Inside Mac OS X

AppleScript Studio Terminology Reference

For AppleScript Studio version 1.2

November 2002

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

Simultaneously published in the United States and Canada.

Contents

Figures, Listings, and Tables

About This Document

AppleScript Studio combines an application framework with a development environment, allowing you to provide sophisticated user interfaces for applications that execute AppleScript scripts. Working with features from AppleScript, Project Builder, Interface Builder, and the Cocoa application framework, you can quickly create applications that support the Aqua human interface guidelines.

This document describes the AppleScript Studio scripting terminology available through AppleScript Studio version 1.2, which first became available with Mac OS X version 10.2. This document is a companion to *Inside Mac OS X: Building Applications With AppleScript Studio*, described in "Related Documentation" (page 19). This document does not describe the AppleScript scripting language in detail—see "Related Documentation" (page 19) for information on where to obtain *AppleScript Language Guide*.

The chapter "Terminology Fundamentals" (page 21) provides some key information about AppleScript Studio's scripting terminology.

The individual AppleScript Studio script suites (or collections of related terminology) are described in these chapters:

- "Application Suite" (page 42)
- "Container View Suite" (page 194)
- "Control View Suite" (page 244)
- "Data View Suite" (page 348)
- "Document Suite" (page 430)
- "Drag and Drop Suite" (page 448)
- "Menu Suite" (page 462)

- "Panel Suite" (page 474)
- "Text View Suite" (page 516)

Enumerations (a form of constant) are available to all suites, and are described in "Enumerations" (page 177) in the Application suite.

For a history of changes to this document, see "Document Revision History" (page 527).

What This Document Includes

This document describes the terminology available through AppleScript Studio version 1.2, which first became available with Mac OS X version 10.2. It lists every AppleScript Studio term, and points out terms in the AppleScript Studio scripting dictionary that are not supported as of version 1.2. For additional information on AppleScript Studio versions, see "Version Information" (page 22).

This document attempts to flag terminology differences between AppleScript Studio versions 1.0, 1.1, and 1.2 by providing Version Notes sections for terminology added or changed after version 1.0. If a class, command, event, or enumeration does not have a Version Notes section, then it has been available, unchanged, since AppleScript Studio version 1.0. See also "Version Information" (page 22).

Most object classes you use in AppleScript Studio scripts (such as the `application` (page 43), `view` (page 226), `button` (page 246), and other classes) are built on corresponding Cocoa classes (NSApplication, NSDocument, NSButton, and so on), although they do not expose all possible functionality from each Cocoa class. This document provides links to the documentation for corresponding Cocoa classes as a source of additional information on the underlying technology.

For more information on how the terminology in this document is organized, see "How Suite Information Is Organized" (page 27).

What This Document Does Not Include

This document does not describe the AppleScript scripting language in detail. For the definitive description of the AppleScript scripting language, see *AppleScript Language Guide*.

If you do not have a working knowledge of scripts and scripting terminologies, see the "Related Documentation" section in *Building Applications With AppleScript Studio*.

Although this document provides many tips on working with Project Builder and Interface Builder to create AppleScript Studio applications, it is not a reference for those tools, both of which provide extensive built-in help.

For information on where to find these documents, see "Related Documentation" (page 19).

In describing the user interface objects available to your application, this document points out some object properties you can set in Interface Builder, as well as some default property values for user interface objects you instantiate. Keep in mind that property values can vary depending on the settings you choose in Interface Builder and on the interaction of objects in your application.

Related Documentation

This document is a companion to *Inside Mac OS X: Building Applications With AppleScript Studio*, which introduces the key features of AppleScript Studio and includes detailed tutorials for creating AppleScript Studio applications. This companion document is available (along with other AppleScript Studio documentation) in Project Builder help, or at http://developer.apple.com/techpubs/macosx/CoreTechnologies/AppleScriptStudio/index.html.

The primary document for the AppleScript scripting language, *AppleScript Language Guide*, is available in Project Builder Help and at http://developer.apple.com/techpubs/macosx/Carbon/interapplicationcomm/AppleScript/applescript.html.

Inside Mac OS X: Aqua Human Interface Guidelines provides guidelines on when to use particular interface items and how to position them. It is available in Project Builder Help (choose Developer Help Center from the Help menu, then click the link for Developer Essentials) and at Apple's Developer website at http://developer.apple.com/ue/. For related information, see "Panels Versus Dialogs and Windows" (page 37).

You can find the latest online documentation for Project Builder and Interface Builder at http://developer.apple.com/techpubs/macosx/DeveloperTools/devtools.html.

You can find the latest Cocoa documentation at http://developer.apple.com/techpubs/macosx/Cocoa/CocoaTopics.html.

Note: To receive notification of documentation updates, you can sign up for ADC's free Online Program and receive their weekly Apple Developer Connection News email newsletter. See http://developer.apple.com/membership/ for more details about the Online Program.

CHAPTER 2

Terminology Fundamentals

The terminology described in this document gives AppleScript Studio applications the ability to work with Cocoa user interface objects in scripts. This chapter provides key information for working with AppleScript Studio's scripting terminology. It contains the following sections:

Version Information

To build AppleScript Studio applications, you must install a version of the Mac OS X Developer Tools that includes AppleScript Studio. To run an AppleScript Studio application, the target machine must have the AppleScript Studio runtime required for the application. An AppleScript Studio runtime is available if `AppleScriptKit.framework` is present in `/System/Library/Frameworks`.

AppleScript Studio attempts to maintain the following:

- An application created and built with an older version of AppleScript Studio can run with a newer runtime.
- An application created and built with a newer version of AppleScript Studio can run with an older runtime, if it doesn't use any features introduced after that runtime.

