Cocoa Scripting Guide

Interapplication Communication

2008-03-11

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

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

The Apple logo is a trademark of Apple Inc.

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

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

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

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

Apple, the Apple logo, AppleScript, Carbon, Cocoa, eMac, Logic, Mac, Mac OS, Objective-C, OpenDoc, Pages, QuickDraw, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Finder and Spotlight are trademarks of Apple Inc.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Simultaneously published in the United States and Canada.

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

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

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

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

Contents

Introduction — **Introduction to Cocoa Scripting Guide 11**

Who Should Read This Document 11
Organization of This Document 11
See Also 12

Chapter 1 — **Overview of Cocoa Support for Scriptable Applications 13**

AppleScript and Scriptable Applications 13
The AppleScript Object Model 14
Scriptability Information 16
    Scriptability Information Formats 16
    Viewing Scripting Terminology 18
    Built-in Support for Standard and Text Suites 19
    Built-in Support for Basic AppleScript Types 20
    Loading Scriptability Information 21
Reliance on Key-Value Coding 22
Interaction With Cocoa Bindings and Core Data 23
Scriptability and Undo 23
Snapshot of Cocoa Scripting 24
A Real World Scripting Example 25
Current Limitations of Cocoa Scripting Support 26

Chapter 2 — **Designing for Scriptability 27**

Designing a New Scriptable Application 27
Adding Scriptability to an Existing Application 29

Chapter 3 — **Implementing a Scriptable Application 31**

Implementation Guidelines 31
Supply a Scripting Definition File 32
Concentrate Scriptable Behavior in Model Objects 33
Provide Keys for Key-Value Coding 33
Add the Scripting Definition File to Your Xcode Project 34
Turn On Scripting Support in Your Application 34
Implement Object Specifier Methods for Scriptable Classes 35
Use the Document Architecture 35
Access the Text Suite 36

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Chapter 4 Preparing a Scripting Definition File 39

Structure of a Scripting Definition File 39
Code Constants Used in Scriptability Information 40
Features Common to Many Scripting Definition File Elements 41
High-level XML Elements 41
Create a Scripting Definition File 43
Add Information to the Scripting Definition File 44
Class Elements 44
Command Elements 47
Enumeration Elements 49
Record-Type Elements 50
Value-Type Elements 51
Cocoa Elements 51

Chapter 5 Getting and Setting Properties and Elements 53

Overview of Getting and Setting Values 53
Key-Value Coding and Cocoa Scripting 53
Maintain KVC Compliance 54
On Omitting KVC Accessors 55
Performance Considerations With KVC 55
Interaction With Key-Value Observing 56
KVC Conversion of Scalar and Structure Values 56
Scripting Additions to KVC 56
Sample KVC-Compliant Accessor Methods 56
Single-Value Access 57
Collection Value Access 58
Special Accessor Methods 59
Support for the Properties Property 59
Coercion 59

Chapter 6 Object Specifiers 61

Overview of Object Specifiers 61
Object Specifiers and KVC 61
When to Implement an Object Specifier Method 62
About Object Specifier Classes 62
A Closer Look at an Object Specifier 63
Evaluation of Nested Specifiers 65
Cocoa Object Specifier Classes 65
Implementing the Object Specifier Method 66
An Object Specifier Method for a Rectangle in Sketch 67
Specifying the Application Object as a Container 67
Implementing A Method for Evaluating Object Specifiers 68
Implicitly Specified Subcontainers 69

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Chapter 7 Script Commands 71

Script Commands Overview 71
- Script Command Classes Supplied by Cocoa 71
- Script Command Scriptability Information 72
- Script Command Components 72
- Script Command Creation 73
- Script Command Execution 74
- Script Commands and Object Specifiers 74
- Error Handling 75
- Object-first Versus Verb-first Script Commands 75
- Steps for Implementing a New or Modified Script Command 78

Implementing an Object-First Command—Rotate 79
Implementing a Verb-First Command—Align 81
Modifying a Standard Command 83
- A Verb-first Move Command 84
- An Object-first Move Command 84

Summary of AppleScript Command Support 85

Chapter 8 Testing, Debugging, and Performance 87

Scriptability Test Plan 87
Use AppleScript Scripts to Test Your Application 87
Turn On Debugging Output for Scripting 88
- Steps for Turning On Cocoa Debugging Output 89
- Sample Output 89

Debugging Scriptability Information 90
- Checking an sdef File with xmllint 91
- Examining Scriptability Information in Your Application 91

Additional Debugging Tips 93
Performance Issues for Scriptability 93

Chapter 9 Cocoa Scripting Classes and Categories 95

Script Commands and Scriptability Information 95
Object Specifiers, Logical Tests, and Related Categories 96
Key-Value Coding and Value Coercion 98
Subclasses for Standard AppleScript Commands 98
Manipulation of Apple Events 99

Chapter 10 How Cocoa Applications Handle Apple Events 101

Apple Event Handling Overview 101
- Basics of Apple Event Handling 101
- Handling Apple Events in a Cocoa Application 101

Apple Events Sent by the Mac OS 102

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Open Application 102
Reopen 103
Open 104
Print 105
Open Contents 105
Quit 106
Constants for Apple Event Handlers Installed by the Application Kit 106
Installing an Apple Event Handler 107
Installing a Get URL Handler 108
Implementing the Get URL Handler 108
Suspending and Resuming Apple Events and Script Commands 109

Appendix A **Evolution of Cocoa Scriptability Information 111**

Scriptability Terms 111
Changes in Scriptability Information Versions 111
Advantages of the Scripting Definition Format 112
Advantages of the Script Suite Format 112
Converting and Updating Scriptability Information 112
Creating Suite Files or 'aete' Files from a Scripting Definition 113
Creating Scripting Definitions from Suite Files or 'aete' Files 113
Updating Older Scripting Definition Files for Mac OS X Version 10.4 114
Editing Scriptability Information 114

Appendix B **Script Suite and Script Terminology Files 117**

Script Suite Files 117
The Structure of a Script Suite File 119
Script Terminology Files 123
The Structure of a Script Terminology File 124
Cocoa Scripting's Built-in Script Suites 127
Creating Your Own Script Suite Files 127

Document Revision History 131

Glossary 133

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Figures, Tables, and Listings

Chapter 1 **Overview of Cocoa Support for Scriptable Applications 13**

Figure 1-1 Object-model containment hierarchy for Sketch application 14
Figure 1-2 Sketch window with graphics 15
Figure 1-3 Structure of an sdef file 17
Figure 1-4 An sdef displayed in a dictionary viewer 19
Figure 1-5 An application's loaded scriptability information 22
Figure 1-6 A Cocoa application responding to an Apple event 25
Table 1-1 Support for basic AppleScript types 20
Listing 1-1 Graphic and rectangle elements from Sketch's sdef file 18

Chapter 3 **Implementing a Scriptable Application 31**

Table 3-1 Standard suite attributes and relationships 36
Table 3-2 Text Suite Attributes and Relationships 37

Chapter 4 **Preparing a Scripting Definition File 39**

Figure 4-1 Structure of an sdef file, revisited 40
Table 4-1 Default naming for attributes of cocoa elements 52
Listing 4-1 Version and document type in an sdef file 41
Listing 4-2 A dictionary element from an sdef file 42
Listing 4-3 A suite element for the Sketch suite 42
Listing 4-4 A class element for the rich text class 45
Listing 4-5 Definition of the save options enumeration 49
Listing 4-6 Definition of the print settings record-type 50
Listing 4-7 Definition of the color value-type 51

Chapter 5 **Getting and Setting Properties and Elements 53**

Listing 5-1 Boolean property getter 57
Listing 5-2 Boolean property setter 57
Listing 5-3 Simple array element accessors 58
Listing 5-4 Array element insert/delete accessors (by index) 58
Listing 5-5 Array element replacement accessor (by index) 58

Chapter 6 **Object Specifiers 61**

Figure 6-1 Sketch object model and containment hierarchy revisited 63
Figure 6-2 Nested object specifiers for a Sketch rectangle 64
Table 6-1 AppleScript reference forms and corresponding object specifier classes 65

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Listing 6-1 An object specifier method for a rectangle 67
Listing 6-2 Specifying the application as a container 67
Listing 6-3 indicesOfObjectsByEvaluatingObjectSpecifier: method from Sketch 68
Listing 6-4 Class definition for text document, containing a contents element 70

Chapter 7 **Script Commands 71**

Figure 7-1 Executing a script command—verb-first versus object-first 76
Table 7-1 Default support for AppleScript commands and how to customize it 85
Listing 7-1 A script to test the rotate command 80
Listing 7-2 A script to test the align command 83

Chapter 8 **Testing, Debugging, and Performance 87**

Listing 8-1 Simple test script 88
Listing 8-2 Debug scripting output for sdef-based Sketch 90
Listing 8-3 Debug scripting output for script suite-based Sketch 90
Listing 8-4 Partial output of NSScriptSuiteRegistry information 91
Listing 8-5 Turning on log statements 92
Listing 8-6 NSLog output for SKTAlignCommand 92

Chapter 9 **Cocoa Scripting Classes and Categories 95**

Table 9-1 Scripting information and command classes 95
Table 9-2 Object specifiers and related classes 96
Table 9-3 Scripting utilities 98
Table 9-4 Subclasses for standard script commands 99
Table 9-5 Classes for manipulating Apple events 99

Chapter 10 **How Cocoa Applications Handle Apple Events 101**

Table 10-1 Event class IDs for Apple events sent by the Mac OS 107
Listing 10-1 Extracting the search text parameter from the current Apple event 104
Listing 10-2 Signature of an event handler function 107
Listing 10-3 Installing an Apple event handler in a Cocoa application 108
Listing 10-4 Implementation of a get URL Apple event handler 108

Appendix B **Script Suite and Script Terminology Files 117**

Figure B-1 Script suite for the Standard suite in Property List Editor 129
Table B-1 Suite dictionary 120
Table B-2 Class list dictionary 120
Table B-3 Class dictionary 120
Table B-4 Property list dictionary 121
Table B-5 Property dictionary 121

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Table B-6 Supported commands dictionary 121
Table B-7 Command list dictionary 121
Table B-8 Command dictionary 121
Table B-9 Argument list dictionary 122
Table B-10 Synonym list dictionary 122
Table B-11 Enumeration list dictionary 122
Table B-12 Enumeration dictionary 122
Table B-13 Enumerators dictionary 123
Table B-14 Terminology dictionary 124
Table B-15 Class list terminology dictionary 124
Table B-16 Class terminology dictionary 125
Table B-17 Attribute list terminology dictionary 125
Table B-18 Attribute terminology dictionary 125
Table B-19 Command list terminology dictionary 125
Table B-20 Command terminology dictionary 125
Table B-21 Argument list terminology dictionary 126
Table B-22 Argument terminology dictionary 126
Table B-23 Class synonym list terminology dictionary 126
Table B-24 Class synonym terminology dictionary 126
Table B-25 Enumeration list terminology dictionary 127
Table B-26 Enumerators list terminology dictionary 127
Table B-27 Enumerator terminology dictionary 127
Listing B-1 NSApplication class from the script suite file for the Standard suite 118
Listing B-2 NSApplication class from the script terminology file for the Standard suite 123

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Introduction to Cocoa Scripting Guide

This document describes how to create scriptable applications using the support provided by the Cocoa application framework. That support, which is referred to as **Cocoa scripting**, includes classes, categories, and scriptability information.

A scriptable application is one that can be controlled by AppleScript scripts. Users write scripts to automate tasks and combine the use of multiple applications. As a developer, you can also use scripts to speed up prototyping or testing of your scriptable applications.

When a script that targets an application is executed, commands are sent to the application in the form of Apple events, a kind of interprocess message. Cocoa scripting helps you create scriptable applications by doing much of the work of receiving these Apple events, extracting information from them, and invoking methods in your scriptable classes.

This document provides conceptual information and examples that are based primarily on the scripting support available in Mac OS X versions 10.4 and 10.3.

Important: You can read about changes in Cocoa scripting support for Mac OS X v10.5 in the Scripting section of *Foundation Release Notes*.

Although some information may be accurate for versions of the Mac OS prior to v10.3, this document has not been reviewed for accuracy on those versions, nor does it attempt to provide details for working with those versions.

Who Should Read This Document

This document is intended for developers who want to make their Cocoa applications scriptable or who need to know more about how Cocoa applications interact with AppleScript and Apple events. It assumes you have some familiarity with Cocoa, Objective-C, and AppleScript. However, if you are unfamiliar with AppleScript, you should start by reading *Getting Started with AppleScript* and *AppleScript Overview*. Many of the terms scripters use are defined in *AppleScript Language Guide*.

Organization of This Document

The following chapters describe how to design, implement, and debug a scriptable Cocoa application:

- "Overview of Cocoa Support for Scriptable Applications" (page 13) provides a brief overview of AppleScript and scriptable applications, and describes the scripting support provided by the Cocoa application framework. This chapter is a prerequisite for the chapters that follow.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

- "Designing for Scriptability" (page 27) provides high-level checklists for designing a new scriptable Cocoa application and making an existing application scriptable.
- "Implementing a Scriptable Application" (page 31) lists the key steps for implementing a scriptable Cocoa application, with links to more detailed information where necessary.
- "Preparing a Scripting Definition File" (page 39) describes the structure of scripting definition, or sdef, files. It also shows how to create an sdef for a scriptable Cocoa application and how to add scriptability information to it.
- "Getting and Setting Properties and Elements" (page 53) describes how to work with Cocoa scripting to get and set the values of properties and elements in your scriptable application. It also provides examples of basic, KVC-compliant accessor methods.
- "Object Specifiers" (page 61) explains the mechanism for locating a scriptable object in the context of its container and provides sample object specifier methods.
- "Script Commands" (page 71) provides additional detail about the script command mechanism Cocoa uses to respond to Apple events and describes how to implement script commands.
- "Testing, Debugging, and Performance" (page 87) provides tips for building up your test plan debugging the scriptability in your Cocoa application, and spotting possible performance issues.

The following chapters provide additional information about Cocoa scripting support:

- "How Cocoa Applications Handle Apple Events" (page 101) describes the default support for handling Apple events in Cocoa applications and how your application interacts with it.
- "Cocoa Scripting Classes and Categories" (page 95) provides brief descriptions of the main classes that make up Cocoa scripting support, including those that you use in creating scriptable applications.
- "Evolution of Cocoa Scriptability Information" (page 111) describes changes to Cocoa scriptability information over time. It includes information about when to use various types of scriptability information and how to convert between them.

The following chapter describes how to specify scriptability information using an earlier format:

- "Script Suite and Script Terminology Files" (page 117) describes another way to provide terminology information and describes the structure of script suite and script terminology files.

See Also

For more information on the basic design patterns used by Cocoa scripting support, see *Key-Value Coding Programming Guide* and Cocoa Design Patterns in *Cocoa Fundamentals Guide*.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Overview of Cocoa Support for Scriptable Applications

This chapter provides an overview of Cocoa scripting and how your application takes advantage of it, and provides links to more detailed information in other chapters and documents.

A scriptable application is one that scripters can control with AppleScript scripts. To create a scriptable application, you specify a dictionary of terms that scripters can use with your application, implement classes and methods to support scriptable features, and provide a road map of scriptability information that AppleScript and Cocoa use to allow scripts to control the application.

Cocoa scripting refers to the support provided by the Cocoa application framework for creating scriptable applications. It includes classes, categories, and scriptability information that specifies the supported AppleScript terminology and the class information needed to work with it.

Cocoa scripting makes use of standard mechanisms and design patterns used throughout Cocoa, including key-value coding (KVC) and Model-View-Controller (MVC). When an AppleScript command targets your application, the goal of the scripting support is to send the command directly to the application's model objects to perform the work. To do that, it relies on the KVC mechanism to get and set values in your application's scriptable model objects, based on a set of keys you define for them.

Through the use of these mechanisms, you can make your application scriptable with a minimum of additional code.

AppleScript and Scriptable Applications

AppleScript is a scripting language that makes possible direct control of scriptable applications and scriptable parts of the Mac OS (such as the Finder). The AppleScript language doesn't supply an exhaustive or a task-specific terminology. Instead, it defines common commands, such as `get`, `set`, `make`, and `delete`, which can be applied to a wide variety of objects or their properties in a scriptable application. Scriptable applications define additional terms as needed for their unique operations.

A **scriptable application** is one that makes its operations and data available in response to AppleScript messages, called Apple events. An **Apple event** is a kind of interprocess message that can specify complex operations and data. Apple events make it possible to encapsulate a high-level task in a single package that can be passed across process boundaries, performed, and responded to with a reply event.

Note: Mac OS X mechanisms for communicating between processes are described in System-Level Technologies in *Mac OS X Technology Overview*.

A scriptable application specifies the set of scripting terms it understands and provides information that AppleScript uses to compile scripts that use those terms. When a user executes a script that targets the application, Apple events are sent to the application. Apple events can also be sent by other applications and by the Mac OS.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Applications handle Apple events by registering with the Apple Event Manager for the events they expect to receive and by supplying handler routines to process the events. Cocoa scripting simplifies this process by automatically registering and responding to Apple events your application can handle, based on the scriptability information you supply. That means you don't need to write low-level code to interact with Apple events and the Apple Event Manager.

AppleScript and Apple events are built on the Open Scripting Architecture (OSA), which is described in Open Scripting Architecture in *AppleScript Overview*.

The AppleScript Object Model

Every scriptable application defines an **AppleScript object model** to specify the classes of objects a scripter can work with in scripts, the accessible properties of those objects, and the inheritance and containment relationships for those classes. *Inheritance* allows a class to access the properties of its ancestors. *Containment* specifies where an object resides within the hierarchy of objects in the running application.

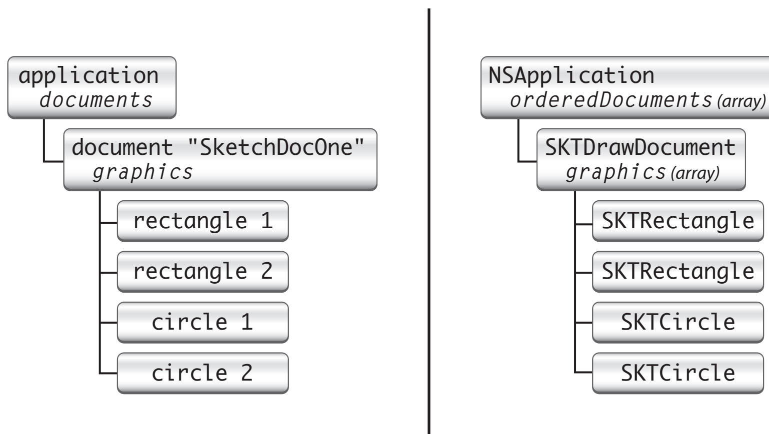
The objects in the object model often correspond closely to object classes in the application that implement scripting support, but there is no requirement that they do so. For example, a text application might provide script access to words in a document, but it would not be efficient for the application to maintain a corresponding object for each word.

Application classes have *attributes*, *to-one relationships*, and *to-many relationships*. AppleScript classes in the object model have *properties* and *elements*—properties are synonymous with attributes and to-one relationships, while elements are synonymous with to-many relationships. For more information on these and related terms, see the "Glossary" (page 133).

Figure 1-1 shows the object-model containment hierarchy for a specific document (shown in Figure 1-2 (page 15)) in the Sketch application. On the left are the objects a scripter uses to work with the application. On the right are the objects that Sketch uses to represent its object model.

Note: Sketch is a sample application available from Apple.

Figure 1-1 Object-model containment hierarchy for Sketch application



2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

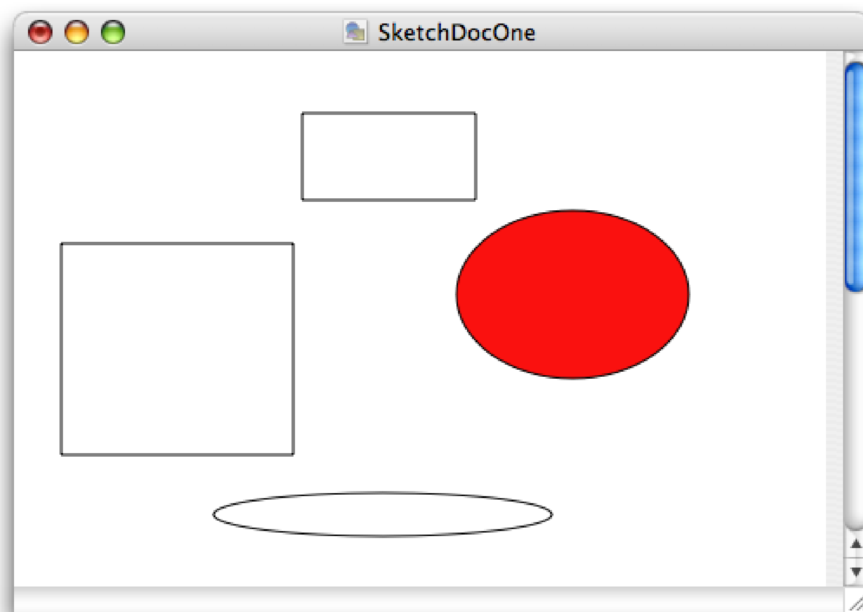
In this object model, documents are elements of the application object and graphics are elements of document objects, while the name of a document is a property of the document. For a script to access an object in the hierarchy, it must locate the object within its containment hierarchy. (In the application, the `orderedDocuments` array is a to-many relationship of the `NSApplication` object.)

Consider, for example, the following sample script:

```
tell app "Sketch" to set the x position of rectangle 1 of document "SketchDocOne"
 to 25
```

This script specifies a rectangle, in a document, in the application. If applied to the following Sketch document, it would set the horizontal (x) position for whichever of the two rectangle shapes is first in the document's ordered list of graphics.

Figure 1-2 Sketch window with graphics



A script can ask for `rectangle 1 of document "SketchDocOne"` without having to specify either `graphics` or `documents` (though they are part of the hierarchy shown above) because an AppleScript class definition implicitly specifies the relationships of its contained elements.

Sketch's object model uses inheritance for its graphics objects. All such objects (`rectangle`, `circle`, `image`, `line`, and `text area`) inherit from a common ancestor (`graphic`). Thus the term `graphic 1 of document "SketchDocOne"` might specify the same object as `rectangle 1 of document "SketchDocOne"`, depending on the ordering of the objects in the graphics array.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Note: Cocoa applications typically follow the Model-View-Controller (MVC) design pattern, where model objects encapsulate and manipulate the data used by the application. It is generally recommended that you support scriptability through your model objects, which tend to be more persistent. For more information, see "Concentrate Scriptable Behavior in Model Objects" (page 33).

Don't confuse the *AppleScript object model* with *MVC model objects*: the former is a structure you define; the latter are objects in your application that may or may not be part of your AppleScript object model.

Scriptability Information

A scriptable application supplies **scriptability information** that formally lays out the AppleScript object model for the application and maps it to application objects. This scriptability information does two things:

- It specifies the terminology available for use in scripts that target the application and supplies comments on the purpose and usage for that terminology.
- It provides information, used by AppleScript and by Cocoa, about how support for that terminology is implemented in the application.

  This information includes the KVC keys that Cocoa scripting uses to gain access to attribute and relationship information in the application's classes. Without these keys, scripts would not be able to manipulate properties of scriptable objects in the application's object model.

To uniquely identify terms in its scriptability information, an application uses constants called four-character codes (or Apple event codes). A **four-character code** is just four bytes of data that can be expressed as a string of four characters in the Mac OS Roman encoding. These codes are described in "Code Constants Used in Scriptability Information" (page 40).

Within an application's scriptability information, terminology associated with related functionality (for example, operations involving text, graphics, or databases) are generally collected into **suites**. Cocoa provides the predefined suites described in "Built-in Support for Standard and Text Suites" (page 19).

AppleScript uses the application's scriptability information, including four-character codes, to compile scripts and send Apple events to the application. Cocoa uses the information to interpret received Apple events and create script command objects to perform the specified operations.

Scriptability Information Formats

You define your application's scriptability information using one of two formats. The first is the **scripting definition** or **sdef** format. This is an XML-based format that describes a set of scriptability terms and the commands, classes, constants, and other information that describe the application's scriptability. The sdef format was introduced in Mac OS X version 10.2 and is used natively by Cocoa starting in Mac OS X version 10.4. In the sdef format, an application's scriptability information is contained in a single **scripting definition (or sdef) file**, with the extension `.sdef`. The word *sdef* can be used to describe either the format or a file in that format.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Important: You can read about additional refinements to sdef usage in Cocoa applications for Mac OS X v10.5 in the Scripting section of *Foundation Release Notes*.

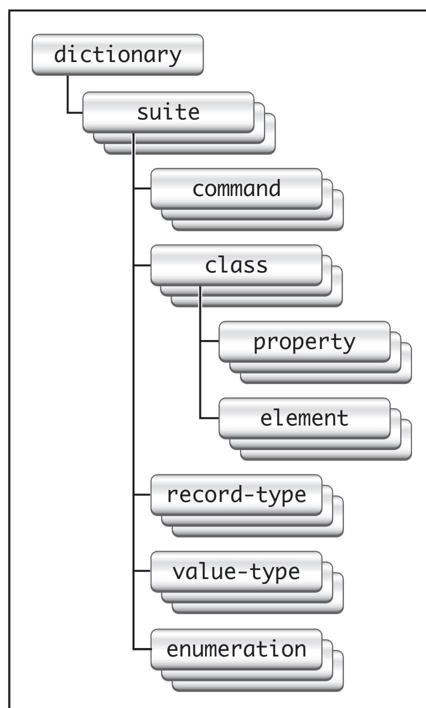
A second, older format uses property list files and is referred to as the **script suite** format. It supplies Cocoa and AppleScript with roughly the same information, in the form of a script suite file and a corresponding script terminology file:

- A **script suite file** describes scriptable objects in terms of their attributes, relationships, and supported commands, and has the extension `.scriptSuite`.
- A **script terminology file** provides AppleScript terminology—the English-like words and phrases a scripter can use in a script—for the class and command descriptions in the corresponding script suite file. Script terminology files have the extension `.scriptTerminology`.

Defining scriptability information with either of these formats is a bit like defining your application interface with nib files, though you work with text rather than a graphic editor. In both cases you provide information up front that Cocoa uses at specific times to create objects and provide support for the task at hand. The information from an sdef (or the older property list form) is loaded just once per application launch, as described in "Loading Scriptability Information" (page 21). The loaded information is then used as needed to create script command objects to perform scriptable operations.

A scripting definition file follows the XML format defined in the sdef man page and described in more detail in "Preparing a Scripting Definition File" (page 39). Figure 1-3 shows the main XML elements in an sdef file.

Figure 1-3 Structure of an sdef file



2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

As an example of specific sdef elements, Listing 1-1 shows the `graphic` and `rectangle` definitions from Sketch's sdef file. The `graphic` class defines several properties (such as `x position` and `y position`) that are inherited by other shape classes. The `rectangle` class adds an `orientation` property and specifies that the object responds to the `rotate` command, which is specific to rectangles.

Listing 1-1 Graphic and rectangle elements from Sketch's sdef file

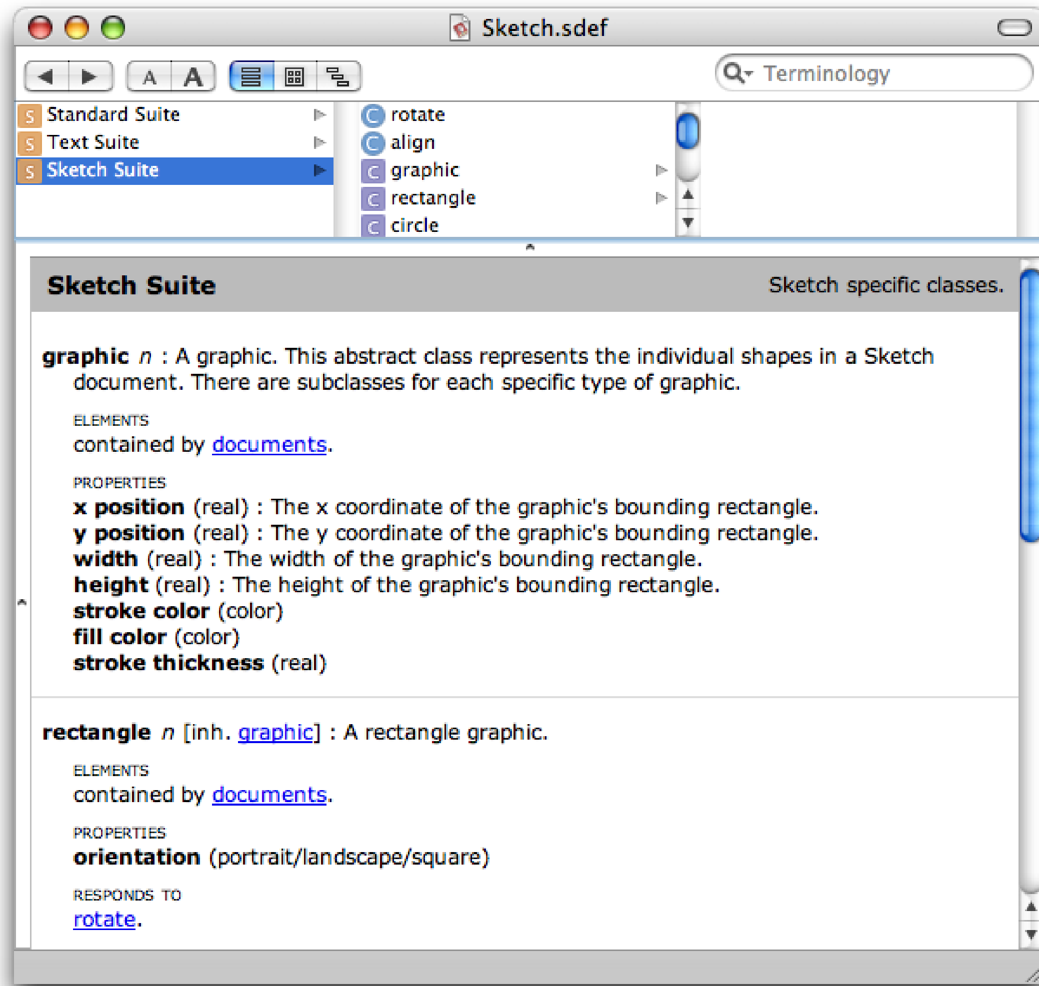
```
... (from the Sketch suite)
<class name="graphic" code="grph"
    description="A graphic. This abstract class represents the
        individual shapes in a Sketch document.
        There are subclasses for each specific type of graphic.">
    <cocoa class="SKTGraphic"/>
    <property name="x position" code="xpos" type="real"
        description="The x coordinate of the graphic's bounding rectangle."/>
    <property name="y position" code="ypos" type="real"
        description="The y coordinate of the graphic's bounding rectangle."/>
    <property name="width" code="widt" type="real"
        description="The width of the graphic's bounding rectangle."/>
... (some properties omitted)
</class>
<class name="rectangle" code="d2rc" inherits="graphic"
        description="A rectangle graphic.">
    <cocoa class="SKTRectangle"/>
    <property name="orientation" code="orin" type="orientation"/>
    <responds-to name="rotate">
        <cocoa method="rotate:"/>
    </responds-to>
</class>
```

Viewing Scripting Terminology

Users typically examine scripting terminology in a dictionary viewer to discover which features are scriptable and how to script an application. You can view the scripting terminology for a scriptable application with Script Editor or Xcode. Figure 1-4 shows the sdef file from the Sketch application in a dictionary viewer, with the graphic and rectangle classes visible.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Figure 1-4 An sdef displayed in a dictionary viewer



Double-clicking an sdef file in the Finder opens it in a dictionary viewer like the one shown above. Double-clicking an sdef file in an Xcode project similarly opens it in a dictionary viewer window. To view or edit the XML for the file, open the sdef file with any plain text editor; in Xcode, select the sdef file and choose File > Open As > Plain Text File.

