it can be assigned as a parent of another script object. This means that the definition of the parent script object (or a command that calls a function that creates the parent script object) must come before the definition of the child in the same script.

How Inheritance Works

To understand how inheritance works, think of a child script object as containing a hidden copy of each of the handlers and properties inherited from its parent. If the child does not have its own definition of a property or handler, it uses the inherited (hidden) property or handler. If the child has its own definition of a particular property or handler, then it ignores the inherited property or handler.

Figure 9-1 shows the relationship between a parent script object called `John` and a simple child script object called `Simple`. The figure includes two versions of the child script object. The version on the left shows the actual script object definition for the child script `Simple`. The version on the right shows how the script object definition would look with the inherited properties and handlers copied in. The inherited properties and handlers are shown between dotted lines, to indicate that they aren't actually a part of the script object definition for `Simple`. As you can see, `Simple` inherits the HowManyTimes property and the `sayHello` handler from its parent.

Figure 9-2 shows another parent-child relationship. As in the previous example, the child script object inherits the HowManyTimes property and the `sayHello` handler from its parent, `John`. But this time, the child script object, called `Rebel`, has its own HowManyTimes property, so it doesn't use the one inherited from the parent. In the figure, the inherited property that is not used is crossed out.

Figure 9-1 Relationship between a simple child script and its parent

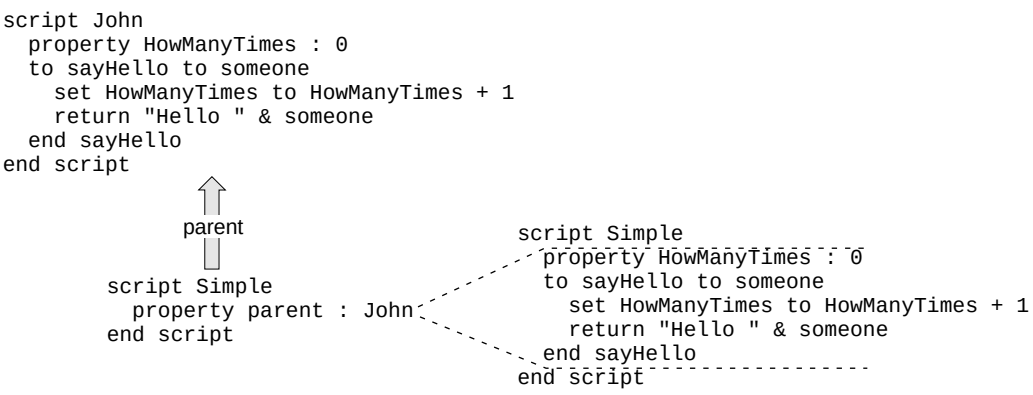
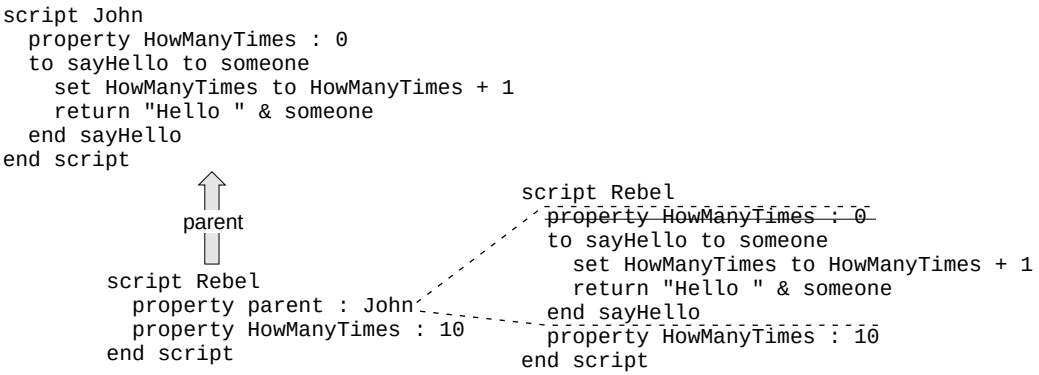


Figure 9-2 Another child-parent relationship

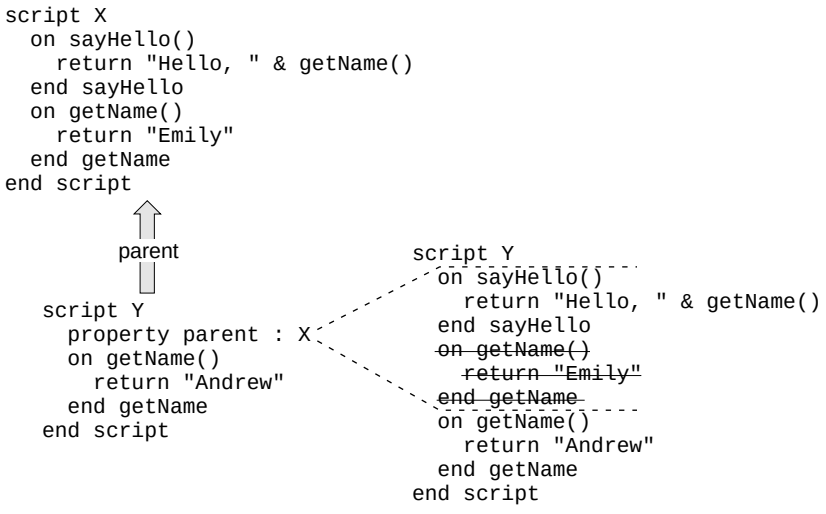


Drawing diagrams like Figure 9-1 and Figure 9-2 can help you understand more complicated relationships between parent and child script objects. For example, if you were to guess the result of the following script without sketching a diagram, you might conclude that the result of the `sayHello` command is

"Hello Emily". However, the correct result is "Hello Andrew", as you can see in Figure 9-3.

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

script Y
   property parent : X
   on getName()
      return "Andrew"
   end getName
end script

tell Y to sayHello()
```

Figure 9-3 A more complicated child-parent relationship



Even though script X in Figure 9-3 sends itself the `getName` command, the command is intercepted by the child script, which substitutes its own version of the `getName` handler. AppleScript always maintains the first target of a command as the "self" to which inherited commands are sent, redirecting to the child any inherited commands the parent sends to itself.

The relationship between a parent script object and its child script objects is dynamic. If the properties of the parent change, so do the inherited properties of the children. For example, the script object `Simple` in the following script inherits its Vegetable property from script object `John`.

```
script John
   property Vegetable : "Spinach"
end script

script Simple
   property parent : John
end script

set Vegetable of John to "Swiss chard"
Vegetable of Simple
--result: "Swiss chard"
```

When you change the Vegetable property of script object `John` with the Set command, you also change the Vegetable property of the child script object `Simple`. The result of the last line of the script is `"Swiss chard"`.

Similarly, if a child changes one of its inherited properties, the value of the parent property changes. For example, the script object `JohnSon` in the following script inherits the Vegetable property from script object `John`.

```
script John
   property Vegetable : "Spinach"
end script

script JohnSon
   property parent : John
   on changeVegetable()
      set my Vegetable to "Zucchini"
   end changeVegetable
end script
```

```
tell JohnSon to changeVegetable()
Vegetable of John
--result: "Zucchini"
```

When you change the Vegetable property of script object `JohnSon` to `"Zucchini"` with the `changeVegetable` command, the Vegetable property of script object `John` also changes.

The previous example demonstrates an important point about inherited properties: to refer to an inherited property from within a child script object, you must use the reserved word `my` or `of me` to indicate that the value to which you're referring is a property of the current script object. (You can also use the words `of parent` to indicate that the value is a property of the parent script object.) If you don't, AppleScript assumes the value is a local variable.

For example, if you refer to `Vegetable` instead of `my Vegetable` in the `changeVegetable` handler in the previous example, the result is `"Spinach"`.

```
script John
   property Vegetable : "Spinach"
end script

script JohnSon
   property parent : John
   on changeVegetable()
      set Vegetable to "Zucchini" (* creates a local variable called
                                     Vegetable; doesn't change value of
                                     the parent's Vegetable property *)
   end changeVegetable
end script

tell JohnSon to changeVegetable()
Vegetable of John
--result: "Spinach"
```

The Continue Statement

Normally, if a child script object and its parent both have handlers for the same command, the child uses its own handler. However, the handler in a child script object can handle a command first, and then use a Continue statement to call the handler for the same command in the parent.

The use of a Continue statement to call a handler in a parent script object is called **delegation.** By delegating commands to a parent script object, a child can extend the behavior of a handler contained in the parent without having to repeat the entire handler definition. After the parent handles the command, AppleScript continues at the place in the child where the Continue statement was called. Handlers in child script objects that contain Continue statements are similar to wrapper methods in object-oriented programming.

The syntax of a Continue statement is

`continue` *commandName parameterList*

where

commandName is the name of the current command.

parameterList is the list of parameters to be passed with the command. The list must follow the same format as the parameter definitions in the handler definition for the command. For handlers with labeled parameters, this means that the parameter labels must match those in the handler definition. For handlers with positional parameters, the parameters must appear in the correct order. You can list actual values or parameter variables. If you list actual values, those values replace the parameter values that were specified in the original command. If you list parameter variables, the Continue statement passes the parameter values that were specified in the original command.

The following script includes two script object definitions similar to those shown in Figure 9-1 on page 273. The first, `Elizabeth`, works just like the script `John` in the figure. The second, `ChildOfElizabeth`, includes a handler with a Continue statement that is not included in the child script object (`Simple`) shown in the figure.

```
script Elizabeth
   property HowManyTimes : 0
   to sayHello to someone
      set HowManyTimes to HowManyTimes + 1
      return "Hello " & someone
   end sayHello
end script

script ChildOfElizabeth
   property parent : Elizabeth
   on sayHello to someone
      if my HowManyTimes > 3 then
         return "No, I'm tired of saying hello."
      else
         continue sayHello to someone
      end if
   end sayHello
end script
```

In the preceding example, the handler defined by `ChildOfElizabeth` for the `sayHello` command checks the value of the HowManyTimes property each time the handler is run. If the value is greater than 3, `ChildOfElizabeth` returns a message refusing to say hello. Otherwise, `ChildOfElizabeth` calls the `sayHello` handler in the parent script object (`Elizabeth`), which returns the standard hello message. The word `someone` in the Continue statement is a parameter variable. It indicates that the parameter received with the original `sayHello` command will be passed to the handler in the parent script.

Note
The reserved word `my` in the statement `if my HowManyTimes > 10` in the previous example is required to indicate that HowManyTimes is a property of the script object. Without the word `my`, AppleScript assumes that HowManyTimes is an undefined local variable. ◆

A Continue statement can change the parameters of a command before delegating it. For example, suppose the following script object is defined in the same script as the preceding example. The first Continue statement changes the direct parameter of the `sayHello` command from `"Bill"` to `"William"`. It does this by specifying the value `"William"` instead of the parameter variable `someone`.

```
script AnotherChildOfElizabeth
   property parent : Elizabeth
   on sayHello to someone
      if someone = "Bill" then
         continue sayHello to "William"
      else
         continue sayHello to someone
      end if
   end sayHello
end script
```

If you override a parent's handler in this manner, the reserved words `me` and `my` in the parent's handler no longer refer to the parent, as demonstrated in the example that follows.

```
script Hugh
   on identify()
      me
   end identify
end script
```

```
script Andrea
   property parent : Hugh
   on identify()
      continue identify()
   end identify
end script
tell Hugh to identify()
--result: <<script Hugh>>

tell Andrea to identify()
--result: <<script Andrea>>
```

Using Continue Statements to Pass Commands to Applications

Scripting addition commands and application commands sent to script objects don't trigger their associated actions until they're received by the default target application. This means you can use a handler for such commands within a script object to modify the way the command works when sent to that script object.

For example, the handler for the Beep command in the example that follows modifies the command by displaying a dialog box and allowing the user to decide whether to continue or not:

```
script Joe
   on beep
      set x to display dialog ¬
         "Do you really want to hear this awful noise?" ¬
         buttons {"Yes", "No"}
      if the button returned of x is "Yes" then ¬
         continue beep
   end beep
end script
tell Joe to beep
```

When AppleScript encounters the Tell statement, it sends a Beep command to script `Joe`. The Beep handler causes the default target application (for example, the Script Editor) to display a dialog box that gives the user a choice about hearing the alert sound. If the user clicks Yes, the handler uses a Continue statement to pass the Beep command to the default target application. If the user clicks No, the target application never receives the Beep command and no alert sound is heard.

In applications that allow you to attach script objects to application objects, you can use a handler for an application command in a script object to modify the way the application responds to the command.

For example, if a drawing application allows you to associate script objects with geometric shapes such as circles or squares, you could include a handler like this in a script object associated with a shape in a document:

```
on move to {x, y}
   continue move to {x, item 2 of my position}
end move
```

Whenever the shape the script object is associated with is named as the target of a Move command, the `on move` handler handles the command by modifying one of the parameters and using the continue statement to pass the command on to the default parent—that is, the drawing application. The location specified by `{x, item 2 of my position}` has the same horizontal coordinate as the location specified by the original Move command, but specifies the shape's original vertical coordinate (item 2 of the circle's original position), thus constraining the shape's motion to a horizontal direction.

The documentation for applications that allow you to associate script objects with application objects in this manner should provide more information about how to write handlers for application commands.

The Parent Property and the Current Application

The default parent property for any script that doesn't explicitly declare one is the default target application—usually, the application that is running the script, such as the Script Editor. You can use the predefined variable `current application` to refer to either the default target application or whatever application is currently set as a script's parent property.

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

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

```
property parent: application "Scriptable Text Editor"
close front window
tell current application to return my name
```

In this case, `my` refers to the current application (Scriptable Text Editor). The Tell statement is optional; just `return the name of me` would produce the same result, because AppleScript sends the command to the Scriptable Text Editor. If you remove the property declaration from the script, the Script Editor becomes the current application. When sent to the Script Editor, the Close command and the Return statement produce errors because the Script Editor doesn't understand them.

In the next example, the script `Bilbo` declares the Scriptable Text Editor as its parent property and includes a handler that modifies the behavior of the scripting addition command Display Dialog.

```
script Bilbo
   property parent : application "Scriptable Text Editor"
   on display dialog x
      tell application "Script Editor" to display dialog ¬
         "Scriptable Text Editor has something to say"
      continue display dialog x
   end display dialog
end script

tell Bilbo to display dialog "Hello"
```

Because the script object `Bilbo` declares the Scriptable Text Editor as its parent property, the `on display dialog` handler must use a Tell statement to send a separate Display Dialog command to the Script Editor. The handler then uses a Continue statement to pass the original Display Dialog command to the Scriptable Text Editor, which becomes the frontmost application and uses the Display Dialog addition to display “Hello”.

Using the Copy and Set Commands With Script Objects

The Copy and Set commands both assign values to variables, but they have different results when the value assigned is a script object. The Copy command makes a new copy of the script object, and the Set command creates a variable that shares data with the original script object.

To see how this works, consider the following example, which defines a script object, called `John`, with a property called Vegetable.

```
script John
   property Vegetable: "Spinach"
end script

set myScriptObject to John
set Vegetable of John to "Swiss chard"
get Vegetable of myScriptObject
--result: "Swiss chard"
```

The first Set command defines a variable, called `myScriptObject`, that shares data with the original script object `John`. The second Set command changes the value of the Vegetable property of script object `John` from `"Spinach"` to `"Swiss chard"`. Because `myScriptObject` shares data with `John`, it shares the change to the Vegetable property of `John`. When you get the Vegetable property of `myScriptObject`, the result is `"Swiss chard"`.

Now consider the following example, which uses the Copy command to define the variable `myScriptObject`.