For example, an application built with AppleScript Studio version 1.1 that uses features added in version 1.1 requires the 1.1 runtime. However, a similar application that doesn't use any features from AppleScript Studio 1.1 can run with the 1.0 runtime. And an application built with AppleScript Studio version 1.0 can run with any runtime, through version 1.2.

Important
An application created with AppleScript Studio version 1.2 currently will not run with any earlier runtimes, even if it does not use any new features of 1.2. The online release notes for AppleScript Studio version 1.2, at Apple's Developer website, will be updated with a fix for this problem.

Table 2-1 lists AppleScript Studio versions, the development environment they are part of, and the system software the corresponding runtime is installed with.

Table 2-1 Availability for AppleScript Studio development environment and runtime

AppleScript Studio Version	Distributed with development environment	Runtime installed with
1.0	December 2001 Developer Tools CD	Developer Tools CD (with AppleScript 1.8), or Mac OS X version 10.1.2 software update (with AppleScript 1.8.1 or later)
1.1	April 2002 Developer Tools CD	Developer Tools CD (with AppleScript 1.8.2 or later)
1.2	Mac OS X version 10.2 Developer Tools CD	Mac OS X version 10.2, and later (with AppleScript 1.9 or later)

For an example of how your application can determine whether the required version of AppleScript Studio is present, see the Examples section for the `will finish launching` (page 167) event handler.

Starting with the version of Interface Builder released with Mac OS X version 10.2, there is a Nib File Compatibility preference on the General pane of the Interface Builder Preferences window. You should select a nib-file preference that suits your compatibility goals, from the following choices (and restart Interface Builder for the changes to take affect):

- **Pre-10.2 format:** applications will run with earlier versions of Mac OS X, but will not have access to new features (such as the circular `progress indicator` (page 296) or the brushed-metal, textured `window` (page 86) appearance)
- **10.2 and later format:** provides access to all new features, but is not guaranteed to run in previous versions of Mac OS X
- **Both Formats:** provides access to new features but will also run in previous versions of Mac OS X (though without the new features)

Building AppleScript Studio Applications

You use Project Builder, distributed with the Mac OS X Developer Tools CD, to create and build AppleScript Studio application projects, using one of the following three templates:

- AppleScript Application: used for applications that don't need to store data in documents
- AppleScript Document-based Application: used for applications that create and manage multiple documents
- AppleScript Droplet: used for creating droplets—script applications that launch when you drag a file or folder icon in the Finder and "drop" it on the droplet's icon

You use Interface Builder, also distributed with the Mac OS X Developer Tools CD, to create application interfaces for AppleScript Studio applications. Interface Builder provides active support for the Aqua human interface guidelines, including feedback lines that show when an object is properly placed. Throughout this document, you will find illustrations of the available Cocoa user interface objects and information on where to find them in Interface builder.

You can also find illustrations of windows, menus, controls, and many other user interface elements, as well as guidelines for when to use them and how to position them, in *Inside Mac OS X: Aqua Human Interface Guidelines*. Please use these guidelines to help create applications that take full advantage of the Aqua interface. "Related Documentation" (page 19) describes how to access this document, as well as documentation for Project Builder and Interface Builder.

All AppleScript Studio applications are Cocoa applications, built on the Cocoa application framework. As such, they are a mix of Cocoa code and AppleScript scripts, though you can write robust applications that require no additional Cocoa code on your part.

Some Cocoa capabilities are not currently available to AppleScript Studio classes. Where this is true, you have several options:

- You can use the `call method` (page 102) command to call methods of Cocoa classes directly.
- You can write your own Cocoa code as part of your application, and access it through the `call method` command. In fact, your application can access C, C++, Objective-C, Objective-C++, and Java (both directly and through the Mac OS X Java bridge), as demonstrated in the Multi-Language sample application distributed with AppleScript Studio.
- In some cases, the feature may become available in a future version of AppleScript Studio.

Sources of AppleScript Studio Terminology

AppleScript allows you to write scripts that control multiple applications, including many parts of the Mac OS itself. There are a number of tools for writing scripts, including the Script Editor application (distributed with the Mac OS) and various third party products. Most of the power in your scripts comes from the scripting terminology provided by applications and the operating system, not from the relatively small number of terms that are native to AppleScript itself. To take full advantage of the capabilities available, you need to know what terminology you can use in your AppleScript Studio application scripts.

Scripts in AppleScript Studio applications have access to the basic terminology that is available to all scripts, including:

- terminology provided by AppleScript
- terminology from scriptable parts of the Mac OS
- terminology from any available Apple and third party scripting additions (a scripting addition is code, stored in a `SystemAdditions` folder in any of the System, Library, User, or Network domains, that makes additional commands or coercions available to scripts on the same computer)
- terminology from any available scriptable applications (whether Carbon or Cocoa applications)

An AppleScript Studio application also has access to:

- terminology that is available to it as a Cocoa application (all Cocoa applications that turn on scripting support gain access to a certain default terminology; this terminology is described in more detail in "Script Suites" (page 26))
- terminology defined by AppleScript Studio itself in the AppleScriptKit framework (a framework is a type of `bundle` (page 51), or directory in the file system, that packages software with the resources the software requires); this terminology makes it possible to script a wide variety of user interface classes from the Cocoa application framework
- terminology from the script suites (described in "Script Suites" (page 26)) of any scriptable frameworks the application uses, and any scriptable bundles it loads

Note: The name spaces of these various terminologies may conflict, which can result in confusing behavior in scripts.

Script Suites

The Cocoa framework provides scripting information in the form of script suites, which are available to any application using the framework. Applications can also define additional suites. A **script suite** consists of at least one suite definition and one suite terminology, contained in external files. A **suite definition** describes scriptable objects in terms of their attributes, relationships, and supported commands. This information is stored as key-value pairs in a property list. A **property list** is a structured, textual representation of data, commonly stored in Extensible Markup Language (XML) format.