For related information, see "Editing Scriptability Information" (page 114).

Built-in Support for Standard and Text Suites

A scriptable application typically supports certain standard AppleScript terms, such as the `count` and `make` commands and the `application` class. Cocoa provides built-in support for these terms in the **Standard suite**. It also provides command classes, including `NSCountCommand` and `NSCreateCommand`, to implement standard commands. In addition, classes such as `NSApplication` and `NSDocument` implement certain aspects of standard scripting support.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Cocoa scripting also provides support for the `get` and `set` commands, including implementation of the command classes `NSGetCommand` and `NSSetCommand`. These commands are not part of the Standard suite, but are considered built-in AppleScript commands. Allowing scripters to get and set the values of properties and elements of scriptable objects is a key part of making an application scriptable.

> **Note:** Though "Standard suite" is the current name, you may see it referred to as the "Core suite" (and you may see filenames that include "Core"). That usage dates back to the early days of AppleScript.

The Standard suite defines the following AppleScript commands: (for all classes) `copy`, `count`, `create`, `delete`, `exists`, and `move`; (for documents and windows) `print`, `save`, `close`. Note that there is no default implementation for the `print` command—see "Print" (page 105) for information on how to support printing.

To support text-related classes such as `rich text`, `word`, and `paragraph`, Cocoa implements the **Text suite**.

Cocoa's built-in support for the Standard and Text suites is described in more detail in "Use the Document Architecture" (page 35) and "Access the Text Suite" (page 36). To include it in your application, you follow the steps described in "Turn On Scripting Support in Your Application" (page 34).

Built-in Support for Basic AppleScript Types

Cocoa scripting provides built-in support for basic AppleScript types and automatically associates them with appropriate Cocoa data types, as shown in Table 1-1. You can use these types without declaring them in your sdef file.

Table 1-1 Support for basic AppleScript types

AppleScript type	Cocoa type	Code
`any`	`NSAppleEventDescriptor`	`"****"`
`boolean`	`NSNumber`	`"bool"`
`date`	`NSDate`	`"ldt "`
`file`	`NSURL`	`"file"`
`integer`	`NSNumber`	`"long"`
`location specifier`	`NSPositionalSpecifier`	`"insl"`
`number`	`NSNumber`	`"nmbr"`
`point`	`NSData` containing a QuickDraw `Point`	`"QDpt"`
`real`	`NSNumber`	`"doub"`

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

AppleScript type	Cocoa type	Code
`record`	`NSDictionary` This type is used as the type of the `properties` property for the `item` class, and the type of the `with properties` parameters of the `duplicate` and `make` commands. Declaring something to be of type `record` doesn't provide enough information to convert Apple event record descriptors to `NSDictionary` objects, so various Cocoa scripting command classes have special code to handle that situation. In your code, instead of using the `record` type, you should declare specific record types (see "Record-Type Elements" (page 50)) and use those for `class` properties and `command` parameters.	`"reco"`
`rectangle`	`NSData` containing a QuickDraw `Rect`	`"qdrt"`
`specifier`	`NSScriptObjectSpecifier`	`"obj "`
`text`	`NSString` Internally, Cocoa scripting always uses Unicode text when converting to get information from or add it to an Apple event.	`"ctxt"`
`type`	`NSNumber`	`"type"`

Loading Scriptability Information

When an application first needs to work with scriptability information, Cocoa scripting uses a global instance of `NSScriptSuiteRegistry` to load information from the application's sdef file:

- For each class described in the sdef file, it instantiates a class description object (`NSScriptClassDescription`) that stores information about objects of that type.

  This information includes the KVC keys that Cocoa scripting uses to access values of the application's scriptable objects.

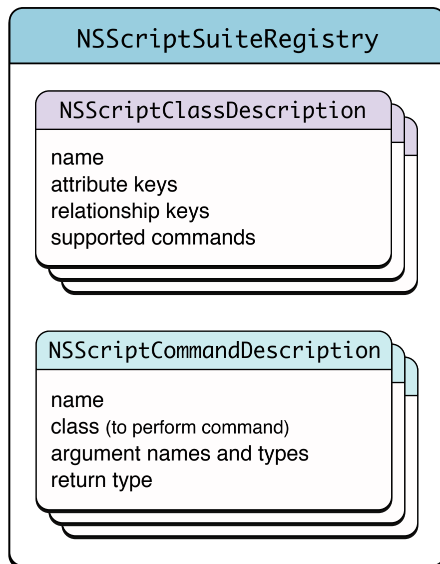
- For each script command described in the sdef file, it registers a handler routine for Apple events that specify that command.

  It also instantiates a command description object (`NSScriptCommandDescription`) that stores information such as the name of the scripting command class to instantiate to perform the command, as well as the command's arguments, and return type (if any).

Figure 1-5 shows the loaded class and command descriptions for an application. Cocoa uses the information to create instances of command classes and descriptions of application objects that it needs to handle Apple events received by the application.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Figure 1-5 An application's loaded scriptability information



Important: You don't typically instantiate scripting commands in your application—you list the commands your application supports in your sdef file and Cocoa instantiates them when your application receives the corresponding Apple events.

See "How Cocoa Applications Handle Apple Events" (page 101) for information on how Cocoa handles Apple events other than those that are part of your application's scriptability information.

Application scriptability information automatically includes information to support the basic AppleScript types listed in Table 1-1 (page 20) and to make the `get` and `set` commands available to all applications.

Reliance on Key-Value Coding

Key-value coding (KVC) is a mechanism for accessing object properties indirectly by key. A **key** is just a string that identifies a property, such as `"xPosition"` for the horizontal coordinate of a graphic object (shown in Listing 1-1 (page 18)). The KVC API provides a generic way to query an object for the values of its keys and to set new values for those keys.

Cocoa scripting relies on KVC for the following purposes:

- Command objects use KVC to find the specified scriptable objects on which to operate.
- For many commands, Cocoa uses KVC to access properties of the specified objects to get or set their values.

As you design the object classes for your application, you also define keys for their scriptable properties. Your application provides these keys in `class` definitions in its sdef file. Then, when you implement accessor methods (or declare instance variables) that correspond to these keys, you make it possible for Cocoa scripting and KVC to get and set the corresponding values.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Default naming for keys is described in "Provide Keys for Key-Value Coding" (page 33). For information on naming accessor methods, see "Maintain KVC Compliance" (page 54).

Interaction With Cocoa Bindings and Core Data

Cocoa scripting, Cocoa bindings, and Core Data are three development technologies available in Mac OS X that rely on key-value coding:

- Cocoa scripting, described throughout this document, provides support for creating scriptable applications.
- The Core Data framework provides generalized and automated solutions to common tasks associated with object life-cycle and object graph management, including persistence. It is described in detail in *Core Data Programming Guide*.
- Cocoa bindings provides a means of keeping model and view values synchronized, without having to write a lot of "glue code," such that a change in one is reflected in the other. It is described in detail in *Cocoa Bindings Programming Topics*.

As noted, all three of these technologies rely on KVC, so you will gain by making your application KVC-compliant. Cocoa bindings and Core Data also make use of key-value observing, but Cocoa scripting does not.

While these technologies are not closely tied together, here are some rules of thumb that may aid in combining their use:

- Core Data and Cocoa bindings: These technologies neither aid nor interfere with each other, so you can mix them as needed for your application.
- Core Data and Cocoa scripting: Because Cocoa scripting deals in ordered to-many relationships, and Core Data deals in unordered to-many relationships, you must do extra work to expose the to-many relationships of your managed objects as scriptable element classes. For each managed-object to-many relationship that you want to make scriptable, you can add an unmanaged, derived, to-many relationship solely for access by Cocoa scripting. For information about how to implement these derived to-many relationships, see "Indexed Accessor Patterns for To-Many Properties" in Key-Value Coding Accessor Methods in *Key-Value Coding Programming Guide* and "Getting and Setting Properties and Elements" (page 53) in this document.
- Cocoa scripting and Cocoa bindings: Again, these technologies neither aid nor interfere with each other. If you're using bindings but not Core Data, you probably have ordered relationships, which will work with Cocoa scripting.

For related information, see "Interaction With Key-Value Observing" (page 56).

Scriptability and Undo

Scriptable applications should generally support undo of scripted operations as they would any user-initiated operation. That is, after a script causes the application to perform one or more operations, a user should be able to sequentially undo the operations in the normal manner.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Cocoa scripting provides no special support for undo, but neither does it interfere with normal undo operations. Applications that take advantage of the Model-View-Controller paradigm and Cocoa's key-value coding mechanism, as scriptable applications are designed to do, are well-positioned to support both scriptability and undo.

For more information on supporting undo, see *Undo Architecture*.

Snapshot of Cocoa Scripting

The work done by your scriptable application is divided between the application and Cocoa scripting. Your application implements classes and methods that perform its scriptable operations; it also provides information that describes its scriptability. Cocoa receives Apple events and uses your scriptability information to interpret them and perform the specified operations, calling on your code to do the actual work.

Here is a summary of how this process works:

1. The application defines scriptability information (in an sdef file, or in the older style script suite and script terminology files) that includes both the terms a scripter can use and the application information for supporting those terms. This information typically includes the Standard suite (implemented by Cocoa scripting), which supports standard AppleScript commands and classes.

   The application defines classes for the scriptable objects it supports and provides keys for their scriptable properties and elements. It also defines additional script command classes if it has scriptable operations that can't be performed by one of the standard command classes provided by Cocoa.

   Scriptability is generally provided through the application's model objects (in terms of the Model-View-Controller paradigm).

2. The application is key-value coding (KVC) compliant in naming the instance variables or accessor methods for the scriptable properties and elements of its scriptable classes.

3. For each scriptable class, the application implements an object specifier method, which locates a scriptable object of that type within the application's containment hierarchy.

4. The application's `Info.plist` file has entries that activate Cocoa scripting and specify an sdef file, as shown in "Turn On Scripting Support in Your Application" (page 34).

5. When it is first needed, Cocoa loads the application's scriptability information and automatically registers event handlers for the supported commands.

6. When the application receives an Apple event for a registered command, Cocoa instantiates a script command object containing all the information needed to identify the application objects on which the command should operate. All command objects use KVC to locate the specified scriptable objects to operate on.

   Cocoa then executes the command, which sends messages to the appropriate application objects to perform the work. For many commands, Cocoa uses KVC to get or set values of the specified objects.

7. When a command needs to return a value, Cocoa scripting packages the information in a reply Apple event and returns it.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

If an error occurs while executing the command, Cocoa returns the error information (including any information added by the application) in the reply Apple event. For details, see "Error Handling" (page 75).

8. If a command requires asynchronous processing (such as the gathering of information through a sheet), the application can suspend it, so that the application doesn't receive additional Apple events during processing. For details, see "Suspending and Resuming Apple Events and Script Commands" (page 109).

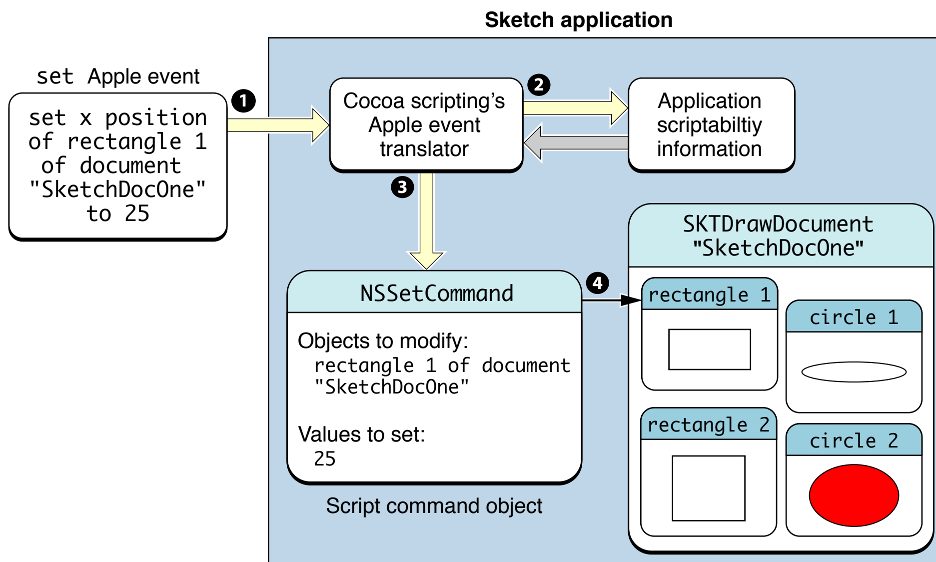
A Real World Scripting Example

To track a scripting command from the execution of an AppleScript script to the operation it performs in the targeted application, consider again the following one-line script, which attempts to set a value in a document of the Sketch application:

```
tell app "Sketch" to set the x position of rectangle 1 of document "SketchDocOne"
 to 25
```

When this script is compiled against Sketch's scriptability information and executed, it results in an Apple event being sent to the Sketch application. The Apple event encapsulates the `set` command and specifies an operation on an object in the `"SketchDocOne"` document shown in Figure 1-2 (page 15). Figure 1-6 illustrates the actions that result when the application receives the `set` Apple event.

Figure 1-6 A Cocoa application responding to an Apple event



Here is a description of the steps shown in this figure:

1. The application receives the Apple event specifying a `set` command.

2. The **Apple event translator**, a part of Cocoa scripting, uses scriptability information supplied by the application to evaluate the Apple event. This information is pictured in more detail in Figure 1-5 (page 22).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

3. The translator creates an instance of an `NSSetCommand` script command and initializes it with the information need to perform the command:

   - The direct parameter of the `set` command specifies the object or objects to set the value for (in this case, the first rectangle), and becomes the receivers specifier in the `NSSetCommand` object.
   - The `to` parameter provides the value to set (in this case, `25`) and becomes the "Value" argument in the `NSSetCommand` object.

   These values are stored in the command object as an argument dictionary. For information on how your application works with that information, see "Script Command Components" (page 72).

4. When the `set` command is executed, it uses KVC to locate the specified object or objects. It also relies on KVC to set the specified value. For additional detail, see "Getting and Setting Properties and Elements" (page 53).

   Some commands return a value, but the `set` command does not.

As this example and the information in "Snapshot of Cocoa Scripting" (page 24) show, an application can support the `set` command with a relatively modest effort: it modifies its `Info.plist` file to indicate it is scriptable and to specify its sdef file; it provides command and class information in the sdef file; and it maintains KVC compliance for scriptable properties in its scriptable classes.

If the application also implements an `objectSpecifier` method for each of its scriptable classes, it can support the `get` command as well. Cocoa scripting uses KVC to get the specified value from the application and package it up in a return Apple event.

As a result, an application can rather quickly supply a great deal of scriptability just by supporting `get` and `set` access to properties of a few key objects.

Current Limitations of Cocoa Scripting Support

There are some scripting-related features for which Cocoa scripting currently provides little or no support.

- Sending individual Apple events. Applications might want to send events to other scriptable applications or to themselves to obtain data or services or to support recording (see next item).

  However, Cocoa applications are free to use C functions from the Apple Event Manager. So, for example, your application could use an instance of `NSAppleEventDescriptor` to assemble the information for an Apple event, invoke the `aeDesc` method to get an Apple event data structure, and use that structure with the C functions described in *Apple Events Programming Guide* to send the Apple event.
- Recording. A recordable application sends itself Apple events, which can be assembled by the Script Editor application into a script that "records" the user's actions (when recording is turned on in Script Editor).

Cocoa scripting does not currently enforce the `read/write` status specified in an sdef, so marking a property as read-only does not ensure that it cannot be written. (Read/write access is described in "Property Elements" (page 45).)

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Designing for Scriptability

This chapter provides high-level checklists for designing a new scriptable Cocoa application and adding scriptability to an existing application.

Whether you're designing a Cocoa application from scratch or adding a new feature to an existing application, you go through a standard set of design steps, tailored to your experience, the systems you work with, and the project at hand. The process is no different when you're designing for scriptability.

The following steps provide an outline for your design process, but are not intended to be comprehensive. However, links are provided to additional design information in this and other documents.

Designing a New Scriptable Application

To design a new scriptable Cocoa application, consider using the following steps:

1. For your overall application design, work with the Model-View-Controller (MVC) design pattern, which is widely followed in Cocoa applications. MVC is described in detail in *Cocoa Fundamentals Guide*.

   Good scripting support generally calls for scripting the model, rather than the view layer. For more information, see "Concentrate Scriptable Behavior in Model Objects" (page 33).

   Cocoa bindings and Core Data also work with the MVC design pattern. See "Interaction With Cocoa Bindings and Core Data" (page 23) for more information.

2. Think about scripting very early in your design.

   When you first start defining the requirements for your application, consider the kinds of features you'll want to make scriptable and how users might script those features to automate operations with your application. This can help provide insight into the overall goals for your application.

   You'll want the terms you specify to look natural in AppleScript's English-like grammar. Scripting terms won't necessarily be an exact match for the terminology you use within your application's code.

   Technical Note TN2106, "Scripting Interface Guidelines," supplies valuable information on how to provide a clean and consistent scripting interface for your application.

3. Work out the design for your AppleScript object model—the objects scripters will use in their scripts, and the inheritance and containment relationships between them.

   This includes identifying the scriptable properties of your scriptable classes and providing a set of keys to identify them. (You'll use this information later, in your sdef file, to provide the keys Cocoa scripting uses with key-value coding. However, in many cases, you won't need to specify keys explicitly—Cocoa can determine them automatically by applying its default rules to the property names.)

   For more information, see "Provide Keys for Key-Value Coding" (page 33).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

4. Determine which of the standard AppleScript commands your application will support. Cocoa provides support for most of the standard scripting commands, including `close`, `count`, `delete`, `duplicate`, `exists`, `get`, `make`, `move`, `open`, `print`, `quit`, `save`, and `set`, but your application has to do some work too.

   Scripters expect your application to support whichever of these commands make sense for it, and to make the commands work in standard ways. By meeting these expectations, you gain two kinds of leverage:

   - You can take advantage of scripters' existing knowledge, so that your application's scriptability is easier for them to understand and use (meaning fewer complaints and support calls).
   - Supporting a few commands can provide a lot of scriptability—for example, if you provide access through the `get` and `set` commands to five properties in each of the two most important classes in your application, that's 20 scriptable operations—2 commands times 5 properties times 2 classes. In many cases, support for the `get` and `set` commands requires relatively little coding effort.

5. To support application-specific operations that can't be handled by one of the Cocoa scripting command classes, define new script command subclasses for those operations.

   Keep in mind that you can often accomplish a goal with an existing command. For example, rather than define a new `rename` command, you can use the existing `set` command to set the `name` property of an object to the new name.

   For more information, see "Script Commands" (page 71).

6. Create the actual scripting definition file (sdef) for your application.

   This is listed as a design step because it is possible to define an sdef and actually compile scripts against it in a test or skeleton application before you've written code to support your scriptability. You can even let scripters in your target audience work with your application's terminology and provide feedback.

   Note: To test changes you make to an sdef file, you must quit the application, rebuild, and relaunch it to pick up the changes. In addition, the Script Editor application caches scriptability information, so you may need to quit and relaunch Script Editor as well.

   Doing this step early can help immensely in refining your scripting support and ensuring that it is user-friendly for scripters. In addition, your AppleScript dictionary can serve as a useful design document for the entire application.

   Although you can wait to write your scripting terminology until you implement the application, writing it first will help to ensure consistency and usability. And during implementation, you don't need to implement support for your terminology all at once—you can do it incrementally as you become more familiar with the process.

   For information on creating an sdef file, see "Preparing a Scripting Definition File" (page 39).

7. Design the application classes that will support scriptability. Make sure these classes support the scripting object model you have defined, including both inheritance and containment relationships.

   You must name the accessor methods for scriptable properties of these classes to match the keys you defined earlier, as described in "Maintain KVC Compliance" (page 54).

8. Plan for testing.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Your testing should include creation and regular execution of AppleScript scripts that exercise the application's functionality. The ability to automate your testing and to more easily perform regression testing is one of the big gains of making your application scriptable.

Performance represents another important area of testing for most applications. For a general introduction, see *Getting Started with Performance*. For issues specific to scriptable applications, see "Performance Issues for Scriptability" (page 93).

And of course, you should also plan for unit testing or other kinds of testing that you customarily use.

For issues to cover in your test plan, see "Scriptability Test Plan" (page 87).

Adding Scriptability to an Existing Application

To add scriptability to an existing Cocoa application, consider using the following steps. (Some steps are abbreviated, where they repeat those found in "Designing a New Scriptable Application" (page 27).)

1. If time allows, and especially if there are other reasons to redesign the application, consider a major redesign, following the steps described previously for designing a new scriptable application.

2. Experiment with creating a scripting definition file for your application.

As noted previously, you can compile scripts against your terminology before you've written any code. That helps you identify the scripting objects and terminology you want to provide for scripters (your AppleScript object model), as well as the underlying application information you need to expose to support that terminology. It also helps determine the scale of the effort to make the application scriptable.

3. In classes containing information you want to make scriptable, make use of any instance variables or accessor methods that are already key-value coding compliant.

If not, you can add KVC-compliant accessors to support scriptability. One convenient way to do this is through the use of an Objective-C category. This approach is most appropriate in the case where you are primarily making existing properties and elements scriptable. It can also be useful when you don't have free access to the existing code.

4. Consider designing scriptable "helper" classes to implement the object hierarchy you expose to scripters. These classes can maintain references to application objects that contain the actual information scripters will be working with.

You can define helper classes from scratch, adding properties and relationships as needed to support your AppleScript object model. To access the underlying data, you have to invoke methods in existing classes—this is where you're likely to find the most complexity in retrofitting an existing application.

This approach is most appropriate in the case where you need to pull together properties from multiple application classes to represent one AppleScript object model class, or where a category won't work because you need to add new instance variables.

5. Determine which of the standard AppleScript commands your application will support and plan new script command subclasses for any additional commands that are needed.

6. Plan for testing, using both AppleScript scripts and other testing options available to you.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Implementing a Scriptable Application

This chapter lists the key steps for implementing a scriptable Cocoa application, with links to more detailed information where necessary.

Implementation Guidelines

Once you have completed one of the design phases described in "Designing for Scriptability" (page 27), you use steps like the following to implement a scriptable Cocoa application:

1. For a new application, implementing scriptability should be an integral part of creating the application. That is, you'll be creating scriptable classes, adding scripting accessor methods, and so on, as you implement other parts of the application.

   When you're adding scriptability to an existing application, there is more opportunity for a staged or incremental approach. That is, you may want to test your approach by retrofitting one or a small number of classes, before extending it to the entire application.

   In either case, your work should include milestones to test each phase of scriptability, as spelled out in your test plan.

2. Cocoa follows the Model-View-Controller (MVC) design pattern, where model objects encapsulate and manipulate the data used by the application. You should generally support scriptability in your model objects, which tend to be more persistent. Although there may be some cases where you want to allow scripting of your view objects, keep in mind that scripts that operate on the user interface tend to be fragile, and they can also be less efficient.

   For more information, see "Concentrate Scriptable Behavior in Model Objects" (page 33).

3. Provide an sdef file with the scriptability information for your application.

   For more information, see "Supply a Scripting Definition File" (page 32).

4. Maintain key-value coding (KVC) naming compliance for instance variables or accessor methods for scriptable properties and elements, based on the keys in your sdef file. Cocoa scripting support relies on this naming compliance.

   For details, see "Maintain KVC Compliance" (page 54).

5. Include the sdef file in the Xcode project for your application, as described in "Add the Scripting Definition File to Your Xcode Project" (page 34).

6. Modify your application's `Info.plist` file to turn on Cocoa scripting support and identify your sdef file, as described in "Turn On Scripting Support in Your Application" (page 34).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

7. Cocoa implements the `NSScriptCommand` class and a number of specific subclasses, such as `NSDeleteCommand`, `NSGetCommand`, `NSMoveCommand`, and `NSSetCommand`. However, for some commands implemented by Cocoa, your application may need to provide a different implementation.

   For information on how to do this, see "Steps for Implementing a New or Modified Script Command" (page 78).

8. Implement `objectSpecifier` methods for scriptable classes in your object model. These methods describe the object and point to its parent in the object containment hierarchy (with the application object generally serving as the outermost container). They are invoked by an instance of `NSGetCommand` when it works with your application to obtain the requested information.

   If you've created helper classes to add scriptability to an existing application, these classes also need to implement object specifier methods.

   For more information, see "Implement Object Specifier Methods for Scriptable Classes" (page 35).

9. Implement any new script command subclasses your application requires.

   Many applications provide unique capabilities, such as rotating an image or converting between two audio formats. To make these features scriptable, you may need to define new script command classes that are subclasses of `NSScriptCommand` or one of the other command classes provided by Cocoa.

   For more information, see "Subclasses for Standard AppleScript Commands" (page 98) and"Script Commands Overview" (page 71).

10. To take advantage of Cocoa scripting support that works with document and window classes, your application should use the Cocoa document architecture.

    For more information, see "Use the Document Architecture" (page 35).

11. To take advantage of Cocoa scripting support that works with text, your application can take advantage of Cocoa's built-in support.

    For more information, see "Access the Text Suite" (page 36).

12. Throughout the implementation process, test your application according to the test plan you developed.

    For tips and suggestions, see "Testing, Debugging, and Performance" (page 87).

Supply a Scripting Definition File

Every scriptable application must provide a definition of its scriptability information—the terminology available for use in scripts that target the application, as well as the implementation information used to support that terminology. This information includes a set of keys for the scriptable properties accessible in the application through key-value coding (described in "Provide Keys for Key-Value Coding" (page 33)).

If you developed an sdef file during the design phase, you've already completed this step. If not, see "Preparing a Scripting Definition File" (page 39) for a description of the steps you take to create an sdef file and add scriptability information to it.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

For information on working with the older scriptability format, see "Script Suite and Script Terminology Files" (page 117).

Concentrate Scriptable Behavior in Model Objects

The Model-View-Controller (MVC) paradigm is one of the central design patterns for Cocoa applications. MVC assigns objects in an application to one of three roles and recommends that you try to maintain a separation among objects of different roles.

- Model objects encapsulate the data and basic behaviors of the application; ideally, they have no explicit connection to the user interface.
- View objects present data to the user; they know how to display and possibly edit data, but typically do not encapsulate any data that is not specific to displaying or editing.
- Controller objects act as an intermediary, coordinating the exchange of data between the model and view objects.

Generally, the objects that you make scriptable should be model objects. The most efficient way for a script to perform a task generally involves modifying the model and is often not the same as the best way for a user to do the same task through the user interface (or view). This is consistent with how AppleScript works, and Cocoa accordingly gears its scripting support to the model layer.

A script should not require the user's involvement, unless it is intended more as a macro than as a form of batch processing. In a macro-like script, the user must prepare things for the script (such as opening a window and creating or selecting certain objects), and then invoke it. If you anticipate that your application will be scripted for this purpose, you may want to provide scriptable behavior to the appropriate nonmodel objects, such as windows and selections. If so, be sure to confine your scripting support in nonmodel objects to those specific purposes.

MVC is described in more detail in Cocoa Design Patterns in *Cocoa Fundamentals Guide*.

Provide Keys for Key-Value Coding

Recall that key-value coding (KVC) is a mechanism for accessing object properties indirectly by key, where a key is just a string that represents a property name (such as `"xPosition"` for the horizontal coordinate of a graphic object). Cocoa scripting relies on KVC, both for finding the specified objects for a command to operate on and for getting and setting values in the specified objects.

Your application provides keys for its scriptable properties and elements in `class` definitions in its sdef file. The property and element names themselves serve as keys, unless you specify a different key explicitly. Cocoa scripting adjusts the key names as necessary according to the following rules, which are consistent with standard Cocoa naming conventions for accessors:

- For single-word property names, the name becomes the key. For example, an sdef property named `"width"` would result in a key of `"width"`.
- For multiple-word property names, Cocoa capitalizes each word of the name except the first word, then removes any spaces. For example, an sdef property named `"desktop position"` would result in a key of `"desktopPosition"`.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

- For element names, Cocoa specifies a key by making a plural from the name. For example, "word" results in a key of "words" and "document" results in "documents".

You can override the default naming conventions to specify arbitrary key values where necessary. For example, suppose you want a scripter to be able to use color in a script, but your application refers to the underlying property as foregroundColor (as in the NSTextStorage class). You can specify "foregroundColor" as the key for the "color" property by adding a cocoa key entry to the sdef property definition:

```
<property name="color" code="colr" ...
    <cocoa key="foregroundColor"/>
```

To support getting and setting scriptable properties and elements in your application, you define accessor methods that match the keys in your sdef, as described in "Maintain KVC Compliance" (page 54). For additional information on default naming and working with keys, see "Cocoa Elements" (page 51).

Add the Scripting Definition File to Your Xcode Project

Once you have created an sdef file, you'll need to add it to the Xcode project for your application. Place the file in the project folder (or other appropriate location) and use Project > Add to Project, which also lets you add the sdef file to application targets. Adding the sdef file to the project automatically adds it to the Copy Bundle Resources build phase, so it will be included in the application.

> **Note:** Double-clicking the sdef in the project opens it in a dictionary viewer window, where you can examine its terminology. To view or edit the XML for the file, select the sdef file and choose File > Open As > Plain Text File.