```
script John
   property Vegetable: "Spinach"
end script

copy John to myScriptObject
set Vegetable of John to "Swiss chard"
get Vegetable of myScriptObject
--result: "Spinach"
```

In this case, the Copy command creates a new script object. Setting the Vegetable property of the original script object has no effect on the new script object. The result of the Get command is `"Spinach"`.

When you copy a child script object to a variable, the variable contains a complete copy of both the child and its parent, including all the parent's properties and handlers. Each new copy, including its inherited properties and handlers, is completely independent of both the original and any other copies.

For example, if you copy a modified version of the `JohnSon` script in this example to two different variables, you can set each variable's Vegetable property independently:

```
script John
   property Vegetable : "Spinach"
end script

script JohnSon
   property parent : John
   on changeVegetable(x)
      set my Vegetable to x
   end changeVegetable
end script
```

```
copy JohnSon to J1
copy JohnSon to J2

tell J1 to changeVegetable("Zucchini")
tell J2 to changeVegetable("Swiss chard")

Vegetable of J1
--result: "Zucchini"

Vegetable of J2
--result: "Swiss chard"

Vegetable of John
--result: "Spinach"
```

You can create handlers that construct copies of script objects for use elsewhere in a script. For example, the script that follows includes a handler that takes an initial balance as a parameter and creates a copy of a script object that acts as an independent account. Each copy includes several properties and an `on deposit` handler that enables the script object to increment its own balance when it receives a Deposit command.

```
on makeAccount(initialBalance)
   script account
      property StartDate : current date
      property Balance : initialBalance
      on deposit(amount)
         set Balance to Balance + amount
      end deposit
   end script
end makeaccount

set a to makeAccount(3300)
set b to makeAccount(33)
```

```
tell a
   deposit(30)
   deposit(60)

end tell
{Balance of a, StartDate of a}
--result: {3390, date "Tuesday, July 6, 1993 2:38:11 PM"}

{Balance of b, StartDate of b}
--result: {33, date "Tuesday, July 6, 1993 2:38:12 PM"}
```

Appendixes

APPENDIX A

The Language at a Glance

This appendix summarizes the commands, reference forms, operators, control statements, and other elements of the AppleScript English dialect. For more detailed information on these elements, see the appropriate chapters in this book.

The placeholder descriptions in the last section of this appendix define the placeholders used in the syntax summaries.

Commands

A command is a request for action. In AppleScript, you can use application commands, which are defined in each application's dictionary; AppleScript commands, which are defined and handled by AppleScript; or scripting addition commands, which are defined and handled by AppleScript extensions called scripting additions.

Table A-1 lists standard application commands and AppleScript commands. (For information about scripting addition commands, see the *AppleScript Scripting Additions Guide*.) The syntax shown for standard application commands is the syntax supported by most applications. Individual applications can extend or change the way the standard application commands work.

For information about how a specific application handles a particular application command, see the application's dictionary. For more detailed descriptions of the commands listed here, see Chapter 4, "Commands."

Table A-1 Command syntax

Command	Syntax	Result
`close` (application command)	`close` *referenceToObject* `close` *referenceToObject* `saving in` *referenceToFile* `close` *referenceToObject* `saving` *saveOption* `close` *referenceToObject* `saving in` *referenceToFile* `saving` *saveOption*	None
`copy` (AppleScript command)	`( copy | put )` *expression* `( to | into)` *variablePattern*	Value copied
`copy` (application command)	`( copy | put )` *expression* `( to | into)` *referencePattern* `( copy | put )` `( copy | put )` *referenceToObject*	Value copied (none if no parameters are included)
`count` (AppleScript command)	`count` *compoundValue* `count [ each | every ]` *className* `( in | of )` *compoundValue* `number of` *compoundValue* `number of` *pluralClassName* `( in | of )` *compoundValue*	Integer

`count` (application command)	`count [ each | every ]` *className*
	`count [ each | every ]` *className* `( in | of )` *referenceToObject* `]`
	`number of` *className*
	`number of` *className* `[ ( in | of )` *referenceToObject* `]`
		Integer or list of integers

continued

Table A-1 Command syntax (continued)

Command	Syntax	Result
`data size` (application command)	`data size of` *referenceToObject* `data size of` *referenceToObject* `as` *className*	Integer or list of integers
`delete` (application command)	`delete` *referenceToObject*	None
`duplicate` (application command)	`duplicate` *referenceToObject* `duplicate` *referenceToObject* `to` *referenceToLocation*	Reference

`error` (AppleScript command)	`error`
	`[` *errorMessage* `]`
	`[ number` *errorNumber* `]`
	`[ from` *offendingObject* `]`
	`[ partial result` *resultList* `]`
	`[ to` *expectedType* `]`	

`exists` (application command)	`exists` *referenceToObject* *referenceToObject* `exists`	Boolean
`get` (AppleScript command)	`get` *expression* `get` *expression* `as` *className*	Value of expression
`get` (application command)	`get` *referenceToObject* `get` *referenceToObject* `as` *className*	Value of reference

continued

Table A-1 Command syntax (continued)

Command	Syntax	Result
`launch` (application command)	`launch` `launch` *referenceToApplication*	None
`make` (application command)	`make [new]` *className* `at` *referenceToLocation* `make [new]` *className* `at` *referenceToLocation* ¬ `with properties {` *propertyLabel*`:`*propertyValue* ¬ ¬ `[,` *propertyLabel*`:`*propertyValue* `]...}` `make [new]` *className* `at` *referenceToLocation* ¬ `with data` *dataValue* `make [new]` *className* `at` *referenceToLocation* ¬ `with properties {`*propertyLabel*`:`*propertyValue* ¬ ¬ `[,` *propertyLabel*`:`*propertyValue* `]...}` ¬ `with data` *dataValue*	Reference to the new object
`move` (application command)	`move` *referenceToObject* `to` *referenceToLocation*	Reference to the moved object
`open` (application command)	`open` *referenceToFile* `open` *listOfFiles*	None
`print` (application command)	`print` *referenceToObject*	None

continued

Table A-1 Command syntax (continued)

Command	Syntax	Result
`quit` (application command)	`quit` *referenceToApplication* `quit` *referenceToApplication* `saving` *saveOption*	None
`run` (AppleScript command)	`run` `run` *scriptObjectVariable*	The value, if any, returned by the script object
`run` (application command)	`run` `run` *referenceToApplication*	None
`save` (application command)	`save` *referenceToObject* `save` *referenceToObject* `in` *referenceToFile*	None
`set` (AppleScript command)	`set` *variablePattern* `to` *expression* *expression* `returning` *variablePattern*	Value assigned
`set` (application command)	`set` *referencePattern* `to` *expression* *expression* `returning` *referencePattern*	Value assigned

References

A reference is a phrase that specifies an object. Table A-2 summarizes the reference forms you can use to specify objects in AppleScript. The first column lists the name of the reference form. The second column lists the syntax for that form.

When you use references to specify objects, you can specify a series of containers, each of which is itself a reference, to identify the object uniquely. Table A-3 lists the ways to specify containers.

For examples and more detailed descriptions of the AppleScript reference forms, see Chapter 5, "Objects and References."

Table A-2 Reference form syntax

Reference form	Syntax
Arbitrary Element	`some` *className*
Every Element	`every` *className*
	pluralClassName
Filter	*referenceToObject* `whose` *Boolean*
	referenceToObject `where` *Boolean*
ID	*className* `ID` *IDvalue*
Index	*className* *integer*
	className `index` *integer*
	`first` *className*
	`second` *className*
	`third` *className*
	`fourth` *className*

continued

Table A-2 Reference form syntax (continued)

Reference form	Syntax
	`fifth` *className*
	`sixth` *className*
	`seventh` *className*
	`eighth` *className*
	`ninth` *className*
	`tenth` *className*
	integer `st` *className*
	integer `nd` *className*
	integer `rd` *className*
	integer `th` *className*
	`last` *className*
	`front` *className*
	`back` *className*
Middle Element	`middle` *className*
Name	*className string*
	className `named` *string*
Property	*propertyLabel*
Range	`every` *className* `from` *reference* `to` *reference*
	pluralClassName `from` *reference* `to` *reference*
	className integer `through` *integer*
	className integer `thru` *integer*
	pluralClassName integer `through` *integer*
	pluralClassName integer `thru` *integer*

continued

Table A-2 Reference form syntax (continued)

Reference form	Syntax
Relative	*className* `before` *reference*
	className `front of` *reference*
	className `in front of` *reference*
	className `after` *reference*
	className `back of` *reference*
	className `in back of` *reference*

Table A-3 Container notation in references

Container notation	Syntax
`in`	*reference* `in` *containerReference*
`of`	*referenc*`e of` *containerReference*
`'s`	*containerReference*`'s` *reference*

Operators

Table A-4 summarizes the operators in the AppleScript English language dialect. The first column lists the operators. The second column shows the syntax for using the operators in expressions. The placeholders in the syntax descriptions correspond to AppleScript value classes, which are described briefly in the last section of this appendix, and in more detail in Chapter 3, "Values."

Synonyms are listed in groups. The table shows the syntax for the first operator, but operators that are synonyms follow the same syntax rules.

Table A-4 Operators

Operator	Syntax
Arithmetic operators	
`*`	*number* `*` *number*
`+`	*number* `+` *number* *date* `+` *number*
`-`	*number* `-` *number* *date* `-` *number* *date* – *date*
`÷` `/`	*number* `( ÷ | / )` *number*
`^`	*number* `^` *number*
`div`	*number* `div` *number*
`mod`	*number* `mod` *number*
Logical operators	
`and`	*Boolean* `and` *Boolean*
`not`	`not` *Boolean*
`or`	*Boolean* `or` *Boolean*
Containment operators	
`start[s] with` `begin[s] with`	*list* `starts with` *list* *string* `starts with` *string*
`end[s] with`	*list* `ends with` *list* *string* `ends with` *string*
`contains`	*list* `contains` *list* *record* `contains` *record* *string* `contains` *string*
`does not contain` `doesn't contain`	*list* `does not contain` *list* *record* `does not contain` *record* *string* `does not contain` *string*

continued

Table A-4 Operators (continued)

Operator	Syntax
`is in` `is contained by`	*list* `is in` *list* *record* `is in` *record* *string* `is in` *string*

`is not in`
`is not contained by`
`isn't contained by`
	list `is not in` *list*
	record `is not in` *record*
	string `is not in` *string*

Comparison operators (equality and inequality)	

`=`
`equal`
`equals`
`equal to`
`is`
`is equal to`
	expression `=` *expression*

`≠`
`does not equal`
`doesn't equal`
`is not`
`is not equal [to]`
`isn't`
`isn't equal [to]`
	expression `≠` *expression*

Comparison operators (precedence)	

`<`
`comes before`
`is less than`
`is not greater than or equal [to]`
`isn't greater than or equal [to]`
`less than`
	date `<` *date*
	integer `<` *integer*
	real `<` *real*
	string `<` *string*

`>`
`comes after`
`greater than`
`is greater than`
`is not less than or equal [to]`
`isn't less than or equal [to]`
	date `>` *date*
	integer `>` *integer*
	real `>` *real*
	string `>` *string*

continued

Table A-4 Operators (continued)

Operator	Syntax

≤
<=
`does not come after`
`doesn't come after`
`is less than or equal [to]`
`is not greater than`
`isn't greater than`
`less than or equal [to]`
	date ≤ *date*
	integer ≤ *integer*
	real ≤ *real*
	string ≤ *string*

≥
>=
`does not come before`
`doesn't come before`
`greater than or equal [to]`
`is greater than or equal [to]`
`is not less than`
`isn't less than`
	date ≥ *date*
	integer ≥ *integer*
	real ≥ *real*
	string ≥ *string*

Miscellaneous operators	
`&`	*expression* `&` *expression*
`as`	*expression* `as` *className*
`a reference to`	`[a] ( ref [to] | reference to )` ¬ *reference*

Control Statements

Control statements are statements that control when and how other statements are executed. Table A-5 summarizes the control statements in the AppleScript English dialect. For more information about control statements, see Chapter 7, “Control Statements.”

Table A-5 Control statements

Control statement	Syntax

`tell`	`tell` *referenceToObject* `to` *statement*
	
	`tell` *referenceToObject*
	`[` *statement* `]...`
	`end [ tell ]`

`if`	`if` *Boolean* `then` *statement*
	
	`if` *Boolean* `[ then ]`
	`[` *statement* `]...`
	`[ else if` *Boolean* `[ then ]`
	`[` *statement* `]...]...`
	`[ else`
	`[` *statement* `]...]`
	`end [ if ]`

`repeat`	`repeat`
	`[` *statement* `]...`
	`end [ repeat ]`
	
	`repeat` *integer* `[ times ]`
	`[` *statement* `]...`
	`end [ repeat ]`
	
	`repeat while` *Boolean*
	`[` *statement* `]...`
	`end [ repeat ]`
	
	`repeat until` *Boolean*
	`[` *statement* `]...`
	`end [ repeat ]`
	
	`repeat with` *variable* `from` *integer* `to` *integer* `[ by` *integer* `]`
	`[` *statement* `]...`
	`end [ repeat ]`
	
	`repeat with` *variable* `in` *list*
	`[` *statement* `]...`
	`end [ repeat ]`

`exit`	`exit`

continued

Table A-5 Control statements (continued)

Control statement	Syntax

`try`	`try`
	`[ statement ]...`
	`on error` ¬
	`[ errorMessageVariable]` ¬
	`[ number errorNumberVariable]` ¬
	`[ from offendingObjectVariable ]` ¬
	`[ partial result resultListVariable]` ¬
	`[ to expectedTypeVariable ]`
	`[ global variable [, variable]...]`
	`[ local variable [, variable]...]`
	`[ statement ]...`
	`end [ error | try ]`

`considering`	`considering attribute [, attribute ... and attribute ]` ¬
	`[ but ignoring attribute [, attribute ... and attribute ] ]`
	`[ statement ]...`
	`end considering`

`ignoring`	`ignoring attribute [, attribute ... and attribute ]` ¬
	`[ but considering attribute [, attribute ... and attribute ] ]`
	`[ statement ]...`
	`end ignoring`

`with timeout`	`with timeout [ of ] integer second[s]`
	`[ statement ]...`
	`end [ timeout ]`

`with transaction`	`with transaction [ session ]`
	`[ statement ]...`
	`end [ transaction ]`

Handlers

Handlers are collections of statements that are executed in response to commands or error messages. Table A-6 summarizes handler definitions and subroutine calls.

Table A-6 Handler definitions and calls

Handler	Syntax

Subroutine definition (labeled parameters)	`( on | to ) subroutineName`
	`  [ of | in directParameterVariable ]`
	`  [ subroutineParamLabel paramVariable ] ...`
	`  [ given label:paramVariable [, label:paramVariable ]...]`
	`[ global variable [, variable ]...]`
	`[ local variable [, variable ]...]`
	`[ statement ]...`
	`end [ subroutineName ]`

Subroutine call (labeled parameters)	`subroutineName`
	`[ ( of | in ) directParameter ]`
	`[ subroutineParamLabel parameterValue ]`
	`  | [ with labelForTrueParam [, labelForTrueParam ]...`
	`  [( and | or | , ) labelForTrueParam ] ]`
	`  | [ without labelForFalseParam [, labelForFalseParam ]...`
	`  [( and | or | , ) labelForFalseParam ] ]`
	`  | [ given label:parameterValue`
	`  [, label:parameterValue ]...]] ...`

Subroutine definition (positional parameters)	`( on | to ) subroutineName ( [ paramVariable [, paramVariable ]...] )`
	`[ global variable [, variable ]...]`
	`[ local variable [, variable ]...]`
	`[ statement ]...`
	`end [ subroutineName ]`

Subroutine call (positional parameters)	`subroutineName ( [ parameterValue [, parameterValue ]...] )`
Return statement	`return expression`

Command handler definition	`( on | to ) commandName`
	`[ [ of ] directParameterVariable ]`
	`[ [ given ] label:paramVariable [, label:paramVariable ]...]`
	`[ global variable [, variable ]...]`
	`[ local variable [, variable ]...]`
	`[ statement ]...`
	`end [ commandName ]`

Script Objects

Script objects are user-defined objects. Table A-7 summarizes the syntax for defining script objects in AppleScript. For more information about script objects, see Chapter 9, "Script Objects."

Table A-7 Script objects

Script object element	Syntax

Script object definition	`script [`*scriptObjectVariable*`]`
	`  [(property | prop)` *propertyLabel* `:` *expression*`]...`
	`  [` *handlerDefinition* `]...`
	`  [` *statement* `]...`
	`end [ script ]`

Continue statement (to pass a command to a handler in the parent script object)	`continue` *commandStatement*

Variable and Property Assignments and Declarations

Table A-8 summarizes the syntax for assigning values to variables and script properties and declaring local and global variables. For information about variables and script properties, see Chapter 3, "Values." For detailed information about the scope of script variables and properties, see "Scope of Script Variables and Properties," which begins on page 252.

Table A-8 Assignments and declarations

Assignment or declaration	Syntax
Variable assignment (and declaration if variable has not previously been declared)	`copy` *expression* `to` *variable* `copy` *reference* `to` *variable* `set` *variable* `to` *expression* `set` *variable* `to` *reference*
Global variable declaration	`global` *variable* `[,` *variable* `]...`
Local variable declaration	`local` *variable* `[,` *variable* `]...`
Script property declaration and assignment	`property` *propertyLabel* `:` *expression* `prop` *propertyLabel* `:` *expression*

The Text Item Delimiters property, which is the only property you can get and set using the global variable `AppleScript`, consists of a list of the delimiters used by AppleScript when coercing lists to strings and when getting text items from strings. This property is declared by AppleScript and is available from any script. You can get and set it using this syntax:

```
AppleScript's text item delimiters
text item delimiters of AppleScript
```

Currently, only the first delimiter in the list is used by AppleScript.

Predefined Variables

Table A-9 lists special variables that are defined by AppleScript. These variables are global, that is, they are available anywhere in a script.

As with all other identifiers, predefined variables are not case sensitive. For example, `result`, `Result`, and `RESULT` are all treated as the same variable.

Table A-9 Predefined variables

Identifier	Class	Description
`it`	Reference	The default target. For more information, see Chapter 7, "Control Statements."
`me`	Reference	The current script (used within Tell statements to refer to handlers or properties of the current script). For more information, see Chapter 7, "Control Statements," and Chapter 8, "Handlers."
`pi`	Real	The value π (roughly 3.14159).
`result`	Any class	The result returned by the most recently executed command or the most recently evaluated expression. If the most recently executed command did not return a result, then the value of `result` is undefined.
`return`	String	A return character.
`space`	String	A space character.
`tab`	String	A tab character.

Constants

Table A-10 lists constants defined by AppleScript.

As with all other identifiers, constants are not case sensitive. For example, `false`, `False`, and `FALSE` are all treated as the same constant.

Table A-10 Constants defined by AppleScript

Identifier	Meaning
Attributes specified in Considering and Ignoring statements	
`application responses`	If ignored, AppleScript doesn't wait for responses from application commands before proceeding to the next statement in a script and ignores any results or errors returned.

continued

Table A-10 Constants defined by AppleScript (continued)

Identifier	Meaning
`case`	If considered, AppleScript distinguishes uppercase letters from lowercase.
`diacriticals`	If ignored, AppleScript ignores diacritical marks in string comparisons.
`expansion`	If ignored, AppleScript treats the characters æ, Æ, œ, and Œ as single characters and thus not equal to the character pairs ae, AE, oe, and OE.
`hyphens`	If ignored, AppleScript ignores hyphens in string comparisons
`punctuation`	If ignored, AppleScript ignores punctuation marks in string comparisons.
`white space`	If ignored, AppleScript ignores spaces, tab characters, and return characters in string comparisons.
Text styles	
`all caps`	All caps
`all lowercase`	All lowercase
`bold`	Boldface
`condensed`	Condensed
`expanded`	Expanded
`hidden`	Hidden
`italic`	Italic
`outline`	Outline
`plain`	Plain text
`shadow`	Shadow
`small caps`	Small caps
`strikethrough`	Strikethrough
`subscript`	Subscript

continued

Table A-10 Constants defined by AppleScript (continued)

Identifier	Meaning
`superscript`	Superscript
`underline`	Underline
Save options	
`ask`	Ask user whether to save modified object or objects.
`no`	Don't save modified object or objects.
`yes`	Save modified object or objects.
Alignment	
`center`	Centered
`full`	Justified
`left`	Flush left
`right`	Flush right
Boolean constants	
`false`	The Boolean `false` value.
`true`	The Boolean `true` value.
Miscellaneous	
`current application`	Either the default target application or whatever application is currently set as a script's parent property.

Placeholders

Table A-11 explains the placeholders used in the syntax descriptions in this appendix.

Table A-11 Placeholders used in syntax descriptions

Placeholder	Explanation
applicationName	A string containing the name of the application as it would be listed in the Application menu, or a string of the form "*Disk*:*Folder1*:*Folder2*:...:*ApplicationName*" that specifies where the application is stored. For more information, see page 146.
attribute	An attribute, identified by a constant, that can be considered or ignored in a Considering or Ignoring control statement. The constants for attributes are `case`, `white space`, `diacriticals`, `hyphens`, `expansion`, `punctuation`, and `application responses`.
Boolean	An expression that evaluates to `true` or `false`. (Boolean is an AppleScript value class. For more information about Boolean values, see page 41.)
className	A class identifier or an expression that evaluates to an object class identifier.
commandName	An identifier (name) for a command.
commandStatement	A statement, consisting of a command with either parameter values or formal parameters, to be passed to a parent script object.
compoundValue	An expression that evaluates to a compound value (a list, record, or string).
containerReference	A reference that specifies a container for another object.
dataValue	An expression that evaluates to a value of the appropriate class for the object being created.
date	An expression that evaluates to a date. (Date is an AppleScript value class. For more information about dates, see page 44.)
directParameter	The direct parameter of a subroutine definition.
directParameterVariable	A parameter variable used as a placeholder for the value of the direct parameter in a subroutine definition.
errorMessage	An expression, usually a string, that describes an error.
errorMessageVariable	A parameter variable for the expression that describes the error.
errorNumber	The error number for the error.

continued

Table A-11 Placeholders used in syntax descriptions (continued)

Placeholder	Explanation
errorNumberVariable	A parameter variable for the error number.
expectedType	A class identifier for the value class to which AppleScript was attempting to coerce a value when an error occurred.
expectedTypeVariable	A parameter variable for the value class to which AppleScript was attempting to coerce a value when an error occurred.
expression	A series of AppleScript words whose value is a Boolean, class identifier, constant, data, date, integer, list, real, record, reference, or string.
handlerDefinition	A command or subroutine handler definition.
IDvalue	An expression that evaluates to an object's ID property. For most objects, the ID property is an integer.
integer	An expression that evaluates to an integer. (Integer is an AppleScript value class. For more information about integers, see page 48.)
label	An identifier for a parameter.
labelForFalseParam	An identifier for a Boolean parameter whose value is `false`.
labelForTrueParam	An identifier for a Boolean parameter whose value is `true`.
list	An expression that evaluates to a list.
listOfFiles	A list of references, each of which has the form `file "`*Disk*`:`*Folder1*`:`*Folder2*`:...:`*Filename*`"` or `alias "`*Disk*`:`*Folder1*`:`*Folder2*`:...:`*Filename*`"` and specifies a file. For more information , see page 144.
nameString	A string of the form `"`*Disk*`:`*Folder1*`:`*Folder2*`:...:`*FileName*`"` that specifies where a file is stored. For more information, see page 144.
number	An expression that evaluates to an integer or real number.
offendingObject	A reference to an object that caused an error.
offendingObjectVariable	A parameter variable for the reference to the object that caused an error.
parameterValue	An expression that evaluates to a value of a parameter.

continued

Table A-11 Placeholders used in syntax descriptions (continued)

Placeholder	Explanation
paramVariable	A parameter variable (also known as a formal parameter) used as a placeholder for the value of a parameter in a handler definition.
pluralClassName	A plural class identifier or an expression that evaluates to a plural class identifier.
propertyLabel	The identifier for a property of an object, or an expression that evaluates to the identifier for a property of an object.
propertyValue	An expression that evaluates to a value of the appropriate class for the property being defined.
real	An expression that evaluates to a real number. (Real is an AppleScript value class. For more information about real numbers, see page 54.)
record	An expression that evaluates to a record. (Record is an AppleScript value class. For more information about records, see page 55.)
reference	A reference that specifies an object or location. (For more information about references, see Chapter 5, "Objects and References.")
referencePattern	A reference, a list of reference patterns, or a record of reference patterns.
referenceToApplication	A reference of the form `application "`*Disk*`:`*Folder1*`:`*Folder2*`:...:`*ApplicationName*`"` that specifies an application. For more information, see page 146.
referenceToFile	A reference of the form `file "`*Disk*`:`*Folder1*`:`*Folder2*`:...:`*Filename*`"` or `alias "`*Disk*`:`*Folder1*`:`*Folder2*`:...:`*Filename*`"` that specifies a file. For more information , see page 144.
referenceToLocation	A reference that specifies a location. (For more information about locations, see "Parameters That Specify Locations" on page 80.)
referenceToObject	A reference that specifies an object or objects. (For more information about references, see Chapter 5, "Objects and References.")
resultList	List of results for objects that were handled before an error occurred.

continued

Table A-11 Placeholders used in syntax descriptions (continued)

Placeholder	Explanation
resultListVariable	A parameter variable for a list of results for objects that were handled before an error occurred.
saveOption	A constant (`yes`, `no`, or `ask`) that specifies whether to save an object that has been modified before closing it.
scriptObjectVariable	A variable whose value is a script object. (For more information about script objects, see Chapter 9, "Script Objects.")
session	An object that specifies a specific session.
statement	An AppleScript statement.
string	An expression that evaluates to a string. (String is an AppleScript value class. For more information about strings, see page 61.)
subroutineName	An identifier (name) for a subroutine.
subroutineParamLabel	Any of the following labels: `above`, `against`, `apart from`, `around`, `aside from`, `at`, `below`, `beneath`, `beside`, `between`, `by`, `for`, `from`, `instead of`, `into`, `on`, `onto`, `out of`, `over`, `thru` (or `through`), `under`.
timeDifference	An integer specifying a time difference in seconds.
variable	A variable (a user-defined identifier that represents a value).
variablePattern	A variable, a list of variable patterns, or a record of variable patterns.

Scriptable Text Editor Dictionary

This appendix defines the AppleScript terms understood by the Scriptable Text Editor, the application used in examples throughout this book. These include both the names of application objects in the Scriptable Text Editor or its documents and the names of commands that specify actions that the Scriptable Text Editor performs. The appendix contains these sections:

- About Text Objects
- Scriptable Text Editor Object Class Definitions
- Scriptable Text Editor Commands
- Scriptable Text Editor Errors

For general information about definitions provided by scriptable applications, see "Commands and Objects" on page 17. For definitions of the terms understood by other scriptable applications, see the documentation for those applications.

About Text Objects

The Scriptable Text Editor deals mainly with text; therefore, most of its objects are text objects. The Scriptable Text Editor defines five text object classes:

- Character
- Word
- Paragraph
- Text
- Text Item

The following sections describe characteristics that these objects have in common. To avoid ambiguity, objects that belong to the class Text are always referred to as "objects of class Text." Objects that belong to the character, word, paragraph, text, or text item classes are called "text objects."

Elements of Text Objects

Each of the text object classes can contain any of the other text object classes as elements. For example, a word can also be a paragraph. A word object can contain character, word, and text item elements.

Classes of text objects are distinguished from each other by the way their boundaries, or delimiters, are defined. Because of this, the same text can be viewed as a collection of character elements, word elements, paragraph elements, or text item elements.

In addition to paragraph, word, character, and text item elements, all text objects can contain elements that belong to the class Text. An object belonging to class text is a series of contiguous characters. In any text object, the elements of class text are all of the possible series of contiguous characters contained within the object. The most common way to use an element of class Text is to specify the contiguous characters in a range of text objects.

For example, the following reference specifies all the characters, including spaces, from the beginning of the fifth word to the end of the twenty-fifth word.

```
text from word 5 to word 25
```

In contrast, the following reference specifies a list of words, with no information about spaces or punctuation between words.