A **suite terminology** provides corresponding AppleScript terminology—the English-like words and phrases you can use in a script—for the class and command descriptions in a suite definition. Suite terminologies are also stored as property lists. Frameworks and applications typically place terminology files in a localized resource directory named `English.lproj`. (English is currently the only supported dialect in AppleScript.)

Note: Carbon applications store similar terminology information in a different format—as an `'aete'` (Apple event terminology) resource.

For more information on Cocoa's built-in scripting support, see the Cocoa Programming Topic Scriptable Applications.

How Suite Information Is Organized

Wherever possible, this document provides connecting links between class, command, event, property, element, and constant definitions. For each AppleScript Studio suite, it provides the following information.

- a brief suite overview
- the terminology for each class
  - plural form, parent class, and Cocoa class the class is based on

Note: Classes that start with "NS" (such as NSButton) are defined in the Cocoa application framework, and include links to Cocoa documentation for the class. Classes that start with "ASK" (such as ASKDataCell) are defined in AppleScript Studio's own framework, the AppleScriptKit framework, and do not have Cocoa documentation.

  - properties, elements, commands supported, events supported
  - version notes (if new or changed since version 1.0)
- the terminology for each command and event (if any in the suite)
  - abstract, syntax, parameter descriptions, return value description (if any)
  - examples
  - discussion (if any)
  - version notes (if new or changed since version 1.0)
- the terminology for each enumeration (all enumerations, which are available to every suite, are described in "Enumerations" (page 177) in the Application suite)
  - a description of the enumeration
  - a description of each constant in the enumeration
  - version notes (if new or changed since version 1.0)

Terminology Supplied by the Cocoa Application Framework

AppleScript Studio applications can take advantage of the built-in Cocoa terminology found in two default suites:

- The Standard suite
  - defines basic classes, including `application` and `window` (though AppleScript Studio defines its own version of these classes in its Application suite).
  - defines terminology for basic events, including `get`, `set`, `count`, `delete`, `print`, `quit`, and others. In Cocoa applications that activate scripting support, objects can support certain important commands, such as `get` and `set`, with little or no special code by the developer. For detailed information, see the Cocoa Programming Topic Scriptable Applications.
- The Text suite defines classes for working with text, including the `character`, `paragraph`, `word`, and `text` classes. For more information on working with text, see the Examples section for the `text view` (page 520) class.

Important
The Cocoa application framework provides default script suites to support scriptable Cocoa applications. While AppleScript Studio applications have access to this terminology, they will primarily use the terminology described in the next section.

Terminology Supplied by AppleScript Studio

AppleScript Studio defines its own script suites, which add to the scripting terminology defined by Cocoa. These suites provide additional terminology you can use to script objects based on many classes from the Cocoa application framework. This terminology is defined in several suite definition and suite terminology files in AppleScript Studio's own framework, the AppleScriptKit framework. Each suite can include class definitions (which include elements and properties), command and event definitions (which have an associated syntax), and enumerations (or predefined constants).

Note: AppleScript Studio draws a distinction between a **command**, which is a word or phrase a script can send to an object to request an action, and an **event**, which is an action, typically generated through interaction with an application's user interface, that causes a handler for the appropriate object to be executed. That is, scripts can send commands to objects, while events, often the result of user actions, generate calls to event handlers in scripts.

Each of the following chapters describes the terminology for one AppleScript Studio script suite. Enumerations (a form of constant) are available to all suites, and are described in "Enumerations" (page 177) in the Application suite:

"Application Suite" (page 42)

"Container View Suite" (page 194)

"Control View Suite" (page 244)

"Data View Suite" (page 348)

"Document Suite" (page 430)

"Drag and Drop Suite" (page 448)

"Menu Suite" (page 462)

"Panel Suite" (page 474)

"Text View Suite" (page 516)

Standard Key Forms

To access a property or element of an object, a script can specify the property or element by any key form it supports. Key forms simply indicate how data should be interpreted. For example, if a property supports the absolute position key form, a script can use a statement like the following:

```
set myWindow to the third window
```

Objects in AppleScript Studio applications are based on Cocoa class definitions, and AppleScript Studio's scripting support allows most classes to conveniently support a broad range of key forms. As a result, most objects you work with in AppleScript Studio applications support the following key forms:

- name

  ```
  tell drawer "mydrawer" to open drawer on top edge
  ```

- absolute position (numeric index)

  ```
  text field 1 of window 1
  ```

- relative position (before/after)

  ```
  window in front of window 3
  ```

- range (as a range of elements)

  ```
  name of windows 1 through 3
  ```

- satisfying a test

  ```
  the first window whose name is "mainwindow"
  ```

- unique ID

  ```
  set windowID to id of window 1
  ```

See also "Accessing Properties and Elements" (page 32).

Naming Conventions for Methods and Handlers

The Cocoa application framework follows a naming convention that helps explain when certain methods are called. This convention, which is reflected in the terminology for AppleScript Studio's event handlers, inserts `should`, `will`, and `did` in method names. Table 2-2 describes the meaning of these terms. Note that to indicate a completed operation, AppleScript Studio uses the past tense, rather than the term `did`.