Turn On Scripting Support in Your Application

To turn on Cocoa's built-in scripting support, you add the following key to your application's Info.plist file:

```
    <key>NSAppleScriptEnabled</key>
    <string>YES</string>
```

To provide your application's scriptability information through an sdef file, you add a second key to the property list to specify the sdef file. Here's the entry for the Sketch application:

```
    <key>OSAScriptingDefinition</key>
    <string>Sketch.sdef</string>
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Important: To take advantage of Cocoa's built-in support for Standard and Text suites, your sdef file must include whichever classes, commands, and other information from those suites it will use. You can obtain this information as described in "Suite Elements" (page 42).

If your application uses the script suite and script terminology format, you automatically gain access to Cocoa's built-in terms by including the `NSAppleScriptEnabled` key, as shown above.

Implement Object Specifier Methods for Scriptable Classes

An object specifier locates a scriptable object or objects within the containment hierarchy in which they reside.

When a script statement targets an application, the application may need to return a reply. For example, the result of a `get` command is an object or a list of objects. When Cocoa returns these objects in the reply Apple event, it does not return pointers to Objective-C objects, it returns object specifiers.

To obtain the object specifiers, Cocoa sends `objectSpecifier` messages to the objects to be returned. Therefore, for any class of object that is part of your containment hierarchy of scriptable objects, you must implement the `objectSpecifier` method. This method is declared in `NSScriptObjectSpecifiers`, a category on `NSObject` that implements a version that just returns `nil`.

For more information, including code examples, see "Object Specifiers" (page 61).

Use the Document Architecture

The `NSApplication`, `NSDocument`, `NSDocumentController`, `NSWindow`, and `NSWindowController` classes form the basic structure for the Cocoa document architecture. Together with the terminology defined in the Standard suite, these classes provide direct support for the standard AppleScript document scripting model, including classes such as `application`, `document`, and `window`. When you use these classes to implement a document-based application, that application automatically supports a number of scripting features.

For example, the `NSApplication`, `NSDocument`, and `NSWindow` classes are KVC-compliant for standard scriptable properties. `NSApplication` provides methods for accessing the application's documents as an ordered list. The `NSDocument` class provides support for the `close`, `print`, and `save` commands by implementing the `handleCloseScriptCommand:`, `handlePrintScriptCommand:`, and `handleSaveScriptCommand:` methods. `NSWindowScripting` also implements default versions of these methods, which in many cases pass control to the window's document. It also implements methods for scriptable access to window attributes, such as the close box, title bar, and so on.

Applications that take advantage of Cocoa's document architecture put themselves in a better position to support scripting generally. A document in Cocoa applications typically owns and manages one or more model objects of the application. It therefore provides a hub for scripted access to the model objects in your application, which are the ones that load and save data.

Table 3-1 lists Cocoa classes that correspond to AppleScript classes in the Standard suite, along with attributes and relationships (properties and elements) used by those classes.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Table 3-1 Standard suite attributes and relationships

Objective-C and AppleScript class	Attributes (script term, if different)	Relationships
`NSObject` Implements the `item` AppleScript class. For any scriptable Objective-C class that inherits from `NSObject`, the AppleScript class it implements inherits from the `item` class (and inherits the `class` property and the `properties` property).	class name (`class`), properties	
`NSApplication` Implements the `application` AppleScript class.	name, active flag (`frontMost`), version	documents, windows (both accessible as ordered relationship)
`NSDocument` Implements the `document` AppleScript class.	location of the document's on-disk representation (`path`); last component of filename (`name`); edited flag (`modified`)	
`NSWindow` Implements the `window` AppleScript class.	title (`name`); various binary-state attributes: closeable, floating, miniaturized, modal, resizable, titled, visible, zoomable	document

Access the Text Suite

The Text suite defines terminology that allows scripts to request or select textual elements at different levels of granularity: character, word, paragraph, or entire body of text. The `NSTextStorage` class, provided by the Application Kit, defines a corresponding set of methods for getting and setting scriptable properties of `NSTextStorage` objects.

To gain access to this text scripting support, use an `NSTextStorage` object as the content for one of your scriptable classes. The TextEdit sample code (available at `<Xcode>/Examples/AppKit/TextEdit`) demonstrates a scriptable application that supports text scripting, as well as scripting support for printing.

> **Note:** The TextEdit sample does not use the sdef file format—it supplies its scriptability information using the older style script suite and script terminology files.

Table 3-2 lists Cocoa classes for working with text, along with attributes and relationships used by those classes.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Table 3-2 Text Suite Attributes and Relationships

Class	Attributes (script term, if different)	Relationships
`NSTextStorage`	font name (`name`), font size (`size`), foreground color (`color`)	attribute runs, characters, paragraphs, text, words

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Preparing a Scripting Definition File

This chapter describes the steps you take to create a new scripting definition file and add scriptability information for your Cocoa application.

Important: You can read about additional refinements to sdef usage in Cocoa applications for Mac OS X v10.5 in the Scripting section of *Foundation Release Notes*.

If your scriptable application will run in versions of the Mac OS prior to version 10.4, it must supply scriptability information in script suite and script terminology files. For more information, see "Converting and Updating Scriptability Information" (page 112) and "Script Suite and Script Terminology Files" (page 117).

This chapter uses examples from the sdef file `Sketch.sdef` from the Sketch sample application, available from Apple.

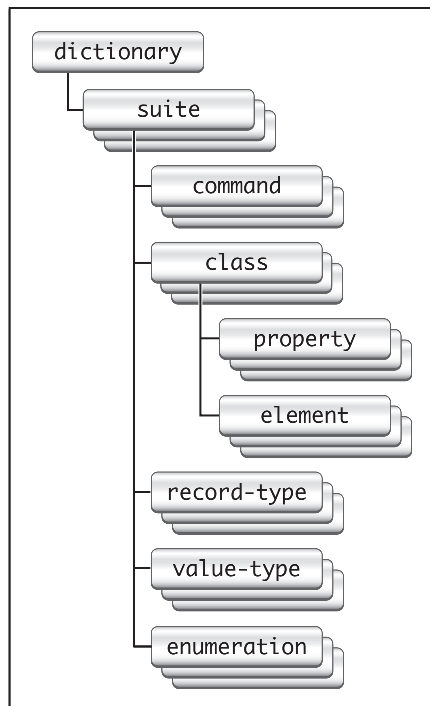
For definitions of terms such as *property* and *element* used throughout this chapter, see the "Glossary" (page 133).

Structure of a Scripting Definition File

A scripting definition file provides a statically stored representation of a set of scriptability terms and the classes, constants, and other information used to support an application's scriptability. Here is another look at the organization of the main XML structures used in the sdef format. All valid sdef files follow this structure, differing primarily in the contents of the application-specific `suite` elements.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Figure 4-1 Structure of an sdef file, revisited



Note: The sdef information in this chapter is provided to demonstrate the general construction of sdefs for scriptable Cocoa applications. It is not meant to be an exhaustive description of the sdef format.

Code Constants Used in Scriptability Information

Four-character codes (or *Apple event codes*, or simply *codes*), are used throughout AppleScript to identify various kinds of information. A **four-character code** is just four bytes of data that can be expressed as a string of four characters in the Mac OS Roman encoding. For example, `'docu'` is the code that specifies the `document` class. The codes defined by Apple and the header files in which they are defined are described in "Apple Event Constants" in Building an Apple Event in *Apple Events Programming Guide*. They are also documented in *Apple Event Manager Reference* and *AppleScript Terminology and Apple Event Codes Reference*.

You use codes in your sdef file to identify scriptable commands, classes, properties, elements, and other items your application supports. When choosing a code, use an existing code from one of the Apple headers for a standard object—for example, use `'capp'` for an application object. If you need to refer to the value in your code, you can use the corresponding Apple constant `cApplication`, but in the sdef you simply use the string literal (`'capp'`). For new codes you define, you can define a corresponding constant, if necessary.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Important: If you use a term (or a code) that is defined in AppleScript's built-in terminology or in the Standard suite, you must match it with the code (or the term) defined for it.

Apple reserves all values that consist entirely of lower-case letters and spaces. You can generally avoid conflicts with Apple-defined codes by including at least one upper-case letter when choosing your own codes.

You should always use a given code with the same term, and vice versa. While it is possible to use a term-code pair for different items (for example, to have a property with the same name as a class), this won't work for all elements, and you may want to avoid such reuse. Your codes do not have to be unique with respect to those used in other applications; in fact, as noted above, it is recommended that you use Apple-defined values for standard items.

Features Common to Many Scripting Definition File Elements

Except for the `dictionary` element (described below), most XML elements in an sdef have a name, a code (or pair of codes), and a description. They may also include documentation and implementation information. XML elements with the `cocoa` tag (described in "Cocoa Elements" (page 51)) contain a particular kind of implementation information for Cocoa scripting. Some elements can be marked as hidden, meaning they won't be displayed when the dictionary is viewed in Script Editor or Xcode, although they will be implemented. Elements can also include synonyms, which define an alternate term or code for the main element.

An XML element in an sdef may refer to any other element in the sdef, regardless of whether they are defined in the same `suite` element. For example, the `word` class defined in the Text suite can be referred to in any other XML elements in any other suites in the same sdef file.

High-level XML Elements

All sdef files used to supply Cocoa scriptability information contain the high-level XML elements described in the following sections.

XML Version and Document Type Definition

The first two lines of an sdef file, shown in the following listing, specify the XML version and DTD (document type definition) file for the sdef format. You don't change these lines.

Listing 4-1 Version and document type in an sdef file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dictionary SYSTEM "file://localhost/System/Library/DTDs/sdef.dtd">
```

Dictionary Element

The root XML element in an sdef file is a `dictionary` element. A dictionary defines all the scriptability information for one application. Every sdef file you create should contain one dictionary definition. You typically name the dictionary after the application.

A `dictionary` element contains one or more `suite` elements. Most Cocoa applications include Cocoa's default suites (the Standard and Text suites), as well as one or more suites of their own scriptability information.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

The following listing shows the format for a `dictionary` element.

Listing 4-2 A dictionary element from an sdef file

```
<dictionary title="Sketch Terminology">
    <!--
        Contains one or more suite XML elements.
    -->
</dictionary>
```

Suite Elements

A suite is a collection of related scriptability information. For example, Cocoa's Standard suite contains default commands, enumerations, and other information, along with Cocoa scriptability information used to implement support for the standard AppleScript commands. Cocoa also implements a Text suite.

Your application supplies its own scriptability information in one or more `suite` XML elements. For example, if your application provides scripting support for graphics-related features and database-related features, you can provide separate graphics and database suites.

For most applications, the sdef should also include a `suite` element for the Standard suite and possibly the Text suite as well. You can copy these suites from Sketch's sdef file, then delete the XML definitions for commands, classes, and other items not supported by your application so they won't be displayed in your dictionary. For example, if your application does not support printing, you should delete the elements for the `print` command and the `print settings` record-type. Otherwise, users will see those terms when they view your application terminology and will expect the application to support them.

Important: You can read about additional refinements to sdef usage in Cocoa applications for Mac OS X v10.5 in the Scripting section of *Foundation Release Notes*. For example, in Mac OS X v10.5, an sdef for Cocoa's Standard suite is provided in the file `CocoaStandard.sdef` in the location `/System/Library/ScriptingDefinitions`, and you can use an `xinclude` statement to include that file into your application's sdef file.

A `suite` element contains one or more of the following elements: `class`, `command`, `enumeration`, `record-type`, and `value-type`. These elements are described in "Add Information to the Scripting Definition File" (page 44).

The following listing (from Sketch's sdef file) shows the XML format for a `suite` element.

Listing 4-3 A suite element for the Sketch suite

```
<suite name="Sketch Suite" code="sktc"
        description="Sketch specific classes.">
    <!--
        Contains one or more of the following sdef elements:
            class, command, enumeration, record-type, and value-type
    -->
</suite>
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Create a Scripting Definition File

One option for creating an sdef file is to simply take an existing sdef, such as the one distributed with the Sketch application, and modify it for your application. You can do that with the following steps:

> **Important:** As noted previously, starting in Mac OS X v10.5, the file `CocoaStandard.sdef` in the location `/System/Library/ScriptingDefinitions` provides an sdef for Cocoa's Standard suite, and you can use an `xinclude` statement to include it into your application's sdef file. For more information, see the Scripting section of *Foundation Release Notes*.

1. Copy the file `Sketch.sdef` and rename it for your application (for example, `MyApplication.sdef`).
2. Edit the sdef file in a text editor (in Xcode, select the file and choose File > Open As > Plain Text File to open the sdef as plain text). Rename the Sketch `dictionary` element, so it looks something like the following:

   ```
   <dictionary title="MyApplication Terminology">
   ```

3. Rename the Sketch `suite` element and change its code, so that it looks something like the following:

   ```
   <suite name="MyApplication Suite" code="MyAp"
       description="MyApplication information.">
   ```

4. Replace the scriptability information in the renamed `suite` element with the information for your application, as described in "Add Information to the Scripting Definition File" (page 44).

Creating an sdef file from scratch is quite similar, and also includes copying information for the Standard and Text suites. To create a new sdef file:

1. Create a plain text file with the extension `.sdef` (for example, `MyApplication.sdef`).
2. Add XML version and document type information. (All sdef files start with this information.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dictionary SYSTEM "file://localhost/System/Library/DTDs/sdef.dtd">
```

3. Your sdef must contain a `dictionary` element as the root element. The dictionary is typically named after your application, as shown in previous steps. The complete `dictionary` element from Sketch is shown in "Dictionary Element" (page 41).
4. If your application supports any of the commands, classes, or other scriptability information defined in the Standard and Text suites, as most do, you should copy the `suite` elements for those suites into your sdef from an existing sdef file, such as `Sketch.sdef`.

   > **Important:** You should delete any information in the default Standard and Text `suite` elements for features your application does not support.

   For more information, see "Suite Elements" (page 42).

5. Add one or more `suite` elements for your application's scriptability information.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

6. Fill in the scriptability information in the `suite` element (or elements) for your application by adding whichever of the following kinds of XML elements your application uses:

    a. Classes (see "Class Elements" (page 44))

    b. Commands (see "Command Elements" (page 47))

    c. Enumerations (see "Enumeration Elements" (page 49))

    d. Simple structures (see "Record-Type Elements" (page 50))

    e. Simple types (see "Value-Type Elements" (page 51))

    Your suite elements can refer to definitions from the Standard or Text suites that you've copied into your sdef.

To add an sdef file to the project for your application, see "Add the Scripting Definition File to Your Xcode Project" (page 34).

Add Information to the Scripting Definition File

The following sections provide descriptions and examples of the XML elements you'll most commonly add to the sdef file for your application. These examples are based on the sdef file from the Sketch sample application, available from Apple.

As you add information to your sdef file, you can verify that the file is still valid at each step. Later, when testing your application, you can examine the information that Cocoa scripting extracts from your sdef file. For information on how to perform these steps, see "Debugging Scriptability Information" (page 90).

Important: To test changes you make to an sdef file, you must quit the application, rebuild, and relaunch it to pick up the changes. In addition, the Script Editor application caches scriptability information, so you may need to quit and relaunch Script Editor as well.

Class Elements

You'll need to add a `class` element to your sdef file for each type of scriptable object in your AppleScript object model—that is, for the objects scripters can access in scripts. A `class` element defines an object class by listing the properties, elements, and supported commands for instances of that class. Using the `rich text` class from the Text suite as an example, here is the information you supply in a `class` element:

- A human-readable name and description, which a scripter can view in the application's dictionary, as well as a four-character code to identify the class. It may optionally specify a plural spelling:

    ```
    <class name="rich text" plural="rich text" code="ricT" description="Rich (styled)
     text">
    ```

    If the plural name is omitted, it defaults to the name of the class with `'s'` appended.

- The Cocoa or application Objective-C class that implements the class:

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
<cocoa class="NSTextStorage"/>
```

- The commands the class can respond to, listed in a `responds-to` element. The `rich text` class doesn't contain a responds-to element, but here is an example for Sketch's rectangle class, which handles the `rotate` command with the `rotate:` method:

  ```
  <responds-to name="rotate">
      <cocoa method="rotate:"/>
  </responds-to>
  ```

  When Cocoa instantiates a command object to perform an AppleScript command (such as `rotate`), if the command operates on an object (such as a `rectangle`) whose class definition includes a `responds-to` entry, the command invokes the method specified in that entry (`rotate:`).

  A class can respond to the standard AppleScript commands, such as `count`, `move`, `delete`, and so on, without listing them in a `responds-to` element. For more information, see "Script Commands Overview" (page 71).

- A class also specifies its properties, elements, and possibly contents (described in subsequent sections).

The following listing shows the full `class` definition for the `rich text` class.

Listing 4-4 A class element for the rich text class

```
<class name="rich text" plural="rich text" code="ricT"
        description="Rich (styled) text">
    <cocoa class="NSTextStorage"/>
    <type type="text"/>
    <property name="color" code="colr" type="RGB color"
        description="The color of the first character.">
        <cocoa key="foregroundColor"/>
    </property>
    <property name="font" code="font" type="text"
        description="The name of the font of the first character.">
        <cocoa key="fontName"/>
    </property>
    <property name="size" code="ptsz" type="integer"
        description="The size in points of the first character.">
        <cocoa key="fontSize"/>
    </property>
    <element type="character"/>
    <element type="paragraph"/>
    <element type="word"/>
    <element type="attribute run"/>
    <element type="attachment"/>
</class>
```

Property Elements

For each `class` element in your sdef file, you'll need to add a `property` element for each accessible property of that class. A property is a unique data member of an object—it is synonymous with an attribute or to-one relationship. (A `property` element can also appear in a `record-type` element; see "Record-Type Elements" (page 50) for more information.)

Using the `color` property of the `rich text` class as an example, here is the information you supply in a `property` element:

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

- A human-readable name and description, which a scripter can view in the application's dictionary, along with a four-character code to identify the property and an entry to define its type. Basic AppleScript types, such as `boolean`, `integer`, and `text`, are listed in Table 1-1 (page 20). You can also use types declared in the suites provided by Cocoa (such as `color`), as well as types you have defined with a `value-type` element (described in "Value-Type Elements" (page 51)):

```
<property name="color" code="colr" type="RGB color"
    description="The color of the first character.">
```

- A KVC key to be used by Cocoa scripting to access the property. By default, the property name serves as the key. However, you can provide a substitute key with a `cocoa key` entry:

```
<cocoa key="foregroundColor"/>
```

  Providing this key specification means that when a script asks for the `color` property of a `rich text` object, Cocoa scripting will use the key `"foregroundColor"` to access the property from the instance of `NSTextStorage` that implements the `rich text` object. If you did not provide this key, Cocoa scripting would use `"color"` for the key.

  For more information on naming and keys, see "Cocoa Elements" (page 51).

- An optional entry that specifies access to the property. A property can be read-only (`access="r"`), write-only (`access="w"`), or read-write (`access="rw"`), which is the default. (When you display an sdef in a dictionary viewer, read-only access is shown as (r/o) and write-only as (w/o). Read-write access, the default, is not shown.)

  The `color` property of the `rich text` class doesn't contain an access entry (meaning access is read-write, the default), but the `name` property of the `window` class from the Standard suite shows how to specify read-only access:

```
<property name="name" code="pnam" type="text" access="r"
    description="The full title of the window.">
    <cocoa key="title"/>
</property>
```

  This property defines a key attribute of `<cocoa key="title"/>`. So when a script asks for the `name` property of a `window` object, Cocoa scripting uses the `"title"` key to get the appropriate information from the window.

Element Elements

For each `class` element in your sdef file, you'll need to add an `element` element for each accessible element of that class. An element defines a class of objects contained in another object and represents a to-many relationship. For each of its defined elements, an object may reference zero or more items of that type. For example, a `rich text` object can have any number of `character` elements, including none. Element definitions define part of the object model for the application.

When you display an sdef in a dictionary viewer, an `element` element shows up as a link to the class for its type. For example, an element that has the class type of `window` will show up as a link to the `window` class.

The `element` elements shown in Listing 4-4 (page 45) for the `rich text` class (such as `<element type="character"/>`) are all simple ones that take the default value for access (read-write) and key (for example, `"characters"`), as described in "Cocoa Elements" (page 51).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Using the `document` element from the `application` class in the Standard suite, here is a slightly more interesting example of the information you can supply for an `element` element:

- The class type and an optional access entry. The access values are the same as those described previously for `property` elements (read-only, write-only, or read-write, which is the default):

  ```
  <element type="document" access="r">
  ```

- An optional KVC key to be used by Cocoa scripting to access the element. By default, the plural of the `element type` name serves as the key, but you can provide a substitute:

  ```
      <cocoa key="orderedDocuments"/>
  ```

  In this case, the `application` class (implemented by `NSApplication`) provides access to `document` objects in the ordered manner expected by an AppleScript script using the key `orderedDocuments`. This overrides the default naming for key-value coding access to this element, which would use the key `documents` (made by generating the plural of the element name, `document`).

Contents Elements

For some classes, you may want to define a `contents` element. A `contents` element is similar to a `property` element, except that the name and code are optional. If you omit them, they default to `"contents"` and `"pcnt"`, respectively.

Cocoa Scripting treats the `contents` property as the implied container for its class. Scripts may refer to elements of the contents property as if they were elements of the class. For example, TextEdit documents have a `text` contents property. Technically, the first word of the first document is `word 1 of text of document 1` but, because `text` is an implied container, a script can also say `word 1 of document 1`.

For related information, see "Implicitly Specified Subcontainers" (page 69).

Command Elements

A `command` element represents an action or "verb" such as `close` that specifies an operation to be performed. You will need to add a `command` element to your sdef file for each scriptable operation your application supports, including those defined in the Standard suite and implemented by `NSScriptCommand` and its subclasses. For the commands supplied by Cocoa, you can obtain information for your sdef as described in "Create a Scripting Definition File" (page 43).

Some commands can be sent directly to the objects they operate on and some cannot. For more information on the different kinds of commands and how to work with them, see "Object-first Versus Verb-first Script Commands" (page 75).

Using commands from the Standard suite as examples, here is the information you supply in a `command` element:

- A human-readable name and description, which a scripter can view in the application's dictionary, along with two four-character codes to identify the command. The following is from the `quit` command:

  ```
  <command name="quit" code="aevtquit" description="Quit the application.">
  ```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

> **Note:** The code for a command totals eight characters in length. Although the code can in fact be an arbitrary value, the first half (`aevt` in this case) has historically represented the suite (the Standard suite, formerly called the Core suite) and the second half (`quit`) the command. See "Code Constants Used in Scriptability Information" (page 40) for more information on using codes.

- The Objective-C class, defined by Cocoa or the application, that Cocoa scripting should instantiate to handle the command:

  ```
  <cocoa class="NSQuitCommand"/>
  ```

  If a command does not specify a command class, the default is the `NSScriptCommand` class. The `NSQuitCommand` class is a subclass of `NSScriptCommand`, and any scripting class you define must also be a subclass of `NSScriptCommand` or one of its subclasses. The command classes defined by Cocoa are described in "Subclasses for Standard AppleScript Commands" (page 98).

- An optional `direct parameter` element definition. A direct parameter is a special case of a `parameter` element (described next) that does not include a name or identifying code and cannot be hidden. If the direct parameter is a class, then it specifies the object to which the message is sent. Otherwise, the message is sent to the `application` object, with the direct parameter's value interpreted as a normal parameter, in which case it typically specifies the objects on which the command is to operate.

> **Note:** The `direct parameter` element, other `parameter` elements, and the `result type` element (if any) in a `command` element correspond to parameter information supplied by an Apple event for that type of command. When Cocoa scripting receives the Apple event, it creates an instance of the specified command and initializes it with information from the event. It stores parameter and result type information from the Apple event in an argument dictionary in the command object.
>
> Your application obtains arguments from the argument dictionary using the keys specified for the corresponding parameters or result type in the sdef file. For information on how to do this, see "Script Command Components" (page 72).

  Here is the `direct parameter` element from the `open documents` command, which specifies a list of documents to be opened by the application object:

  ```
  <direct-parameter description="The file(s) to be opened.">
      <type type="file" list="yes"/>
  </direct-parameter>
  ```

  This definition contains a text description and a data type declaration. The data for the direct parameter is a list (`list="yes"`; the default is `"no"`) of files to be opened.

  (For information on how Cocoa applications handle the `open` command, see "How Cocoa Applications Handle Apple Events" (page 101).)

- Optional `parameter` element definitions for the command. These parameters are optional in the sense that you need not define any parameters at all. However, if you do define a parameter, you can also specify that it is optional (the default is `"no"`, indicating the parameter is required). If a script omits an optional parameter, the command should perform its default operation with respect to that parameter.

  A `parameter` element contains a human-readable name and text description, four-character code, and type declaration for the data type of the parameter.

  The `quit` command includes this `parameter` element, which specifies how to handle saving of modified documents:

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
<parameter name="saving" code="savo" type="save options" optional="yes"
   description="Whether changed documents should be saved before closing.">
    <cocoa key="SaveOptions"/>
</parameter>
```

A `parameter` element also contains a dictionary key used by Cocoa scripting. In this case, the line `<cocoa key="SaveOptions"/>` specifies a dictionary key to identify the entry in the `NSQuitCommand` argument dictionary whose value is the `saving` parameter. The type (`type="save options"`) indicates that supported values for the `saving` parameter come from the `save options` enumeration, defined in the Standard suite.

- An optional `result type` element definition for the command, which specifies the type of value generated when a command is executed. A result is a special case of a parameter definition that has only `type` and `description` attributes and may not be hidden or optional. If the command has no result, omit this element.

  When you write Objective-C code to handle a command, it should return information of the type specified by the `result type` element. If there is no result type specified, return `nil`.

  Here is the result definition for the `count` command, which returns an integer value for the count.

  ```
  <result type="integer" description="the number of elements"/>
  ```

Scriptability information for a command is packaged in the command object created to handle a received Apple event. For information on how your application works with that information, see "Script Command Components" (page 72), as well as other sections in "Script Commands" (page 71).

Enumeration Elements

You may need to define enumerated constants in your sdef. For example, you may want to provide constants a scripter can use to specify information to a command. The Standard suite does this with the `save options` enumeration, which provides values a scripter can use with either the `close` command or the `quit` command to specify how to handle unsaved documents.

An `enumeration` element is a collection of symbolic constants, referred to as enumerators. As shown in the following listing for the `save options` enumeration, you provide a name and a code for the enumeration as a whole. You also provide a name, code, and description for each enumerator.

Listing 4-5 Definition of the save options enumeration

```
<enumeration name="save options" code="savo">
    <enumerator name="yes" code="yes " description="Save the file."/>
    <enumerator name="no" code="no  " description="Do not save the file."/>
    <enumerator name="ask" code="ask "
        description="Ask the user whether or not to save the file."/>
</enumeration>
```

An enumeration element can optionally contain an inline entry, indicating how many of the enumerators should be shown with a term that uses the enumeration when the dictionary defined by the sdef is displayed. By default (if you do not use the inline element), all enumerators are shown. If you specify `inline="0"`, just the enumeration name (`save options`, in the example above) is displayed. However, a user viewing the definition can click the enumeration name to view the actual enumeration definition.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

The following line shows how to modify the `save options` definition to limit the display to just the first two enumerators (`yes`/`no`).

```
<enumeration name="save options" code="savo" inline="2">
```

Record-Type Elements

If you need to define a simple structure, rather than a class, for your sdef, you can add a `record-type` element. For example, the following listing shows the `record-type` definition for `print settings` from the Standard suite. The `cocoa key` entries in this definition match the names that `NSPrintInfo` uses in its attributes dictionary.

In addition to `property name`, `code`, and `description`, a `record-type` element contains "Property Elements" (page 45), described previously. You can use `record-type` elements you've defined anywhere you specify the `type` of an `element`, `property`, or `parameter` in your sdef.

Listing 4-6 Definition of the print settings record-type

```
<record-type name="print settings" code="pset">
    <property name="copies" code="lwcp" type="integer"
        description="the number of copies of a document to be printed">
        <cocoa key="NSCopies"/>
    </property>
    <property name="collating" code="lwcl" type="boolean"
        description="Should printed copies be collated?">
        <cocoa key="NSMustCollate"/>
    </property>
    <property name="starting page" code="lwfp" type="integer"
        description="the first page of the document to be printed">
        <cocoa key="NSFirstPage"/>
    </property>
    <property name="ending page" code="lwlp" type="integer"
        description="the last page of the document to be printed">
        <cocoa key="NSLastPage"/>
    </property>
    <property name="pages across" code="lwla" type="integer"
        description="number of logical pages laid across a physical page">
        <cocoa key="NSPagesAcross"/>
    </property>
    <property name="pages down" code="lwld" type="integer"
        description="number of logical pages laid out down a physical page">
        <cocoa key="NSPagesDown"/>
    </property>
    <property name="error handling" code="lweh" type="printing error handling"
        description="how errors are handled">
        <cocoa key="NSDetailedErrorReporting"/>
    </property>
    <property name="fax number" code="faxn" type="text"
        description="for fax number">
        <cocoa key="NSFaxNumber"/>
    </property>
    <property name="target printer" code="trpr" type="text"
        description="for target printer">
        <cocoa key="NSPrinterName"/>
    </property>
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
</record-type>
```

Value-Type Elements

If you need to define a new basic type for your scripting support, you can do so with a `value-type` element. A `value-type` element defines a simple type that has no properties or elements accessible through scripting. The following listing shows the value-type definition for `color` from the Text suite. In addition to the type name and code, Cocoa value-type definitions should specify a corresponding Objective-C class, such as `NSData` or `NSNumber` (or your class that supports your value-type). The built in AppleScript types supported by Cocoa are listed in Table 1-1 (page 20).

You can use `value-type` elements you've defined anywhere you specify the `type` of an element, property, or parameter in your sdef.

Listing 4-7 Definition of the color value-type

```
<value-type name="RGB color" code="cRGB">
    <cocoa class="NSColor"/>
</value-type>
```

Cocoa Elements

Some of the information in your sdef describes implementation details from your application. For example, property names in the sdef serve as KVC keys for accessing property values of scriptable application objects, as described in "Provide Keys for Key-Value Coding" (page 33) and "Maintain KVC Compliance" (page 54). Similarly, your sdef contains other information that directly identifies classes or methods that are part of your application's scripting support.

To supply this implementation information for use by Cocoa scripting, you use `cocoa` elements. A `cocoa` element can contain the following attributes:

- `class`: Specifies an Objective-C class name for `class` elements and `command` elements.
- `key`: Specifies a string key for a dictionary (`NSDictionary`) of command parameters, or a key to be used by Cocoa scripting to access a property or element through key-value coding.
- `method`: Specifies an Objective-C method name. Used to specify `responds-to` methods for `class` elements.

See the previous sections in this chapter for examples of these attributes.

Cocoa scripting will generate default keys for property and element attributes and for commands, if you do not specify them. For a property, it capitalizes each word of the property's name except the first word, then removes any spaces. For an element, it specifies the plural of the element type. For a command, the default is `NSScriptCommand`.