```
word 5 thru 25
```

Special Properties of Scriptable Text Editor Text Objects

Like most scriptable applications, the Scriptable Text Editor supports standard groups, or **suites,** of AppleScript objects and commands. Different applications of the same type have many of the same objects and commands in their dictionaries. For example, most text-processing applications have paragraph objects, word objects, character objects, text item objects, and objects of class text. For the most part, these objects have the same properties in different applications.

Sometimes scriptable applications add properties to standard objects that are not included in the standard suites. In the Scriptable Text Editor, each of the

text object classes—paragraph, word, character, text item, and text—includes the following special properties:

- The Length property specifies the number of characters in the object.
- The Offset property specifies the offset, in characters, from the beginning of a Scriptable Text Editor document to the first character of the text object. For example, if a document begins with the words “We the people”, the offset of the word “We” is 1 and the offset of the word “people” is 8.

Text Styles

The text styles of Scriptable Text Editor text objects are defined by two properties: Style and Uniform Styles:

- The Style property specifies the styles of the text object. If the styles vary within the object, the Style property specifies the style of the first character of the object.
- The Uniform Styles property specifies the styles that are the same for all the characters in the text object.

The value of a Style or Uniform Styles property is a record with two properties: On Styles and Off Styles. The On Styles property specifies styles that are “on,” that is, that apply to the text object. The Off Styles property specifies styles that are “off,” that is, that do not apply to the text object. Both On Styles and Off Styles are lists of constants that specify text styles. For the Scriptable Text Editor, these constants are `bold`, `italic`, `outline`, `underline`, and `shadow`.

The following example shows the value of a Style property for a text object whose characters are bold and underlined:

```
{On Styles:{bold, underline}, Off Styles:{italic, outline, shadow}}
```

The On Styles property lists the two styles (`bold` and `underline`) that apply to the text object, while the Off Styles property lists the styles that do not apply to the text object. Together, the On Styles and Off Styles properties of a Style property include all of the text styles that the Scriptable Text Editor supports.

The Uniform Styles property specifies the styles that are uniform throughout a text object, that is, that are the same for every character in a text object. The On Styles property specifies the styles that apply to every character in the object. The Off Styles property specifies styles that do not apply to any character in the

text object. If any of the style constants (`bold`, `italic`, `outline`, `underline`, and `shadow`) do not appear in either field, it is because the styles apply to some, but not all, of the characters in the object.

The following example shows the value of a Uniform Styles property for a text object in which every character is bold and no characters are outlined or shadowed:

```
{On Styles:{bold}, Off Styles:{outline, shadow}}
```

The fact that the `italic` and `underline` constants do not appear in either field means that at least one, but not all, of the characters in the object are italicized and underlined. To find out which characters are italicized or underlined, you must examine the Style properties of each character in the text object.

To set the styles of a text object, use a Copy or Set command to set the Style property of the text object to one of the following:

- a two-part record like the one described earlier for the value of a Style property
- a one-part record that specifies either On Styles or Off Styles
- a list specifying the On Styles
- a constant specifying a single style

For example, here are four ways to use the Set command to set text styles:

```
set the style of word 1 to {On Styles:{italic}, ¬
   Off Styles:{bold, shadow}}

set the style of word 1 to {On Styles:{italic, bold}}

set the style of word 1 to {bold, italic}

set the style of word 1 to italic
```

- In the first example, the styles specified in the On Styles property are added to the active styles of word 1. The styles specified in the Off Styles property are removed from the active styles of word 1. Styles that are not specified in either property remain the same. For example, if word 1 is originally

underlined, shadowed, and bold, then after the Set command, word 1 is underlined and italicized.

- In the second example, the styles specified in the On Styles property are added to the active styles of word 1. Styles that are not specified remain the same. For example, if word 1 is originally underlined and shadowed, then after the Set command, word 1 is italicized, underlined, shadowed, and bold.
- In the third example, the styles specified in the list are added to the active styles of word 1. Styles that are not specified remain the same. For example, if word 1 is originally underlined and shadowed, then after the Set command, word 1 is italicized, underlined, shadowed, and bold.
- In the fourth example, the style specified in the command is added to the active styles of word 1. All other styles remain the same. For example, if word 1 is originally underlined, shadowed, and bold, then after the Set command, word 1 is italicized, underlined, shadowed, and bold.

When you set the style of a text object, if you include the same style constant in both the On Styles property and the Off Styles property, the Scriptable Text Editor returns the error `Bad data`.

When setting text styles, you can use a special constant—`plain`—to specify that the text object is to be plain, that is, have no text styles. If you include constants other than `plain` in the On Styles property, the Scriptable Text Editor ignores the other constants. If you include `plain` in the Off Styles property, the Scriptable Text Editor returns the error `Bad data`.

AppleScript and Non-Roman Script Systems

A **script system** is a collection of system software facilities that allow for the visual representation of a particular writing system. Script systems include Roman, Japanese, Hebrew, Greek, and Thai. Each script system has a corresponding **script code,** a constant used, for example, to identify the script system in which some text was prepared. AppleScript and the Scriptable Text Editor can handle text prepared in a variety of script systems, provided the appropriate software is installed on your computer.

A character in an AppleScript string or a Scriptable Text Editor document takes up either 1 byte or 2 bytes, depending on the script system the character belongs to. Thus, the size in bytes of a text object may differ from the number of characters it contains.

AppleScript honors the script codes of characters as much as possible when comparing them. This means that characters from different script systems may represent different characters and may be of unequal length even though their character codes are the same.

Scriptable Text Editor Object Class Definitions

This section defines the application object classes to which Scriptable Text Editor objects belong. For an introduction to application objects and references, see Chapter 5, "Objects and References."

Application

The Application object class defines the characteristics of the Scriptable Text Editor application.

PROPERTIES

`Clipboard`	Information stored in the Clipboard. The Clipboard is an area in the computer's memory that functions as a holding place for information that is cut or copied. The Scriptable Text Editor uses the Clipboard that is shared by all applications. *Class:* List of data objects (see "Notes" later in this section) *Modifiable?* Yes
`Frontmost`	A Boolean parameter that indicates if the application is the active application. If the value is `true`, the Scriptable Text Editor is the active application. If the value is `false`, the Scriptable Text Editor is not the active application. *Class:* Boolean *Modifiable?* No
`Name`	The name of the application. *Class:* String *Modifiable?* No

`Selection` The current selection. The object or objects in the selection are the ones that would be cut by a Cut command or copied by a Copy command. If no objects are selected, the value of this property is a reference to an insertion point object.
Class: Reference
Modifiable? Yes

`Text Item Delimiters`
The characters that are used to separate text item objects. By default, the value of this property is a single comma, {","} (see "Notes").
Class: List of text objects
Modifiable? Yes

`Version` The version of the Scriptable Text Editor.
Class: String
Modifiable? No

ELEMENT CLASSES

document — Scriptable Text Editor documents that are currently open (see "Notes").

window — Scriptable Text Editor windows that are currently open (see "Notes").

COMMANDS HANDLED

Open, Print, Quit, Run

DEFAULT VALUE CLASS RETURNED

None

EXAMPLES

```
tell application "Scriptable Text Editor" to quit

cut the selection of application "Scriptable Text Editor"
```

NOTES

The AppleScript English language dialect supports the abbreviation `app` for referring to objects of class application. For example, the following statements are equivalent:

```
cut the selection of application "Scriptable Text Editor"

cut the selection of app "Scriptable Text Editor"
```

The window and document element classes can be used interchangeably. This is because each open document has a window, whose elements and properties are always the same as those of its document. For example, if the first open document is named Giant, the first window is also named Giant. A Scriptable Text Editor document and its window contain the same text elements, so references to text elements in a document and its window can be used interchangeably. For example, `word 1 of document "Giant"` and `word 1 of window "Giant"` refer to the same word.

Text item objects are series of contiguous characters that are separated from each other by special characters called delimiters. The Text Item Delimiters property of the Scriptable Text Editor application consists of a list of characters that can be used to separate text item objects. At startup the default value of the Scriptable Text Editor's Text Item Delimiters property is a single-item list that contains a comma: {","}. You can set this delimiter to a different value or add additional delimiters, but the new values are lost when the user quits the application. You may find it convenient to alter the Text Item Delimiters property temporarily if you are dealing with multilingual text or text formatted with other delimiters.

The Clipboard property is a list of objects of class Data, but you can use the As operator to get the Clipboard as a string, reference, or styled text. The Scriptable Text Editor coerces the data to the requested type, if possible.

```
tell application "Scriptable Text Editor"
   cut word 1 of front document
   get clipboard as string
end tell
--result: the first word of the document as a string
```

Character

An object of class Character is a text character.

PROPERTIES

`Font` The name of the font of the character or characters. If the font varies within a range of characters, the Font property specifies the font of the first character.
Class: String
Modifiable? Yes

`Length` The number of characters in a text object.
Class: Integer
Modifiable? No

`Offset` The offset, in characters, from the beginning of the document to the first character of a text object.
Class: Integer
Modifiable? No

`Size` The size, in points, of the character or characters. If the size varies within a range of characters, the Size property specifies the size of the first character.
Class: Integer
Modifiable? Yes

`Style` The text styles of the character or characters. If the text styles vary within a range of characters, the Style property specifies the styles of the first character.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to the text object. The Off Styles property specifies the styles that do not apply to the text object. Both properties are lists of constants that specify text styles. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Style property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? Yes

`Uniform Styles`

The text styles that are uniformly "on" or "off" for all characters of a text object.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to all the characters of the text object. The Off Styles property specifies the styles that do not apply to any of the characters of the text object. Both properties are lists of constants that specify text styles. If a style does not appear in either list, the style applies to some but not all of the characters in the object. See "Elements of Text Objects" on page 314 for more information. The constants that can be used in Uniform Styles property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? No

ELEMENT CLASSES

See "Elements of Text Objects" on page 314 for a general discussion of these element classes.

Character	Characters contained in the characters
Paragraph	Paragraphs contained in the characters
Text	Series of characters contained in the characters
Text Item	Text items contained in the characters
Word	Words contained in the characters

COMMANDS HANDLED

Copy, Count, Cut, Data Size, Delete, Duplicate, Exists, Get, Make, Move, Select, Set

DEFAULT VALUE CLASS RETURNED

Styled Text

EXAMPLES

```
delete the last character of word 5

tell document "Intro" of app "Scriptable Text Editor"
   move (text from character 1 to character 21) to end
end tell
```

Document/Window

A document object is an open Scriptable Text Editor document. The window and document objects have the same elements and properties. They can be used interchangeably.

PROPERTIES

`Bounds` The rectangle that bounds the content region of the window (the portion of the window that contains the text of the document; the "window frame"—the title bar and scroll bars—are not part of the content region).
Class: List of four integers. The first two integers specify the coordinates of the upper-left corner of the window, and the last two integers specify the coordinates of the lower-right corner of the window. (For information about window coordinates, see "Notes" later in this section.)
Modifiable? Yes

`Closable` A Boolean parameter that indicates whether the window has a close box. The value `true` specifies that the window has a close box, and the value `false` indicates that it doesn't. All Scriptable Text Editor windows have close boxes.
Class: Boolean
Modifiable? No

`Contents` All the text contained in the window.
Class: Text
Modifiable? Yes

`Floating` A Boolean parameter that indicates whether the window is a floating window (a window that appears in front of all other windows). The value `true` indicates that the window is a floating window, and the value `false` indicates that it isn't. No Scriptable Text Editor windows are floating windows.
Class: Boolean
Modifiable? No

`Index` The number of the window (`window 1` is the frontmost window, `window 2` is the window immediately behind `window 1`, and so on).
Class: Integer
Modifiable? Yes

`Modal` A Boolean parameter that indicates whether the window is modal (one that requires a response from the user before the user can perform any other tasks). The value `true` indicates that the window is modal, and the value `false` indicates that it isn't. No Scriptable Text Editor windows are modal.
Class: Boolean
Modifiable? No

`Modified` A Boolean parameter that indicates whether the document has been modified since it was last saved. The value `true` indicates that the document has been modified, and the value `false` indicates that it hasn't.
Class: Boolean
Modifiable? No

`Name` The name of the window (see "Notes" later in this section).
Class: Text
Modifiable? Yes

`Position` The upper-left corner of the content region of the window (the portion of the window that contains the text of the document; the "window frame"—the title bar and scroll bars—are not part of the content region).
Class: List of two integers that specify the coordinates of the upper-left corner (for information about window coordinates, see "Notes" later in this section).
Modifiable? Yes

`Resizable` A Boolean parameter that indicates whether the window can be resized. The value `true` indicates that the window can be resized, and the value `false` indicates that it can't. All of the Scriptable Text Editor's windows can be resized.
Class: Boolean
Modifiable? No

`Selection` The text selected in the window.
Class: Selection
Modifiable? Yes

`Titled` A Boolean parameter that indicates whether the window has a title bar. The value `true` indicates that the window has a title bar, and the value `false` indicates that it doesn't. All Scriptable Text Editor windows have title bars.
Class: Boolean
Modifiable? No

`Visible` A Boolean parameter that indicates whether the window is visible. The value `true` indicates that the window is visible, and the value `false` indicates that it isn't.
Class: Boolean
Modifiable? No

`Zoomable` A Boolean parameter that indicates whether the window can be zoomed. The value `true` indicates that the window can be zoomed, and the value `false` indicates that it can't. All of the Scriptable Text Editor's windows can be zoomed.
Class: Boolean
Modifiable? No

`Zoomed` A Boolean parameter that specifies whether the window is full size or not. The value `true` specifies that the window is full size, and the value `false` specifies that it is not.
Class: Boolean
Modifiable? Yes

ELEMENT CLASSES

See "Elements of Text Objects" on page 314 for a general discussion of these element classes.

Character Characters contained in the document

Paragraph Paragraphs contained in the document

Text	Series of characters contained in the document
Text Item	Text items contained in the document (see "Elements of Text Objects" on page 314)
Word	Words contained in the document

COMMANDS HANDLED

Close, Copy, Count, Delete, Duplicate, Exists, Get, Make, Move, Print, Revert, Save, Select, Set

DEFAULT VALUE CLASS RETURNED

None; use Contents property to get the data of a document.

EXAMPLE

```
tell document "Colossal" of app "Scriptable Text Editor"
    delete paragraphs 1 thru 10
end tell
```

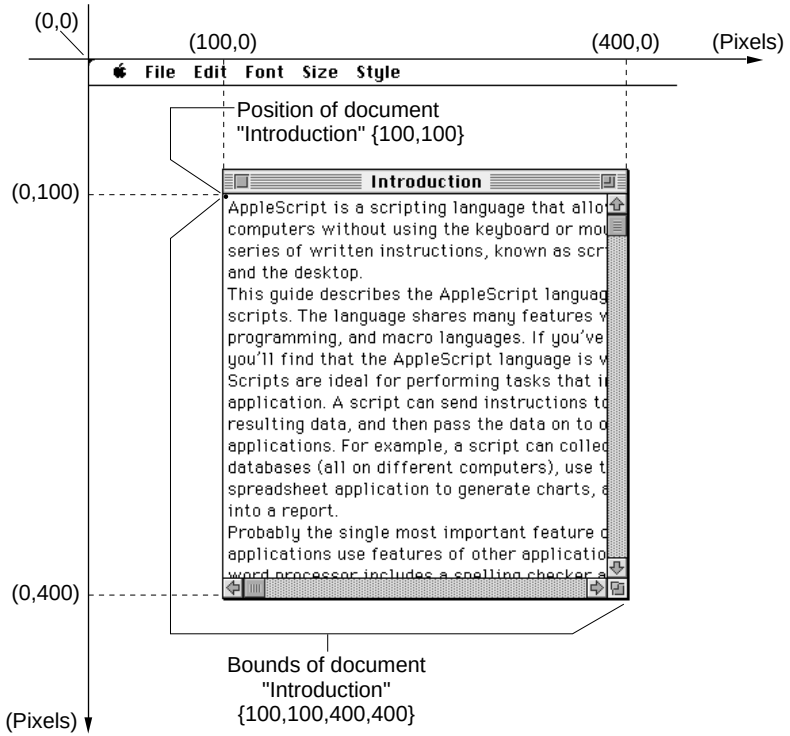
NOTES

The window and document object classes can be used interchangeably. This is because each open document has a window, whose elements and properties are always the same as those of its document. For example, if the first open document is named Giant, the first window is also named Giant. A Scriptable Text Editor document and its window contain the same text elements, so references to text elements in a document and its window can be used interchangeably. For example, `word 1 of document "Giant"` and `word 1 of window "Giant"` refer to the same word.

Both the Bounds and Position properties are specified in terms of points on the display. The Position property contains a single point: the upper-left corner of the window's content region (the portion of the window within the "window frame" made up of the title bar and scroll bars). The Bounds property contains two points: the upper-left corner of the content region followed by the lower-right corner. In AppleScript, points are specified by pairs of integers known as coordinates. The first coordinate in the pair, known as the x-coordinate, specifies the distance from the left edge of the display to the

point. The other coordinate, known as the y-coordinate, specifies the distance from the top of the display to the point. (These distances are measured in pixels, which are the picture elements that make up the display.) Figure B-1 illustrates the Bounds and Position properties of a sample window.

Figure B-1 Bounds and Position properties of a Scriptable Text Editor window



You specify points in AppleScript with lists. Figure B-1 shows that the point for the Position property is specified by the list `{100, 100}`. The first item in this list is the x-coordinate value of the point, and the second is the y-coordinate value. The figure also shows that the Bounds property is specified by the list `{100, 100, 400, 400}`. You specify the two points that define a rectangle in a single list. In this list, `{100, 100}` are the coordinates of the upper-left corner, and `{400, 400}` are the coordinates of the lower-right corner.

Although the object class defined for the Bounds and Position properties is List, the Scriptable Text Editor actually stores Bounds and Position property data in a different form. To get Bounds or Position data as a List, you must copy the data to AppleScript. For example,