Table 2-2 Naming conventions in Cocoa and AppleScript Studio

Cocoa phrase	Explanation	AppleScript Studio examples
`should`	Asks whether an operation should take place. You can cancel the operation by returning `false`.	`should open` `should close`

`will`	An operation is about to take place. You can prepare for it, but not prevent it.	`will resize`
		`will hide`
		`will quit`

`did`	An operation has completed. You can perform actions in response to it. AppleScript Studio uses past tense, rather than the term `did`.	`activated`
		`launched`
		`miniaturized`
		`zoomed`

So, for example, you can add a `should close` (page 157) handler to a `window` (page 86) object. When the handler is called, it can determine whether the user has performed some essential task—if not, it can return `false` and refuse to allow the window to close. A `will close` (page 166) handler cannot cancel the close operation, but it can perform any necessary tasks to prepare for closing. Finally, a `closed` (page 136) handler can perform any tasks required after closing.

At application startup time, handlers connected to the `application` (page 43) object are called in this order:

1. `will finish launching` (page 167)
2. `awake from nib` (page 130)

   Important: If you are working with the version of Interface Builder distributed with Mac OS X version 10.2, see "Version Information" (page 22) for information on setting a Nib File Compatibility preference.
3. `launched` (page 142)
4. `will become active` (page 165)
5. `activated` (page 130)
6. `idle` (page 139)

Accessing Properties and Elements

While elements typically have a singular name, such as `document`, `window`, and `pasteboard` for certain elements of the `application` (page 43) class, you use the plural form to access these elements in scripts. For example, you could use the following script in Script Editor to get a list of windows from an AppleScript studio application (and you can use similar terminology within the application, where you won't need the `tell application` block):

```
tell application "MyApp"
    set windowList to windows
end
```

A class may list properties that you cannot access in a particular application or at a particular time. For example, the `application` class lists a `key window` property that identifies the window that is the current target for keystrokes. But if an application has no windows open, it will not have a key window, and attempting to access the `key window` property will result in a nil value. (For more information on the key window, see the `key` and `main` property descriptions for the `window` (page 86) class.)

Starting with AppleScript Studio 1.2, you can use the `properties` property to access the properties of most AppleScript Studio objects. This feature relies on changes present in the version of Cocoa distributed with Mac OS X version 10.2. The following Script Editor script shows how to display the properties of a button in an AppleScript Studio application:

```
tell application "simple"
    properties of button 1 of window 1
end tell
```

The results of this script might be something like the following, where `missing value` indicates a value couldn't be obtained for a certain property:

```
{visible:true, content:false, double value:0.0, ignores multiple
clicks:false, title:"Button", image:missing value, name:missing value,
bordered:true, enabled:true, window:window id 1 of application
"TestingControls", allows mixed state:false, float value:0.0, image
position:no image, string value:"0", id:2, flipped:true, roll over:false,
opaque:false, alternate image:missing value, bounds:{47, 45, 131, 77}, bounds
rotation:0.0, tag:0, class:button, alignment:center text alignment, enclosing
scroll view:missing value, can draw:true, menu:missing value, target:missing
value, bezel style:rounded bezel, alternate title:"", needs display:false,
current cell:item id 3 of application "TestingControls", auto resizes:false,
key equivalent modifier:0, position:{47, 45}, formatter:missing value,
continuous:false, size:{84, 32}, cell:item id 3 of application
"TestingControls", state:0, button type:momentary push in button, font:item
id 4 of application "TestingControls", sound:missing value, integer value:0,
tool tip:missing value, super view:content view of window id 1 of application
"TestingControls", key equivalent:"", current editor:missing value,
transparent:false, visible rect:{0, 0, 84, 32}, contents:false}
```

If your application accesses a property that may not have a current value, you should enclose statements that use the property in a `try`, `on error` block. For an example of such a block, see the Examples section for the `path for` (page 120) command.

Similarly, an object may have zero or more of each kind of element listed for its class. If you attempt to access an element of a type for which an object currently has no instances, you will get back an empty list. You won't need a `try`, `on error` block, but depending on the logic in your script, you may need to verify that the list is not empty.

Important
This document points out some properties you can set in Interface Builder, and some default property values, but it does not attempt to be an exhaustive guide to Interface Builder, which includes its own built-in help.

Event Handler Parameters

In an AppleScript Studio application, you use Interface Builder to connect application objects to event handlers in application scripts. For example, if you connect a `clicked` (page 337) handler to a button, when a user clicks the button, the handler is called in your application script.

When you first connect a handler in Interface Builder, it installs a template in your script file. Listing 2-1 shows the template for a `clicked` handler. The template matches the full syntax for the event handler, which in this case has just one parameter, `theObject`. Nearly every event handler template includes the `theObject` parameter (which is almost always a reference to the object for which the handler is called), though some contain additional parameters as well.

Listing 2-1 A new clicked handler

```
on clicked theObject
    (* Add your script here. *)
end clicked
```

The syntax table for each event handler shows the complete syntax that is used for constructing a template. Once a template has been inserted into your application script, you are free to change the names of the parameters and to delete (or ignore) any parameters that are marked as optional in the syntax table.

One way to help document your scripts is, where possible, to change the name of the `theObject` parameter to reflect knowledge of the object it refers to. For example, if a `clicked` handler is only called in response to a click on a Search button, you might change the event handler declaration to `on clicked searchButton`. Changing a parameter name has no effect on how the handler operates.

If the handler may be called for one of several objects (say a series of buttons on one pane of a `tab view` (page 219)), you can both change the parameter name and add a comment to clarify its usage:

```
on clicked importButton
    (* This handler handles buttons on the Import pane. *)
    -- Your script statements here
end clicked
```

Connecting Key and Mouse Event Handlers

AppleScript Studio provides a number of key and mouse event handlers, such as `keyboard down` (page 141), `keyboard up` (page 141), `mouse down` (page 144), `mouse up` (page 148), `mouse dragged` (page 145), and so on. For objects of certain types, particularly for `application` (page 43) objects, key and mouse event handlers that you connect may never get called because they are handled by other objects before they get to the `application` object. If your application really needs to deal with these events, consider connecting them to objects in the user interface that inherit from the `control` (page 270) class, such as `button` (page 246), `slider` (page 306), `stepper` (page 310), or `text field` (page 314) objects.