The following table shows some examples of default naming.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Table 4-1 Default naming for attributes of cocoa elements

Attribute	Default value
`<property name="current resolution">`	`currentResolution` (key name)
`<element type="monitor">`	`monitors` (pluralized key name)
`<command name="someCommand"...` (with no command class specified)	`NSScriptCommand` (class name for command)

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Getting and Setting Properties and Elements

This chapter describes how to work with Cocoa scripting to allow it to get and set the values of properties and elements in your scriptable application. It also provides examples of KVC-compliant accessor methods.

Overview of Getting and Setting Values

Allowing scripters to get and set values of scriptable objects is an important part of scriptability. In your sdef, you define keys for scriptable properties and elements. In your application, properties are equivalent to attribute and to-one relationships, while elements are represented by to-many relationships.

Cocoa scripting supports getting and setting values through the use of key-value coding (KVC). For your application to work with this support, you must comply with KVC conventions in naming the keys in your sdef file and the corresponding accessor methods (or instance variables) in your scriptable classes:

- In your sdef file, you define keys for the scriptable properties and elements of your scriptable classes, as described in "Provide Keys for Key-Value Coding" (page 33). Cocoa scripting uses these keys when it invokes KVC to get and set values.
- In your application code, accessors (or instance variables) for scriptable properties and elements must match the keys specified in your sdef file and the naming conventions described in "Maintain KVC Compliance" (page 54).

In handling `get` and `set` scripting commands, Cocoa scripting takes care of the following automatically:

- It provides the `NSGetCommand` and `NSSetCommand` classes and automatically instantiates the appropriate object when your application receives a `get` or `set` Apple event. You rarely need to subclass these classes.
- It automatically invokes key-value coding (KVC) methods to get or set values when a `get` or `set` command is executed—your application doesn't need to call KVC methods directly.

For a description of how Cocoa scripting and an application work together to perform a `set` command, see "A Real World Scripting Example" (page 25).

Key-Value Coding and Cocoa Scripting

When Cocoa scripting uses KVC to get and set values, it calls one of several KVC methods, depending on the type of relationship involved (properties or elements) and on whether the operation is a get or a set. These include two primitive instance methods of the `NSObject` class: `valueForKey:` and `setValue:forKey:`. These methods are used primarily in dealing with properties, though they can also handle element collections in a simple but potentially inefficient way. For more efficient handling of collections, KVC invokes `mutableArrayValueForKey:`.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

These KVC methods in turn search your method list for specific methods in a specific order, as described in *Key-Value Coding Programming Guide*. Templates for the methods KVC looks for are described in "Maintain KVC Compliance" (page 54).

Your application doesn't invoke or override KVC methods—it just names its accessor methods or instance variables so that KVC can find them.

Maintain KVC Compliance

For Cocoa scripting to work successfully with KVC, every scriptable class in your application must be KVC-compliant for every key corresponding to its scriptable properties and elements. To ensure this, you must define keys in your sdef file as described in "Provide Keys for Key-Value Coding" (page 33), and you must adhere to the following naming conventions for your accessor methods or instance variables (where you replace <key> or <Key> with the key string to construct the actual method name):

- For properties (attributes and to-one relationships), you implement <key> and, if the property is not read-only, set<Key>: accessors.
- For element classes (to-many relationships), you implement <key> and, if the element class is not read-only, insertObject:in<Key>AtIndex: and removeObjectFrom<Key>AtIndex: methods. For better performance, you may also need to implement replaceObjectIn<Key>AtIndex:withObject:.

  If there is no way to efficiently implement <key> (for example, if the value of the to-many relationship is not naturally stored in an instance of NSArray), you can instead implement countOf<Key> and objectIn<Key>AtIndex:. KVC's valueForKey: and mutableArrayValueForKey:, which are invoked by Cocoa scripting, will return collection proxies that invoke your methods when appropriate.
- To let Cocoa automatically create an array proxy to handle element collections, you can implement countOf<Key> and objectIn<Key>AtIndex: methods.

  If the to-many relationship is mutable, you should also implement insertObject:in<Key>AtIndex: and removeObjectFrom<Key>AtIndex: methods.

  And for better performance, you may need to implement replaceObjectIn<Key>AtIndex:withObject:.

  KVC invokes these methods regardless of the class used to model the relationship, so you can use array-based or non-array-based collections, including custom collections you have defined. But it's up to you to implement the methods to work with your underlying data. The code should be quite straightforward; if you are working with arrays, it often requires just a call to an array method that corresponds to the accessor method.
- As an alternative to implementing accessor methods, you can take advantage of KVC's direct instance variable access feature. You do this by merely giving an instance variable a name that matches a key defined in your sdef, with an optional leading underscore—for example, xPosition or _xPosition,

  Instance variables corresponding to writable element classes must be of type NSMutableArray if you do not implement the insertObject:in<Key>AtIndex: and removeObjectFrom<Key>AtIndex: methods.
- For element classes that can be accessed by name or ID (that is, the class has properties identified by the four character code 'pnam' or 'ID '), you can optimize the evaluation of name and ID specifiers by implementing the methods valueIn<Key>WithName: or valueIn<Key>WithUniqueID:, respectively.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Important: In your implementation of `valueIn<Key>WithName:`, string matching should honor the current AppleScript considerations, such as `considering case` or `ignoring diacriticals`. Name data may come to your application as plain or Unicode text, so be prepared to handle either, even if you do not use full Unicode internally.

In your implementation of `valueIn<Key>WithUniqueID:`, ID data should not honor the current AppleScript considerations, such as `considering case`. Treat IDs as always case-insensitive, or as always case-sensitive if case is relevant (that is, if `"ijkl"` and `"IJKL"` are both valid and different).

If you follow standard Cocoa naming conventions for accessors, you're already on the way to KVC compliance:

- A "get" accessor should start with a lower case letter and have the same name as the variable it is accessing—for example, `xPosition`.
- A "set" accessor should start with "set", followed by the name of the variable, with the first letter capitalized—for example, `setXPosition`.

For examples of KVC-compliant accessor methods based on the templates described here, see "Sample KVC-Compliant Accessor Methods" (page 56).

On Omitting KVC Accessors

The use of accessor methods supports data encapsulation, a standard coding practice. However, in cases where you are providing KVC access to your data solely to support scriptability (or Cocoa bindings), encapsulation may be less important, and you might choose to omit accessor methods altogether. If so, you need only maintain KVC compliance in naming your instance variables to match the keys in your sdef.

You might also choose to use direct instance variable access as a convenience during prototyping or feasibility testing, then add accessor methods later in the development cycle.

Performance Considerations With KVC

KVC access to simple values is fast enough that it should not be a performance bottleneck. In general, KVC is smart enough to search for the most efficient methods your classes provide that match the KVC naming conventions. For example, for to-many relationships, KVC looks first for collection mutation methods, then for setter methods; if neither is available, it tries for direct instance variable access.

"Maintain KVC Compliance" (page 54) describes the collection methods you should implement for best performance (`insertObject:in<Key>AtIndex:` and `removeObjectFrom<Key>AtIndex:`).

You can instead merely implement a setter method (as in, `set<Key>:`) and KVC will invoke it, but be aware of what KVC, as used by Cocoa Scripting, must do in that case: for an insertion or removal, it must get the value array, make a mutable copy of it, mutate the copy, and then set it again. This can be slow.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Interaction With Key-Value Observing

Because Cocoa scripting invokes `setValue:forKey:` instead of `takeValue:forKey:` (a deprecated KVC method), changes to model objects made by AppleScript scripts are observable using automatic key-value observing. However, if the container in question overrides `takeValue:forKey:`, Cocoa scripting invokes `takeValue:forKey:` for backward binary compatibility.

Note: In Mac OS version 10.3, Cocoa scripting did not use key-value coding methods introduced in that version of the OS, such as `setValue:forKey:` and `mutableArrayValueForKey:`, so automatic key-value observing (KVO) was not supported for model object changes caused by scripts.

Cocoa scripting does not depend on KVO. However, if your application uses Cocoa bindings, you should follow the guidelines described in *Key-Value Observing Programming Guide*. For related information in this document, see "Interaction With Cocoa Bindings and Core Data" (page 23).

KVC Conversion of Scalar and Structure Values

The default implementations of the KVC methods `valueForKey:` and `setValue:forKey:` provide support for automatic object wrapping of scalar data types such as `BOOL`, `char`, `double`, `float`, `int`, and structures such as `NSPoint`, `NSRange`, and `NSRect`. When Cocoa scripting invokes `valueForKey:` to get a value, KVC automatically converts the value to an `NSNumber` object (for scalars) or `NSValue` object (for structures) if necessary.

Similarly, `setValue:forKey:` determines the data type required by the appropriate accessor or instance variable for the specified key. If the data type is not an object, then the value is extracted from the passed object using the appropriate `NSNumber` or`NSValue` method.

For more information, including a table of the supported types, see Scalar and Structure Support in *Key-Value Coding Programming Guide*.

Scripting Additions to KVC

Cocoa defines the category `NSScriptKeyValueCoding` to provide scripting-related additions to the key-value coding methods implemented in `NSObject`. The methods in this category on `NSObject` provide additional capabilities for working with key-value coding, including getting and setting key values by index in a collection and coercing (or converting) a key value. Additional methods allow the implementer of a scriptable container class to provide fast access to elements that are referenced by name and unique ID.

Sample KVC-Compliant Accessor Methods

Suppose you have a `DrawingCanvas` class that you want to make scriptable. In a simple scenario, you want scripts to be able to access the properties of graphical shapes associated with a canvas object, and you also want scripts to be able to access a boolean value indicating whether the canvas has been modified. Assume that scripters can get and set properties of a shape, but can only read the current value of the modified property.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

First, you specify the keys "shape" and "modified" in the scriptability information for your application. To do this, you could define the following entries in the class definition for the drawing canvas scriptable object class in your sdef file:

```
<property name="modified" code="iMod" type="boolean" access="ro"
    description="Has the canvas been modified since the last save?">
</property>
<element type="shape">
```

The two keys also become instance variables of the DrawingCanvas class. Remember that Cocoa scripting pluralizes the key for an element, as shown in the declared array variable, shapes. However, the instance variable names don't have to match the keys if you provide complete KVC-compliant method coverage.

```
@interface DrawingCanvas: NSObject <NSCoding> {
    NSMutableArray *shapes; // An array of shape objects
    BOOL modified; // Whether the canvas has been modified
    // ...
```

The following sections provide sample accessors for these properties, based on the KVC accessor method templates described in "Maintain KVC Compliance" (page 54). Depending on the implementation, some examples might require additional application-dependent code that is not shown here.

Single-Value Access

To provide scriptable access to the boolean property of the DrawingCanvas class, based on the key "modified", you can define a KVC-compliant getter method like the following:

Listing 5-1 Boolean property getter

```
- (BOOL)modified
{
    // If necessary, obtain or update value
    return modified;
}
```

If the modified property were settable (if the property were defined as read/write, the default), you could define a KVC-compliant setter method like the following:

Listing 5-2 Boolean property setter

```
- (void)setModified:(BOOL)flag
{
    modified = flag;
}
```

As noted in "On Omitting KVC Accessors" (page 55), in cases where you are providing KVC access to your data solely to support scriptability, you might choose to omit accessor methods. If so, you need only name your instance variables to match the keys in your sdef—in this case, modified or _modified.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Collection Value Access

For simple scriptable access to a to-many relationship, you can implement accessor methods like those in Listing 5-3. However, the setter method may not be efficient, for reasons described in "Performance Considerations With KVC" (page 55).

Listing 5-3 Simple array element accessors

```
- (NSArray *)shapes
{
    return [[shapes retain]autorelease];
}
- (void)setShapes:(NSArray *)newShapes
{
    if (shapes != newShapes)
    {
        [shapes release];
        shapes = [newShapes copy];
    }
}
```

You can support KVC access for to-many relationships by implementing indexed accessor methods. These accessors work whether or not the related objects are stored in an array—Cocoa automatically creates an array proxy for you if needed.

For best performance, you should implement the two KVC-compliant methods shown in the next listing, instead of the `setShapes:` method:

Listing 5-4 Array element insert/delete accessors (by index)

```
-(void)insertObject:(id)anObject inShapesAtIndex:(unsigned)index
{
    [shapes insertObject:anObject atIndex:index];
}

-(void)removeObjectFromShapesAtIndex:(unsigned)index
{
    [shapes removeObjectAtIndex:index];
}
```

And finally, to provide even more efficient mutation of the array, you can also implement a KVC-compliant method like the following:

Listing 5-5 Array element replacement accessor (by index)

```
-(void)replaceObjectInShapesAtIndex:(unsigned)index withObject:(id)anObject
{
    // possible implementation-specific code
    [shapes replaceObjectAtIndex:index withObject:anObject];
}
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Special Accessor Methods

Sometimes an accessor method must do something special to support scripting. For example, consider the documents managed by a document-based application. When a script asks for an application's documents, the application could invoke the `documents` method of the `NSDocumentController` object to obtain all currently opened documents, ordered by creation.

However, this is not what is expected by the scripter. AppleScript has the notion of a `first document` and a `last document` (as well as `front document`, `document 1`, and related notations), and this implies an ordering of documents visible on the screen. The `NSApplication` class therefore implements the `orderedDocuments` method, which, in response to a request for its documents, returns an array of document objects, where the position of a document in the array is based on the front to back ordering of its associated (on-screen) window.

Sometimes an application's object model provides scripting access to objects at a level of granularity that would be impractical to implement with individual objects. For example, an AppleScript script can ask for the `characters` of a text document, but it would be quite expensive for an application to represent each character as an object. The `NSTextStorage` class handles this case with a special accessor method, `characters`

Support for the Properties Property

It is common for good scriptable applications to make the complete set of properties for each scriptable object available in the form of a single record that is the value of the `properties` property. A category on `NSObject` (defined in the Foundation file `NSObjectScripting.h`) declares the following public KVC accessor methods for this property:

```
- (NSDictionary *)scriptingProperties;
- (void)setScriptingProperties:(NSDictionary *)properties;
```

For all scriptable classes that inherit from `item`, Cocoa scripting provides automatic scriptability support for the `properties` property. So if your scriptable classes that descend from `NSObject` provide KVC-compliant accessors for their individual scriptable properties and elements, you will automatically have support for the `properties` property.

Note: For applications that supply scriptability information through script suite and script terminology files, the `AbstractObject` scriptable class (which corresponds to `NSObject`) specifies a base class that scriptable classes can inherit from. Cocoa Scripting uses this class, declared in the Foundation files `NSCoreSuite.scriptSuite` and `NSCoreSuite.scriptTerminology`, to provide support for the `properties` property.

Coercion

Coercion is the process of converting data from one type to another. AppleScript provides many default (or built-in) coercions. For example, when it executes the statement `set theValue to "20.05" as number`, AppleScript converts the string `"20.05"` to the numeric value `20.05`.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Cocoa scripting supports the types shown in "Built-in Support for Basic AppleScript Types" (page 20), and makes use of the default coercions as needed when it handles Apple events received by a Cocoa application. For information on the default coercions, see Default Coercion Handlers in *Apple Events Programming Guide*.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Object Specifiers

An **object specifier** locates a scriptable object within an application's containment hierarchy. Cocoa scripting makes use of object specifiers to find objects in your application while executing a script command and to return information requested by a script. This chapter describes how Cocoa scripting uses object specifiers and how your application provides object specifiers for its scriptable objects.

Overview of Object Specifiers

An AppleScript script contains statements that manipulate the objects in an application's AppleScript object model. The part of a script statement that identifies an object, such as `fourth word`, is called a **reference**. A reference rarely occurs in isolation. Usually a script statement consists of a series of references, preceded by a command and typically connected to each other by `in` or `of`, such as `get the fourth word in the third paragraph of document "Quarterly Report"`.

An Apple event encapsulates the operation specified by a script statement and delivers it to the application. For Apple events that correspond to commands defined in the application's sdef file, Cocoa scripting converts the Apple event into a script command that contains all the information necessary to perform the operation.

To describe the objects specified by a reference, the command uses object specifiers. Where a script statement identifies an object in the AppleScript object model, an object specifier identifies the corresponding object in the application itself. When the application must return an object to the calling script, Cocoa scripting also uses an object specifier, supplied by your application, to identify the object.

There is not always a one-to-one correspondence between AppleScript objects and objects in your application's implementation. For example, a `character` object in a script does not have a corresponding `character` object in the application.

Object Specifiers and KVC

Cocoa scripting relies on key-value coding (KVC) when evaluating object specifiers. When an application first needs to work with scriptability information, Cocoa scripting loads information from the application's sdef file into a global instance of `NSScriptSuiteRegistry`. At that time, it registers class descriptions for your scriptable classes. As a result, Cocoa and your application can obtain script class description information, including keys, about scriptable classes for use with object specifiers.

For more information, see "Loading Scriptability Information" (page 21).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

When to Implement an Object Specifier Method

Any class in your application that is part of the containment hierarchy for your scriptability support will most likely require an object specifier method. When a script statement targets your application, the application may need to return a reply. For example, the result of a `get` command is often an object or a list of objects. When Cocoa returns these objects in the reply Apple event, it does not return pointers to Objective-C objects, it returns object specifiers that locate scriptable objects within the containment hierarchy. As part of building these specifiers, Cocoa calls on your scriptable objects to supply specifiers for themselves.

You provide an object specifier for a scriptable class by implementing the `objectSpecifier` method. That method is defined in `NSScriptObjectSpecifiers` (note the trailing "s"), a category on `NSObject` that implements a version that just returns `nil`.

About Object Specifier Classes

An `objectSpecifier` method returns an instance of one of the classes listed in Table 6-1 (page 65), which are subclasses of `NSScriptObjectSpecifier`. Classes such as `NSNameSpecifier` and `NSUniqueIDSpecifier` represent the standard AppleScript reference forms, so you shouldn't need to subclass them. You pick the best version for the object you need to specify—see Technical Note TN2106, "Scripting Interface Guidelines," for guidance. For example, if the object has a unique name or unique ID, use the corresponding specifier type. For an unnamed object with no ID, such as a rectangle or a paragraph of text, you may want to return an index specifier.

Most subclasses of `NSScriptObjectSpecifier` have a specific initializer that you should use.

Most initializer methods for an object specifier class include the following parameters:

- `(NSScriptClassDescription *)classDescription`

  Here you supply a class description for the container class of the specified object. You must always supply a class description for a specifier.

  If you already have an object specifier for a container, you can obtain a class description for that container object using the `NSScriptObjectSpecifier` method `keyClassDescription`, as shown in Listing 6-1 (page 67).

  When you can get the class of a container object (for example, by invoking `[NSApp class]`), you can use the `classDescriptionForClass` method to determine the container description, as shown in Listing 6-2 (page 67). If you have an instance of the container's class, you can instead use the `NSObject` instance method `classDescription` to obtain a container description.

- `(NSScriptObjectSpecifier *)specifier`

  Here you supply an object specifier for the parent container of the specified object. You typically obtain it by invoking the `objectSpecifier` method of the containing object.

  An object that has no container specifier is known as the **top-level specifier**. In most cases, the top-level specifier is the application itself. You can specify the top-level specifier by passing `nil` for this parameter. (That is the only time you can pass `nil` for a container specifier, and even then you must specify a container class description.)

- `(NSString *)key`

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

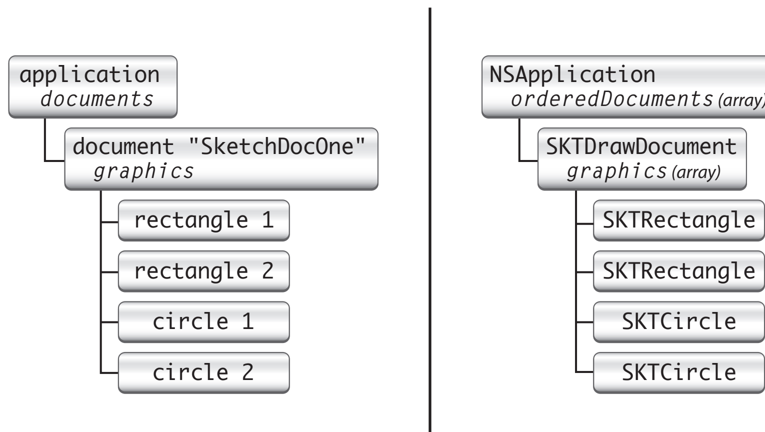
Here you supply a key that tells the parent container how to find the specified object. The key you pass is based on information supplied in your sdef file. For example, Sketch defines the "graphics" key to identify the graphics array in a Sketch document. (Sketch actually defines the singular "graphic" as an element of the document class, but Cocoa scripting applies its default rules, described in "Provide Keys for Key-Value Coding" (page 33), to convert this key to the string "graphics".)

For an annotated example of an objectSpecifier method, see "An Object Specifier Method for a Rectangle in Sketch" (page 67).

A Closer Look at an Object Specifier

Consider the object model and containment hierarchy diagram for the Sketch sample application, shown in Figure 6-1. The left side of this illustration shows scriptable Sketch objects, the right side the corresponding application containment hierarchy. For a script statement that identifies a rectangle object on the left, Cocoa scripting supplies an object specifier that locates the rectangle and its containing document within the containment hierarchy on the right.

Figure 6-1 Sketch object model and containment hierarchy revisited

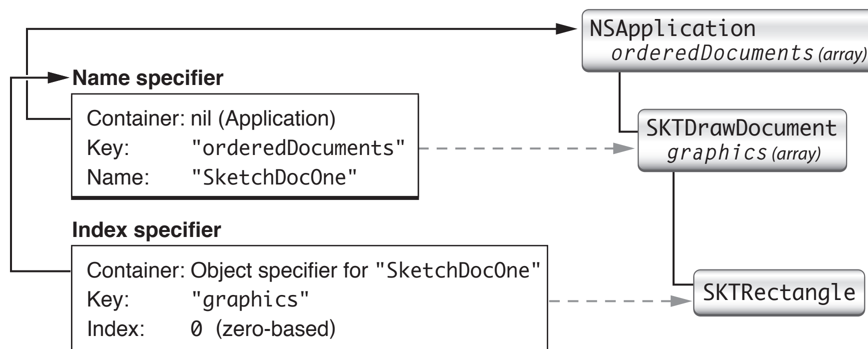


Each object specifier contains a reference to its containing object, which allows object specifiers to represent objects that can be deeply nested within an object hierarchy. The outermost container, represented by a nil object specifier, is usually the application object, although it can be an object specifier involved in a whose clause (NSWhoseSpecifier) or the container for a range evaluation.

Figure 6-2 shows the nested object specifiers for the statement first rectangle in document "SketchDocOne":

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Figure 6-2 Nested object specifiers for a Sketch rectangle



Here is how to interpret the information in this figure:

1. An index specifier specifies the first rectangle, which is an object of class `SKTRectangle`. The specifier has these components:

   - The index for the specified object, which in this example has the value `0`.

     This is the zero-based index of the specified rectangle in its containing array. Because Sketch's graphics arrays can contain other kinds of graphics, the index for the first rectangle won't always have the value `0`—for example, if preceded by four circle graphics, the index would have the value `4`.

   - A key that specifies the collection for the specified object, which in this example has the value `"graphics"`.

     The graphics array is the collection for the indexed object. The key matches information provided by your sdef and made available in the running application as described in "Object Specifiers and KVC" (page 61).

   - A container reference that specifies the parent for this object specifier. In this example the container is the object specifier for the document `"SketchDocOne"`.

2. A name specifier specifies the document containing the first rectangle, which is an object of class `SKTDrawDocument`. The specifier has these components:

   - The name for the specified object, which in this example has the value `"SketchDocOne"`.

   - A key that specifies the collection for the specified object, which in this example has the value `"orderedDocuments"`.

     The application's ordered array of documents is the collection for the named document, though in this case, the order is unimportant.

   - A container reference that specifies the parent for this object specifier. In this example, the reference is `nil`, specifying that the array of documents is contained by the `application` object.

A script statement, and thus an object specifier, can specify objects that don't currently exist in the application, causing an error condition when the script is executed. For example, the specified document may not exist, or it may not contain the specified graphic. Code within a command class should check for and be prepared to handle common error conditions. (Validation is not considered appropriate for KVC getter and setter methods.)

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Evaluation of Nested Specifiers

When Cocoa scripting prepares a script command for a script that contains a reference, it converts the reference into a nested series of object specifiers, such as the ones shown in Figure 6-2 (page 64). It packages these nested object specifiers as a receiver of the command (or in some cases as an argument of the command). When the command is executed, the application evaluates the object specifiers in its own context to discover the specified objects.

Evaluation starts with the top-level specifier and proceeds down the chain of object specifiers, evaluating and resolving each until the identity of the final, nested child object is determined. This object is a receiver (or receivers) of the command or one of the command's arguments. Key-value coding is used as the standard mechanism for evaluation. An object specifier queries its evaluated container for the value of the key associated with the object specifier.

Evaluation of object specifiers is described in more detail in "Script Commands and Object Specifiers" (page 74).

Cocoa Object Specifier Classes

AppleScript recognizes many types of references. For the standard reference forms, Cocoa defines subclasses of the abstract class `NSScriptObjectSpecifier`. These classes are shown in Table 6-1. Each of these object specifier classes deals with identifying objects in collections (`NSArray` objects).

These classes are unusual in that they provide one of the few examples where your application routinely creates instances of scripting classes defined by Cocoa. You do that in the object specifier methods for your scriptable classes. For more information on these and related classes, see also Table 9-2 (page 96).

Table 6-1 AppleScript reference forms and corresponding object specifier classes

Reference forms	Cocoa Class	Description
Arbitrary	`NSRandomSpecifier`	Specifies an arbitrary object in a collection. Example: `any word`, `some document`
Filter	`NSWhoseSpecifier`	Specifies every object in a particular container that matches specified conditions defined by a Boolean expression. Example: `words whose color is blue` or `document whose name is "Letter to Santa Claus"`
ID	`NSUniqueIDSpecifier`	Specifies an object in a collection by unique ID. Example: `window id 739`
Index	`NSIndexSpecifier`	Specifies an object in a collection by index. Examples: `word 5`, `front document`
Middle	`NSMiddleSpecifier`	Specifies the middle object in a collection. Example: `middle word of paragraph 2`

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Reference forms	Cocoa Class	Description
Name	NSNameSpecifier	Specifies an object in a collection by name. See class documentation for evaluation mechanism. Example: window named "Report"
Property, Every	NSPropertySpecifier	Specifies an attribute (property) or relationship of an object. Example: color (Property), every graphic of the front document (Every)
Range	NSRangeSpecifier	Specifies a range of objects in a collection. Example: words 5 through 10
Relative	NSRelativeSpecifier	Specifies the position of an object in relation to another object. Example: before word 5

It is unlikely that you will need to subclass any of these classes, as they already cover the standard set of valid AppleScript reference forms. However, if you do need to create a subclass, you must override the primitive method indicesOfObjectsByEvaluatingWithContainer:count: to return indices to the elements within the container whose values are matched with the child specifier's key. In addition, you may need to declare any special instance variables and implement an initializer that invokes the designated initializer of its superclass, initWithContainerClassDescription:containerSpecifier:key:, and initializes these variables.

In addition to the concrete subclasses of NSScriptObjectSpecifier shown in Table 6-1, Cocoa provides other classes that assist the object-specifier classes in evaluation. Instances of these classes help to indicate relative position and represent Boolean and logical expressions in which object specifiers are involved.

A script statement can also contain filter references, which identify objects in containers based on the conditions specified in Boolean expressions. These expressions can be linked together by logical operators (AND, OR, NOT) and return the appropriate true or false value. Filter references begin with the words whose or where, as in get words where color is blue or color is red and rectangles whose x position is greater than 45. These references can contain phrases such as is, is equal to or is greater than, as well as their symbolic equivalents (such as = and >).

Instances of the NSWhoseSpecifier class represent filter reference forms in Cocoa. These instances hold a “test” instance variable, which is an NSScriptWhoseTest object.

For information about these and related classes, see “Object Specifiers, Logical Tests, and Related Categories” (page 96).

Implementing the Object Specifier Method

The following sections provide examples of how to implement the object specifier defined by NSScriptObjectSpecifiers.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

An Object Specifier Method for a Rectangle in Sketch

Listing 6-1 shows how the Sketch sample application (available from Apple) implements the `objectSpecifier` method for the `SKTGraphic` class.

Listing 6-1 An object specifier method for a rectangle

```
- (NSScriptObjectSpecifier *)objectSpecifier{
    NSArray *graphics = [[self document] graphics];                          // 1
    unsigned index = [graphics indexOfObjectIdenticalTo:self];               // 2
    if (index != NSNotFound) {
       NSScriptObjectSpecifier *containerRef = [[self document] objectSpecifier];   // 3
        return [[[NSIndexSpecifier allocWithZone:[self zone]]
            initWithContainerClassDescription:[containerRef keyClassDescription]
            containerSpecifier:containerRef key:@"graphics" index:index] autorelease]; // 4
    } else {
        return nil;                                                          // 5
    }
}
```

Here’s a description of how this method provides an object specifier for a graphics object. The returned specifier consists of an index specifier for the graphic in its container, and a container specifier for the document that contains the array of graphics:

1. From its document, it gets the array of graphics and determines the index of the receiving graphic, if it is contained in the array.

2. It gets the object specifier of the document that contains the graphic.

   Sketch defines the `SKTDrawDocument` class as a subclass of `NSDocument`. The object specifier method for `NSDocument` returns an instance of `NSNameSpecifier` that identifies the document by name within the application's array of ordered documents.

3. It creates and initializes an index specifier (type `NSIndexSpecifier`) for the receiving graphic and returns it.

   The specifier locates the graphic by index within the graphics in its document. Although an AppleScript script asks for indexed items as though they are one-based, this index specifier, which locates objects within an array that correspond to the specified items, works with zero-based values. For information about the other parameters, see “About Object Specifier Classes” (page 62).

Specifying the Application Object as a Container

To specify the application object as the top-level container for a specifier, you can use code like the following. This snippet is part of the document class and obtains an object specifier for the document by name (rather than by index, which may become invalid if the document ordering changes):

Listing 6-2 Specifying the application as a container