```
copy (Bounds of front window) to x
get item 2 of x
```

returns the y-coordinate of the upper-left corner of the window, but

```
get item 2 of (Bounds of front window)
```

returns an error.

The Name property of a document is an object of class Text. You can change the characters of the name of a document, but unlike other objects of class Text, you cannot change its Font, Size, or Style properties.

File

An object of class File contains the data for a Scriptable Text Editor document. When a file object is opened, a corresponding document object is created. (Only open documents are Scriptable Text Editor document objects.) When a document object is saved, the data for the document is saved in the corresponding file object, which is represented by a Scriptable Text Editor document icon on the desktop.

PROPERTIES

`Name` The name of the file.
Class: String of the form "*Disk*:*Folder1*:*Folder2*:...:*Filename*" (see "Notes")
Modifiable? No

ELEMENT CLASSES

None

COMMANDS HANDLED

Open, Print

DEFAULT VALUE CLASS RETURNED

None

EXAMPLE

```
tell application "Scriptable Text Editor"
   print file "MyDisk:Status Reports:Status - 11/12/92"
end tell
```

NOTES

To specify the name of a file, use a string of the form "*Disk*:*Folder1*:*Folder2*:...:*Filename*"; for details, see "References to Files," which begins on page 144. You can also specify a string with only a filename ("*Filename*"). In this case, AppleScript attempts to find the file in the current directory.

Insertion Point

An object of class Insertion Point is a location between characters where text can be inserted. The first insertion point in a container is the place between the beginning of the container and the first character. In the Scriptable Text Editor, insertion points are zero-length text objects. As a result, they have the same properties and elements as text objects.

PROPERTIES

`Font` The name of the font of the insertion point.
Class: String
Modifiable? Yes

`Length` The number of characters in a text object. For insertion points, the value of the Length property is always 0.
Class: Integer
Modifiable? No

`Offset` The number of possible insertion points from the beginning of the document to the insertion point. The first insertion point of a document is the insertion point before the first character.
Class: Integer
Modifiable? No

`Size` The point size that is active at an insertion point.
Class: Integer
Modifiable? Yes

`Style` The text styles that are active at an insertion point.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that are active. The Off Styles property specifies the styles that are not active. Both properties are lists of constants that specify text styles. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Style property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? Yes

`Uniform Styles`
The text styles that are uniformly "on" or "off" for all characters of a text object. For insertion points, the Uniform Styles property indicates which text styles are active at the insertion point. It always has the same value as the Style property.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Uniform Styles property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? No

ELEMENT CLASSES

None

COMMANDS HANDLED

Copy, Count, Data Size, Exists, Get, Set

DEFAULT VALUE CLASS RETURNED

Reference

EXAMPLES

The following statement moves the first word of a document to the insertion point after the tenth word.

```
tell document "Intro" of app "Scriptable Text Editor"
   move word 1 to insertion point after word 10
end tell
```

The following statement does the same thing:

```
tell document "Intro" of app "Scriptable Text Editor"
   move word 1 to after word 10
end tell
```

AppleScript allows you to leave out the words `insertion point` when specifying locations with the Relative reference form. For more information about the Relative reference form, see “Relative” on page 139.

Paragraph

An object of class Paragraph is a text object that’s delimited by return characters or by the beginning or end of the container.

PROPERTIES

`Font` The name of the font of the characters of the paragraph. If the font varies within the paragraph, the Font property specifies the font of the first character.
Class: String
Modifiable? Yes

`Length` The number of characters in a text object.
Class: Integer
Modifiable? No

`Offset` The offset, in characters, from the beginning of the document to the first character of a text object.
Class: Integer
Modifiable? No

`Size` The size, in points, of the characters of the paragraph. If the size varies within the paragraph, the Size property specifies the size of the first character.
Class: Integer
Modifiable? Yes

`Style` The text styles of the characters of the paragraph. If the text styles vary within the paragraph, the Style property specifies the styles of the first character.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to the text object. The Off Styles property specifies the styles that do not apply to the text object. Both properties are lists of constants that specify text styles. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Style property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? Yes

`Uniform Styles`
The text styles that are uniformly "on" or "off" for all characters of a text object.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to all the characters of the text object. The Off Styles property specifies the styles that do not apply to any of

the characters of the text object. Both properties are lists of constants that specify text styles. If a style does not appear in either list, the style applies to some but not all of the characters in the object. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Uniform Styles property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? No

ELEMENT CLASSES

See "Elements of Text Objects" on page 314 for a general discussion of these element classes.

Character	Characters contained in the paragraphs
Paragraph	Paragraphs contained in the paragraphs
Text	Series of characters contained in the paragraphs
Text Item	Text items contained in the paragraphs
Word	Words contained in the paragraphs

COMMANDS HANDLED

Copy, Count, Cut, Data Size, Delete, Exists, Get, Make, Move, Select, Set

DEFAULT VALUE CLASS RETURNED

Styled Text

EXAMPLE

```
tell document "Memo" of app "Scriptable Text Editor"
   set style of (paragraphs whose first word = "Hello") to bold
end tell
```

Selection

An object of class Selection is text that is currently selected. A blinking insertion point is a zero-length selection.

PROPERTIES

`Contents` The text in the selection (because Scriptable Text Editor documents only contain text, a selection can only contain text).
Class: Text
Modifiable? Yes

`Font` The name of the font of the text object. If the font varies within the object, the Font property specifies the font of the first character.
Class: String
Modifiable? Yes

`Length` The number of characters in a text object.
Class: Integer
Modifiable? No

`Offset` The offset, in characters, from the beginning of the document to the first character of a text object.
Class: Integer
Modifiable? No

`Size` The size, in points, of the text object. If the size varies within the object, the Size property specifies the size of the first character.
Class: Integer
Modifiable? Yes

`Style` The text styles of a text object. If the text styles vary within the object, the Style property specifies the styles of the first character.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to the text object. The Off Styles property specifies the styles that do not apply to the text object. Both properties are lists of constants that specify text styles. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Style property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? Yes

`Uniform Styles`

The text styles that are uniformly "on" or "off" for all characters of a text object.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to all the characters of the text object. The Off Styles property specifies the styles that do not apply to any of the characters of the text object. Both properties are lists of constants that specify text styles. If a style does not appear in either list, the style applies to some but not all of the characters in the object. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Uniform Styles property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? No

ELEMENT CLASSES

See "Elements of Text Objects" on page 314 for a general discussion of these element classes.

Character	Characters contained in the selection
Paragraph	Paragraphs contained in the selection
Text	Series of characters contained in the selection
Text item	Text items contained in the selection
Word	Words contained in the selection

COMMANDS HANDLED

Copy, Count, Cut, Data Size, Delete, Duplicate, Exists, Get, Make, Move, Set

DEFAULT VALUE CLASS RETURNED

Reference

EXAMPLES

```
tell front document of application "Scriptable Text Editor"
   copy word 1
   set selection to end of paragraph 1
   paste
end tell

tell application "Scriptable Text Editor"
   set style of the selection of front document to bold
end tell
```

NOTES

In the Scriptable Text Editor, the only objects of class selection are the Selection properties of the application, document, and window objects.

Text

An object of class Text is a series of one or more contiguous characters.

PROPERTIES

`Font`	The name of the font of the text object. If the font varies within the object, the Font property specifies the font of the first character. *Class:* String *Modifiable?* Yes
`Length`	The number of characters in a text object. *Class:* Integer *Modifiable?* No
`Offset`	The offset, in characters, from the beginning of the document to the first character of a text object. *Class:* Integer *Modifiable?* No

`Size` The size, in points, of the text object. If the size varies within the object, the Size property specifies the size of the first character.
Class: Integer
Modifiable? Yes

`Style` The text styles of a text object. If the text styles vary within the object, the Style property specifies the styles of the first character.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to the text object. The Off Styles property specifies the styles that do not apply to the text object. Both properties are lists of constants that specify text styles. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Style property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? Yes

`Uniform Styles`
The text styles that are uniformly "on" or "off" for all characters of a text object.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to all the characters of the text object. The Off Styles property specifies the styles that do not apply to any of the characters of the text object. Both properties are lists of constants that specify text styles. If a style does not appear in either list, the style applies to some but not all of the characters in the object. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Uniform Styles property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? No

ELEMENT CLASSES

See "Elements of Text Objects" on page 314 for a general discussion of these element classes.

Character Characters contained in the text

Paragraph Paragraphs contained in the text

Text	Series of characters contained in the text
Text Item	Text items contained in the text
Word	Words contained in the text

COMMANDS HANDLED

Copy, Count, Cut, Data Size, Delete, Duplicate, Exists, Get, Make, Move, Select, Set

DEFAULT VALUE CLASS RETURNED

Styled Text

EXAMPLE

```
tell document "Memo" of app "Scriptable Text Editor"
   move (the text from paragraph 10 to paragraph 12) to end
end tell
```

NOTES

The Scriptable Text Editor does not allow you to refer to an object of class Text with the Index reference form (such as `text 5 of characters 1 thru 1024`) because it does not make sense to number the many possible objects of class Text in a text object.

Text is a plural class name. You cannot use it in places where AppleScript expects a singular class name. For example, the following statement results in a compilation error.

```
tell document "Intro" of app "Scriptable Text Editor"
   get every text from paragraph 1 to paragraph 2
end tell
--causes an error because text is a plural class name
```

Text Item

Objects of class Text Item are delimited by any of the characters specified by the Scriptable Text Editor's Text Item Delimiters property or by the beginning or end of a container. You can modify the Text Item Delimiters property, but such changes are lost when the user quits the application.

PROPERTIES

`Font` The name of the font of the text item. If the font varies within the text item, the Font property specifies the font of the first character.
Class: String
Modifiable? Yes

`Length` The number of characters in a text object.
Class: Integer
Modifiable? No

`Offset` The offset, in characters, from the beginning of the document to the first character of a text object.
Class: Integer
Modifiable? No

`Size` The size, in points, of the text item. If the size varies within the text item, the Size property specifies the size of the first character.
Class: Integer
Modifiable? Yes

`Style` The text styles of a text object. If the text styles vary within the text item, the Style property specifies the styles of the first character.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to the text object. The Off Styles property specifies the styles that do not apply to the text object. Both properties are lists of constants that specify text styles. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Style property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? Yes

`Uniform Styles`

The text styles that are uniformly "on" or "off" for all characters of a text object.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to all the characters of the text object. The Off Styles property specifies the styles that do not apply to any of the characters of the text object. Both properties are lists of constants that specify text styles. If a style does not appear in either list, the style applies to some but not all of the characters in the object. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Uniform Styles property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? No

ELEMENT CLASSES

See "Elements of Text Objects" on page 314 for a general discussion of these element classes.

Character	Characters contained in the text items
Paragraph	Paragraphs contained in the text items
Text	Series of characters contained in the text items
Text Item	Text items contained in the text items
Word	Words contained in the text items

COMMANDS HANDLED

Copy, Count, Cut, Data Size, Delete, Duplicate, Exists, Get, Make, Move, Select, Set

DEFAULT VALUE CLASS RETURNED

Styled Text

EXAMPLE

```
tell document "Report" of app "Scriptable Text Editor"
   copy text item 2 to x
end tell
```

In this example, if the document Report begins with “Tomato, Banana, Pineapple”, then the value of `x` is `"Banana"`.

NOTE

The characters used to separate text item objects are specified in the Scriptable Text Editor’s Text Item Delimiters property. At startup the default value of the Scriptable Text Editor’s Text Item Delimiters property is a single-item list that contains a comma: {",}. You can set this delimiter to a different value or add additional delimiters, but the new values are lost when the user quits the application.

Text Style Info

An object of class Text Style Info specifies the styles of a text object. The Style and Uniform Styles properties of text objects are text style info objects.

PROPERTIES

`On Styles` The styles that apply to the text object.
Class: List of constants (see “Notes” later in this section)
Modifiable? Yes

`Off Styles` The styles that do not apply to the text object.
Class: List of constants (see “Notes” later in this section)
Modifiable? Yes

ELEMENT CLASSES

None

COMMANDS HANDLED

Copy, Data Size, Exists, Get, Set

DEFAULT VALUE CLASS RETURNED

Record; text style info

EXAMPLES

```
set the style of word 1 to {On Styles:{italic}, ¬
   Off Styles:{bold, shadow}}

set the style of word 1 to {On Styles:{italic, bold}}

set the style of word 1 to {bold, italic}

get words whose style contains bold
```

NOTES

The constants that can be used in Style property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`. For a complete description of text styles, see "Elements of Text Objects" on page 314.

Window

See the definition of the document object class on page 323.

Word

Generally speaking, words in English are text objects delimited by spaces, return characters, or the beginning or end of a container. (For a more precise definition, see page 61.) Words in other languages are defined by the script system for each language if the appropriate script system is installed.

PROPERTIES

`Font` The name of the font of the word. If the font varies within the word, the Font property specifies the font of the first character.
Class: String
Modifiable? Yes

`Length` The number of characters in a text object.
Class: Integer
Modifiable? No

`Offset` The offset, in characters, from the beginning of the document to the first character of a text object.
Class: Integer
Modifiable? No

`Size` The size, in points, of the word. If the size varies within the word, the Size property specifies the size of the first character.
Class: Integer
Modifiable? Yes

`Style` The text styles of a text object. If the text styles vary within the word, the Style property specifies the styles of the first character.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to the text object. The Off Styles property specifies the styles that do not apply to the text object. Both properties are lists of constants that specify text styles. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Style property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? Yes

`Uniform Styles`
The text styles that are uniformly "on" or "off" for all characters of a text object.
Class: Text Style Info. Objects of this class have two properties: On Styles and Off Styles. The On Styles property specifies the styles that apply to all the characters of the text object. The Off Styles property specifies the styles that do not apply to any of the characters of the text object. Both properties are lists of constants that specify text styles. If a style does not appear in either list, the style applies to some but not all of the characters

in the object. See "Elements of Text Objects" on page 314 for information about how styles are used. The constants that can be used in Uniform Styles property lists are `bold`, `italic`, `outline`, `underline`, and `shadow`.
Modifiable? No

ELEMENT CLASSES

See "Elements of Text Objects" on page 314 for a general discussion of these element classes.

Character	Characters contained in the words
Paragraph	Paragraphs contained in the words
text	Series of characters contained in the words
Text Item	Text items contained in the words
Word	Words contained in the words

COMMANDS HANDLED

Copy, Count, Cut, Data Size, Delete, Duplicate, Exists, Get, Make, Move, Select, Set

DEFAULT VALUE CLASS RETURNED

Styled Text

EXAMPLES