Scripting Error Messages

AppleScript Studio applications are Cocoa applications, and when your application gets a script-related error message, that message is generated by Cocoa's scripting support. The following table lists scripting errors, through Mac OS X version 10.2.

The table includes the current error number. (For a brief description of some related AppleScript terminology, see Glossary of AppleScript Terms in the Cocoa Programming Topic Scriptable Applications.)

These messages may not always provide the detail you'd like for debugging your application, but other options are available. Among the most useful are using Project Builder's debugging support to set breakpoints, examine variables, and single-step through scripts; and using AppleScript Studio's `log` (page 118) command to output values and messages from your running application.

Table 2-3 Cocoa scripting error messages

Error constant (value)	Description
`NSNoScriptError` (0)	no error
`NSReceiverEvaluationScriptError` (1)	the object or objects specified by the direct parameter to a command could not be found
`NSKeySpecifierEvaluationScriptError` (2)	the object or objects specified by a key could not be found
`NSArgumentEvaluationScriptError` (3)	the object specified by an argument to a command could not be found
`NSReceiversCantHandleCommandScriptError` (4)	the object doesn't support the command that was sent to it
`NSRequiredArgumentsMissingScriptError` (5)	one or more required arguments are missing
`NSArgumentsWrongScriptError` (6)	an argument (or more than one argument) is of the wrong type or otherwise invalid
`NSUnknownKeyScriptError` (7)	an unidentified error occurred; indicates an error in the scripting support of a scriptable application, or of AppleScript Studio (or Cocoa) itself

Table 2-3 Cocoa scripting error messages (continued)

Error constant (value)	Description
`NSInternalScriptError` (8)	an unidentified error occurred; indicates an error in the scripting support of a scriptable application, or of AppleScript Studio (or Cocoa) itself
`NSOperationNotSupportedForKeyScriptError` (9)	the requested operation is not supported for the specified key
`NSCannotCreateScriptCommandError` (10)	the application received an invalid or unrecognized Apple event

Using the Sample Scripts

This document contains many sample scripts and event handlers, both complete and partial. In some cases, long statements in the samples contain hard carriage returns to make them easier to read. If you cut and paste samples from this document into AppleScript Studio application scripts, or into scripts you use with Script Editor or another script-editing application, you may have to remove the hard carriage returns to compile the sample.

Panels Versus Dialogs and Windows

You'll have to get used to a bit of naming inconsistency regarding the use of panels, dialogs, and windows. For historical reasons, the Cocoa application framework uses "panel" in many cases where *Inside Mac OS X: Aqua Human Interface Guidelines*, the final arbiter on Mac OS interface issues, would call for "dialog" or "window". The use of "panel" is even embedded in Cocoa class names, such as NSPanel,

NSFontPanel, NSOpenPanel, and so on. AppleScript Studio terminology, which is based closely on Cocoa user interface classes, generally adopts similar names, so that it provides the "Panel Suite" (page 473), which includes the `color-panel` (page 476), `font-panel` (page 480), `open-panel` (page 482), and related classes.

Given that history, it isn't possible for this document to be completely consistent when using these terms. However, the following list shows how some usages of "panel" correspond to terms used by the Aqua human interface guidelines. See "Related Documentation" (page 19) for information on where you can obtain *Inside Mac OS X: Aqua Human Interface Guidelines*.

- A `panel` (page 487) (based on the NSPanel class) is a special kind of window that typically serves an auxiliary function in an application, such as providing status to the user. The Aqua human interface guidelines refer to these kinds of windows as "dialogs" or "utility windows."
- A `color-panel` (page 476) (based on the NSColorPanel class) provides a standard user interface for selecting color in an application. The Aqua guidelines would call this a "Colors window."
- A `font-panel` (page 480) (based on the NSFontPanel class) displays a list of available fonts, allowing for preview and selection of the text display font. The Aqua guidelines call this a "Fonts window."
- An `open-panel` (page 482) (based on the NSOpenPanel class) provides a standard mechanism to query a user for the name of a file to open. The Aqua guidelines call this an "Open dialog."
- A `save-panel` (page 490) (based on the NSSavePanel class) allows users to specify the directory and name under which a file is saved. The Aqua guidelines call this a "Save dialog."

A dialog can be displayed as a sheet (attached to a window), in which case it is document modal, and a user can work with other windows in the application before dismissing the sheet. For example, you can use the optional `attached to` parameter of the `display` (page 496) command to display the `open-panel` (page 482) dialog as a sheet.

A Word on Unicode Text

When you get text back from AppleScript Studio, it will almost always be supplied as Unicode text. In some cases you may need to convert from Unicode text to plain text. For a discussion and example of how to convert Unicode text to plain text, see the Discussion section for the `default entry` (page 58) class.

In some cases, when you supply text to AppleScript Studio, such as for use with the `localized string` (page 116) command, you should ensure that you are working with Unicode text. See the Examples section for `localized string` for information on how to specify UTF-8 format for a file in your application project.

CHAPTER 2

Terminology Fundamentals

Application Suite

This chapter describes the terminology in AppleScript Studio's Application suite. For other suites, see the following:

- "Container View Suite" (page 194)
- "Control View Suite" (page 244)
- "Data View Suite" (page 348)
- "Document Suite" (page 430)
- "Drag and Drop Suite" (page 448)
- "Menu Suite" (page 462)
- "Panel Suite" (page 474)
- "Text View Suite" (page 516)

Application Suite

The Application suite defines its own version of certain classes that are defined in the Standard suite. These include the `application` (page 43) and `window` (page 86) classes. The Standard suite is described in "Script Suites" (page 26).