```
NSScriptClassDescription *containerClassDesc = (NSScriptClassDescription *)
    [NSScriptClassDescription classDescriptionForClass:[NSApp class]];      // 1
return [[[NSNameSpecifier alloc]
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
    initWithContainerClassDescription:containerClassDesc
    containerSpecifier:nil key:@"orderedDocuments"
    name:[self lastComponentOfFileName]] autorelease];                        // 2
```

Here's a description of what this code snippet does:

1. The application is the container for a list of ordered documents, so this code first obtains a class description from the global application object, `NSApp`. (You never pass `nil` for the class description.)

2. It then creates and returns an autoreleased object specifier of type `NSNameSpecifier`, passing `nil` for the `containerSpecifier` parameter to specify the top-level container, the application. (This is the only case in which you can pass `nil` for the container.)

   It invokes the `NSDocument` method `lastComponentOfFileName` to obtain the name of the document and uses it to construct a name specifier object.

Implementing A Method for Evaluating Object Specifiers

Container classes that want to evaluate certain object specifiers on their own should implement the `indicesOfObjectsByEvaluatingObjectSpecifier:` method defined by `NSScriptObjectSpecifiers` (a category on `NSObject`). For example, you might choose to implement this method if you find that `whose` clause evaluation is too slow and you want to do your own evaluation to speed it up (though for most applications, performance of the default `whose` mechanism should be sufficient).

If this method returns `nil`, the object specifier method for the class does its own evaluation. If this method returns an array, the object specifier uses the `NSNumber` objects in the array as the indices of the specified objects.

Therefore, if you implement this method and when you evaluate the specifier there are no objects that match, return an empty array, not `nil`. If you find only one object, you return its index in an array. Returning an array with a single index where the index is –1 is interpreted to mean all the objects match.

The Sketch application implements this method in its document class, as shown in Listing 6-3. This allows Sketch to directly handle some range and relative specifiers for graphics, so it can support script statements such as `graphics from circle 3 to circle 5`, `circles from graphic 1 to graphic 10`, or `circle before rectangle 3`.

Listing 6-3 indicesOfObjectsByEvaluatingObjectSpecifier: method from Sketch

```
- (NSArray *)indicesOfObjectsByEvaluatingObjectSpecifier:(NSScriptObjectSpecifier
 *)specifier {
    if ([specifier isKindOfClass:[NSRangeSpecifier class]]) {                 // 1
       return [self indicesOfObjectsByEvaluatingRangeSpecifier:(NSRangeSpecifier
 *)specifier];
    } else if ([specifier isKindOfClass:[NSRelativeSpecifier class]]) {       // 2
        return [self
indicesOfObjectsByEvaluatingRelativeSpecifier:(NSRelativeSpecifier *)specifier];
    }
    return nil;                                                               // 3
}
```

Here's a description of how this method works:

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

1. If the passed specifier is a range specifier, it returns the result of invoking another Sketch method to evaluate the specifier.

   The method `indicesOfObjectsByEvaluatingRangeSpecifier:` allows more flexible range specifiers to be used with the different graphic keys of a `SKTDrawDocument`.

   You can examine this method in Sketch. Describing it in full is beyond the scope of this discussion.

2. If the passed specifier is a relative specifier, it returns the result of invoking another Sketch method to evaluate the specifier.

   The method `indicesOfObjectsByEvaluatingRelativeSpecifier:` allows more flexible relative specifiers to be used.

   Again, this method is available in Sketch, but is beyond the scope of this discussion.

3. For any other type of specifier, this method returns `nil` so that the default object specifier evaluation will take place.

Implicitly Specified Subcontainers

Cocoa Scripting provides support for implicitly specified subcontainers. An **implicitly specified subcontainer** is an object container that can be specified in an AppleScript script by context, rather than by an explicit reference. Without this feature, a script would have to fully specify the path to a word in, for example, a TextEdit document:

```
fourth word of text of front document
```

You can make life easier for scripters who use your application by saving them the trouble of writing full references to objects in situations where part of the reference can be safely assumed. For example, the following would be a simpler, but reasonable way to refer to the same word, since text is the obvious container of words in a document:

```
fourth word of front document
```

That is, you can allow `of text` to be implicitly specified by context, instead of explicitly specified in the script.

To support an implicitly specified subcontainer, you add a `contents` element to the `class` element for the containing class in your sdef file. The `contents` element indicates that a scripter can obtain the contents of an object of this class type without specifying the container that holds the contents.

For example, suppose your application has a document class, `MyTextDocument`, that handles text with an instance of the `NSTextStorage` class. `NSTextStorage` supports scriptability for words, paragraphs, and the other text items listed in Table 3-2 (page 37).

Your application can turn the text storage instance into an implicitly specified subcontainer, so that a user won't have to specify it in a script, but can specify just the document, as in the sample shown above. To do so, you add a `contents` entry for the `document` class to support, as shown in Listing 6-4. This `contents` element specifies that for a script to access the text-related contents of a `document` object, it can just specify the `document` object, without having to specify a container within the object.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

The MyTextDocument class would implement accessors that match the Cocoa key defined in the sdef: that is, contents and setContents:. For an idea of what these accessors might look like, see the similarly named accessors for the text area class (SKTTextArea) in the Sketch application.

Listing 6-4 Class definition for text document, containing a contents element

```
<class name="document" code="docu"
        description="A text document.">
    <cocoa class="MyTextDocument"/>
    <contents name="text contents" code="TeCo" type="rich text"
        description="The text of the document.">
        <cocoa key="contents"/>
    </contents>
</class>
```

When you supply a contents entry, an appropriate NSPropertySpecifier will be inserted into the object specifier containment chain when a reference using that container class would otherwise be invalid.

Note: For applications that supply scriptability information through script suite and script terminology files, implicit subcontainers are supported by a DefaultSubcontainerAttribute entry in the class dictionary. The value of the entry must be the key of one of the entries in the Attributes or ToOneRelationships dictionary of the class. For example, the TextEdit application's TextEdit.scriptSuite file includes an entry in its document dictionary to support implicitly specified text storage. The following is an excerpt from that file:

```
{
 [...]
      "Classes" = {
          [...]
          "Document" = {
              "Superclass" = "NSCoreSuite.NSDocument";
              "AppleEventCode" = "docu";
              "DefaultSubcontainerAttribute" = "textStorage";
              "ToOneRelationships" = {
                  "textStorage" = {
                      "Type" = "NSTextStorage";
                      "AppleEventCode" = "ctxt";
                  };
              };
          };
      };  }
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Script Commands

Cocoa scripting uses script command objects to handle scripting-related Apple events received by your application. This chapter describes how script commands work, and how your application uses them to support its scriptable features.

Script Commands Overview

When a user runs an AppleScript script, script statements that target your application are converted into corresponding Apple events and sent to the application. For each Apple event that corresponds to a command defined in your sdef, Cocoa scripting instantiates a **script command object** that contains all the information needed to describe the specified operation. It then executes the script command, which works with objects in your application to perform the operation. The flow of Apple events is bidirectional and script commands can return values to the originating script.

Note: Cocoa's `NSApplication` class automatically registers handlers for certain Apple events, such as `open application` and `open documents`. These handlers are described in "How Cocoa Applications Handle Apple Events" (page 101).

Cocoa scripting provides default support for many basic AppleScript commands, such as `delete`, `move`, `get`, and `set`. This support is implemented by the `NSScriptCommand` class and a number of subclasses. In some cases, it also relies on command information that you must insert into your sdef file. Beyond that, however, the default support generally requires only that your scriptable objects follow the key-value coding guidelines described in "Maintain KVC Compliance" (page 54) and that you implement object specifier methods for your scriptable classes, as described in "Object Specifiers" (page 61).

You can define new subclasses of Cocoa's script command classes to modify their default behavior or to implement new AppleScript commands specific to your application. Or, in some cases, you can simply add a command-handling method to a scriptable class and provide information in your sdef file to specify when it should be called. This chapter includes detailed information on how to perform these kinds of operations.

Script Command Classes Supplied by Cocoa

Cocoa defines Objective-C script command classes to implement the AppleScript commands from the Standard suite. These classes are listed in Table 9-4 (page 99). As part of Cocoa scripting's standard implementation, `NSScriptCommand` and its subclasses can handle the `close`, `copy`, `count`, `create`, `delete`, `exists`, `move`, `open`, and `print`commands for most applications without any subclassing. It also handles the `get` and `set` commands, which are technically not part of the Standard suite, but rather considered built-in, or intrinsic, AppleScript commands. However, if your application needs to modify any of these commands, you can do so in one of two ways:

- Supply a method to handle the command and list it in your sdef file.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

- Create a subclass of `NSScriptCommand` or one of its subclasses and override `performDefaultImplementation`.

Table 7-1 (page 85) lists the AppleScript commands supported by Cocoa scripting and summarizes their default behavior and how you can customize it.

Important: When your application defines a new Objective-C script command class, it must be a subclass of `NSScriptCommand` or of one of its subclasses.

Script Command Scriptability Information

In working with script commands, Cocoa scripting relies on scriptability information in your sdef. That includes `command` elements, which provide information about specific AppleScript commands, and `responds-to` elements, which specify that objects of a particular class can respond to a specified command.

Your sdef includes a `command` element for every new AppleScript command you create. For these commands, you specify all the appropriate information described in "Command Elements" (page 47). That includes command name, code, and description. It can also include a direct parameter, other parameters, and result type.

Note: When a command object is instantiated, parameters and result type are stored in an argument dictionary in the command. See "Script Command Components" for information on how to access items in the argument dictionary.

If you implement a new Objective-C class for the command, you also supply the name in the `command` element. If you do not supply a class name, Cocoa scripting uses the default, `NSScriptCommand`.

Your sdef should also include `command` element definitions for most of the AppleScript commands Cocoa scripting supports, if they can be used in your application:

- Your sdef should not include `command` elements for the `get` and `set` commands—they are automatically available to every application.
- Your sdef should include `command` elements for the commands in the Standard suite, such as `count`, `duplicate`, and `move`. You can obtain these definitions by copying them from the file `Sketch.sdef`, as described in "Create a Scripting Definition File" (page 43).

Cocoa also provides automatic support for the AppleScript types described in "Built-in Support for Basic AppleScript Types" (page 20).

Script Command Components

A script command object is an instance of `NSScriptCommand` or one of its subclasses, including classes defined by Cocoa and those defined by your application. A script command can have several components, which vary by command and are described in the `command` element definition for the command in the sdef file.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

- **The receiver or receivers for the command (if any):** The object or objects designated to receive the command in the application.

  If the originating Apple event specifies receivers, it does so in its direct parameter. If the Apple event does not specify receivers, it may still have a direct parameter, which is then interpreted to be the object to which to send the command.

  If a command specifies receivers, you can retrieve them from the command object with the `evaluatedReceivers` method, which converts object specifiers into references to actual objects. If the command doesn't specify receivers, you can retrieve the direct parameter with the `directParameter` method, or you can retrieve it from the arguments list, as described in the next item.

- **The arguments of the command (if any):** The arguments provide access to information from the parameters of the originating Apple event. The arguments can include direct parameter, other parameters, and result type.

  You can retrieve a dictionary of arguments from the command object with the `evaluatedArguments` method. You obtain individual arguments from this dictionary by key, where the parameter names defined in the sdef serve as keys. You can use the empty key (an empty string: `@""`) to retrieve the direct parameter (or "unnamed argument") from the argument dictionary, if the direct parameter is not the receivers specifier.

- **The class description for the command:** From the class description, you can obtain information such as argument names, command result type, AppleScript command name, and name of the Objective-C class Cocoa instantiates to perform the command. This information is used by Cocoa scripting, but less commonly by your application. See "Script Command Creation" (page 73) for a description of how Cocoa scripting obtains this information.

- **An Apple event descriptor:** This Cocoa object, of type `NSAppleEventDescriptor`, represents the Apple event itself. Your application won't necessarily need to work directly with the Apple event, but it's available if you do.

Your application typically only needs to access the components of a command when you want to modify the default behavior or implement a new script command. In those situations, the `NSScriptCommand` class provides a number of methods for obtaining the information you need, including those mentioned here (`evaluatedReceivers`, `evaluatedArguments`, and `directParameter`).

Script Command Creation

It is important to note that your application doesn't typically instantiate a script command directly. Instead, it lists the commands it can handle in its sdef file and Cocoa scripting instantiates a command object when the application receives the corresponding Apple event.

Cocoa extracts information from the Apple event and stores it in the command object. To do this, it uses application scriptability information (loaded from your sdef and stored in a global instance of `NSScriptSuiteRegistry`) to obtain the keys for the specified objects and to get data from class and command descriptions. Application scriptability information automatically includes information to support the `get` and `set` commands.

Because at this point the command's receivers and arguments are probably known only as AppleScript reference forms (for example, `graphic 3 of document "SketchDocOne"`), they are represented in the command object as nested object specifiers. See "Script Commands and Object Specifiers" (page 74) for more information.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Script Command Execution

Once Cocoa scripting has created and prepared a script command, it executes it in a series of steps:

1. It uses key-value coding (KVC) to evaluate the receivers specifier in the script command (see "Script Commands and Object Specifiers" (page 74) for details).

2. It determines which method to use for executing the command. For some commands (such as `get` and `set`), it invokes KVC methods, based on keys supplied in the sdef file, to access values of the specified objects.

   For other commands, it looks in the class descriptions of the receivers to see if any has specified a selector for the command. If not, or if there are no receivers, it selects the default implementation for the command. This mechanism is described in more detail in "Object-first Versus Verb-first Script Commands" (page 75).

3. It invokes the method indicated by the selector (which has a single argument, the script command object) or the method that implements the default behavior for the command (`performDefaultImplementation`).

4. When a command needs to return a value, Cocoa scripting packages the information in a reply Apple event and returns it. If an error occurs while executing the command, Cocoa returns the error information (including any information added by the application) in the reply Apple event. For details, see "Error Handling" (page 75).

5. If a command requires asynchronous processing, the application can suspend it, so that the application doesn't receive additional Apple events during processing. For details, see "Suspending and Resuming Apple Events and Script Commands" (page 109).

Most standard commands perform their operations automatically. For an example where you might want to modify or replace the default behavior, see "Modifying a Standard Command" (page 83).

Script Commands and Object Specifiers

When a script command is ready for execution in your application, the receiver or receivers have been set as object specifiers and any arguments may also have been set as object specifiers (arguments can be actual values as well). To represent a series of reference forms (such as `first word of second paragraph of document "Stock Alert"`), each object specifier is nested inside its container object specifier; the innermost object specifier indicates the final object to be evaluated, while the outermost object is usually the application.

The keys to an attribute or relationship are often not the same words expressed by the corresponding reference forms. For example, the key for an array of document objects is `orderedDocuments`, but the actual scripting term used is `document`. The mapping between key name and script name is provided in the application's sdef. When Cocoa scripting converts an Apple event into an objective-C script command object, it obtains the mapping between a four-character code in the Apple event and the corresponding key for the specified class, attribute, or relationship in the application. It can then locate the language-independent information (specifically, class and command descriptions) needed to compose the script command, including the object specifiers for its arguments and receivers.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

In the normal course of script-command execution, an application invokes `evaluatedReceivers` on a script command to get the receiver or receivers of the command and invokes `evaluatedArguments` to get any arguments of the command. These methods in turn invoke `objectsByEvaluatingSpecifier` on the object specifiers representing command arguments or receivers. The object specifier receiving the message is the innermost specifier as nested in its containers.

The `objectsByEvaluatingSpecifier` method goes up the chain of nested containers by asking each specifier for its container until it comes to the top-level object specifier, which has no container. The top-level object is usually the application object, but it can be an object specifier involved in a `whose` clause (`NSWhoseSpecifier`) or the container for a range evaluation. The method then invokes `objectsByEvaluatingWithContainers:` on this top-level specifier, which then proceeds down the chain of nested specifiers, evaluating each through key-value coding and using the evaluated object as the basis for the next evaluation. Evaluating the innermost specifier yields the real command receiver or receivers or any object used as a command argument.

For related information, see "A Closer Look at an Object Specifier" (page 63).

Error Handling

Your application can signal error information during script command execution by providing the command object with an error number, an error string, or both. The error information is returned in the reply Apple event. If an error occurs and your application does nothing, Cocoa scripting will supply the most applicable error number it can, along with an error string for that number. The error codes that Cocoa scripting uses for general command execution problems are listed with the documentation for the `NSScriptCommand` class.

`NSScriptCommand` supplies the `setScriptErrorNumber:` and `setScriptErrorString:` methods for setting error information.

Your command handler should only provide error information if it is specific to the operation of your application. On occasion, you may be able to use one of the codes defined by Cocoa scripting. You can also choose an error number from constants supplied by the Apple Event Manager (described in "Apple Event Manager Result Codes" in *Apple Event Manager Reference*). When you choose one of these constants, such as `errAENotASingleObject`, Cocoa scripting will supply the corresponding error string (`"Handler only handles single objects"`). You can also supply general Mac OS system error numbers (defined in `MacErrors.h`). For example, if you return `fnfErr`, the error number for `"file not found"`, AppleScript will attempt to supply an appropriate error string.

Object-first Versus Verb-first Script Commands

Cocoa script commands can be described as object-first or verb-first, depending on the receivers for the command. When a script command is executed, it looks first for receivers that can perform the desired action directly. If it finds any, it invokes the specified method on each receiver. A command of this type is called an **object-first command**—the objects perform the specified action on themselves.

Sketch implements the `rotate` command as an object-first command—for details, see "Implementing an Object-First Command—Rotate" (page 79).

If no receiver can perform the desired action, or if there are no receivers specified, the script command invokes its `performDefaultImplementation` method. When a command invokes this method, it is called a **verb-first command**—a single method performs the action (or verb) on any number of objects. To create a verb-first

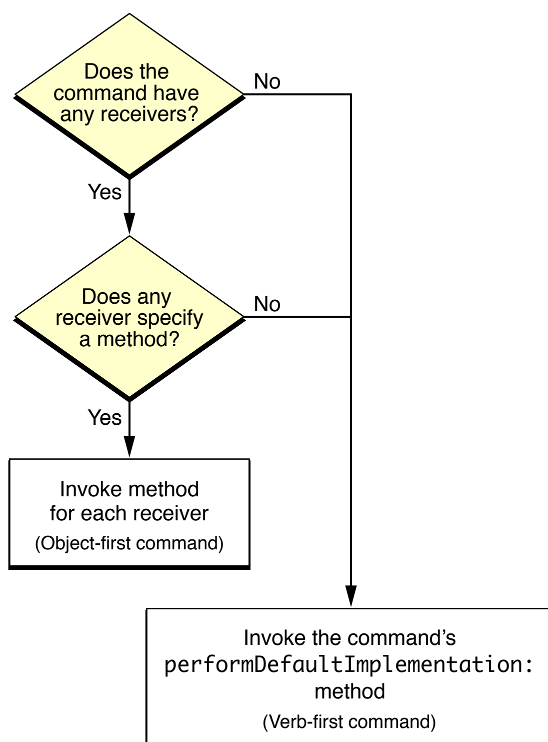
2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

command, you define a subclass of one of Cocoa's script command classes and override the `performDefaultImplementation` method (which does nothing in `NSScriptCommand`) to perform your version of the command action.

Sketch implements the `align` command as a verb-first command—for details, see "Implementing a Verb-First Command—Align" (page 81). Several of the standard commands are also verb-first commands—see Table 9-4 (page 99) for details.

Figure 7-1 shows Cocoa scripting's decision tree for executing a script command.

Figure 7-1 Executing a script command—verb-first versus object-first



About Object-first Script Commands

You can use an object-first command to modify the behavior of a standard AppleScript command (except for the `get` and `set` commands). You can also use an object-first command to quickly add a new AppleScript command to your application. For these tasks, you don't need to implement a new Objective-C command class. However, you can implement such a class if desired—for example, if you want to supply functionality in the command class that any of the receivers can invoke.

An object-first command is appropriate for an action that can benefit from polymorphism, because the same message can result in different behavior depending on the receiver. It is also appropriate for operations on relatively small numbers of objects or for simple actions that don't require peripheral information—for example, an action that calls for a simple reversal of state. Although each receiver can, if necessary, extract information from the command to aid in performing the action, such an approach could conceivably lead to performance problems, if operating on large numbers of objects.

Another advantage of using an object-first command is that you can often implement it by creating a category on an existing class.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

To support an object-first command, you perform the following steps:

- In your sdef, provide a `responds-to` entry for each scriptable class that can handle the command. Specify the same method name for each class, matching the template `<methodName>:`.
- For a new AppleScript command you have defined, add a `command` element to your sdef file. Specify all the appropriate information described in "Command Elements" (page 47).

  If you will define a new Objective-C class to implement the command, supply the class name in the `command` element.
- To use the object-first approach to change the behavior of a standard AppleScript command (but not the `get` and `set` commands), make sure your sdef includes the `command` element definition for that command, as described in "Create a Scripting Definition File" (page 43). You don't need to modify the `command` definition unless you implement a new Objective-C class for the command.

  For example, if your sdef includes a `responds-to` element that specifies a `specialMove:` method that Cocoa scripting should invoke for the `move` command, there is no need to modify the `move` command in your sdef or to subclass `NSMoveCommand`. But if you do provide a subclass of `NSMoveCommand`, you must list it in the `command` element.
- In the implementation for the corresponding scriptable classes, implement the named method. The method declaration must match one of the following two templates (depending on whether the command returns a value):

  ```
  -(id)<methodName>:(NSScriptCommand*)command
  ```

  ```
  -(void)<methodName>:(NSScriptCommand*)command
  ```
- If you are using one, implement the Objective-C class for the command; it must inherit from one of the script command classes defined by Cocoa.

For a detailed example, see "Implementing an Object-First Command—Rotate" (page 79).

About Verb-first Script Commands

A verb-first command is appropriate for an action that requires the interaction of objects, so that it cannot be handled by individual objects. It is also appropriate for operations that require significant overhead, such that it would be inefficient to invoke the same method on many objects, with each duplicating the overhead. A verb-first command can also be appropriate to customize the behavior of an existing verb-first command.

To implement a verb-first command, you perform the following steps:

- To handle a new AppleScript command for your application, add a `command` element for the class to your sdef file. Specify all the appropriate information described in "Command Elements" (page 47). That includes specifying the name of your new Objective-C class that implements the command.

  To handle a standard AppleScript command with a new command subclass, make sure your sdef includes the `command` element definition for that command (except for the `get` and `set` commands), as described in "Create a Scripting Definition File" (page 43). Modify that element to specify the name of your new Objective-C class.
- Define the command in your code as a subclass of `NSScriptCommand` or of one of the script command subclasses listed in Table 9-4 (page 99):

  Subclass `NSScriptCommand` to perform an operation which is not supported by any of the standard Cocoa script command classes.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Subclass one of Cocoa scripting's other script command classes to perform a variation of its standard action. For example, you may want to perform a custom `move` operation in some cases, but otherwise fall back on the default behavior.

- Override the `performDefaultImplementation` method. In that method, you can use methods of `NSScriptCommand` to extract any information you need from the command object.

  For example, you can examine the objects on which the command should be performed and decide whether to perform the customized version of the command. If not, you can invoke the method of the superclass:

  ```
  return [super performDefaultImplementation];
  ```

  This method either returns an `id` or, if there is nothing to return, returns `nil`. The version in `NSScriptCommand` does nothing and returns `nil`, but most subclasses of `NSScriptCommand` override this method to perform an action.

For a detailed example, see "Implementing a Verb-First Command—Align" (page 81).

Mixing Object-first and Verb-first Behavior

When implementing a new command class or overriding a Cocoa command class, you might choose to mix the verb-first and object-first approaches. For example, you might support a command that most objects in your application can handle in a specified method (object-first), but that for a certain class of objects, it is necessary to handle the action in the `performDefaultImplementation` method (verb-first).

To implement a command that mixes these behaviors, you use a combination of the same implementation steps described in "About Object-first Script Commands" (page 76) and "About Verb-first Script Commands" (page 77). That is, you provide both `responds-to` elements and a `command` element in your sdef file and you specify an Objective-C class for the command; in your code, you implement a script command subclass that overrides `performDefaultImplementation`, as well as versions of the method specified by the `responds-to` element in individual classes that can respond to the command. The result of handling an instance of the command will then depend on the class types of the objects on which it operates.

Important: Cocoa scripting cannot mix verb-first and object-first approaches in response to a single Apple event. It does the right thing if all the objects specified by an Apple event provide a method with the same signature to handle the action (object first—invoke the method) or if none of them do (verb-first—invoke the command's `performDefaultImplementation` method). However, if a single Apple event includes some objects that specify a method and some that don't, or objects that specify methods with different names, the result is undefined.

Steps for Implementing a New or Modified Script Command

To summarize from previous sections, you use these steps to implement a new command or to modify the behavior of the script commands provided by Cocoa scripting:

- For either an object-first (except the `get` and `set` commands) or a verb-first command:

  In the sdef, define a `command` element for the AppleScript command, if it doesn't already have one.

- For an object-first command:

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

- ❏ In the sdef, add a `responds-to` element to the `class` definition of each scriptable class that can handle the command. Specify the same method name for each class.
- ❏ Implement the specified method in the classes that can respond to the command. The declaration should match one of these templates:

  ```
  -(id)<methodName>:(NSScriptCommand*)command
  ```

  ```
  -(void)<methodName>:(NSScriptCommand*)command
  ```

- For a verb-first command (and if needed, for an object-first command)
  - ❏ Implement an Objective-C script command class that inherits from `NSScriptCommand` or one of its subclasses.
  - ❏ Specify the name of the Objective-C command class in the `command` element in the sdef.

For a verb-first command, override the `performDefaultImplementation` method.

Implementing an Object-First Command—Rotate

The Sketch sample application implements the `rotate` command as an object-first command—it's a logical task for a `rectangle` object to rotate itself.

To implement the object-first AppleScript command `rotate`, the Sketch application does the following:

1. It defines the `rotate` command in the file `Sketch.sdef`:

```
        <command name="rotate" code="sktcrota"
            description="Rotate objects.">
            <direct-parameter type="graphic"/>
            <parameter name="by" code="by  " type="real"
                description="degrees to rotate; positive numbers rotate
counter-clockwise.">
                <cocoa key="byDegrees"/>
            </parameter>
```

   Here's what this command definition specifies:

   a. The command name is "`rotate`" and its two-part code is "`sktcrota`". This code is used in installing a handler to respond to Apple events that specify this command.

   b. The command has a direct parameter which specifies one or more graphic objects to be rotated.

   c. The command has a parameter with the key "`byDegrees`" that specifies the degrees by which the specified objects should be rotated.

   d. This command definition does not specify a command class to implement the command, because the `rotate` command does not require a new command class. Because it doesn't specify a command class, the default class will be used (`NSScriptCommand`).

   e. This command definition does not include a `result type` element, so the Objective-C method that handles the command should return `nil`.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

2. In its sdef, Sketch also adds a responds-to element to the rectangle class (rectangles are the only graphics that can be rotated by this command).

```
<responds-to name="rotate">
    <cocoa method="rotate:"/>
</responds-to>
```

   This element definition specifies that for a rotate command, Cocoa scripting should call the rotate: method of the rectangle object to be rotated.

3. In the implementation for the SKTRectangle class, it implements the rotate: method. Here is a summary of how rotate: works:

   a. It invokes [command evaluatedArguments] on the passed command object to get a dictionary (theArgs) containing the evaluated arguments for the command. The arguments have been evaluated from object specifiers to objects if necessary. The keys in the dictionary are the argument names, specified in the sdef.

   b. It invokes [theArgs objectForKey:@"byDegrees"] to obtain the argument for the number of degrees to rotate, as an instance of NSNumber. The key "byDegrees" corresponds to the Cocoa key defined for the "by" parameter in the rotate definition in Sketch's sdef file.

   c. It obtains the number of degrees to rotate by from the number object and performs some math operations to determine whether to rotate the rectangle.

   d. In the case that it needs to rotate the rectangle, it does so by flipping the width and height and modifying the position.

   e. If there is an error, it invokes [self setScriptErrorNumber:theError] to supply an error number.

      You can also invoke setScriptErrorString: to supply an error message.

   f. Because the rotate command does not declare a result type element in its sdef definition, this method should return nil.

Note: To support various AppleScript commands, scriptable Sketch classes such as SKTGraphics and SKTDrawDocument maintain KVC compliance in naming the accessors for their scriptable properties. In addition, SKTGraphics implements an object specifier method (shown in Listing 6-1 (page 67), as does NSDocument (the superclass of SKTDrawDocument).

Here is an AppleScript script that exercises the rotate command:

Listing 7-1 A script to test the rotate command

```
tell application "Sketch"
    with timeout of 60 * 60 seconds
        tell document 1
            get orientation of every rectangle
            set x to every rectangle
            repeat with y in x
                rotate y by 90
            end repeat
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
            try
                rotate rectangle 1 by 80
            on error eMsg number eNum
                log {eNum, eMsg}
            end try
            get orientation of every rectangle
        end tell
    end timeout
end tell
```

Here's what this script does:

1. It sets a long timeout value so it won't time out (and be interrupted) during execution if you break in the application to debug its scriptability support.
2. It performs a series of tests on the first document:
   a. It gets the `orientation` property of every `rectangle` object.
   b. It rotates every `rectangle` object by 90 degrees.
   c. It uses a `try` block to test for an error condition (rotating by a value that is not a multiple of 90).
   d. It gets the `orientation` property of every `rectangle` object after rotating.

You might add additional tests to this script, such as the following:

1. Try to rotate by different degrees.
2. Try to rotate objects that aren't rectangles.
3. Delete all graphics, then try to rotate with no rectangles in the document.
4. Rotate each rectangle by flipping its orientation property.

Implementing a Verb-First Command—Align

The Sketch sample application implements the `align` script command as a verb-first command. It makes sense to let the `performDefaultImplementation` method align all the specified objects in an array of objects, whereas asking each object to align itself would require the objects to know or find out about other objects to align with.

To implement the verb-first AppleScript command `align`, the Sketch application does the following:

1. It defines the `align` command in Sketch's sdef file

```
<command name="align" code="sktcalig"
    description="Align a set of objects.">
    <cocoa class="SKTAlignCommand"/>
    <direct-parameter>
        <type type="graphic" list="yes"/>
    </direct-parameter>
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
        <parameter name="to" code="to  " type="edge">
            <cocoa key="toEdge"/>
        </parameter>
    </command>
```

Here's what this command definition specifies:

a. The command name is "align" and its two-part code is "sktcalig". This code is used in installing a handler to respond to Apple events that specify this command.

b. To handle the command, Cocoa scripting should implement an instance of SKTAlignCommand.

c. The command has a direct parameter which supplies a list of graphics to be aligned.

d. The command has a parameter that specifies the edge to which the objects should be aligned.

e. This command definition does not include a result type element, so the Objective-C code that handles the command should return nil.

In its sdef, Sketch also defines an edge enumeration (not shown) to define edge constants for use with the align command.

2. Sketch adds two files to its Xcode project to define the SKTAlignCommand class: SKTAlignCommand.h and SKTAlignCommand.m.

   This command is a subclass of NSScriptCommand, containing one method, performDefaultImplementation. That method overrides the version in NSScriptCommand.