```
move word 2 to end of document "Huge"

set size of word 10 to 18
```

Scriptable Text Editor Commands

This section defines the commands that are understood by the Scriptable Text Editor and its objects.

The Scriptable Text Editor supports most of the standard application commands listed in Chapter 4, "Commands." As is true for most scriptable applications, the Scriptable Text Editor's definitions for some of these commands differ slightly from the standard definitions. Table B-1 summarizes the differences between the defined behavior of the standard commands and the Scriptable Text Editor behavior.

Table B-1 Variations from standard behavior in Scriptable Text Editor versions of standard application commands

Standard application command	Scriptable Text Editor version
Close	Identical to the standard version defined on page 87.
Copy	Behaves like the standard version defined on page 88, except that the direct parameter must be a single object. The Scriptable Text Editor cannot copy a range of objects.
Count	Identical to the standard version defined on page 92.
Data Size	Behaves like the standard version defined on page 97, except that the Scriptable Text Editor returns the size of text objects as a data class that includes writing code information. The writing code information adds 4 bytes to the data size.
Delete	Identical to the standard version defined on page 98.
Duplicate	Behaves like the standard version defined on page 99, except that the direct parameter must be a single object. The Scriptable Text Editor cannot duplicate a range of objects.
Exists	Identical to the standard version defined on page 99.
Get	Identical to the standard version defined on page 100.

continued

Table B-1 Variations from standard behavior in Scriptable Text Editor versions of standard application commands (continued)

Standard application command	Scriptable Text Editor version
Make	Behaves like the standard version defined on page 105, with some exceptions. The default location for new text objects is the current selection; the default location for new document and window objects is in front of other Scriptable Text Editor windows. When you create text objects with the Make command, the Scriptable Text Editor automatically adds the appropriate delimiters before or after the new text objects, depending on the script code for the new text, the script code for the surrounding text, and, in the case of text items, the value of the Text Item Delimiters property. Do not include delimiters in the `with data` parameter.
Move	Behaves like the standard version defined on page 106, except that the direct parameter must be a single object. The Scriptable Text Editor cannot move a range of objects.
Open	Behaves like the standard version defined on page 107, except that the Scriptable Text Editor can only open text files or its own files. If you use the Open command to open a file of type `"TEXT"`, the Scriptable Text Editor automatically converts the file to a Scriptable Text Editor file and appends "`.s`" to the end of the filename.
Print	Identical to the standard version defined on page 108.
Quit	Identical to the standard version defined on page 109.
Run	Identical to the standard version defined on page 110.
Save	Behaves like the standard version defined on page 112, except that it has an optional parameter that allows you to save Scriptable Text Editor files as text files. The direct parameter of the Save command must be a single object. The Scriptable Text Editor cannot save a range of objects.
Set	Identical to the standard version defined on page 113.

Table B-2 summarizes the commands other than the standard application commands that are supported by the Scriptable Text Editor.

Table B-2 Other Scriptable Text Editor commands

Command	Description
Cut	Removes an object and puts it on the Clipboard
Paste	Copies the object or objects on the Clipboard to the current selection
Revert	Replaces a document with the most recently saved version
Select	Selects an object or document

The sections that follow describe both the commands listed in Table B-1 that differ from the standard versions and the commands listed in Table B-2.

Copy

The standard Copy command is defined on page 88. The Scriptable Text Editor version of the Copy command behaves like the standard version, except the direct parameter of the Scriptable Text Editor command cannot be a list; it must be a reference to a single object. For example, the following statement results in an error:

```
tell document "Intro" of app "Scriptable Text Editor"
   copy words whose style contains bold to beginning
end
--result: error; Copy can handle single objects only
```

Cut

The Cut command is a request to remove an object and put it on the Clipboard. If the Cut command includes a direct parameter, it removes the object specified in the direct parameter. If the command does not include a direct parameter, it removes the object in the current selection.

The Cut command has the same result as choosing the Cut menu item in the Scriptable Text Editor: the object that was cut replaces any objects that were previously on the Clipboard.

SYNTAX

```
cut [ referenceToObject ]
```

PARAMETER

referenceToObject
A reference to the object to be cut.
Class: Reference

RESULT

None

EXAMPLE

Both of the following statements cut the first paragraph of a document.

```
tell application "Scriptable Text Editor"
   set the selection to paragraph 1 of document "Test"
   cut
end tell

tell application "Scriptable Text Editor"
   cut paragraph 1 of document "Test"
end tell
```

NOTES

If you do not specify a direct parameter and there is no selection to be cut, the Scriptable Text Editor does not change the contents of the Clipboard.

If you specify a direct parameter, it must be a reference to a single object. The Scriptable Text Editor cannot cut ranges of objects. For example, the following statement results in an error.

```
tell application "Scriptable Text Editor"
   cut words of paragraph 5 of front document
end tell
--result: error, because Cut can handle single objects only
```

Data Size

The standard Data Size command is defined on page 97. The Scriptable Text Editor version of the Data Size command behaves like the standard version. It returns the size of the data (a value) that would result from a Get command on an object or objects.

By default, the Scriptable Text Editor returns text objects as class Styled Text, including font and style information. The additional information adds a variable number of bytes to the data size. As a result, the size of a text object will be greater than the number of bytes contained in the corresponding text without the font and style information. To get the data size of a text object without font and style information, use the `as` *className* parameter to request the size of the data as text; for example,

```
data size of word 1 of front document as text
```

Duplicate

The standard Duplicate command is defined on page 99. The Scriptable Text Editor version of the Duplicate command behaves like the standard version of the command, except the direct parameter cannot be a list; it must be a

reference to a single object. For example, the following statement results in an error:

```
tell document "Intro" of app "Scriptable Text Editor"
   duplicate words whose style contains bold to beginning
end
--result: error; Duplicate can handle single objects only
```

Make

The standard Make command is defined on page 105. The Scriptable Text Editor version of the Make command behaves like the standard version except that the *referenceToLocation* parameter, which is required in the standard version of the command, is optional in the Scriptable Text Editor version:

```
make [ new ] className [ at referenceToLocation ]                ¬
   [ with properties                                             ¬
      { propertyName:propertyValue [, propertyName:propertyValue]...}]¬
   [ with data dataValue ]
```

In addition, the Scriptable Text Editor automatically adds delimiters as needed around new text objects. When you create text objects with the Make command, do not include delimiters in the data.

When making a new text item, the Scriptable Text Editor uses the first delimiter listed in its Text Item Delimiters property as the item's delimiter.

Not all languages require delimiters between each word. When making new words, the Scriptable Text Editor relies on both the script code of the text to be inserted and the script code of the text into which it is being inserted to determine whether to use delimiters and, if so, which delimiters to use and where to place them.

Finally, the Scriptable Text Editor provides default values for the data and properties of objects created with the Make command. If you do not specify values for all the properties or data of a new object, the Scriptable Text Editor provides the default values.

Move

The standard Move command is defined on page 106. The Scriptable Text Editor version of the Move command behaves like the standard version, except that the direct parameter cannot be a list; it must be a reference to a single object.

For example, the following statement results in an error:

```
tell document "Intro" of app "Scriptable Text Editor"
   move words whose style contains bold to beginning
end
--result: error; Move can handle single objects only
```

Open

The standard Open command is defined on page 107. The Scriptable Editor can open its own files and text files (files of type `"TEXT"`). If you use the Open command to open a file of type `"TEXT"`, the Scriptable Text Editor automatically converts the file to a Scriptable Text Editor file and appends "`.s`" to the end of the filename.

Paste

The Paste command is a request to make a copy of the objects on the Clipboard and replace the current selection with them. The Paste command has the same effect as choosing the Paste command from the Scriptable Text Editor's Edit menu.

SYNTAX

```
paste
```

PARAMETERS

None

RESULT

None

EXAMPLE

```
tell application "Scriptable Text Editor"
   select paragraph 1 of document "Report"
   cut
   select the last insertion point of document "Report"
   paste
end tell
```

NOTES

The Paste command replaces the current selection with the contents of the Clipboard. If the current selection is an insertion point, the Paste command inserts the contents of the Clipboard at the insertion point.

Revert

The Revert command is a request to replace one or more documents with the versions of the documents that were most recently saved.

SYNTAX

`revert` *referenceToDocument*

PARAMETER

referenceToDocument
A reference to the document or documents to be replaced by the versions of the documents that were most recently saved. Because the Scriptable Text Editor's document and window objects are interchangeable, either document or window objects can be specified.
Class: Reference

RESULT

None

EXAMPLE

```
revert document "Really Big"
```

Save

The standard Save command is defined on page 112. The Scriptable Text Editor version behaves like the standard version, except that it has an extra optional parameter (`as text`) that allows you to save Scriptable Text Editor files as text files (files of type `"TEXT"`). Without this parameter, files are saved as Scriptable Text Editor files.

`save` *referenceToObject* `[in` *referenceToFile* `] [as text]`

In addition, the direct parameter of the Scriptable Text Editor version cannot be a list; it must be a reference to a single object. For example, the following statement results in an error:

```
tell application "Scriptable Text Editor"
   save documents 1 thru 3
end tell
--result: error; Save can handle single objects only
```

Select

A Select command is a request to select a text object or bring a Scriptable Text Editor document or window to the front. Using the Select command to select a text object is the same as highlighting the text object by double-clicking it or dragging the cursor across it while holding down the mouse button.

SYNTAX

`select` *referenceToObject*

PARAMETER

referenceToObject
A reference to the text object, document, or window to select.
Class: Reference

RESULT

None

EXAMPLE