The Application suite also defines the `item` (page 73) class, which has `id` and `name` properties, and the `responder` (page 80) class, which inherits from the `item` class and serves as a superclass for the `window` (page 86), `view` (page 226), and `control` (page 270) classes. These and other classes that inherit from `responder` can respond to user actions.

To work with the many high-level classes it contains, the Application suite defines a large number of commands and event handlers for working with the application, windows, mouse and keyboard events, and so on.

The classes, commands, events, and enumerations in the Application suite are described in the following sections:

Classes

The Application suite contains the following classes:

application

Plural: applications
Inherits from: responder (page 80)
Cocoa Class: NSApplication

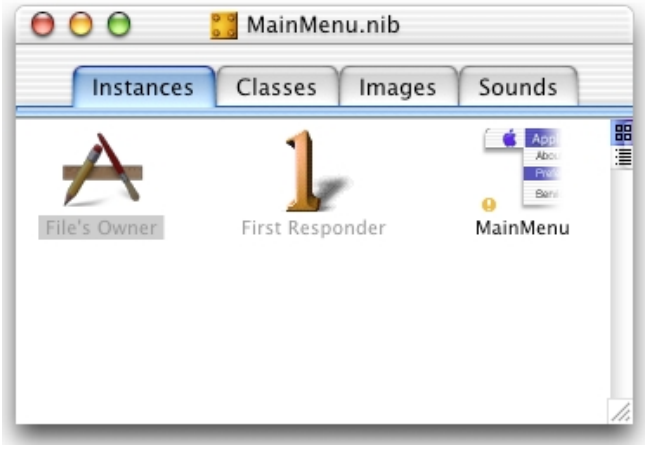
Manages an the main event loop of an application object, as well as resources used by all of the application's objects.

The main purpose of an application object is to receive events and distribute them to the appropriate objects that can respond to them (typically view subclasses). For example, all keyboard and mouse events go directly to the window (page 86) object associated with the event. In an AppleScript Studio application, these events can be

dispatched to script event handlers you connect in Interface Builder. See the `responder` (page 80) class for more information about how an application handles mouse and key events.

Every AppleScript Studio (and Cocoa) application has exactly one instance of the `application` object, created automatically as part of the application project in Project Builder. Figure 3-1 shows the File's Owner instance in the main nib window. In the main nib file, File's Owner always represents `NSApp`, a global constant that references the NSApplication object for the application. The `application` object serves as the master controller for the application. You attach handlers to an `application` object in Interface Builder through the File's Owner instance. For more information on nibs and File's Owners, see the `awake from nib` (page 130) command.

Figure 3-1 File's Owner instance (representing the application object) in an Interface Builder nib window



The `application` object supplies shared instances of a `color-panel` (page 476), `font-panel` (page 480), `open-panel` (page 482), and `save-panel` (page 490) that you can use in your application scripts. For related information, see the Programming Topic Choosing Colors With Color Wells and Color Panels. The `application` object also provides elements for storing items such as `image` (page 72), `movie` (page 75), and `sound` (page 81) objects with the application.

Properties of application objects

In addition to the properties it inherits from the `responder` (page 80) class, an `application` object has these properties (see the Version Notes section for this class for the AppleScript Studio version in which a particular property was added):

`active`

Access: read/write
Class: *boolean*
Is the application active?

`color panel`

Access: read/write
Class: *color-panel* (page 476)
the color panel for the application; created automatically for every AppleScript Studio application

`font panel`

Access: read/write
Class: *font-panel* (page 480)
the font panel for the application; created automatically for every AppleScript Studio application

`hidden`

Access: read/write
Class: *boolean*
Is the application hidden? setting the `hidden` property to `true` will set the `visible` property of all application `window` (page 86) objects to `false`; setting the property back to `false` will restore the previous `visible` property settings for the windows

`icon image`

Access: read/write
Class: *image* (page 72)
the icon for the application; the current default icon shows a pencil, brush, and ruler in the shape of a letter "A"; you can change the application icon by specifying an icon file in Project Builder (start by selecting the current target in the Targets pane; in Project Builder version 2.0.1, distributed with Mac OS X version 10.2, you set the icon file under Icon in the Info.plist Entries section)

`key window`

Access: read only
Class: *window* (page 86)
the current key window (the current target for keystrokes; see the `key` and `main` properties of the `window` (page 86) class for more information)

`main bundle`

Access: read/write
Class: *bundle* (page 51)
the main bundle of the application; the main bundle is the default location for all application resources, including compiled scripts; it is created automatically by Project Builder

`main menu`

Access: read/write
Class: *menu* (page 463)
the main menu of the application; the menu items on the main menu typically represent other menus; see the Examples section for more information

`main window`

Access: read only
Class: *window* (page 86)
the current main window of the application; the main window is the current focus of user activity; see the `key` and `main` properties of the `window` (page 86) class for more information)

`name`

Access: read only
Class: *Unicode text*
the name of the application

`open panel`

Access: read/write
Class: *open-panel* (page 482)
the open panel for the application; created automatically for every AppleScript Studio application

`save panel`

Access: read/write
Class: *save-panel* (page 490)

the save panel for the application; created automatically for every AppleScript Studio application

`services menu`

Access: read/write
Class: *menu* (page 463)
the services menu of the application; services allow an application to take advantage of features supplied by other applications, such as spell checking or mailing selected text

`user defaults`

Access: read/write
Class: *user-defaults* (page 83)
the user defaults key-value pairs for the application (see also `default entry` (page 58))

`version`

Access: read only
Class: *Unicode text*
the version of the application, from the short version string; by default, "0"

`windows menu`

Access: read/write
Class: *menu* (page 463)
the Windows menu of the application; by default, the Windows menu in Interface Builder contains Minimize and Bring All to Front items; in the running application, it also lists each open application window

Elements of application objects

An `application` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30). See the Version Notes section for this class for the AppleScript Studio version in which a particular element was added.