3. Here is a summary of how performDefaultImplementation works for the align command:

   a. It determines the receivers for the command (an array of graphic objects to align).

   b. It invokes [self evaluatedArguments] to get a dictionary (theArgs) containing the evaluated arguments for the command. The arguments have been evaluated from object specifiers to objects if necessary. The keys in the dictionary are the argument names, specified in the sdef.

   c. It invokes [theArgs objectForKey:@"toEdge"] to obtain the argument for the edge to align to. The key "toEdge" corresponds to the Cocoa key defined for the "to" edge parameter in the align definition in Sketch's sdef file.

      From that argument, it obtains the value of the edge to align to.

   d. It gets the bounds for the first object in the array of graphics objects. That is the object to which any other objects will be aligned.

   e. It iterates over the array of graphics objects to align, using a mechanism to align them that depends on the specified edge to align to.

   f. If there is an error, it invokes [self setScriptErrorNumber:theError] to supply an error number.

      You can also invoke setScriptErrorString: to supply an error message.

   g. Because the align command does not declare a result type element in its sdef definition, this method should return nil.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

The `performDefaultImplementation` method for `SKTAlignCommand` never invokes the implementation of its superclass (`NSScriptCommand`) for two reasons:

- The version in `NSScriptCommand` doesn't do anything—it exists only to be overridden.
- The implementation in `SKTAlignCommand` does all the work necessary to implement the `align` command.

Here is an AppleScript script that exercises the `align` command:

Listing 7-2 A script to test the align command

```
tell application "Sketch"
    with timeout of 60 * 60 seconds
        tell document 1
            align every graphic to vertical centers
            delay 3
            set x to every graphic
            align x to horizontal centers
        end tell
    end timeout
end tell
```

Here's what this script does:

1. It sets a long timeout value so it won't time out (and be interrupted) during execution if you break in the application to debug its scriptability support.
2. It tells the first document to align every graphic vertically.
3. After a 3 second delay, it tells the document to align every graphic horizontally.

You can make this test script more complete by, for example, adding statements to:

1. Test the other alignment options defined in Sketch's sdef: `left edges`, `right edges`, `horizontal centers`, `top edges`, and `bottom edges`.
2. Align a range of graphics.
3. Delete all graphics, then try to align with no graphics in the document.

Modifying a Standard Command

The `NSMoveCommand` is part of Cocoa's built-in scripting support for standard AppleScript commands. It works automatically to support the `move` command through key-value coding. However, there are situations where you might want to override this command, using either a verb-first or an object-first approach. This section provides some tips for this task—a complete solution is beyond the scope of this document.

Here is the sdef entry for the `move` command, which shows the direct parameter and other parameters to the command:

```
<command name="move" code="coremove"
        description="Move object(s) to a new location.">
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
            <cocoa class="NSMoveCommand"/>
            <direct-parameter type="specifier" description="the object(s) to
move"/>
            <parameter name="to" code="insh" type="location specifier"
                description="The new location for the object(s).">
                <cocoa key="ToLocation"/>
            </parameter>
```

Important: The `NSMoveCommand` class overrides the `setReceiversSpecifier:` method of `NSScriptCommand`, so that the container, rather than the objects to be moved, becomes the receiver for the command. So the specifier returned by `receiversSpecifier` may be different than the specifier set by `setReceiversSpecifier:`. For more information, see the documentation for `NSMoveCommand`.

Other command classes that override `setReceiversSpecifier:` are: `NSCloneCommand`, `NSDeleteCommand`, and `NSSetCommand`.

A Verb-first Move Command

Suppose your application lets scripters work with terms such as `file` and `folder`, providing a familiar terminology to access data that is actually stored in a database. However, your underlying implementation does not support operations on this data using KVC accessors whose keys can be mapped to `"file"` and `"folder"`. Or perhaps you tried the standard `move` support and found that you need increased performance for moving large numbers of objects.

Using the verb-first approach, you can subclass `NSMoveCommand` and override `performDefaultImplementation`. The `NSMoveCommand` class is already a verb-first command, and it generally makes sense to have a higher-level object within your application supervise the movement of contained objects, rather than telling each object to move itself.

In your override of `performDefaultImplementation`, you can extract information from the command and translate it into appropriate operations on the underlying data. For example, you can examine the objects to move:

- If they are `file` or `folder` objects, you make the appropriate changes in the database to reflect the changes expected in the user's (AppleScript object model) view of the world.
- For other objects, you can use the following statement to apply the standard behavior supplied by `NSMoveCommand`:

  ```
  return [super performDefaultImplementation];
  ```

An Object-first Move Command

Suppose, on the other hand, that certain objects in your application do know how to move themselves, and they use a mechanism different from the KVC-based moves supported by `NSMoveCommand`, which work with standard containers. In this situation, you could use the object-first mechanism to allow objects of a certain class to handle the `move` command directly in a method defined for that class.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

To implement this approach, you use the steps described previously in "About Object-first Script Commands" (page 76). In your handler methods, you extract the information you need from the command object (the location to move to), which is passed as a parameter to the method, then perform the move for the current object.

Summary of AppleScript Command Support

Table 7-1 lists AppleScript commands supported by Cocoa scripting. For each command, it shows the Objective-C class that executes the command. It also describes the default handling for the command and how to customize it.

Though not mentioned in the table, for classes that recommend verb-first modification, you can generally use the object-first approach as well. This is particularly useful if you want to modify behavior on a class-by-class basis.

Remember too that most Cocoa script commands rely on you to maintain KVC-compliance in the naming of scriptable properties in your scriptable classes, and in some cases to implement object specifier methods for those classes. Otherwise, the commands cannot identify objects in your application on which to operate, or get or set values in those objects.

For additional information on default handling, see "Apple Events Sent by the Mac OS" (page 102).

Table 7-1 Default support for AppleScript commands and how to customize it

AppleScript command	Objective-C class	Default support and how to customize it
`close`	`NSCloseCommand`	Cocoa scripting automatically handles `close` for windows and documents, with the window version commonly passing control to the window's document. Customize by overriding the `handleCloseScriptCommand:` method in a subclass of `NSDocument` or `NSWindow`, depending on the object. (See also the documentation for `NSWindowScripting`.) To support `close` for a different class, you can use the mechanism described in "About Object-first Script Commands" (page 76).
`count`	`NSCountCommand`	Handled by verb-first command. Customize by implementing a verb-first subclass, as described in "About Verb-first Script Commands" (page 77).
`delete`	`NSDeleteCommand`	Handled by verb-first command. Customize by implementing a verb-first subclass, as described in "About Verb-first Script Commands" (page 77).
`duplicate`	`NSCloneCommand`	Handled by a verb-first command that invokes `copyWithZone:` on the objects to be duplicated. Customize in your implementation of that method or by implementing a verb-first subclass, as described in "About Verb-first Script Commands" (page 77).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

AppleScript command	Objective-C class	Default support and how to customize it
exists	NSExistsCommand	Handled by verb-first command. Customize by implementing a verb-first subclass, as described in "About Verb-first Script Commands" (page 77).
get	NSGetCommand	Handled by verb-first command. Customize by implementing a verb-first subclass, as described in "About Verb-first Script Commands" (page 77). You do not need a command element in your sdef file for the get command.
make	NSCreateCommand	Handled by verb-first command. Customize by implementing a verb-first subclass, as described in "About Verb-first Script Commands" (page 77).
move	NSMoveCommand	Handled by verb-first command. Customize by implementing either a verb-first or object-first approach, as described in "Modifying a Standard Command" (page 83).
open documents	NSScriptCommand	Cocoa automatically handles open documents by invoking the methods described in "Open" (page 104). You can modify the default behavior by implementing or overriding those methods.
print documents	NSScriptCommand	Cocoa automatically handles print for documents by invoking the methods described in "Print" (page 105). You can modify the default behavior by implementing or overriding those methods.
quit	NSQuitCommand	The NSApplication class automatically handles this command for applications by invoking the methods described in "Quit" (page 106). You can modify the default behavior by implementing or overriding those methods.
save	NSScriptCommand	Cocoa automatically handles save for windows and documents, with the window version commonly passing control to the window's document. Customize saving behavior by overriding the handleSaveScriptCommand: method in a subclass of NSDocument or NSWindow, depending on the object. (See also the documentation for NSWindowScripting.) To support save for a different class, you can use the mechanism described in "About Object-first Script Commands" (page 76).
set	NSSetCommand	Handled by verb-first command. Customize by implementing a verb-first subclass, as described in "About Verb-first Script Commands" (page 77). You do not need a command element in your sdef file for the set command.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Testing, Debugging, and Performance

This chapter lists tactics you can use to test and debug your scriptable Cocoa application. It also provides a brief list of possible performance issues, including links to performance information elsewhere in this document.

Scriptability Test Plan

As noted in "Designing for Scriptability" (page 27), planning for testing should be an integral part of your application design and implementation. Your test plan should include regular milestones to confirm each step of scriptability. These can include:

1. Validating your terminology (verifying that your sdef is semantically correct).

2. Verifying that the application is receiving Apple events.

3. Verifying that your script commands get instantiated and executed when the corresponding Apple events are received.

4. Verifying that the expected methods are called to get and set scriptable values.

5. Verifying that support for more complex script statements you support, such as `every` and `whose` statements, is working. (For examples of these types of statements, see the object specifiers for the filter (`whose`) and `every` reference forms in Table 6-1 (page 65).)

Your test plan should also include performance testing. For more information, see "Performance Issues for Scriptability" (page 93).

Use AppleScript Scripts to Test Your Application

As you implement your application, you should build up a suite of test scripts to exercise its scriptability. And the more features you make scriptable, the more your script testing can serve as a testbed for your entire application.

From Script Editor, you can cut and paste the Event Log output from your test scripts and save it for later regression testing. As you make changes to your application, whether in script-specific code or not, you can compare the current output to the saved version to help you find possible side-effects.

Make your test scripts as complete as possible by following these guidelines:

- Exercise the entire hierarchy of your AppleScript object model: for example, have your scripts get and set every accessible property of every accessible object, using `repeat` loops and `every` statements.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

After getting and setting values, use a `log` statement to log the `properties` property of the objects to record the changes.

- Test access by every applicable reference form: for example, iterate over scriptable collections by name, by ID, by index, and by any other form that applies. (See Table 6-1 (page 65) for a list of reference forms.)
- Test creating new objects by using the `make` command and specifying properties.

  For example, this script tests Sketch by creating a new graphic object in an existing document:

Listing 8-1 Simple test script

```
tell application "Sketch"
    tell document "Test Doc.sketch"
        make new circle at end with properties {x position:50, y position:30,
width:60, height:60}
    end tell
end tell
```

- Test using `whose` clauses: they're a powerful scripting feature that is important to users.

  A `whose` clause can provide a good stress test for your scriptability, and may turn up performance issues.
- Run scripts that make changes then reverse them, then check whether the end result matches the starting point.
- Run your test scripts regularly, even after code changes that you don't expect to affect your application's scriptability.

For additional information on using test scripts, including examples, see "Implementing a Verb-First Command—Align" (page 81) and "Implementing an Object-First Command—Rotate" (page 79).

Turn On Debugging Output for Scripting

Turning on Cocoa's debugging output for scripting will help you debug your scriptable application. Debugging output shows information such as

- loaded scriptability information (from an sdef file or from script suite and script terminology files)
- executed script commands
- the arguments to executed commands

To turn on this support, you set a debugging value in the user defaults system provided by Mac OS X. Default values in the global domain are accessible to any application. For example, during development, an application can set a user default value that a debugger checks to determine whether to display certain debug information. That's the approach that Cocoa takes to log debug information for scripting.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Note: You can read more about Cocoa's support for the defaults system in the document *User Defaults Programming Topics*.

Steps for Turning On Cocoa Debugging Output

To turn on Cocoa's debugging output for scripting, you do the following:

1. Open a Terminal window. Terminal is available in `/Applications/Utilities`.
2. Enter the following line and press Return to execute it:

   ```
   defaults write NSGlobalDomain NSScriptingDebugLogLevel 1
   ```

   You can turn off debugging output with a line like this:

   ```
   defaults write NSGlobalDomain NSScriptingDebugLogLevel 0
   ```

   You can display all the current values in the global domain with a line like this:

   ```
   defaults read NSGlobalDomain
   ```

3. If your application is already running, quit the application.
4. If you launch your application from the Finder, look for debug information in the Console application (available in `/Applications/Utilities`). If you launch your application in Xcode, look for debug information in the Debug Console pane or in the Run pane.

You will not see any debugging information until the application receives an Apple event that causes it to execute a script command (and in turn, to load the application's scriptability information). You can view debugging information for either a development or a deployment build of your application.

If you only want to turn on script debugging for a particular application, you can use the application domain. The application domain is identified by the bundle identifier of the application, typically a string in the form of a Java-style package name (think of it as a reverse URL). For example, you could turn on script debug logging for the Sketch sample application (available from Apple) by executing the following line:

```
defaults write com.apple.CocoaExamples.Sketch NSScriptingDebugLogLevel 1
```

To read this value or to reset it to zero, use one of the following two lines:

```
defaults read com.apple.CocoaExamples.Sketch NSScriptingDebugLogLevel
defaults write com.apple.CocoaExamples.Sketch NSScriptingDebugLogLevel 0
```

Sample Output

The following listing shows a script that changes the height of a circle in a Sketch application window.

```
tell application "Sketch"
    tell the first document
        set height of first circle to 100
    end tell
end tell
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Listing 8-2 shows the debug output from running the script shown above on a version of Sketch that supplies its scriptability information with an sdef file. The output shows the command listed as `Command: Intrinsics.set`, because Cocoa scripting automatically supplies information for various "intrinsic" AppleScript terminology.

The debug information includes the direct parameter (the property to set, `height`), the receivers for the command (a specifier for `circles 1 of orderedDocuments 1`), and the arguments for the command (the value to set, `100`). You can also see that Sketch returns a result of `null`, because no return value is needed for a `set` command.

Listing 8-2 Debug scripting output for sdef-based Sketch

```
2006-02-24 13:51:29.951 Sketch[8245] Command: Intrinsics.set
    Direct Parameter: <NSPropertySpecifier: height>
    Receivers: <NSIndexSpecifier: circles 1 of orderedDocuments 1>
    Arguments: {Value = <NSAppleEventDescriptor: 100
2006-02-24 13:51:29.953 Sketch[8245] Result: (null)
```

Listing 8-2 shows output from running the same script on a version of the Sketch application that supplies its scriptability information in the script suite format. There are two main differences from the previous listing:

- First, the output shows the Standard suite (formerly called the Core suite), Text suite, and Sketch suites—you will see this information the first time the application loads those suites in response to receiving an Apple event.
- Second, the command is listed as `Command: NSCoreSuite.Set`, reflecting the suite format for providing intrinsic scriptability information.

Listing 8-3 Debug scripting output for script suite-based Sketch

```
2005-05-20 13:38:39.736 Sketch[2005-05-20 13:40:55.215 Sketch[516] Suite NSCoreSuite,
apple event code 0x3f3f3f3f
2005-05-20 13:40:55.223 Sketch[516] Suite NSTextSuite, apple event code 0x3f3f3f3f
2005-05-20 13:40:55.253 Sketch[516] Suite Sketch, apple event code 0x736b7463
2005-05-20 13:40:55.259 Sketch[516] Command: NSCoreSuite.Set
    Direct Parameter: <NSPropertySpecifier: height>
    Receivers: <NSIndexSpecifier: circles 1 of orderedDocuments 1>
    Arguments: {Value = 100; }
    Key Specifier: <NSPropertySpecifier: height>
2005-05-20 13:40:55.261 Sketch[516] Result: (null)
```

Debugging Scriptability Information

To view an sdef file in a dictionary viewer, double-click it in the Finder. Double-clicking an sdef file in an Xcode project similarly opens it in a dictionary viewer window. To view or edit the XML for the file, open the sdef file with any plain text editor; in Xcode, select the sdef file and choose File > Open As > Plain Text File.

As you add information to an sdef file, you can verify that the file is still valid (semantically correct) by opening it with Script Editor or Xcode (in Mac OS X version 10.4). If those applications cannot parse the file, you will see “Nothing to see here; move along.” displayed in the dictionary viewer. You can then open the Console log to see specific parsing errors.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Important: To test your application's scriptability after making changes to your sdef file, you must quit the application, rebuild, and relaunch it to pick up the changes. In addition, the Script Editor application caches scriptability information, so you may need to quit and relaunch Script Editor as well.

Checking an sdef File with xmllint

You can also validate your sdef with the `xmllint` tool, using a command line like the following:

```
xmllint --valid YourApp.sdef
```

The `--valid` parameter causes the `xmllint` tool to determine if the document is a valid instance of the included Document Type Definition (DTD). By default, `xmllint` also checks to determine if the document is well-formed. (In "Version and document type in an sdef file" (page 43), you can see how the `sdef.dtd` file is specified.)

Examining Scriptability Information in Your Application

The `description` methods of various Cocoa scripting classes can provide useful information. You can view this information during a debugging session with the `print object` command in the GDB debugger. (This command can be abbreviated as `po`.) For example, you can examine the information Cocoa scripting has extracted from your sdef file or script suite and script terminology files. To do so, follow these steps:

1. Debug the application in Xcode.
2. Break anywhere in the application.
3. Open the Debugger Console Log and enter this command:

```
print object [NSClassFromString(@"NSScriptSuiteRegistry") sharedScriptSuiteRegistry]
```

As a result of these steps, you will get a detailed listing of the scriptability information `NSScriptSuiteRegistry` has loaded for the application. Listing 8-4 shows just the `get` and `set` command information from that output.

Listing 8-4 Partial output of NSScriptSuiteRegistry information

```
 Command: get ('core'/'getd')
        Implementation class: NSGetCommand
        Name: "get", description: "Returns the value of the specified object(s)."
        Unnamed argument ('----'), type: specifier ('obj '), optional: no
            (No user-readable name or description needed for unnamed arguments)
        Result type: any ('****')
            Description: <none>
    Command: set ('core'/'setd')
        Implementation class: NSSetCommand
        Name: "set", description: "Sets the value of the specified object(s)."
        Unnamed argument ('----'), type: specifier ('obj '), optional: no
            (No user-readable name or description needed for unnamed arguments)
        Argument: Value ('data'), type: any ('****'), optional: no
            Name: "to", description: "The new value."
        Result type: <none> ('null')
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
  Description: <none>
```

Similarly, you can get descriptions of instances of the `NSScriptCommand` and `NSAppleEventDescriptor` classes. For example, you can use the following command at a break point during script command execution:

```
print object [NSClassFromString(@"NSScriptCommand") currentCommand]
```

If you need to know more about specific Apple events received by your application, you can examine the information for the Apple event that triggered the current scripting command with a statement like the following:

```
print object [[NSClassFromString(@"NSScriptCommand") currentCommand] appleEvent]
```

You can get similar information by using `NSLog` statements in your application, allowing you to track execution through your code as you implement scriptability support. For example, the Sketch application contains the file `my.h`. That file provides definitions to turn logging on or off based on the value of `myMasterSwitch`, shown in Listing 8-5. Set it to `1`, recompile, and any `NSLog` statements in your application will be executed:

Listing 8-5 Turning on log statements

```
#define    myMasterSwitch    ( 0 )

#if        myMasterSwitch
#define    myLog1(x)    NSLog(x)
#define    myLog2(x,y)    NSLog(x,y)
#else
#define    myLog1(x)
#define    myLog2(x,y)
#endif
```

To dump information for a command object in Sketch, you could place this `NSLog` statement in the `performDefaultImplementation` method of the `SKTAlignCommand` class:

```
myLog2(@"SKTAlignCommand performDefaultImplementation command = %@", self);
```

Here's the result, from Xcode's Console window (after setting `myMasterSwitch` to `1` in `my.h` and recompiling):

Listing 8-6 NSLog output for SKTAlignCommand

```
2006-02-01 13:15:31.622 Sketch[1662] ME SKTAlignCommand
performDefaultImplementation command = Sketch Suite.align
    Direct Parameter: <CFArray 0x3d3780 [0xa073a150]>{type = mutable-small,
count = 2, values = (
    0 : <NSIndexSpecifier: graphics 1 of orderedDocuments named "SketchDoc">
    1 : <NSIndexSpecifier: graphics 2 of orderedDocuments named "SketchDoc">
)}
    Receivers: (null)
    Arguments: {
        "" = (
          <NSIndexSpecifier: graphics 1 of orderedDocuments named "SketchDoc">,
           <NSIndexSpecifier: graphics 2 of orderedDocuments named "SketchDoc">
        );
        toEdge = 1986359907;
    }
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

In this output, you can see that the direct parameter uses index specifiers to specify two graphics. The direct parameter (with the same two specifiers) is also displayed in the arguments array, identified by the empty string `("")` key. The edge to align the graphics to is also provided as an argument, with the key `"toEdge"`.

Additional Debugging Tips

There are a number of Apple event debugging tips that work well both for applications that use the Apple Event Manager directly and those that use frameworks such as Cocoa. For example, the chapter Testing and Debugging Apple Event Code in *Apple Events Programming Guide* describes how you can:

- Determine if your application is receiving Apple events (and log the information in those events).
- Use Script Editor as a test tool to send events to your application.
- Observe Apple events for multiple applications.
- Find third-party resources for monitoring and debugging Apple events and scriptable applications.

Performance Issues for Scriptability

Performance considerations should be an integral part of your test plan, so that as you implement your application, you always know when its performance increases or decreases. You may have a pretty good idea of which code paths are most likely to cause trouble, but acquiring regular timing information will help avoid surprises.

In general, you won't have to worry about performance in receiving Apple events and translating them into command objects, as applications don't commonly receive great numbers of Apple events, and Cocoa can decode them quite rapidly. However, your application should not rely on Apple events to communicate information that is more suitable to a lighter weight form of communication, such as notifications. See System-Level Technologies in *Mac OS X Technology Overview* for more information on the interprocess communications options available in Mac OS X.

If an Apple event initiates a script command that requires your application to perform an asynchronous (and perhaps lengthy) operation, you can suspend the command and resume it when the operation is complete, as described in "Suspending and Resuming Apple Events and Script Commands" (page 109). Depending on the operation, you may want to provide progress information while the command is suspended. However, suspending a command (or an Apple event) does not prevent the originating script from timing out if your application takes too long to respond to an event.

Performance can also be an issue in determining how to implement a particular application script command. For example, it may be easier to implement an object-first command, where the command calls a method for each object on which it operates. However, using an object-first approach may have performance consequences in cases where significant overhead is incurred for each object, especially when dealing with large numbers of objects. As an alternative, using the verb-first approach may allow you to minimize overhead and optimize operations on multiple objects. These approaches are described in "Object-first Versus Verb-first Script Commands" (page 75).

For most applications, performance of Cocoa scripting's default handling of `whose` clauses should be sufficient. However, if you find the need to speed it up, you can use the mechanism described in "Implementing A Method for Evaluating Object Specifiers" (page 68).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Cocoa scripting relies on key-value coding (KVC) to get and set values and find objects in your application. Performance of KVC can vary depending on the methods you implement to support it. This is especially true for operations on arrays. For tips on handling these issues, see "Performance Considerations With KVC" (page 55).

For a general introduction to performance issues, see *Getting Started with Performance*.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Cocoa Scripting Classes and Categories

The tables in this chapter provide brief descriptions for the listed Cocoa scripting classes. The accompanying material provides information on when your application uses these classes, as well as hints on which ones you might need to subclass.

About thirty public classes in Cocoa's Foundation framework support the basic structure of Cocoa scripting. Several methods in the Application Kit framework add scriptability features for applications, windows, documents, and text objects. Together, this provides support for the AppleScript commands listed in "Summary of AppleScript Command Support" (page 85) (such as `get`, `set`, `move`, `delete`, and so on).

In many cases, Cocoa scripting creates and manipulates instances of these classes, such that your application needs only to respond when a method of a particular application object is invoked. That is, your application rarely needs to declare or instantiate any of the basic Cocoa scripting classes.

On the other hand, some applications will need to define subclasses of one or more of Cocoa scripting's command classes to provide support for operations specific to the application. Even in those cases, however, the application is not responsible for creating instances of the commands—Cocoa scripting does that, based on the scriptability information provided in the application's sdef file. The process of working with commands is described in detail in "Script Commands" (page 71).

There is one case where your application typically creates instances of Cocoa scripting classes. In object specifier methods for your scriptable classes, you'll create instances of the object specifier classes listed in Table 9-2 (page 96).

Script Commands and Scriptability Information

The following classes provide the base class for script commands, the context in which commands are executed, and the scriptability information associated with an application. Instances of these classes are created automatically by the Cocoa scripting, in a process described in "Script Commands Overview" (page 71). Except for `NSScriptCommand`, most applications will not need to subclass or even call methods of these classes.

Table 9-1 Scripting information and command classes

Class	Description
`NSScriptSuite-Registry`	A shared instance of this class loads and registers the scriptability information associated with an application, whether from sdef files or the older script suite and script terminology files. Provides methods to get loaded suites, class descriptions, and command descriptions, but applications rarely call these methods.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Class	Description
NSClassDescription	Abstract class that provides the interface for querying the properties of a class. Instantiated by the global instance of NSScriptSuiteRegistry when it loads the application's scriptability information.
NSScriptClass-Description	A subclass of NSClassDescription that represents a description of a scriptable class in a script suite. Provides methods to get attributes, relationships, supported commands, and related information for a scriptable class. Instantiated by the global instance of NSScriptSuiteRegistry when it loads the application's scriptability information.
NSScriptCommand-Description	A subclass of NSObject that represents a definition of a command supported by a suite. Provides methods to get command class and return and argument types for a script command class. Instantiated by the global instance of NSScriptSuiteRegistry when it loads the application's scriptability information.
NSScriptCommand	Encapsulates an AppleScript command sent to an application as an Apple event. Uses its methods to evaluate object references (receivers and arguments) and execute the command. Cocoa scripting subclasses for the major AppleScript commands are described in Table 9-4 (page 99). You can create your own subclasses to handle operations specific to your application. For more information, see "Object-first Versus Verb-first Script Commands" (page 75).
NSScriptExecution-Context	Represents the context in which an AppleScript command is executed and tracks global state related to that command. You do not need to subclass this class.

Object Specifiers, Logical Tests, and Related Categories

NSScriptObjectSpecifier, an abstract class. Instances of these classes—object specifiers—know how to evaluate themselves within the context of a container object specifier. Some of these classes represent relative or logical tests performed with object specifiers (particularly NSWhoseSpecifier objects).

These are among the few classes provided by Cocoa scripting that your application routinely instantiates. It does so when creating object specifiers. You shouldn't need to subclass these classes, but you will need to implement some of the methods in the described categories, particularly in providing object specifier methods for your scriptable objects. For detailed information, see "Object Specifiers" (page 61).

Table 9-2 Object specifiers and related classes

Class or category	Description
NSScriptObject-Specifier	An abstract parent class for subclasses that represent AppleScript references. An object specifier knows how to evaluate itself (to actual objects) in the context of a container specifier.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Class or category	Description
NSIndexSpecifier	A subclass of NSScriptObjectSpecifier for object specifiers that specify an object in a collection by index. Though scripters typically specify one-based items, an index specifier, which typically locates objects within an array that correspond to the specified items, uses zero-based values.
NSMiddleSpecifier	A subclass of NSScriptObjectSpecifier for object specifiers that specify the middle object in a collection.
NSNameSpecifier	A subclass of NSScriptObjectSpecifier for object specifiers that specify an object in a collection by name.
NSPositionalSpecifier	A subclass of NSObject for object specifiers that represent an insertion point by reference to a point before or after another object, or at the beginning or end of a collection. It contains an object specifier that represents the object referred to for position.
NSPropertySpecifier	A subclass of NSScriptObjectSpecifier for object specifiers that represent an attribute or relationship of an object.
NSRandomSpecifier	A subclass of NSScriptObjectSpecifier for object specifiers that specify an arbitrary object in a collection.
NSRangeSpecifier	A subclass of NSScriptObjectSpecifier for object specifiers that specify a range of objects in a collection by indexes. Though scripters typically specify a one-based range of items, a range specifier, which typically locates objects within an array that correspond to the specified items, uses zero-based values.
NSRelativeSpecifier	A subclass of NSScriptObjectSpecifier for object specifiers that specify the position of an object in relation to another object.
NSUniqueIDSpecifier	A subclass of NSScriptObjectSpecifier for object specifiers that specify an object in a collection by unique ID.
NSWhoseSpecifier	A subclass of NSScriptObjectSpecifier for object specifiers that specify an object in a collection that matches a specified condition defined by a Boolean expression.
NSScriptObject-Specifiers	Category on NSObject that defines methods that scriptable objects can implement to provide a fully specified object specifier to themselves within an application, and to perform their own specifier evaluation.
NSLogicalTest	A subclass of NSScriptWhoseTest for objects that represent the Boolean operations AND, OR, and NOT; used with one or more instances of NSSpecifierTest.
NSSpecifierTest	A subclass of NSScriptWhoseTest for objects that represent a comparison between two objects (which can be object references before being evaluated) using a given comparison method.
NSScriptWhoseTest	An abstract class for objects that represent Boolean expressions (qualifiers) involving object specifiers (also called whose clauses, as in every word whose color is blue).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Class or category	Description
NSComparisonMethods	Category on NSObject that defines a set of default comparison methods useful for the comparisons in NSSpecifierTest.
NSScripting-ComparisonMethods	Category on NSObject that defines a set of additional comparison methods you may need to implement for comparisons in cases where the correct way to compare two objects for scripting is different from the correct way to compare objects otherwise.

Note: For additional information on subclasses of NSScriptObjectSpecifier and the reference forms they represent, see Table 6-1 (page 65) in "Object Specifiers" (page 61).

Key-Value Coding and Value Coercion

The following perform essential functions related to scripting. For information on the use of these classes, see "Key-Value Coding and Cocoa Scripting" (page 53) and "Coercion" (page 59).

Table 9-3 Scripting utilities

Class or category	Description
NSScriptCoercion-Handler	A shared instance of this class coerces object values to objects of another class, using information supplied by classes who register with it. Coercions frequently are required during key-value coding. For more information, see "Coercion" (page 59).
NSScriptKeyValueCoding	Category on NSObject that defines additions to the implementation of key-value coding related to scripting, such as getting and setting key values by index in collections and coercing (or converting) a key value.

Subclasses for Standard AppleScript Commands

The following classes implement standard AppleScript commands. They are all subclasses of NSScriptCommand, which is described in Table 9-1 (page 95). Your application can create a subclass of one of these classes to replace the default behavior, or to selectively modify that behavior in some circumstances. In most cases, the default behavior should be sufficient.