```
tell application "Scriptable Text Editor"
   select paragraph 5 of document "Introduction"
   copy
   select paragraph 10 of document "New Introduction"
   paste
end tell
```

Scriptable Text Editor Errors

This section lists error numbers and messages returned by the Scriptable Text Editor. For information about writing handlers for these and other errors, see Chapter 8, "Handlers."

Error number	Error message
–15301	You must run on 512Ke or later.
–15302	Application Memory Size is too small.
–15303	Not enough memory to run Scriptable Text Editor.
–15304	Cannot exceed 32,000 characters.
–15305	Could not create self-address descriptor.
–15306	Cannot have more than 6 documents open at a time.
–15307	Could not open a new window.
–15308	Could not get style scrap.
–15309	User canceled.
–15310	Bad data.
–15311	Cannot handle list - need single item.
–15312	Had trouble with the Print job.
–15313	Not a valid font size.
–15314	No such property.
–15315	Property is read-only.
–15316	Cannot select that thing.
–15317	Cannot perform that comparison.
–15318	Invalid comparison.
–15319	Cannot handle that file type.
–15320	Need an open window to do that.
–15321	Cannot perform that operation on text lists - try a text range or repeat loop.
–15322	Cannot get data for that kind of object.
–15323	Window and file names must be 31 characters or less.
–15324	You must run on System 7 or later.

Error Messages

This appendix lists error numbers and error messages for the following types of errors:

- Operating System errors are errors that occur when AppleScript or an application requests services from the Operating System. They are rare, and more important, there's usually nothing you can do about them in a script. A few, such as `"File <name>wasn't found"` and `"Application isn't running"`, make sense for scripts to handle.
- Apple event errors occur when Apple events sent by AppleScript fail. Many of these errors, such as `"No user interaction allowed"`, are of interest to users. Also of interest to users are errors that have to do with reference forms, as well as errors like `"No such object"`.
- *Apple Event Registry* errors are errors returned by applications when handling standard AppleScript commands (commands that apply to all applications). Many of these, such as `"The specified object is a property, not an element"`, are of interest to users and should be handled.
- AppleScript errors are errors that occur 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.

Operating System Errors

Error number	Error message
0	No error.
–34	Disk <name> is full.
–35	Disk <name> wasn't found.
–37	Bad name for file.
–38	File <name> wasn't open.
–39	End of file error.
–42	Too many files open.
–43	File <name> wasn't found.
–44	Disk <name> is write protected.
–45	File <name> is locked.
–46	Disk <name> is locked.
–47	File <name> is busy.
–48	Duplicate file name.
–49	File <name> is already open.
–50	Parameter error.
–51	File reference number error.
–61	File not open with write permission.
–108	Out of memory.
–120	Folder <name> wasn't found.
–124	Disk <name> is disconnected.
–128	User canceled.
–192	A resource wasn't found.
–600	Application isn't running.
–601	Not enough room to launch application with special requirements.
–602	Application is not 32-bit clean.

continued

Error number	Error message
–605	More memory is needed than is specified in the size resource.
–606	Application is background-only.
–607	Buffer is too small.
–608	No outstanding high-level event.
–609	Connection is invalid.
–904	Not enough system memory to connect to remote application.
–905	Remote access is not allowed.
–906	<name> isn't running or program linking isn't enabled.
–915	Can't find remote machine.
–30720	Invalid date and time <date string>.

Apple Event Errors

Error number	Error message
–1700	Can't make some data into the expected type.
–1701	Some parameter is missing for <commandName>.
–1702	Some data could not be read.
–1703	Some data was the wrong type.
–1704	Some parameter was invalid.
–1705	Operation involving a list item failed.
–1706	Need a newer version of the AppleEvent manager.
–1707	Event isn't an AppleEvent.
–1708	<reference> doesn't understand the <commandName> message.
–1709	AEResetTimer was passed an invalid reply.
–1710	Invalid sending mode was passed.
–1711	User canceled out of wait loop for reply or receipt.

continued

–1712	AppleEvent timed out.
–1713	No user interaction allowed.
–1714	Wrong keyword for a special function.
–1715	Some parameter wasn't understood.
–1716	Unknown AppleEvent address type.
–1717	The handler <identifier> is not defined.
–1718	Reply has not yet arrived.
–1719	Can't get <reference>. Invalid index.
–1720	Invalid range.
–1721	<expression> doesn't match the parameters <parameterNames> for <commandName>.
–1723	Can't get <expression>. Access not allowed.
–1725	Illegal logical operator called.

Error number	Error message
–1726	Illegal comparison or logical.
–1727	Expected a reference.
–1728	Can't get <reference>.
–1729	Object counting procedure returned a negative count.
–1730	Container specified was an empty list.
–1731	Unknown object type.
–1750	Scripting component error.
–1751	Invalid script id.
–1752	Script doesn't seem to belong to AppleScript.
–1753	Script error.
–1754	Invalid selector given.
–1755	Invalid access.
–1756	Source not available.
–1757	No such dialect.
–1758	Data couldn't be read because its format is obsolete.
–1759	Data couldn't be read because its format is too new.
–1760	Recording is already on.

Apple Event Registry Errors

Error number	Error message
–10000	AppleEvent handler failed.
–10001	A descriptor type mismatch occurred.
–10002	Invalid key form.
–10003	Can't set <object or data> to <object or data>. Access not allowed.
–10004	A privilege violation occurred.
–10005	The read operation wasn't allowed.

–10006	Can't set <object or data> to <object or data>.
–10007	The index of the event is too large to be valid.
–10008	The specified object is a property, not an element.
–10009	Can't supply the requested descriptor type for the data.
–10010	The AppleEvent handler can't handle objects of this class.
–10011	Couldn't handle this command because it wasn't part of the current transaction.
–10012	The transaction to which this command belonged isn't a valid transaction.
–10013	There is no user selection.
–10014	Handler only handles single objects.
–10015	Can't undo the previous AppleEvent or user action.

AppleScript Errors

Error number	Error message
–2701	Can't divide <number> by zero.
–2702	The result of a numeric operation was too large.
–2703	<reference> can't be launched because it is not an application.
–2704	<reference> isn't scriptable.
–2705	The application has a corrupted dictionary.
–2706	Stack overflow.
–2707	Internal table overflow.
–2708	Attempt to create a value larger than the allowable size.
–2709	Can't get the event dictionary.
–2720	Can't both consider and ignore <attribute>.
–2721	Can't perform operation on text longer than 32K bytes.
–2729	Message size too large for the 7.0 Finder.
–2740	A <language element> can't go after this <language element>.
–2741	Expected <language element> but found <language element>.
–2750	The <name> parameter is specified more than once.
–2751	The <name> property is specified more than once.
–2752	The <name> handler is specified more than once.
–2753	The variable <name> is not defined.
–2754	Can't declare <name> as both a local and global variable.
–2755	Exit statement was not in a repeat loop.
–2760	Tell statements are nested too deeply.
–2761	<name> is illegal as a formal parameter.
–2762	<name> is not a parameter name for the event <event>.
–2763	No result was returned for some argument of this expression.

APPENDIX C

Error Messages

Glossary

Apple event A high-level message that adheres to the interprocess messaging protocol on which AppleScript is based.

AppleScript A scripting language defined by Apple Computer, Inc., that allows you to control Macintosh computers without using the keyboard or mouse.

AppleScript command A command handled by AppleScript. AppleScript commands do not have to be included in Tell statements.

application command A command handled by an application or its objects. An application command must either be included in a Tell statement or include the name of the application in its direct parameter.

application object An object stored in an application or its documents and managed by the application.

Arbitrary Element reference form A reference form that specifies an arbitrary object in a container. If the container is a value, AppleScript uses a random-number generator to choose the object. If the container is an application object, the application chooses the object.

assignment statement A statement that assigns a value to a variable. Assignment statements begin with Set or Copy.

attachable application An application that can be customized by attaching scripts to specific objects in the application, such as buttons and menu items.

attaching a script to an application object The process of associating a script with a specific application object. Each application determines which, if any, of its objects can have scripts attached.

attribute A characteristic that can be considered or ignored in a Considering or Ignoring statement.

binary operator An operator that derives a new value from a pair of values.

Boolean A logical truth value. The two possible Boolean values are `true` and `false`. Boolean is an AppleScript value class.

Boolean expression An expression whose value can be either `true` or `false`.

child script object A script object that inherits properties and handlers from another script object, called the parent.

Class The name of the AppleScript value class for a class identifier, a reserved word that specifies the class to which an object or value belongs. See also **object class, value class.**

coercion The process of converting a value from one class to another.

command A word or phrase that requests an action. In AppleScript, there are four types of commands: AppleScript commands, application commands, scripting additions, and user-defined commands.

command handler A handler for an application or system command. Command handlers are similar to subroutines, but instead of defining responses to user-defined commands, they define responses to commands, such as Open, Print, or Move, that are sent to application objects.

comment Descriptive text that is ignored by AppleScript when a script is executed.

compile In AppleScript, to convert a script from the form typed into a script editor to a form that can be used by AppleScript. The process of compiling a script includes syntax and vocabulary checks. A script is compiled when you first run it and again when you modify it and then run it again, save it, or check its syntax.

compiled script The form to which a script is converted when you compile it. The form of a compiled script is independent of the dialect in which a script is written.

complete reference A reference that has enough information to identify an object or objects uniquely. For a reference to an application object to be complete, its outermost container must be the application itself.

composite value A value that contains other values. AppleScript has three types of composite values: lists, records, and strings.

compound statement A statement that occupies more than one line and contains other statements. A compound statement begins with a reserved word indicating its function and ends with the word `end`.

conditional statement See **If statement.**

Considering statement A control statement that lists a specific set of attributes to be considered when AppleScript performs operations on strings or sends commands to applications.

constant A reserved word defined by AppleScript or an application in its dictionary. Constant is an AppleScript value class.

container An object that contains one or more other objects, known as elements. You specify containers with the reserved words `of` or `in`.

continuation character A character (¬) used in the Script Editor to extend a statement to the next line.

control statement A statement that controls when and how one or more other statements are executed. The types of control statements you can use in AppleScript are Tell, If, Repeat, Considering and Ignoring, With Timeout, and With Transaction.

current application Either the default target application or whatever application is currently set as a script's parent property.

current directory The folder or volume whose contents you can see when you choose Open or the equivalent command from an application's File menu.

current script The script from which a user-defined command is executed.

Data An AppleScript value class used for data that do not belong to any of the other AppleScript value classes. In AppleScript, a value that belongs to the class Data can be stored in a variable, but cannot be manipulated.

Date An AppleScript value class used for a value that specifies a time, day of the month, month, and year.

declaration The first occurrence of a variable or property identifier in a script. The form and location of the declaration determine how AppleScript treats the identifier in that script—for example, as a property, global variable, or local variable.

default target The object that receives a command if no object is specified or if the object is incompletely specified in the command. Default targets are specified in Tell statements.

delegation The use of a Continue statement to call a handler in a parent script object or the current application.

dialect A version of the AppleScript language that resembles a specific human language or programming language; for example, AppleScript English resembles English, AppleScript Japanese resembles Japanese, and so on. All scripts, regardless of the dialects in which they are written, are converted to the same form, called a compiled script, when they are compiled.

dictionary The set of commands, objects, and other words that are understood by a particular application or by a version of the system software. Each application or version of the system software has its own dictionary.

direct parameter The parameter immediately following a command.

element An object contained within another object, or a type of object that can be contained in another object. For example, a word object is an element of a paragraph object, but it is possible to have a paragraph with no words.

empty list A list with no items.

error expression An expression, usually a string, that describes an error.

error handler A collection of statements that are executed in response to an error message.

error message A message that is returned by an application, by AppleScript, or by the Macintosh Operating System if an error occurs during the handling of a command.

error number An integer that identifies an error.

evaluation The conversion of an expression to a value.

Every Element reference form A reference form that specifies every object of a particular class in a container.

Exit statement A statement used in the body of a Repeat statement to exit the Repeat statement.

explicit Run handler A handler at the top level of a script or a script object that begins with `on run` and ends with `end`. A single script or script object can include an explicit Run handler or an implicit Run handler, but not both.

expression In AppleScript, any series of words that has a value.

filter A phrase, added to a reference to a system or application object, that specifies elements in a container that match one or more conditions.

Filter reference form A reference form that specifies all objects in a container that match one or more conditions specified in a Boolean expression.

formal parameter See **parameter variable.**

global variable A variable that is available anywhere in the script in which it is defined.

handler A collection of statements that are executed in response to a command or an error message.

identifier A series of characters that identifies a value or handler in AppleScript. Identifiers are used to name variables, subroutines, parameters, properties, and commands.

ID reference form A reference form that specifies an object by the value of its ID property.

If statement A control statement that contains one or more Boolean expressions whose results determine whether to execute other statements within the If statement.

Ignoring statement A control statement that lists a specific set of attributes to be ignored when AppleScript performs operations on strings or sends commands to applications.

implicit Run handler All the statements at the top level of a script except for property declarations, script object definitions, and other command handlers. A single script or script object can include an explicit Run handler or an implicit Run handler, but not both.

Index reference form A reference form that specifies an object or location by describing its position with respect to the beginning or end of the container.

infinite loop A Repeat statement that does not specify when repetition stops.

inheritance The process by which a child script object receives the properties and handlers of a parent script object.

initializing a script object The process of creating a script object from the properties and handlers listed in a script object definition. AppleScript creates a script object when it runs a script or handler that contains a script object definition.

insertion point An object class, supported by many applications, that specifies a place where another object or objects can be added.

integer A positive or negative number without a fractional part. In AppleScript, Integer is a value class.

item A value in a list or record. An item is specified by its offset from the beginning or end of the list or record.

labeled parameter A parameter that is identified by a label. See also **positional parameter.**

list An ordered collection of values. Lists are enclosed by braces. The values in a list are separated by commas. List is an AppleScript value class.

literal expression An expression that evaluates to itself.

local variable A variable that is available only in the handler in which it is defined. Variables that are defined within subroutines, command handlers, and error handlers are local unless they are explicitly declared as global variables.

loop A series of statements that is repeated.

looping variable A variable whose value controls the number of times the statements in a Repeat statement are executed.

Middle Element reference form A reference form that specifies the middle object of a particular class in a container.

Name reference form A reference form that specifies an object by the value of its Name property.

nested control statement A control statement that is contained within another control statement.

Number A synonym for the AppleScript value classes Integer and Real.

object An identifiable part of an application, or thing within an application, that can respond to commands.

object class A category for objects that share characteristics such as properties and element classes and respond to the same commands.

operand A value from which an operator derives another value.

operation An expression that derives a new value from one or more other values. An operator, such as the addition operator (+), concatenation operator (&), or Contents Of, determines how the new value is derived.

operator An AppleScript language element (a word, series of words, or symbol) used in an expression to derive a value from another value or pair of values.

optional parameter A parameter that need not be included for a command to be successful.

parameter variable An identifier in a subroutine definition that represents the actual value of a parameter when the subroutine is called. Also called **formal parameter**.

parent script object A script object from which another script object, called the child, inherits properties and handlers.

partial reference A reference that does not include enough information to identify an object or objects uniquely. When AppleScript encounters a partial reference, it uses the default object specified in the Tell statement to complete the reference.

positional parameter A subroutine parameter that is identified by the order in which it is listed. In a subroutine call, positional parameters are enclosed in parentheses and separated by commas. They must be listed in the order in which they appear in the corresponding subroutine definition.

property A characteristic of an object that has a single value and is identified by a label. See also **script property.**

Property reference form A reference form that specifies a property of an application object, record or script object.

Range reference form A reference form that specifies a series of objects of the same class in the same container.

real A number that can include a decimal fraction. Real is an AppleScript value class.

record An unordered collection of properties. Properties within a record are identified by labels that are unique within the record. Record is an AppleScript value class.

recordable application An application that uses Apple events to report user actions for recording purposes. When recording is turned on, the Script Editor creates statements corresponding to any significant actions you perform in a recordable application.

recursive subroutine A subroutine that calls itself.

reference A phrase that specifies one or more objects using the reference forms defined in the AppleScript dialect you are using. Reference is an AppleScript value class.

reference form The syntax for referring to objects. The reference forms for the AppleScript English dialect include Arbitrary Element, Every Element, Filter, ID, Index, Middle Element, Name,Property, Range, and Relative.

Relative reference form A reference form that specifies an object or location by describing its position in relation to another object, known as the base, in the same container.

Repeat statement A control statement that contains a series of statements to be repeated and, in most cases, instructions that specify when the repetition stops.

required parameter A parameter that must be included for a command to be successful.

reserved words The words in system and application dictionaries, including object and command names, constants, parameters, and properties.

result A value generated when a command is executed or an expression evaluated.

scope The visibility and context of a variable or property, which determines where else in a script you may refer to the same variable. The scope of a variable depends on where you declare it and