`data source` (page 371)
Specify by: "Standard Key Forms" (page 30)
the application's data sources

`document` (page 432)
Specify by: "Standard Key Forms" (page 30)

the application's documents

`drag info` (page 449)
Specify by: "Standard Key Forms" (page 30)
for internal use by AppleScript Studio only

`event` (page 62)
Specify by: "Standard Key Forms" (page 30)
the current event—the last event retrieved from the event queue

`image` (page 72)
Specify by: "Standard Key Forms" (page 30)
the application's images; by default an application has access to only one image, the default icon image (described in the Properties section above); you can add images (including additional icon images) to a project in Project Builder or Interface Builder, but they aren't added to the application's elements until you've loaded them with the `load image` (page 108) command

`item` (page 73)
Specify by: "Standard Key Forms" (page 30)
for internal use by AppleScript Studio only

`movie` (page 75)
Specify by: "Standard Key Forms" (page 30)
the application's movies; by default an application has no movies; you can add movies to the project in Project Builder or Interface Builder, but they aren't added to the application's elements until you've loaded them with the `load movie` (page 112) command

`pasteboard` (page 76)
Specify by: "Standard Key Forms" (page 30)
the application's pasteboards

`sound` (page 81)
Specify by: "Standard Key Forms" (page 30)
the application's sounds; by default an application has access to the sounds shown in Figure 3-6 (page 82) (the available sounds may differ for your system); you can add sounds to the project in Project Builder or Interface Builder, but they aren't added to the application's elements until you've loaded them with the `load sound` (page 115) command

`window` (page 86)

Specify by: "Standard Key Forms" (page 30)
the application's windows

Commands supported by application objects

Your script can send the following commands to an `application` object:

`display dialog` (page 503)
`path for` (page 120)
`quit` (from Cocoa's Core suite, described in Core Suites in the Cocoa Programming Topic Scriptable Applications)

Events supported by application objects

An `application` object supports handlers that can respond to the following events. Note that Key and Mouse commands may be handled by other objects without ever propagating their way back to the `application` object.

Application

`activated` (page 130)
`idle` (page 139)
`launched` (page 142)
`open` (from Cocoa's Core suite, described in Core Suites in the Cocoa Programming Topic Scriptable Applications)
`open untitled` (page 150)
`resigned active` (page 151)
`should open untitled` (page 159)
`should quit` (page 160)
`should quit after last window closed` (page 161)
`shown` (page 163)
`was hidden` (page 164)
`will become active` (page 165)
`will finish launching` (page 167)
`will hide` (page 168)
`will quit` (page 171)
`will resign active` (page 172)
`will show` (page 174)

Document

`document nib name` (page 138)

Key

keyboard down (page 141)
keyboard up (page 141)

Mouse

mouse down (page 144)
mouse dragged (page 145)
mouse entered (page 146)
mouse exited (page 146)
mouse up (page 148)
right mouse down (page 154)
right mouse dragged (page 155)
right mouse up (page 156)
scroll wheel (page 156)

Nib

awake from nib (page 130)

Examples

The following launched (page 142) handler, connected to the application object through the File's Owner instance in Interface Builder, sets the color of the application's color panel to red and makes the panel visible when the application is launched. AppleScript Studio script statements can refer to application properties without explicitly targeting the application object.

```
on launched theObject
    set color of color panel to {65535, 0, 0}
    set visible of color panel to true
end launched
```

You can use the following script in Script Editor to get the titles of the menu items in the main menu for the Drag Race application distributed with AppleScript Studio. Similar statements will work within an AppleScript Studio application script (though you won't need the tell statements).

```
tell application "Drag Race"
    title of menu items of main menu
    -- result: {"", "File", "Edit", "Window", "Help"}
end tell
```

Your script can dig down to get information about the menu items of the main menu as well. For example, the following line (inserted in the previous `tell` statement) gets the title of a menu item in the File menu:

```
title of menu item 10 of sub menu of menu item 2 of main menu
    -- result: "Page Setup…"
```

It can be convenient to use Script Editor (or a third party script application) to target an AppleScript Studio application, as shown here, to help determine the correct terminology for identifying objects in the application, especially in the case where you haven't given AppleScript names to the objects in Interface Builder. If you did name the menu items, then you could use a line like the following:

```
title of menu item "page setup" of sub menu item "file" of main menu
```

Version Notes

The following properties were added to the `application` object in AppleScript Studio version 1.1: `color panel`, `font panel`, `open panel`, `save panel`, `user defaults`.

The following elements were added to the `application` object in AppleScript Studio version 1.1: `data source`, `item`, `sound`.

The following elements were added to the `application` object in AppleScript Studio version 1.2: `drag info`, `pasteboard`.

bundle

Plural: `bundles`
Inherits from: `item` (page 73)
Cocoa Class: NSBundle

Represents a location in the file system that groups code and resources that can be used in a program. Every `application` (page 43) object has a `main bundle` property that represents the main bundle for that application. Although it is uncommon for AppleScript Studio applications, an application can contain additional bundles.

A `bundle` object corresponds to a directory where related resources—including executable code—are stored. A bundle can find requested resources in the directory and can dynamically load executable code (though that won't be necessary for most AppleScript Studio applications). A `bundle` object has properties that specify its

location in the file system, as well as the location of various items within the bundle. You can also use the `path for` (page 120) command to obtain the path for items in the application's bundle.