For more information on working with commands, see "Script Commands" (page 71).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Table 9-4 Subclasses for standard script commands

Class	Description
`NSCloneCommand`	Copies the specified scriptable object or objects (such as words, paragraphs, images, and so on) and inserts them in the specified location. This class handles the `duplicate` AppleScript command.
`NSCloseCommand`	Closes the specified scriptable object or objects—typically a document or window.
`NSCountCommand`	Counts the number of items of a specified class in the specified object container (such as the number of rectangles in a document).
`NSCreateCommand`	Creates the specified scriptable object (such as a document or graphic), optionally supplying the new object with the specified attributes. This class handles the `make` AppleScript command.
`NSDeleteCommand`	Deletes the specified scriptable object or objects.
`NSExistsCommand`	Determines whether a specified scriptable object, such as a word, paragraph, or image, exists.
`NSGetCommand`	Gets the specified value or object from the specified scriptable object. For related information, see "Getting and Setting Properties and Elements" (page 53).
`NSMoveCommand`	Moves the specified scriptable object or objects. For related information, see "Modifying a Standard Command" (page 83).
`NSQuitCommand`	Quits the specified application.
`NSSetCommand`	Sets one or more attributes or relationships of the specified scriptable object to one or more values. For related information, see "Getting and Setting Properties and Elements" (page 53).

Manipulation of Apple Events

You can use the following classes to directly manipulate Apple events and the data structures they contain. However, you can make your application scriptable with little or no direct use of these classes.

Table 9-5 Classes for manipulating Apple events

Class	Description
`NSAppleEvent-Descriptor`	Represents a descriptor, the basic building block for Apple events. Descriptors can consist of arbitrarily nested lists of other descriptors. Every Apple event is itself a descriptor and is made up of descriptors. For information on the underlying structure of descriptors, see Building an Apple Event in *Apple Events Programming Guide*.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Class	Description
`NSAppleEventManager`	Provides access to a small set of Apple Event Manager features. Used primarily for directly registering Apple event handlers and for suspending and resuming Apple events (described in "Suspending and Resuming Apple Events and Script Commands" (page 109)). For background information, see *Apple Event Manager Reference* and *Apple Events Programming Guide*.
`NSAppleScript`	Provides the ability to load, compile, and execute scripts.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

How Cocoa Applications Handle Apple Events

Whether your application is scriptable or not, Cocoa provides automatic handling for certain Apple events that all applications receive. This chapter describes Cocoa's default support for handling those Apple events and what your application must do to support or modify the default behavior.

Apple Event Handling Overview

All Mac OS X applications that present a graphical user interface should be able to respond to certain Apple events that are sent by the Mac OS. These events, sometimes called the **required events**, include those the application can receive at launch (the `open application`, `open documents`, `print documents`, and `open contents` events), as well as others it receives when already running (again including `open documents` and `print documents`, as well as the `reopen` and `quit` events). These events can also be sent by other applications and by users executing scripts.

If an application is scriptable, it can receive additional events that target the scriptable features it supports.

Basics of Apple Event Handling

Applications work with the Apple Event Manager to receive and extract information from Apple events. The processing of Apple events typically follows this pattern:

1. For the Apple events an application expects to receive, it registers callback routines, called Apple event handlers, with the Apple Event Manager.
2. When the application receives an Apple event, it uses the Apple Event Manager to dispatch the event to the appropriate event handler and to identify the object or objects in the application on which to perform the specified operation. (The details vary depending on the programming environment in use.)

You can read more about this process in *Apple Events Programming Guide* and *Apple Event Manager Reference*. The implementation of code to handle Apple events can be somewhat complex. However, Cocoa provides a lot of built-in support for handling Apple events, minimizing the need for your application to work directly with Apple event data structures or the Apple Event Manager.

Handling Apple Events in a Cocoa Application

For every Cocoa application, the Application Kit automatically installs event handlers for Apple events it knows how to handle, including those sent by the Mac OS. These event handlers implement a default behavior, described in "Apple Events Sent by the Mac OS" (page 102), that in some cases depends on code implemented by your application.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Nonscriptable Applications

A nonscriptable application can support or modify the default behavior provided by the Application Kit's Apple event handlers in these ways:

- It can implement or override the appropriate methods invoked by the Cocoa handlers, as described in "Apple Events Sent by the Mac OS" (page 102). This is the standard mechanism.
- It can install handlers to supersede the ones installed by Cocoa. To replace a specific event handler, read the information about that event in "Apple Events Sent by the Mac OS" (page 102), then read how to install a new handler in "Installing an Apple Event Handler" (page 107).

A nonscriptable application can also install handlers for other Apple events. But if your nonscriptable application needs to install many handlers, you should consider making it scriptable.

Scriptable Applications

A scriptable application can support or modify the default behavior provided by the Application Kit's Apple event handlers in the same ways a nonscriptable application can (by implementing or overriding the appropriate methods, or by installing a replacement event handler).

For other Apple events, a scriptable application doesn't typically install handlers directly (although it is free to do so) because it can use the script command mechanism. That mechanism, which automatically installs handlers based on information in the application's sdef file, is summarized in "Snapshot of Cocoa Scripting" (page 24) and described in more detail in "Script Commands" (page 71).

Important: Your scriptable application should not attempt to provide script command classes to override the Apple event handlers installed by the Application Kit. Because your scriptability information is not loaded until your application receives an Apple event for which no command has been registered, the handlers installed by the Application Kit can be invoked before your handlers are installed.

Apple Events Sent by the Mac OS

The following sections describe Apple events your application is likely to receive from the Mac OS, the response that is expected, and the default behavior supplied by the Application Kit. Your application can implement, or in some cases override, the appropriate methods to support or modify the default behavior. (These events can also be sent by scripts or by other applications.)

Open Application

The `open application` (or `launch`) Apple event is received when the application is launched. The application should perform any tasks required when a user launches it without opening or printing any documents.

Note: The application will instead receive an `open documents` event if it is launched in response to a user double-clicking a document icon.

Here is how the `open application` event is handled in Mac OS X version 10.4:

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

1. The application is launched.
2. The application determines if a new, untitled document should be created by invoking the application delegate's `applicationShouldOpenUntitledFile:` method. If the application delegate does not implement this method, a new untitled document is created. Implement this method if you want to control that behavior (whether or not your application is `NSDocument`-based.)
3. If the application delegate responds to `applicationOpenUntitledFile:`, that message is sent.

   Otherwise:

   - If the application is `NSDocument`-based, the shared `NSDocumentController` is sent this message:

     ```
     openUntitledDocumentAndDisplay:error:
     ```

     You can modify the default behavior by overriding this method. Or you can override the methods this method invokes (described in the Mac OS X version 10.4 documentation for `NSDocumentController`).
   - It the application is not document-based, for a document to be created, your application delegate must implement this method:

     ```
     applicationOpenUntitledFile:
     ```

Starting in Mac OS X version 10.4, an `open application` Apple event may contain information about whether the application was launched as a login item or as a service item. If so, the application typically should only perform actions suitable to the environment in which it is launched. For example, an application launched as a service item may not wish to open an untitled document.

For more information on how your application can check for the presence of launch information, see "Launch Apple Event Constants" in "Apple Event Manager Constants" in *Apple Event Manager Reference*.

Reopen

The `reopen` (or `reopen application`) Apple event is received when the application is reopened—for example, when the application is open but not frontmost, and the user clicks its icon in the Dock. The application should perform appropriate tasks—for example, it might create a new document if none is open.

Here is how the `reopen application` event is handled:

1. The application delegate is sent this message (if it implements the method):

   ```
   applicationShouldHandleReopen:hasVisibleWindows:
   ```
2. If the method returns `YES`, and if there are no open windows, it attempts to create a new untitled document in the same way as for the `open application` event.

You can modify the default behavior by implementing the method of the application delegate.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Open

The `open` (or `open documents`) Apple event is received when the application should open a list of one or more documents. For example, the user may have selected document files in the Finder and double-clicked.

Here is how the `open documents` event is handled in Mac OS X version 10.4 (v10.4):

- If the application is `NSDocument`-based, the `NSDocumentController` is sent this message, once for each document to be opened:

  ```
  openDocumentWithContentsOfURL:display:error:
  ```

  For your application to customize opening of documents in response to the `open documents` Apple event, it can override this method. Or it can override the methods this method invokes. These methods are described in the Mac OS X v10.4 documentation for `NSDocumentController`, which also describes how the default implementation handles compatibility with previous versions of the Mac OS.

- If the application is not `NSDocument`-based:

  1. The application delegate is sent this message, if it responds to it:

     ```
     application:openFiles:
     ```

  2. If the delegate does not respond to that message, the handler checks, in the order listed, whether the application delegate responds to one of the following messages, and if so, sends it:

     ```
     openTempFile:
     openFiles:
     openFile:
     ```

     For your application to open documents in response to this Apple event, it must implement one of these methods.

Starting in Mac OS X version 10.4, the `open documents` Apple event may contain an optional parameter containing search text from a Spotlight search that specified the documents to be opened. This parameter is identified by the keyword `keyAESearchText`. The application should make a reasonable effort to indicate occurrences of the search text in each opened document—for example by selecting the text and scrolling the first or primary occurrence into view. The Application Kit does not currently handle this parameter, but you can provide your own implementation by adding code like that shown in Listing 10-1 (page 104) to your method that opens files, such as `application:openFiles:`.

Listing 10-1 Extracting the search text parameter from the current Apple event

```
NSString    *searchString = nil;
searchString = [[[[NSAppleEventManager sharedAppleEventManager]         // 1
    currentAppleEvent]                                                  // 2
    paramDescriptorForKeyword:keyAESearchText]                          // 3
    stringValue];                                                       // 4
// Application-specific code to highlight the searched-for text, if any.  // 5
```

Here's what this code snippet does:

1. It invokes the `sharedAppleEventManager` class method of the `NSAppleEventManager` class to get an instance of the shared manager.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

2. It invokes the `currentAppleEvent` method of the `NSAppleEventManager` object to get the Apple event that is being handled on the current thread, if any, as an `NSAppleEventDescriptor`.

3. It invokes the `paramDescriptorForKeyword:` method of that class, passing the key `keyAESearchText` (and obtaining another instance of `NSAppleEventDescriptor`).

4. It invokes the `stringValue` method of that class to get the search text, if any, coercing it to Unicode text.

5. It selects instances of the specified text, if any, in the document, as appropriate for the application (not shown). For example, it might scroll the primary instance into view.

Note: For information on the document scriptability features provided by Cocoa, see "Use the Document Architecture" (page 35).

Print

The `print` (or `print documents`) Apple event is received when the application should print a list of one or more documents.

Here is how the `print` event is handled in Mac OS X version 10.4:

- If the application is `NSDocument`-based, each document to be printed is sent this message:

```
printDocumentWithSettings:showPrintPanel:delegate:didPrintSelector:contextInfo:
```

  Documents are opened automatically before printing, if necessary. If a document is opened just for printing, it is closed when printing is complete.

  You can modify the default behavior by overriding the methods this method invokes (described in the Mac OS X v10.4 documentation for `NSDocumentController`).

- If the application is not `NSDocument`-based, the handler checks, in the order listed, whether the application delegate responds to one of the following messages, and if so, sends it:

```
printFiles:withSettings:showPrintPanels:
printFiles:
printFile:
```

  For your application to print documents in response to this Apple event, it must implement one of these methods.

Open Contents

The `open contents` Apple event is available starting in Mac OS version 10.4. This Apple event is sent when content, such as text or an image, is dropped on the application icon—for example, when a dragged image is dropped on the icon in the Dock. The application should use the content in an appropriate way—for example, if a document is open, it might insert the content at the current insertion point; if no document is open, it might open a new document and insert the provided text or image.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

If your application provides a service that can accept the type of data in a received `open contents` Apple event, the default handler will use the service.

Here is how the `open contents` event is handled:

1. It invokes this method of the application delegate, to give the application a chance to set up services: `applicationDidFinishLaunching:`

   This is unique, in that for all the other Apple events described here, `applicationDidFinishLaunching:` is called *after* the described event handling behavior takes place.

2. It invokes Carbon's default `open contents` Apple event handler, which uses the application-provided service. (Carbon behavior is described in "Common Apple Events Sent by the Mac OS" in Responding to Apple Events in *Apple Events Programming Guide*.

You can modify the default behavior by installing your own `open contents` Apple event handler. The structure of the event is similar to the `open documents` event. The direct parameter consists of a list of content data items to be opened. The descriptor type for each item in the list indicates the type of the content (`'PICT'`, `'TIFF'`, `'utf8'`, and so on).

Quit

The `quit` (or `quit application`) Apple event is received when your application is terminated.

Here is how the `quit` event is handled:

- If the application is `NSDocument`-based, the behavior depends on the `saving` parameter, which has one of these three values:
  - `NSSaveOptionsNo`: The application quits without sending a `close` message to any document.
  - `NSSaveOptionsYes`: Each unmodified document is sent a `close` message; each modified document is sent the following message: `saveDocumentWithDelegate:didSaveSelector:contextInfo:`
  - `NSSaveOptionsAsk`: (This is the default value if no `saving` parameter is supplied in the event.) If there are modified documents open, the `NSDocumentController` sends itself this message:

`reviewUnsavedDocumentsWithAlertTitle:cancellable:delegate:didReviewAllSelector:contextInfo:`

- If the application is not `NSDocument`-based, the application delegate is sent this message (if it is implemented):

  `applicationShouldTerminate:`

  You can modify the default behavior by implementing this method.

Constants for Apple Event Handlers Installed by the Application Kit

Table 10-1 shows the constants for the Apple event handlers installed by the Application Kit to handle events sent by the Mac OS. Each of these events has the same event class, `kCoreEventClass`, which has the value `'aevt'`. Your application can use these constants for the event class and event ID when installing replacement handlers to modify standard behavior.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

These Apple event constants are defined in `AppleEvents.h`, a header in `AE.framework`, a subframework of `ApplicationServices.framework`. These and other constants are described in *Apple Event Manager Reference*.

Table 10-1 Event class IDs for Apple events sent by the Mac OS

Apple event	Event ID	Value
`open application` (or `launch`)	`kAEOpenApplication`	`'oapp'`
`reopen`	`kAEReopenApplication`	`'rapp'`
`open` (or `open documents`)	`kAEOpenDocuments`	`'odoc'`
`print` (or `print documents`)	`kAEPrintDocuments`	`'pdoc'`
`open contents`	`kAEOpenContents`	`'ocon'`
`quit` (or `quit application`)	`kAEQuitApplication`	`'quit'`

Installing an Apple Event Handler

Your application, whether scriptable or not, can install Apple event handlers directly. This generally makes sense in the following cases:

- You want to customize the default handling of one of the events for which the Application Kit installs a handler, but you can't do so in the standard way, by implementing or overriding the methods described in "Apple Events Sent by the Mac OS" (page 102).

  For example, you might want to install a handler for the `open contents` Apple event to override its default behavior.

- You have not made your application scriptable, but you need to handle certain Apple events not supported automatically by the Application Kit.

  If you find yourself handling more than a few Apple events this way, consider making your application scriptable to take advantage of Cocoa scripting's built-in support for handling Apple events.

To install an Apple event handler, you invoke this method of the `NSAppleEventManager` class:

```
setEventHandler:andSelector:forEventClass:andEventID:
```

The signature for the new handler should match the one shown in Listing 10-2.

Listing 10-2 Signature of an event handler function

```
- (void)handleAppleEvent:(NSAppleEventDescriptor *)event
withReplyEvent:(NSAppleEventDescriptor *)replyEvent;
```

A good place to install event handlers is in the `applicationWillFinishLaunching:` method of the application delegate. At that point, the Application Kit has installed its default event handlers, so if you install a handler for one of the same events, it will replace the Application Kit version.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Installing a Get URL Handler

Listing 10-3 shows how you could install a handler for the `get URL` Apple event.

> **Note:** To work with URL events, your application will have to specify one or more keys for the `CFBundleURLTypes` dictionary in its information property list file. These keys are described in the section "CFBundleURLTypes" in Property List Key Reference in *Runtime Configuration Guidelines*.

Listing 10-3 Installing an Apple event handler in a Cocoa application

```
NSAppleEventManager *appleEventManager = [NSAppleEventManager                // 1
sharedAppleEventManager];
[appleEventManager setEventHandler:self                                      // 2
andSelector:@selector(handleGetURLEvent:withReplyEvent:)
forEventClass:kInternetEventClass andEventID:kAEGetURL];
```

Here's what the code in Listing 10-3 does:

1. It gets a reference to the shared Apple Event Manager object.
2. It invokes a method of that object to install the new handler, passing:
   - A reference to the delegate object, `self`, which will handle the event.
   - A selector for the new `get URL` handler (shown in Listing 10-4 (page 108)).
   - The event class constant for the Apple event (from the header `HIServices/InternetConfig.h` in the Application Services framework).
   - The event ID constant for the Apple event (from the header `HIServices/InternetConfig.h` in the Application Services framework).

   If an event handler is already installed for the specified event class and event ID, it is replaced.

Implementing the Get URL Handler

Listing 10-4 provides a template for the event handler. This code resides in the implementation of your application delegate. After this handler has been installed, it is invoked whenever the application receives a `get URL` Apple event. The Apple event is passed to the handler as an instance of `NSAppleEventDescriptor`.

Listing 10-4 Implementation of a get URL Apple event handler

```
- (void)handleGetURLEvent:(NSAppleEventDescriptor *)event
withReplyEvent:(NSAppleEventDescriptor *)replyEvent
{
    // Extract the URL from the Apple event and handle it here.
}
```

The implementation details are left to you, but they require using methods of the `NSAppleEventDescriptor` class to extract the URL from the direct parameter of the Apple event, then performing the required operation with it (typically displaying the referenced page in a window). For a similar example, see Listing 10-1 (page 104).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Suspending and Resuming Apple Events and Script Commands

Starting with Mac OS X version 10.3, the `NSAppleEventManager` class provides methods for suspending and resuming an individual Apple event. These methods are of use for applications that respond to Apple events without using the built-in Cocoa scripting support. For applications that do take advantage of the Cocoa scripting support, `NSScriptCommand` provides methods for suspending and resuming execution of a script command.

An application typically suspends an Apple event (or a script command) when it performs an asynchronous operation, so that the application won't receive any more Apple events from the same script until it completes handling of the current event (or script command). For example, suppose the application must display a sheet as part of obtaining information to return in a reply Apple event. If so, it can suspend Apple events (or a script command) before displaying the sheet, insert information into the reply Apple event after the user dismisses the sheet, then resume. This need to suspend and resume can occur with other asynchronous operations (including those that may be open-ended or take a long time to complete).

To suspend an Apple event, you use the `NSAppleEventManager` method `suspendCurrentAppleEvent`, which returns a suspension ID (`NSAppleEventManagerSuspensionID`) for an Apple event being handled on the current thread. You can pass this suspension ID to `appleEventForSuspensionID:` to get an Apple event descriptor for the suspended event, or to `replyAppleEventForSuspensionID:` to get a descriptor for the corresponding reply Apple event. To resume a suspended Apple event, you pass the associated suspension ID to `resumeWithSuspensionID:`. In Mac OS X version 10.4, this method can be invoked in any thread.

To suspend a script command, use the `NSScriptCommand` method `suspendExecution`. This method suspends execution of a script command if the receiver is being executed in the current thread by the built-in Cocoa scripting support (that is, the receiver would be returned by `[NSScriptCommand currentCommand]`). You use `resumeExecutionWithResult:` to resume a suspended script command. In Mac OS X version 10.4, this method can be invoked in any thread.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Evolution of Cocoa Scriptability Information

Mac OS X and Cocoa provide various tools and formats for working with scriptability information. This appendix describes the history of those tools and shows how to convert between various scriptability formats and versions.

Scriptability Terms

AppleScript is a mature technology that was first introduced in the early 1990's. When scripting support was added to the Cocoa application framework, a significant part of its ease of use depended on its adoption of key-value coding (KVC), a mechanism for accessing an object's properties indirectly.

When terms from KVC (some of which come from entity-relationship modeling) were used with Cocoa scripting, some terms overlapped or conflicted with terms already in use by AppleScript. For example, the term *property* has more than one distinct meaning. For definitions of these terms, see the "Glossary" (page 133).

Additionally, there are some differences in how you use the original format for Cocoa scriptability information (provided in script suite and script terminology files) and the current format (based on the sdef file format). These differences are described throughout this chapter.

Changes in Scriptability Information Versions

You provide scriptability information for your application in one of the two formats described in "Scriptability Information Formats" (page 16):

- The scripting definition (or sdef) format first became available in Mac OS X version 10.2. Cocoa scripting can interpret sdef files natively starting in Mac OS X version 10.4.

  For Mac OS X version 10.3, an sdef file can be converted to a corresponding script suite and script terminology file pair.

- The script suite format, consisting of a script suite file and a corresponding script terminology file, has been in use since scriptability support first became available in Cocoa and works in any version. This is the only format Cocoa scripting can interpret natively prior to Mac OS X version 10.4.

The traditional Carbon mechanism for supplying scriptability information is the `'aete'` resource file. Cocoa scripting doesn't use `'aete'` resources, but you can add one to your application to control how your scriptability information is displayed by Script Editor or other dictionary viewers. Starting in Mac OS version 10.4, however, the preferred mechanism for controlling how your scriptability information is displayed is to use an sdef file.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Advantages of the Scripting Definition Format

Important: You can read about additional refinements to sdef usage in Cocoa applications for Mac OS X v10.5 in the Scripting section of *Foundation Release Notes*.

There are a number of advantages to using the native sdef format in Mac OS X version 10.4, including:

- You can assemble information describing the terminology and implementation details for your application all in one file, so there is no need to synchronize separate files.
- You have more options for specifying scriptability information. For example, you can control the order in which information is displayed (without having to add an `'aete'` resource to your application). You can also use the `hidden` attribute to hide deprecated terms while still keeping them available for backward compatibility. Or you can hide terms that are not yet ready for release.
- With an sdef, only the parts of Cocoa's default scriptability information that you specifically include are used by Cocoa scripting and are visible when your dictionary is displayed.

  By comparison, applications that use only script suite and script terminology files will include the terminology from all such files defined in any framework the application links to or bundles that it loads (including the files for the full Standard and Text suites defined in the Cocoa framework).
- Script Editor can display your application's dictionary without launching the application. However, through Mac OS X version 10.4, it does still need to open your application to compile a script that targets the application.

Advantages of the Script Suite Format

The main advantage of the script suite format is that it can be used in any scriptable version of Cocoa. If your scriptable application will run in versions of the Mac OS prior to version 10.4, it must include script suite and script terminology files. However, it can also contain an sdef file, so that it can gain the advantages that sdef files provide when running in Mac OS X version 10.4.

Script suite and script terminology files do not allow detailed control of how your scriptability information is displayed in a dictionary viewer. But if you need finer control of the look of your dictionary, you can add an `'aete'` resource to your application bundle. You can generate the `'aete'` by creating an sdef file for your application, then using the `sdp` tool to create an `'aete'` resource. Or you can create an `'aete'` resource directly (some third party tools can aid in doing so).

Suite information is described in detail in "Script Suite and Script Terminology Files" (page 117).

Converting and Updating Scriptability Information

There are several options available for converting between scriptability formats and updating scriptability information. Remember that after making any changes to your scriptability information, you should run your full suite of test scripts to make sure your scriptability support is working as expected.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Creating Suite Files or 'aete' Files from a Scripting Definition

You can use the `sdp` tool to convert an sdef file into a pair of script suite and script terminology files that Cocoa can work with in current or earlier versions of the operating system. However, the script suite files may require some modification, particularly if you use `sdp` in Mac OS X version 10.4 to create script suite files you will use with earlier OS versions.

You can also use `sdp` to create an `'aete'` file from an sdef file. A Cocoa application that provides its scriptability information in the script suite format may want to also include an `'aete'` resource to provide more control over how its scriptability information is displayed. That comes in handy in the following situation:

1. A user tries to display your application's scripting dictionary with Script Editor (or another dictionary viewer). Script Editor sends your application an Apple event asking for its scriptability information.
2. If your application has an `'aete'` resource, Cocoa returns that. Only the information you choose to put in the `'aete'` resource is displayed.
3. If your application does not have an `'aete'` resource, Cocoa scripting creates one from the information in any script suites available to the application. The information that gets displayed may include terminology that you do not wish to expose.

However, you do not need an `'aete'` for this purpose in Mac OS X version 10.4 if your application uses an sdef file, because the sdef format gives you full control over what gets displayed.

You can execute a statement like the following in the Terminal application to create both script suite files and an `'aete'` resource from an sdef file:

```
sdp -fast -o ~myHome MyApplication.sdef
```

By specifying `"ast"` with the `-f` parameter, this command tells `sdp` to create `'aete'`, script suite, and script terminology files. It actually creates a pair of script suite files for each `suite` element defined in the sdef, placing them in the directory specified by the `-o` argument.

You can invoke `sdp` in a shell script build phase in Xcode if you want to make it part of your build process. If you do so, your command invocation should include the build style as part of the target directory. Here is an example of such a command:

```
/usr/bin/sdp -fast -o
"$BUILD_DIR/$BUILD_STYLE/$FULL_PRODUCT_NAME/Contents/Resources"
"$SOURCE_ROOT/MyApplication.sdef"
```

For more information, see the `sdp` man page and the Xcode documentation.

Creating Scripting Definitions from Suite Files or 'aete' Files

In Mac OS X version 10.4, if you have existing script suite and script terminology files or a resource file containing an `'aete'` resource, you can use the `desdp` tool to convert them to the sdef format. The resulting sdef will contain all the information in the original dictionary, but is likely to require some changes, because the sdef format is more expressive than the older formats. For example, in the older format you cannot specify the ordering of terms. For more information on the `desdp` tool, see its man page.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Updating Older Scripting Definition Files for Mac OS X Version 10.4

The scripting definition format experienced a number of changes for Mac OS X version 10.4. (For a complete listing, see the History section of the `sdef` man page.) If you have an existing sdef file created for an earlier version of the Mac OS, you can use the `xsltproc` tool to upgrade it for Mac OS X version 10.4. You use this command line tool to apply XSLT stylesheets (such as the Mac OS X file `/usr/share/sdef/upgrade.xsl` referred to in the example that follows) to XML documents (in this case, the sdef file to be upgraded).

You can use a line like the following in the Terminal application to upgrade an sdef file named `MyApplication.sdef`:

```
xsltproc --novalid -o MyNewApplication.sdef /usr/share/sdef/upgrade.xsl
MyApplication.sdef
```

In this invocation:

- The `--novalid` argument causes the tool to skip loading the document's DTD file.
- The `-o` argument specifies the name of the new file to create.
- The `/usr/share/sdef/upgrade.xsl` argument specifies the file from which to obtain the upgrade information.
- The final argument, `MyApplication.sdef`, specifies the existing sdef file to upgrade.

For more information, see the `xsltproc` man page.

Editing Scriptability Information

In Mac OS X version 10.4, both Xcode and Script Editor understand the sdef format. Double-clicking an sdef file in the Finder will launch Script Editor and display the scripting terminology in a dictionary viewer, while double-clicking an sdef file in an Xcode project will open it in a dictionary viewer in Xcode. Figure 1-4 (page 19) shows an sdef displayed in a dictionary viewer.

To edit an sdef file in Xcode, select the sdef file and choose File > Open As > Plain Text File. You can also view and edit the XML for an sdef file by opening it with any plain text editor or XML editor.

You can use File > Open Dictionary in Script Editor or Xcode to choose any scriptable application and display its dictionary.

In Mac OS X version 10.3 and earlier, Cocoa scripting cannot directly parse sdef files, and neither Xcode nor Script Editor can display native sdef files in a dictionary viewer.

For all scriptable versions of Cocoa, you can supply scriptability information in script suite files and script terminology files. Dragging a scriptable Cocoa application that contains these files onto Script Editor will display its dictionary. However, neither Script Editor nor Xcode can interpret script suite and script terminology files natively to display a dictionary.

Double-clicking a script suite or script terminology file will typically open it in Property List Editor (if you have not changed the Finder default). After opening a file in Property List Editor, you can save it in XML format, or in a plain ASCII format that may be somewhat easier to read and work with. You can edit suite files in any of these formats, though editing property lists with Property List Editor has some limitations.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

For information on creating a new sdef file, see "Create a Scripting Definition File" (page 43). For information on creating a new script suite, see "Creating Your Own Script Suite Files" (page 127). If you already have existing scriptability information, see "Converting and Updating Scriptability Information" (page 112).

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Script Suite and Script Terminology Files

Cocoa applications can provide scriptability information in the form of script suite and script terminology files. This format, sometimes referred to as the **script suite** format, has been supported since the first version of Cocoa scripting. This chapter provides a detailed look at the use and structure of these files.

Important: Starting in Mac OS X version 10.4, Cocoa applications can supply scripting information in the sdef format, described in "Preparing a Scripting Definition File" (page 39). This is the preferred format going forward.

However, if your scriptable application will run in versions of the Mac OS prior to version 10.4, it must include script suite and script terminology files. For information on working with multiple scriptability formats, see "Evolution of Cocoa Scriptability Information" (page 111).

When supplying scripting information in the script suite format, an application provides at least one nonlocalized script suite file and a corresponding script terminology file. These files together describe the scripting capabilities of the application and the terminology a scripter uses to access those capabilities.

Script Suite Files

A **script suite file** describes scriptable objects in terms of their attributes, relationships, and supported commands. You can think of a script suite file as supplying scripting information for use by Cocoa's internal scripting support; that information can also be used by your application. "Script Terminology Files" (page 123) describes the files used to store the scripting terminology that corresponds to the information in a script suite file. That terminology defines the terms actually used by scripters to control the application.

The information in a script suite file consists of a nested list of key-value pairs. Script suites are always property lists, but in addition they must conform to the format described in "The Structure of a Script Suite File" (page 119). For information on creating them, see "Creating Your Own Script Suite Files" (page 127).

Frameworks, loadable bundles, and applications that support scripting can include a script suite file as a language-independent resource. The name of the file takes the form *suiteName.scriptSuite*, where *suiteName* uniquely identifies the script suite file. An example would be `MyApplication.scriptSuite`.

Script suites are located in the (nonlocalized) `Resources` directory of an application, framework, or bundle. For example, for a framework named `MyStuff.framework`, the script suite file (named `MyStuff.scriptSuite`) would reside in `MyStuff.framework/Resources/`.