whether you declare it as global or local. The scope of a property extends to the entire script or script object in which it is declared.

script A series of written instructions that, when executed, cause actions in applications and on the desktop.

scriptable application An application that can respond to application commands sent to it when an application such as Script Editor runs a script.

script application An application whose only function is to run the script associated with it.

script code A constant that identifies a particular script system for use on Macintosh computers.

script editor An application used to create and modify scripts.

Script Editor The script-editing application that comes with the AppleScript English language dialect.

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

script object A user-defined object in a script that combines data (in the form of properties) and potential actions (in the form of handlers).

script object definition A compound statement that contains a collection of properties, handlers, and other AppleScript statements. A script object definition begins with the reserved word `script`, followed by an optional variable name, and ends with the keyword `end` (or `end script`).

script property A named container in which to store a value. Script properties are similar to variables, but they are persistent. Unlike variable values, script property values are saved when you save a script.

script system A collection of system software facilities that allow for the visual representation of a particular writing system. Script systems include Roman, Japanese, Hebrew, Greek, and Thai.

simple statement A statement that is contained on a single line and ends with a return character. See also **compound statement.**

simple value A value, such as an integer or a constant, that does not contain other values.

statement A series of AppleScript words, similar to an English sentence, that contains a request for an action or an expression to be evaluated. See also **compound statement, simple statement.**

string An ordered series of characters (a character string). String is an AppleScript value class.

Styled Text A synonym for the AppleScript value class String. A string referred to as Styled Text may include style and font information.

subroutine A collection of statements that are executed in response to a user-defined command.

suite A set of AppleScript words that are related.

synonym An AppleScript word, phrase, or language element that has the same meaning as another AppleScript word, phrase, or language element. For example, the operator `does not equal` is a synonym for ≠.

syntax The arrangement of words in an AppleScript statement.

syntax description A template for using a command or control statement in a script.

target The recipient of a command. Potential targets include application objects, script objects, the current script, and the current application.

Tell statement A control statement that specifies the default target for the statements it contains.

test A Boolean expression that specifies the conditions of a filter or an If statement.

Text A synonym for the AppleScript value class String.

Try statement A two-part compound statement that contains a series of AppleScript statements, followed by an error handler to be invoked if any of those statements cause an error.

unary operator An operator that derives a new value from a single value.

user-defined command A command that triggers the execution of a collection of statements, called a subroutine, elsewhere in the same script.

value A type of data that can be manipulated by and stored in scripts. The AppleScript value classes are Boolean, Class, Constant, Data, Date, Integer, List, Real, Record, Reference, and String.

value class A category of values with similar characteristics. Values that belong to the same class respond to the same operators.

variable A named container in which to store a value.

With Timeout statement A control statement that allows you to change the amount of time AppleScript waits for application commands to complete before stopping execution of the script.

With Transaction statement A control statement that allows you to take advantage of applications that support the notion of a transaction—a sequence of related events that should be performed as if they were a single operation.

Index

Symbols

" character 62
* operator 166
/ operator 167
– operator 167
& operator 163, 177–178
() in syntax descriptions xix
+ operator 166
<= operator 165
< operator 164
= operator 163
>= operator 164
> operator 164
[] in syntax descriptions xix
\ character 62
^ operator 167
{} 49
| in syntax descriptions xix
≠ operator 164
≤ operator 165
≥ operator 164
¬ 25
«» 83–84
÷ operator 167

A

adding values to lists 51
addition of date and number values 180–181
addition operator 166
`after` reserved word 139
`alias` versus `file` 144–145
And operator 163
angle brackets in scripts, terms within 83–84
Apple event errors 205, 359–360
Apple Event Registry errors 205, 361
Apple events 14, 314
AppleScript
  commands 73–74, 84–117
  defined 3
  errors 205, 362
  extension 14
  and non-Roman script systems 317–318
  Text Item Delimiters property 158–160
AppleTalk networks 146
AppleTalk zones 146, 148
application commands 72–73, 84–117
application errors 205
Application object class 318–320
application objects 17, 119
`application responses` attribute 215
applications
  customizing using AppleScript 7, 11
  integrating using AppleScript 7–8
  references to 146–148
  on remote computers 148
`app` reserved word 29
Arbitrary Element reference form 126–127
A Reference To operator 153–154, 168
arithmetic, date-time 180–181
arithmetic operators 166–167, 297
ASCII collating sequence 173
As operator 67, 167, 225
assignment statements 22, 151
associativity, of operators 178–179
attachable applications 11
attaching scripts to objects 241
attributes 214–217
automating activities 5

B

`back of` reserved words 139
`back` reserved word 131, 132
backslash character in strings 62
`"Bad Data"` error message 317
`before` reserved word 139
`beginning` reserved word 132
Begins With operator 165
binary operator 21
Boolean expressions 190
Boolean value class 40
Bounds property 323, 327–328
brackets xix

C

capitalization in AppleScript 28
`case` attribute 214
case sensitivity 28
cell object 186
Character object class 321–323
characters
  in different script systems 318
  elements of a string 60
child script objects 271–286
classes
  of operands 161–168
  of parameters 78, 225
Class value class 41
Clipboard property 318
Closable property 323
Close command 87–88
coercion
  of parameters 81
  of values 67–70
coercion operator 167
Comes After operator 164
Comes Before operator 164
command definitions
  AppleScript
    Copy 88–91
    Count 92–96
    Error 210–212
    Get 100–102
    Run 110–112
    Set 113–117
  Scriptable Text Editor 345–355
    Copy 347
    Cut 348
    Data Size 349
    Duplicate 349
    Make 350
    Move 351
    Open 351
    Paste 351
    Revert 352
    Save 353
    Select 354
  standard application commands
    Close 87–88
    Copy 88–91
    Count 92–96
    Data Size 97–98
    Delete 98
    Duplicate 99
    Exists 99–100
    Get 100–102
    Launch 103–104
    Make 105–106
    Move 106–107
    Open 107–108
    Print 108–109
    Quit 109–110
    Save 112–113
    Set 113–117
  using 77–80
command handlers 241–252
  in script applications 243–252
  in stay-open script applications 247–250
commands 17–18, 71–117
  AppleScript 73–74, 84–117
  application 72–73, 84–117
  defined 71
  handlers for. *See* command handlers
  objects that can respond to 122

commands *(continued)*
parameters of 78–79
Scriptable Text Editor 345–354
scripting addition 23, 74–75
summarized 289–293
syntax of 78
targets of 71
user-defined 76, 221–240
waiting for completion of 215, 217
comments 26–27
comparison operators 163–165, 298
comparisons 213
compiling a script 30
complete reference 124
completion of commands 215, 217
composite values 33
compound statements 16
concatenation operator (&) 163, 177–178
conditional statement. *See* If statements
Considering statements 213–217
constants, listed 305
Constant value class 42
constructor functions 270, 285–286
containers 123–124, 296
containment operators 165–166, 297
Contains operator 165, 175–176
Contents property 58, 154, 323
continuation characters 25
Continue statements 277–283
passing commands to applications with 280–283
control statements 183–218
defined 183
listed 300
nested 184
Copy command
in assignment statements 22, 150
defined 88–91
definition 347
with script objects 283–286
Count command 50, 92–96
`current application` reserved words 281–283
current directory 144
current script 76
customizing applications 7, 11
Cut command 348

D

data sharing 117, 154, 283
Data Size command 97–98, 349
Data value class 43, 81
date-time arithmetic 180–181
Date value class 43–47
default object 185, 186
default target 18
delegation 277–286
Delete command 98
`diacriticals` attribute 215
dialects
defined 24
introduced 5
dictionaries 18–20
defined 19
Scriptable Text Editor 19, 185, 313–355
direct parameter 72, 78
division operator (÷) 167
`div` operator 167
Document object class 323–328
Does Not Come After operator 165
Does Not Come Before operator 164
Does Not Contain operator 166
Does Not Equal operator 164
double-quote character 62
Duplicate command 99, 349

E

`eighth` reserved word 131
elements
of objects 120
of text objects 314
of values 37
Else clause 191

Else If clause 191
empty list 49
`end` reserved word 132
Ends With operator 165, 173–174
Equal operator 163, 168–172
Error command 210–212
error handlers 204–212
  defining 206–209
error messages 204–212, 357–362
  defined 79
  Scriptable Text Editor 355
errors
  Apple event 205, 359–360
  Apple Event Registry 205, 361
  AppleScript 205, 362
  application 205
  Operating System 205, 358
  resignaling in scripts 211–212
  returned by commands 79
  script 206
  Scriptable Text Editor 355
  signaling in scripts 210–212
  types of 205–206
evaluation
  defined 21
  of expressions 149
  of expressions containing operators 162
`"event timed out"` error message 217
Every Element reference form 127–128
`every` reserved word 127
Exists command 99–100
Exit statements 194, 204
explicit Run handlers 244
exponent operator (^) 167
expressions 21–23, 149–181
  Boolean 190
  evaluation of 149, 162
  literal 36

F

`fifth` reserved word 131
filename 329
File object class 328–329
files, specifying 143–145
`file` versus `alias` 144–145
Filter reference form 129–143
`first` reserved word 131
Floating property 324
Font property 321
`fourth` reserved word 131
`from` reserved word 136
Frontmost property 318
`front of` reserved words 139
`front` reserved word 131, 132

G

Get command 100–102
Getting Started With AppleScript xv, xviii
`given` parameter label 230–234
global variables 155, 158
  persistence of 255, 259–260
  scope of 252–264
Greater Than operator 164, 172–173
Greater Than Or Equal To operator 164

H

handlers 221–264, 267
  for application commands 241–242
    in script applications 243–252
  defined 221
  for errors 204–212
  for Idle command 248
  interrupting 250
  for Open command 246–247
  for Quit command 249
  for Run command 243–246
  scope of identifiers declared within 263–264
  for stay-open script applications 247–250
  syntax summary 302
  for user-defined commands 221–240
`hyphens` attribute 215

I, J, K

identifiers 27–28, 150
Idle command, and stay-open script applications 248
ID reference form 130–131
`id` reserved word 130
If statements 190–193
  compound 193
  simple 192
Ignoring statements 213–217
implicit Run handlers 244
`in back of` reserved words 139
Index property 324
Index reference form 131–133
`index` reserved word 131
infinite loop 194
`in front of` reserved words 139
inheritance 271–286
initializing script objects 266, 269–271
`in` reserved word in references 123
insertion point object 80
  and Index reference form 132
  and Relative reference form 139
Insertion Point object class 329–331
instance variables 267
Integer value class 47–48
integral division operator 167
integrating applications 7–8
Is Contained By operator 166, 175–176
Is Equal To operator 163
Is Not Contained By operator 166
Is Not Equal To operator 168–172
Is Not Greater Than operator 165
Is Not Less Than operator 164
Is Not operator 164
Is operator 163
items 48
`it` variable 186–188

L

labeled parameters 78, 223
`last` reserved word 131
Launch command 103–104
Length property 315, 321
  of a list 49
  of a record 55
  of a string 60, 64
Less Than operator 164, 172–173
Less Than Or Equal To operator 165
library 226
lists
  adding values to 51
  merging 51
List value class 48–51
literal expressions 36
Load Script command 226
local variables 155, 270
  scope of 252–264
location parameters 80, 140
logical operators 163, 168, 297
looping variable 195, 196, 201–203
loops. *See* Repeat statements
lowercase letters 28, 214

M

macro languages 10
Make command 105–106, 350
merging lists 51
messages. *See* Apple events
methods 267
`me` variable 186–188, 224, 279–280
Middle Element reference form 133
`middle` reserved word 133
minus symbol (–) 167
Modal property 324
Modified property 324
`mod` operator 167
Move command 106–107, 351
multiplication operator 166
`my` reserved word 186–188, 224, 279–280

N

`named` reserved word 134
Name property 187, 318, 324
Name reference form 134
nested control statements 184
networks
  AppleTalk 146
  zones of 148
`ninth` reserved word 131
non-English text 66
`not` operator 168
Number value class 52–53

O

object class definitions
  Scriptable Text Editor
    Application 318–320
    Character 321–323
    Document 323–328
    File 328–329
    Insertion Point 329–331
    Paragraph 331–333
    Selection 334–336
    Text 336–338
    Text Item 339–341
    Text Style Info 341–342
    Window. *See* Document object class
    Word 342–344
  using 119–122
object-oriented design 265
objects 17–18, 119–148
  in applications 119
  commands that act on 122
  default 185, 186
  elements of 120
  properties of 120
  script
    child 271–286
    initializing 269–271
    parent 268, 271–286
    sending commands to 268–269
    Scriptable Text Editor 318–344
    text 313–317
    user-defined. *See* script objects
  values of 161
Offset property 75, 315, 321
Offset scripting addition command 75
Off Styles property 315–317
`of` reserved word in references 123
`on error` reserved words 207
`on` reserved word 229, 236
On Styles property 315–317
Open command 107–108, 351
  handlers for, in script applications 246–247
operands, defined 161
Operating System errors 205, 358
operations 21, 161–181
operators 21–23, 37, 161–179
  A Reference To 153–154
  arithmetic 166–167, 297
  binary 21
  comparison 163–165, 298
  containment 165–166, 297
  defined 21
  listed, by category 297
  listed, with descriptions 163–168
  logical 163, 168, 297
  precedence 178–179
  unary 21
optional parameters 79
order of operations. *See* precedence
Or operator 163

P

paragraph element of a string 60
Paragraph object class 331–333
parameters 80–82
  for application commands 78–79
  coercion of 81
  in Continue statements 277
  defined 78
  direct 72, 78

parameters *(continued)*
labeled 78, 223
location 80, 140
optional 79
patterned 238–239
positional 223
raw data in 81
required 79
parameter variables 270, 277
parentheses xix
Parent property 271
and current application 281–283
parent script objects 268, 271–286
partial references 124, 185
`partial result` parameter 210
Paste command 351
patterned parameters 238–239
persistence
of global variables 255, 259–260
of script properties 254, 259–260
placeholders xix, 307–311
plural object names 127
plus symbol (+) 166
positional parameters 223
Position property 324, 327–328
possessive object names 124
precedence
of attributes 216
of operations 178–179
predefined variables
introduced 156
listed 305
Print command 108–109
properties
of AppleScript 158–160
of objects 120
scope of 252–264
of script objects 267
of scripts 156–158
of values 36
Property reference form 135
`property` reserved word 157
`prop` reserved word 157
`punctuation` attribute 215

Q

Quit command 109–110
handlers for, in stay-open script applications 249

R

Range reference form 136–139
Real value class 53–54
recordable applications 11
Record button 10
Record value class 54–57
recursion 225–226
reference forms 125–143
Arbitrary Element 126–127
defined 125
Every Element 127–128
Filter 129–143
ID 130–131
Index 131–133
Middle Element 133
Name 134
Property 135
Range 136–139
Relative 139–140
and values 38
`reference` reserved word. *See* A Reference To operator
references 122–148
complete 124
defined 122
as expressions 160–161
to files and applications 143–148
partial 124, 185
Reference value class 57–59
`ref` reserved word. *See* A Reference To operator
Relative reference form 139–140
remainder operator 167
Repeat statements 194–204
Repeat (forever) 197
Repeat (number) Times 198

Repeat statements *(continued)*
Repeat Until 200
Repeat While 199
Repeat With (loopVariable) From (startValue) to (stopValue) 201–202
Repeat With (loopVariable) In (list) 202–203
required parameters 79
Required suite, of application commands 85
reserved words 20
Resizable property 325
Rest Of property 49
results 79, 82–83
`result` variable 83
result window 150
return character in strings 62
Return statements 222, 228, 240
Reverse property 49
Revert command 352
Run command 110–112
handlers for. *See* Run handlers 243
and Launch command 104
Run handlers
in script applications 243–246
in script objects 267–269
running scripts 9

S

`'s` notation 124, 296
sample application xvii
Save command 112–113, 353
`saving` parameter 87, 109
scope, of variables and properties 252–264
scriptable applications 11
Scriptable Text Editor
commands 345–354
dictionary 19, 185, 313–355
and non-Roman script systems 317–318
object class definitions 318–344
script applications 243–252
calling 251–252
interrupting handlers in 250
script codes, and AppleScript 317
Script Editor 5, 30
script errors 206
scripting additions 23
introduced 74–75
in With Timeout statements 217
script objects 265–286
child 271–286
defined 265
initializing 266, 269–271
introduced 23
parent 268, 271–286
scope of identifiers declared within 258–262
sending commands to 268–269
script properties 156–158, 304
persistence of 254, 259–260
scope of 252–262
`script` reserved word 267
scripts
defined 3
running 9
scope of identifiers declared at top level of 254–257
script systems and AppleScript 317–318
`second[s]` reserved word 218
`second` reserved word 131
Select command 354
Selection object class 334–336
Selection property 319, 325
Set command
in assignment statements 22, 150
defined 113–117
scope of variables set with 252–264
with script objects 283–284
`seventh` reserved word 131
short-circuiting, during evaluation 163
simple statements 16
simple values 33
`sixth` reserved word 131
Size property 321
slash symbol (`/`) 167
`some` reserved word 126
special characters
in identifiers 28
in strings 62

Standard suite, of application commands 85
Starts With operator 165, 173–174
statements 14–17
  compound 16
  defined 14
  simple 16
storing values in variables 150
strings, special characters for 62
String value class 60–64
Styled Text value class 64–66
Style property 315–317, 321
subroutines 221–240
  calling
    labeled parameters 230–232
    no parameters 223
    positional parameters 236–240
  defined 221
  defining
    labeled parameters 229–234
    no parameters 223
    positional parameters 235–239
  libraries of 226
subtraction of date and number values 180–181
subtraction operator 167
suites 314
synonyms
  for operators 143, 163–168
  for value classes 38
syntax conventions xix
syntax description, defined 78

T

Tab character in strings 62
target 71
targets 18
Tell statements 72, 185–190
  compound 189–190
  introduced 16
  simple 188
`tenth` reserved word 131
terminating
  handler execution 240
  Repeat statement execution 204
tests 190
text
  element of a string 60
  non-English 66
  styled 64–66
  synonym for string 66
Text Item Delimiters property 319
  of AppleScript 158–160
Text Item object class 339–341
Text object class 336–338
text objects 313–317
Text Style Info object class 341–342
text styles 315–317
Text value class 66
`the`, use of in AppleScript 16
`third` reserved word 131
`through` reserved word 136
`thru` reserved word 136
timeout. *See* With Timeout statements
Titled property 325
`to` reserved word 229, 236
`try` reserved word 207
Try statements 204–209
  defined 206
typographic conventions xix

U

unary operators 21
Uniform Styles property 315–317, 322
uppercase letters 28, 214
user-defined commands 76, 221–240
user-defined objects. *See* script objects

V

value classes 38–67
  Boolean 40
  Class 41
  Constant 42
  Data 43, 81
  Date 43–47
  default, returned by Get command 122
  defined 33
  Integer 47–48
  List 48–51
  Number 52–53
  Real 53–54
  Record 54–57
  Reference 57–59
  String 60–64
  Styled Text 64–66
  summary of 39
  Text 66
  using definitions of 33–38
values 33–70
  characteristics of 33–38
  coercing 38, 67–70
  composite 33
  defined 20
  elements of 37
  of objects 161
  properties of 36
  responses to commands 37
  simple 33
variables 22–23, 150–156
  assignment statements 151, 304
  defined 150
  global 155, 158, 252–264
  instance 267
  local 155, 252–264, 270
  looping 195, 196, 201–203
  predefined 156
    listed 305
  scope of 155, 252–264
Version property 319
vertical bars xix
Visible property 325

W, X, Y

`where` reserved word 129
`white space` attribute 214
`whose` reserved word 129
Window object class. *See* Document object class
With clause 231
Without clause 231
With Timeout statements 217–218
With Transaction statements 219–220
word element of a string 61
Word object class 342–344
wrapper method 277

Z

zones, AppleTalk 146
Zoomable property 325
Zoomed property 325

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter IINTX printer. Final page negatives were output directly from the text and graphic files. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITERS
Sean Cotter, Mitchell Gass, and Pegi Wheeler

DEVELOPMENTAL EDITORS
Jeanne Woodward and Beverly Zegarski

ILLUSTRATOR
Deborah Dennis

PRODUCTION EDITOR
Rex Wolf

Special thanks to William Cook and Warren Harris.

Acknowledgments to Kathleen Carter, Dan Clifford, Sue Dumont, Ron Karr, Kazuhisa Ohta, Donald Olson, Jon Pugh, Brett Sher, Peter Sparks, and the entire AppleScript team.