Listing B-1 shows the class description for the `NSApplication` class, taken from `NSCoreSuite.scriptSuite`, Cocoa's script suite file for the AppleScript Standard suite. The class description is shown as exported in "XML Property List File" format by the Property List Editor application. You can find the full version of this script suite file on your system by following the `Resources` symbolic link at `/System/Library/Frameworks/Foundation.framework`.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Listing B-1 NSApplication class from the script suite file for the Standard suite

```
<key>NSApplication</key>
<dict>
    <key>AppleEventCode</key>
    <string>capp</string>
    <key>Attributes</key>
    <dict>
        <key>isActive</key>
        <dict>
            <key>AppleEventCode</key>
            <string>pisf</string>
            <key>ReadOnly</key>
            <string>YES</string>
            <key>Type</key>
            <string>NSNumber&lt;Bool&gt;</string>
        </dict>
        <key>name</key>
        <dict>
            <key>AppleEventCode</key>
            <string>pnam</string>
            <key>ReadOnly</key>
            <string>YES</string>
            <key>Type</key>
            <string>NSString</string>
        </dict>
        <key>version</key>
        <dict>
            <key>AppleEventCode</key>
            <string>vers</string>
            <key>ReadOnly</key>
            <string>YES</string>
            <key>Type</key>
            <string>NSString</string>
        </dict>
    </dict>
    <key>Superclass</key>
    <string>NSCoreSuite.AbstractObject</string>
    <key>SupportedCommands</key>
    <dict>
        <key>NSCoreSuite.Open</key>
        <string>handleOpenScriptCommand:</string>
        <key>NSCoreSuite.Print</key>
        <string>handlePrintScriptCommand:</string>
        <key>NSCoreSuite.Quit</key>
        <string>handleQuitScriptCommand:</string>
    </dict>
    <key>ToManyRelationships</key>
    <dict>
        <key>orderedDocuments</key>
        <dict>
            <key>AppleEventCode</key>
            <string>docu</string>
            <key>LocationRequiredToCreate</key>
            <string>NO</string>
            <key>ReadOnly</key>
            <string>YES</string>
            <key>Type</key>
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
                <string>NSDocument</string>
            </dict>
            <key>orderedWindows</key>
            <dict>
                <key>AppleEventCode</key>
                <string>cwin</string>
                <key>ReadOnly</key>
                <string>YES</string>
                <key>Type</key>
                <string>NSWindow</string>
            </dict>
        </dict>
    </dict>
```

Each scriptable class in a script suite file must have a "class description" which, in key-value form, declares the attributes and relationships of the class in terms of type and four-character code (or Apple event code). For example, the four-character code for `NSApplication` in Listing B-1 (page 118) is the string `"capp"`. You can read more about four-character codes in "Code Constants Used in Scriptability Information" (page 40). The primary key for a class description must be a class name such as `NSApplication` that identifies a real Objective-C class.

The class description also declares the AppleScript commands the class supports and specifies the superclass, if the superclass also supports scripting of its objects. In this case, the superclass for `NSApplication` is `AbstractObject`, which is the root of the scriptable class hierarchy, and corresponds to the `NSObject` class.

The Structure of a Script Suite File

A script suite file is a text file containing key-value pairs in the form of a series of nested dictionaries, with two main categories: class descriptions and command descriptions.

A **class description** describes the attributes and relationships of a scriptable class. Relationships can be one-to-one or one-to-many. A class description also lists the commands a class supports and specifies whether a particular method of the class handles the command or the command's default implementation is used to execute the command. A description of a class can designate a scriptable superclass, and thus inherit the attributes, relationships, and supported commands of that class. Class descriptions for the classes defined in an application's script suite file are instantiated by the global instance of `NSScriptSuiteRegistry` when it loads the application's scriptability information.

A **command description** defines the characteristics of an AppleScript command that the application, framework, or bundle specifically supports. This information includes the class of the command, the type of the return value, and the number and types of arguments. Many of the commands defined in the Standard suite (such as `copy`, `duplicate`, `move`, and so on) have default implementations in subclasses of `NSScriptCommand`, listed in "Subclasses for Standard AppleScript Commands" (page 98). Command descriptions for the commands defined in an application's script suite file are instantiated by the global instance of `NSScriptSuiteRegistry` when it loads the application's scriptability information. For more information on the script command mechanism, see "Script Commands" (page 71).

A script suite file may contain additional declarations, such as enumerations. For example, `NSCoreSuite.scriptSuite` contains a declaration for the `SaveOptions` enumeration, which defines the AppleScript values for `yes`, `no`, and `ask` that are used when closing a file.

The following tables describe the structure of a script suite file, including its optional and required keys.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Table B-1 Suite dictionary

Key	Value type or reference	Description
"*name*"	`NSString`	Name of suite (required); the name can be placed anywhere in the definition, as long as it is a first-level element
"`AppleEventCode`"	`NSString`	Four-character code for this suite (required)
"`Classes`"	Class list dictionary (Table B-2)	optional (no classes defined by default)
"`Commands`"	Command list dictionary (Table B-7 (page 121))	optional (no classes defined by default)
"`Synonyms`"	Synonym list dictionary (Table B-10 (page 122))	optional (no synonyms defined by default)
"`Enumerations`"	Enumeration list dictionary (Table B-11 (page 122))	optional (no enumerations defined by default)

Table B-2 Class list dictionary

Key	Reference	Description
"*className*"	Class dictionary (Table B-3)	One per each scriptable class. Must be the name of an Objective-C class defined by Cocoa or the application.

Table B-3 Class dictionary

Key	Value type or reference	Description
"`Superclass`"	`NSString`	Scriptable superclass; must be the name of an Objective-C class. All attributes, relationships, and supported commands are inherited and can be overridden. You can use the notation *suiteName.className* to designate the class. (Optional.) You can also use the `AbstractObject` class to specify a base class your scriptable classes can inherit from that contains no scriptability of its own.
"`AppleEventCode`"	`NSString`	Four-character code for this class (required)
"`Attributes`"	Property list dictionary (Table B-4)	Attributes of the class (optional)
"`ToOneRelationships`"	Property list dictionary (Table B-4)	One-to-one relationships of the class (optional)
"`ToManyRelationships`"	Property list dictionary (Table B-4)	One-to-many relationships of the class (optional)

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Key	Value type or reference	Description
"SupportedCommands"	Supported commands dictionary (Table B-6 (page 121))	Commands supported by the class (optional)

Table B-4 Property list dictionary

Key	Reference	Description
"*propertyName*"	Property dictionary (Table B-5)	Definition of an attribute or relationship. *attributeName* should map to an instance variable of the class for which there are accessor methods.

Table B-5 Property dictionary

Key	Value type	Description
"Type"	NSString	Name of class of values of this property (required)
"AppleEventCode"	NSString	Four-character code for this suite (required)
"ReadOnly"	NSString	"Yes" or "No" (optional; "No" by default)

Table B-6 Supported commands dictionary

Key	Value type	Description
"*commandName*"	NSString	Name of method this class uses to implement the command or "" if the default implementation is sufficient. *commandName* should be in *suiteName.commandName* notation if command is not in the same suite as the class.

Table B-7 Command list dictionary

Key	Reference	Description
"*commandName*"	Command dictionary (Table B-12)	Command definition.

Table B-8 Command dictionary

Key	Value type or reference	Description
"CommandClass"	NSString	Class of command. Set this value to NSScriptCommand for default behavior. If not using default, must be set to a subclass of NSScriptCommand (required).
"AppleEventCode"	NSString	Four-character code for this command (required)

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Key	Value type or reference	Description
“AppleEventClassCode”	NSString	Four-character class constant for this command (optional; by default, is the constant for the script suite)
“Type”	NSString	Class name of result of command or “” if no result (optional; no result by default)
“ResultAppleEventCode”	NSString	Four-character code for the return type of the command. Must be present if “Type” value is assigned. Can be “****” if the return type is variable.
“Arguments”	Argument list dictionary (Table B-13)	Arguments of command (optional; no arguments by default)

Table B-9 Argument list dictionary

Key	Value type or reference	Description
“Type”	NSString	Name of class for this argument (required)
“AppleEventCode”	NSString	Four-character code for this argument (required)
“Optional”	NSString	“Yes” or “No” (optional; “No” by default)

Table B-10 Synonym list dictionary

Key	Value type or reference	Description
“Apple event code”	NSString	Class name for which four-character code is a synonym.

Table B-11 Enumeration list dictionary

Key	Reference	Description
“*enumerationName*”	Enumeration dictionary (Table B-12)	One per enumeration.

Table B-12 Enumeration dictionary

Key	Value type or reference	Description
“AppleEventCode”	NSString	Four-character code for this enumeration (required)
“Enumerators”	Enumerators list dictionary (Table B-13)	Enumerators in the enumeration (required)

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Table B-13 Enumerators dictionary

Key	Value type or reference	Description
"enumeratorName"	`NSString`	Four-character Apple event code for this enumerator (required)

Script Terminology Files

A **script terminology file** maps AppleScript terminology—the English-like words and phrases a scripter can use in a script, such as `the first word in the first paragraph`—to the class and command descriptions in a script suite file. A script terminology file also provides valuable documentation about an application's scripting support, which users can examine in the Script Editor and Xcode applications.

Note: If an application also includes an `'aete'` resource, Script Editor displays the contents of that resource instead of the script terminology. (In Mac OS X version 10.4 and later, if an application uses an sdef file, any `'aete'` file is ignored.)

Like a script suite file, a script terminology file is stored as a nested list of key-value pairs. Script terminologies are always property lists, but in addition they must conform to the format described in "The Structure of a Script Terminology File" (page 124). See "Creating Your Own Script Suite Files" (page 127) for information on how to create and edit these files.

Note: English is currently the only supported dialect in AppleScript. It is not recommended that you localize your terminology for other languages.

Listing B-2 (page 123) shows the terminology for the `NSApplication` class, taken from `NSCoreSuite.scriptTerminology`, the script terminology file for the Standard suite (provided by Cocoa). The terminology is shown as exported in "ASCII Property List File" format by the Property List Editor application. You can find the full version of this file on your system by following the `Resources` symbolic link at `/System/Library/Frameworks/Foundation.framework`.

Listing B-2 NSApplication class from the script terminology file for the Standard suite

```
    NSApplication = {
        Attributes = {
            isActive = {
                Description = "Is this the frontmost (active) application?";
                Name = frontmost;
            };
            name = {Description = "The name of the application."; Name = name; };
            version = {Description = "The version of the application."; Name =
version; };
        };
        Description = "An application's top level scripting object.";
        Name = application;
        PluralName = applications;
    };
    NSColor = {Description = "A color."; Name = color; PluralName = colors; };
```


2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

```
        NSDocument = {
            Attributes = {
                fileName = {Description = "The document's path."; Name = path; };
                isDocumentEdited = {
                   Description = "Has the document been modified since the last save?";
                    Name = modified;
                };
                lastComponentOfFileName = {Description = "The document's name."; Name
= name; };
            };
            Description = "A document.";
            Name = document;
            PluralName = documents;
        };
```

The Structure of a Script Terminology File

Script terminologies, like script suite files, are stored as text files of key-value pairs. As with a script suite file, a script terminology file consists of a series of nested dictionaries. Many of the subdictionaries (class, command, argument, and so on) should have counterparts in the script suite file.

The following tables describe the optional and required keys for a script terminology file.

Table B-14 Terminology dictionary

Key	Value type or reference	Description
"Name"	NSString or NSArray of NSString objects	Human-readable name of suite (required); the name can be placed anywhere in the definition, as long as it is a first-level element
"Description"	NSString	Human-readable description of suite (optional; but highly recommended)
"Classes"	Class list terminology dictionary (Table B-15)	Required only if there is a corresponding definition in the script suite file
"Commands"	Command list terminology dictionary (Table B-19 (page 125))	Required only if there is a corresponding definition in the script suite file
"Synonyms"	Class synonym list terminology dictionary (Table B-23 (page 126))	Required only if there is a corresponding definition in the script suite file
"Enumerations"	Enumeration terminology dictionary (Table B-25 (page 127))	Required only if there is a corresponding definition in the script suite file

Table B-15 Class list terminology dictionary

Key	Reference	Description
"*className*"	Class terminology dictionary (Table B-16)	One per each scriptable class. Must be the name of an Objective-C class.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Table B-16 Class terminology dictionary

Key	Value type or reference	Description
"Name"	NSString or NSArray of NSString objects	Human-readable name of class (required)
"Description"	NSString	Human-readable description of class (optional; but highly recommended)
"PluralName"	NSString	Human-readable name for plural form of class (required)
"Attributes"	Attribute list terminology dictionary (Table B-17)	Attributes of the class (required only if there is a corresponding definition in the script suite file)

Table B-17 Attribute list terminology dictionary

Key	Reference	Description
"*attributeName*"	Attribute terminology dictionary (Table B-18)	Description of attribute of the class

Table B-18 Attribute terminology dictionary

Key	Value type or reference	Description
"Name"	NSString or NSArray of NSString objects	Human-readable name of attribute (required).
"Description"	NSString	Human readable description of attribute (optional, but highly recommended)
"Number"	NSString or NSArray of NSString objects	"plural" or "singular" (default).

Table B-19 Command list terminology dictionary

Key	Reference	Description
"*commandName*"	Command terminology dictionary (Table B-20)	One per each supported script command.

Table B-20 Command terminology dictionary

Key	Value type or reference	Description
"Name"	NSString or NSArray of NSString objects	Human-readable name of command (required)
"Description"	NSString	Human-readable description of command (optional; but highly recommended)

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Key	Value type or reference	Description
"Arguments"	Argument list terminology dictionary (Table B-21)	Description of command arguments (required only if there is a definition)

Table B-21 Argument list terminology dictionary

Key	Reference	Description
"*argumentName*"	Argument terminology dictionary (Table B-22)	Descriptions of command arguments.

Table B-22 Argument terminology dictionary

Key	Value type or reference	Description
"Name"	NSString or NSArray of NSString objects	Human-readable name of argument (required)
"Description"	NSString	Human-readable description of argument (optional, but highly recommended)
"Sex"	NSString or NSArray of NSString objects	"masculine", "feminine", "none" (default)
"Number"	NSString or NSArray of NSString objects	"plural" or "singular" (default)

Table B-23 Class synonym list terminology dictionary

Key	Reference	Description
"Apple event code"	Class synonym terminology dictionary (Table B-24)	Descriptions of constant synonyms for class

Table B-24 Class synonym terminology dictionary

Key	Value type or reference	Description
"Name"	NSString or NSArray of NSString objects	Human-readable name of class (required)
"Description"	NSString	Human-readable description of class (optional)
"PluralName"	NSString	Human-readable name of plural form of class (required)

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Table B-25 Enumeration list terminology dictionary

Key	Reference	Description
"enumerationName"	Enumerators list terminology dictionary (Table B-26)	One per enumeration (required).

Table B-26 Enumerators list terminology dictionary

Key	Value type or reference	Description
"enumeratorName"	Enumerator terminology dictionary (Table B-27)	One per enumerator (required).

Table B-27 Enumerator terminology dictionary

Key	Value type or reference	Description
`"Description"`	`NSString`	Human-readable description of enumerator (optional)
`"Name"`	`NSString`	Human-readable name of enumerator (required)

Cocoa Scripting's Built-in Script Suites

Cocoa scripting provides two suites of standard scripting information: the Standard (or Core) suite and the Text suite. These suites define scriptability information for basic AppleScript commands, including `copy`, `count`, `create`, `delete`, `exists`, and `move`. They also provide information for basic AppleScript classes, such as `application`, `document`, `window`, and `text`, and for corresponding Cocoa classes, such as `NSApplication`, `NSDocument`, `NSWindow`, and `NSTextStorage`.

The Standard and Text suites are introduced in "Built-in Support for Standard and Text Suites" (page 19) and described in more detail in "Use the Document Architecture" (page 35) and "Access the Text Suite" (page 36).

> **Note:** This chapter describes how to work with Cocoa scriptability information in the script suite and script terminology format. If your application supplies its scriptability information through an sdef, you declare scriptability information for the Standard and Text suites as described in "Create a Scripting Definition File" (page 43).

Creating Your Own Script Suite Files

To add to Cocoa's built-in scripting support using the script suite and script terminology approach, your application supplies files describing the scriptability information for the objects, properties, and commands it supports. For example, an application that can draw shapes (such as the Sketch application) might specify that it supports `circle`, `rectangle`, and `line` objects, with `color` and `location` properties. It might also support commands such as `rotate` and `scale`, in addition to standard commands such as `get`, `set`, and `delete`.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

To create a script suite file or script terminology file, you can either use a plain text or XML editor, or use an application such as Property List Editor, which provides built-in support for creating property lists. Property List Editor is included with the Mac OS X developer tools. If you have existing scriptability information, see "Converting and Updating Scriptability Information" (page 112) for information on how to convert between various formats.

You construct files that contain entries for the classes, commands, and enumerations used by your application, including the codes, method, and class information used by Cocoa and the terminology used by scripters. Your files must follow the formats described in "The Structure of a Script Suite File" (page 119) and "The Structure of a Script Terminology File" (page 124). If you use the Property List Editor application, you can ensure that you are creating a valid property list. You can also save your scriptability information in several formats:

- As a script suite file or script terminology file: these files are stored in XML format.
- As a text file with XML tags: for an example, see Listing B-1 (page 118).
- As a text file in plain ASCII format: for an example, see Listing B-2 (page 123).

These formats can be opened by various text editors or by Property List Editor. As a result, you can freely work with a script suite file in whichever format is most convenient for you.

Figure B-1 shows the Standard suite (from the file `NSCoreSuite.scriptSuite`) as displayed in the Property List Editor application. In this figure, the Classes and the `NSApplication` class definition are expanded so that attributes, supported commands, and relationships for `NSApplication` are visible.

The `AbstractObject` class specifies a base class that your scriptable classes can inherit from when their actual superclass has no scriptability support.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Figure B-1 Script suite for the Standard suite in Property List Editor

NSCoreSuite.scriptSuite

New Sibling | Delete | Dump

Property List	Class	Value
▼Root	Dictionary	7 key/value pairs
AppleEventCode	String	????
▼Classes	Dictionary	5 key/value pairs
▶AbstractObject	Dictionary	3 key/value pairs
▼NSApplication	Dictionary	5 key/value pairs
AppleEventCode	String	capp
▼Attributes	Dictionary	3 key/value pairs
▶isActive	Dictionary	3 key/value pairs
▶name	Dictionary	3 key/value pairs
▶version	Dictionary	3 key/value pairs
Superclass	String	NSCoreSuite.AbstractObject
▼SupportedCommands	Dictionary	3 key/value pairs
NSCoreSuite.Open	String	handleOpenScriptCommand:
NSCoreSuite.Print	String	handlePrintScriptCommand:
NSCoreSuite.Quit	String	handleQuitScriptCommand:
▼ToManyRelationships	Dictionary	2 key/value pairs
▶orderedDocuments	Dictionary	4 key/value pairs
▶orderedWindows	Dictionary	3 key/value pairs
▶NSColor	Dictionary	2 key/value pairs
▶NSDocument	Dictionary	4 key/value pairs
▶NSWindow	Dictionary	5 key/value pairs
▶Commands	Dictionary	13 key/value pairs
▶Enumerations	Dictionary	1 key/value pair
Name	String	NSCoreSuite

For examples of script suites and sample code for scriptable applications, see the Sketch and TextEdit example projects, .

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Document Revision History

This table describes the changes to *Cocoa Scripting Guide*.

Date	Notes
2008-03-11	Minor corrections for working with sdefs.
	In the examples in "Class Elements" (page 44), now using the four-character code "ricT" for the rich text class.
2007-10-31	Added pointers to new information in Mac OS X version 10.5.
	Noted in several places that you can read about changes in Cocoa scripting support for Mac OS X v10.5, including changes in sdef usage, in the Scripting section of *Foundation Release Notes*.
	Converted "Evolution of Cocoa Scriptability Information" (page 111) to an Appendix.
	Converted "Script Suite and Script Terminology Files" (page 117) to an Appendix and noted that the information it contains is useful primarily if your scriptable application will run in versions of the Mac OS prior to version 10.4. (Otherwise, you can use the sdef format, described in "Preparing a Scripting Definition File" (page 39).)
2007-07-23	Added information on handling the open application Apple event.
	For details, see "Open Application" (page 102) section.
	In the section "Maintain KVC Compliance" (page 54), added implementation details for your overrides of valueIn<Key>WithName: and valueIn<Key>WithUniqueID:.
2006-04-04	Added information about NSObject and the item AppleScript class. Added reference and related document links to the HTML table of contents.
	Updated scriptability information in Table 3-1 (page 36).
	Modified test script in Listing 8-1 (page 88).
2006-03-08	Combines and updates information formerly in two documents, "Sdef Scriptability Guide for Cocoa" and "Scriptable Application Programming Guide for Cocoa."
	Revised "Overview of Cocoa Support for Scriptable Applications" (page 13) to provide a more complete introduction to Cocoa scripting. Changes include a description of the AppleScript object model, a table of the basic data types supported by Cocoa scripting, and expanded information on scriptability information formats.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Date	Notes
	Provided "Implementing a Scriptable Application" (page 31), a revision to the former "Key Steps" chapter. It includes a revised set of implementation steps (in the section "Implementation Guidelines" (page 31)).
	Provided "Preparing a Scripting Definition File" (page 39), which revises and merges two former chapters, "Creating a Scripting Definition" and "Scripting Definition Reference."
	Added the largely new chapters "Getting and Setting Properties and Elements" (page 53) and "Object Specifiers" (page 61). The former provides many details on working with KVC (including revised accessor samples), as well as a section on the `properties` property. The latter includes new illustrations and code samples, and includes conceptual information previously located in `NSScriptObjectSpecifier` and `NSScriptObjectSpecifiers`.
	Revised and expanded the chapter "Script Commands" (page 71), adding an illustration, sample code, and a comprehensive table that describes default support for AppleScript commands and how to customize it (Table 7-1 (page 85)).
	Revised the chapter "Testing, Debugging, and Performance" (page 87), adding new testing tips and a section on performance.
	Revised the chapters "Cocoa Scripting Classes and Categories" (page 95), "How Cocoa Applications Handle Apple Events" (page 101), and "Evolution of Cocoa Scriptability Information" (page 111) (formerly a section in "Scripting Definition File Reference").
	Merged script suite information into the chapter "Script Suite and Script Terminology Files" (page 117). Added tables for enumerations and deleted deprecated information (such as "Working With ASCII Script Suite Files").

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

Glossary

'aete' resource A resource that serves as the traditional mechanism for providing scriptability information in a Carbon application. An 'aete' resource can also be included in a Cocoa application to control how scriptability information is displayed in a dictionary viewer, by applications such as Script Editor and Xcode. Starting in Mac OS X version 10.4, it is not needed for this purpose, for applications that supply their scriptability information in the sdef format.

Apple event An interprocess message that can specify complex operations and data. An Apple event encapsulates a high-level task in a single package that can be passed across process boundaries, performed, and responded to with a reply event.

Apple Event Manager The Mac OS X API for creating and sending Apple events, and for receiving, extracting information from, and responding to them.

Apple event translator A part of Cocoa scripting that uses scriptability information supplied by an application to evaluate an Apple event received by the application. In many cases, an Apple event is "translated" into a script command object that performs the action specified by the event.

AppleScript A scripting language that makes possible direct control of scriptable applications and scriptable parts of the Mac OS. See also Open Scripting Architecture (OSA).

AppleScript object model A hierarchical structure that, for a given application, specifies the classes of objects a scripter can work with in scripts, the accessible properties of those objects, and the inheritance and containment relationships for those objects.

attribute (1) In key-value coding (and in suite files), refers to a property that is a simple value, such as a scalar, string, or Boolean value, or to immutable objects such as NSColor and NSNumber objects. In a scripting definition file, the equivalent of an attribute is a property. A graphic object might have a color attribute (or property). (2) In AppleScript, one of the two main descriptor data types that make up an Apple event. Not commonly used by scriptable Cocoa applications. (3) In an XML file, a name/value pair that specifies a single property for an element.

class description A scripting definition file (sdef) entry that describes a scriptable class, including its attributes and relationships and the KVC keys that Cocoa scripting uses to gain access to its values. When the sdef is loaded, the information is stored in an instance of NSScriptClassDescription.

Cocoa scripting In the Cocoa application framework, the support for creating scriptable applications. Cocoa scripting includes classes, categories, and scriptability information, which together support much of AppleScript's Standard suite.

command description A scripting definition file (sdef) entry that describes the characteristics of an AppleScript command, including argument names (if any), command result type (if any), AppleScript command name, and name of the Objective-C class Cocoa instantiates to perform the command. When the sdef is loaded, the information is stored in an instance of NSScriptCommandDescription.

element (1) In a scripting definition file (or an AppleScript dictionary viewer), a characteristic of an object that refers to a contained collection of related objects. Synonymous with a key-value coding to-many relationship. A document object might have a graphics element (or to-many relationship). (2) In an XML file, such as a scripting definition file, a tag-delimited unit of data.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

four-character code Four bytes of data that can be expressed as a string of four characters in the Mac OS Roman encoding. Used to uniquely identify terms and other items in an application's scriptability information.

implicitly specified subcontainer An object container that can be specified in an AppleScript script by context, rather than by an explicit reference. For example, explicitly specifying a `word` object in a `document` object might require this script reference: `fourth word of text of front document`. But if the application provides support for implicitly specifying the `text` container, the script reference can be simplified to this: `fourth word of front document`.

key (1) In key-value coding, a string that identifies a property. (2) In property lists, the part of a key-value pair that identifies a value in the list.

key-value coding (KVC) A mechanism (widely used in Cocoa) for accessing object properties indirectly by key. Cocoa scripting relies on KVC both to get and set properties of scriptable objects, and to identify the objects on which commands should operate.

keyword A four-character code used by the Apple Event Manager to identify a specific descriptor within an Apple event. Cocoa applications don't typically access the contents of individual Apple events directly, so they don't work with keywords, although they can do so by using methods of `NSAppleEventDescriptor` or by directly calling C functions of the Apple Event Manager.

KVC See key-value coding (KVC).

Model-View-Controller (MVC) A design pattern that assigns objects in an application to one of three roles and recommends a distinct separation among model, view, and controller objects. This is one of the central design patterns for Cocoa applications.

MVC See Model-View-Controller (MVC).

object containment hierarchy The hierarchy of objects in a running application. AppleScript and Cocoa scripting depend on the object containment hierarchy to locate the objects on which to perform an operation. See also AppleScript object model.

object-first command A script command that invokes a specified method of each specified receiver. With an object-first command, objects perform the specified action on themselves. Compare verb-first command.

object specifier Locates a scriptable object within an application's containment hierarchy. Cocoa scripting makes use of object specifiers to find objects in your application while executing a script command and to return information requested by a script. See also object containment hierarchy.

Open Scripting Architecture (OSA) Provides a standard and extensible mechanism for interapplication communication in Mac OS X. Implemented by a number of Mac OS X frameworks and subframeworks, including the AE framework (which implements the Apple Event Manager) and the OpenScripting framework. Also includes the AppleScript component, which implements the AppleScript language.

property (1) In a scripting definition file, a characteristic of a class that has a single value and is identified by a label. Synonymous with a key-value coding (KVC) attribute or to-one relationship. A window's `name` property would be equivalent to a KVC attribute, while its `document` property would be equivalent to a KVC to-one relationship. (2) In KVC, can refer to any of the three different kinds of object values that KVC can access: attributes, to-one relationships, and to-many relationships.

receivers The object or objects in an application designated to receive an AppleScript command.

receivers specifier In a script command object, the object specifier that specifies the objects in the application that should receive an command.

reference In an AppleScript script, the part of a script statement that identifies an object. Constructions such as `first rectangle` and `document "My Notes"` are references. Cocoa scripting provides built-in support for AppleScript's standard reference forms, listed in Table 6-1 (page 65).

required events Certain Apple events that all Mac OS X applications that present a graphical user interface should be able to respond to. These events can be sent by the Mac OS, as well as by other applications and by users executing scripts. They

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

include the `open application`, `open documents`, `print documents`, `open contents`, `reopen`, and `quit` events.

scriptability information Formally lays out the AppleScript object model for an application and maps it to application objects. Scriptability information specifies the terminology available for use in scripts that target the application. It also provides information, used by AppleScript and by Cocoa, about how support for that terminology is implemented in the application. See also scripting definition format, script suite format.

scriptable application An application that makes its operations and data available in response to Apple events, which are AppleScript messages.

script command object An object that encapsulates all the information needed to perform an AppleScript command. Cocoa scripting creates script command objects in response to Apple events received by the application. A script command object is instantiated from `NSScriptCommand` or from one of its subclasses—either those provided by Cocoa scripting to handle standard AppleScript commands, or those defined by your application to perform its unique operations.

scripting definition file A file in the scripting definition format that provides the scriptability information for an application. A scripting definition file has the extension `.sdef` and is also called an *sdef file* or simply an *sdef*. Compare script suite file, script terminology file.

scripting definition format An XML-based format that describes a set of scriptability terms and the commands, classes, constants, and other information used to support an application's scriptability. This format was introduced in Mac OS X version 10.2 and is used natively by Cocoa scripting starting in Mac OS X version 10.4. Also called *sdef format*. Compare script suite format.

script suite file A property list file, in a specific format, that describes scriptable classes in terms of their attributes, relationships, and supported commands and that has the extension `.scriptSuite`. Script suite files, together with corresponding script terminology files, declare the scriptability information for a scriptable application. See also script terminology file.

script suite format A format for providing scriptability information in the form of property list files, consisting of a script suite file together with a corresponding script terminology file. Compare scripting definition format.

script terminology file A property list file, in a specific format, that provides AppleScript terminology—the English-like words and phrases a scripter can use in a script—for the class and command descriptions in the corresponding script suite file. A script terminology file has the extension `.scriptTerminology`. Together with a corresponding script suite file, it declares the scriptability information for a scriptable application. See also script suite file.

sdef See scripting definition file.

Standard suite The scriptability information for a set of standard AppleScript terms that scriptable applications should support if possible. The Standard suite contains commands such as `count`, `delete`, `duplicate`, and `make`, and classes such as `application`, `document`, and `window`. Cocoa scripting provides a great deal of automatic support for the Standard suite.

suite Within an application's scriptability information, terms associated with related operations. For example, operations involving text, graphics, or databases are generally collected into separate text, graphics, and database suites.

to-many relationship In key-value coding (and in suite files), a property whose value is a collection of related objects. In a scripting definition file, represented by an `element` element.

to-one relationship In key-value coding (and in suite files), a property whose value has properties of its own. In a scripting definition file, represented by a `property` element.

top-level specifier In a nested object specifier, an object that has no container specifier. It represents the outermost container in the containment hierarchy. In most cases, the application object is the top-level specifier.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.

verb-first command A script command that invokes its `performDefaultImplementation` method. With a verb-first command, a single method performs the action (or verb) on any number of objects. Compare object-first command.

2008-03-11 | © 2008 Apple Inc. All Rights Reserved.