A bundle can contain images, sounds, localized character strings, and plug-ins. It also contains the application's `Info.plist` file, which specifies various information about the application or bundle that can be used at runtime, including document types and version and copyright information. For an example of how to check for the presence of the minimum version of the AppleScript Studio runtime needed by your application, see the Examples section of the `will finish launching` (page 167) event handler.

You build a bundle in Project Builder using one of these project types: Application, Framework, Loadable Bundle, or Palette. An AppleScript Studio application automatically contains a main application bundle, even if you take no special steps to create one or to specify its contents. Along with an `Info.plist` file, it contains a Scripts folder (in the Resources folder) that contains the application's compiled script files, each of which ends with the extension `.scpt`. For related information, see the Examples section below.

In Mac OS X, you can examine the contents of an application that is built as a bundle by Control-clicking the application icon and choosing Show Package Contents in the resulting contextual menu.

For additional information on working with bundles, see the Cocoa Programming Topic Loading Resources, as well as the `path for` (page 120) command.

Properties of bundle objects

In addition to the properties it inherits from the `item` (page 73) class, a `bundle` object has these properties:

`executable path`

Access: read only
Class: *Unicode text*
the path to executables in the bundle (by default an AppleScript Studio application has only one executable)

`frameworks path`

Access: read only
Class: *Unicode text*

the path to frameworks in the bundle (a framework is itself a type of bundle that packages software with the resources that software requires, including its interface)

`identifier`

Access: read only
Class: *Unicode text*
the identifier for the bundle, which you can specify in the Identifier field in Project Builder's target editor (the details depend on which version of Project Builder you are using); typical identifier names look like `com.yourcompany.somedirectorylocation.YourAppName`; you can see examples in the names of `.plist` files in `~yourUserName/Library/Preferences`

`path`

Access: read only
Class: *Unicode text*
the path to the bundle; not supported (through AppleScript Studio version 1.2); however, the following is a workaround that uses the `call method` (page 102) command to get the path to the bundle (in this example, to the main bundle of the application):

```
set thePath to call method "bundlePath" of object main bundle
```

`resource path`

Access: read only
Class: *Unicode text*
the path to resources in the bundle; depending on the location of the AppleScript Studio application, the path might be something like `/Users/userName/TestApp/build/TestApp.app/Contents/Resources`

`scripts path`

Access: read only
Class: *Unicode text*
the path to the scripts in the bundle; depending on the location of the AppleScript Studio application, the path might be something like `/Users/userName/TestApp/build/TestApp.app/Contents/Resources/Scripts`

`shared frameworks path`

Access: read only
Class: *Unicode text*

the path to the shared frameworks in the bundle; depending on the location of the AppleScript Studio application, the path might be something like `/Users/userName/TestApp/build/TestApp.app/Contents/SharedFrameworks`

`shared support path`

Access: read only
Class: *Unicode text*
the path to the shared support items in the bundle; depending on the location of the AppleScript Studio application, the path might be something like `/Users/userName/TestApp/build/TestApp.app/Contents/SharedSupport`

Commands supported by bundle objects

Your script can send the following commands to a `bundle` object:

`path for` (page 120)

Events supported by bundle objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The following `clicked` (page 337) handler shows how to use a bundle's `scripts path` property obtain the path to the scripts directory in an application's main bundle. It uses the `log` (page 118) command to display the result.

```
on clicked theObject
    set thePath to scripts path of main bundle
    log thePath
end clicked
```

Depending on the name and location of the project, the results of the previous handler would be something like the following:

```
2002-09-03 19:59:56.032 test project[667] "/Volumes/Projects/test project//
build/test project.app/Contents/Resources/Scripts"
```

The Examples section of the `path for` (page 120) command shows how to get the full, slash-delimited path of the compiled main script in an AppleScript Studio application, and how to find and load a script from the application's main bundle.

The following clicked (page 337) handler shows how to use the call method (page 102) command to access an external bundle, in this case the Terminal application that ships with Mac OS X. It uses the log (page 118) command to display the result. Once you have a reference to the external bundle, you can get information from it through its properties or with the path for command.

This example uses the call method (page 102) command to get the path to the bundle because, as noted in the Properties section above, the path property isn't currently supported (through AppleScript Studio version 1.2). This handler uses a try, on error block to handle the case where call method may be unable to return a bundle (if, for example, the Terminal application is not present in the Utilities directory).

```
on clicked theObject
    set theBundle to call method "bundleWithPath:" of class "NSBundle"
        with parameter "/Applications/Utilities/Terminal.app"
    try
        set thePath to call method "bundlePath" of object theBundle
        log thePath
    on error
        log "Problem getting path to Terminal.app"
    end try
end clicked
```

Note: Starting with AppleScript Studio version 1.2, you can say of theBundle, rather than of object theBundle.

The following example shows how to target a bundle outside the application:

```
on clicked theObject
    set myLibBundle to call method "bundleWithPath:" of class "NSBundle"
        with parameter "/Users/MyUser/MyStudioLib/build/MyStudioLib.app"
    try
        tell myLibBundle
            set scriptPath to path for script "MyStudioLib"
                extension "scpt"
        end tell
        log scriptPath
    on error
        log "Problem getting path to MyStudio.app"
    end try
```

```
end clicked
```

Between the examples shown here and the example in the Examples section of the `path for` (page 120) command, it is possible to load an external bundle and find and load scripts from the bundle. That means you can package frequently used scripts in a form that is easily accessible to any AppleScript Studio applications you write.

Important
When you load a script, you get a copy of the script, including a new copy of any global properties or variables. AppleScript Studio currently does not provide a convenient way to share data between scripts. You can use files to read and write data, but the overhead is at minimum inconvenient.

The following provides a simple example of how you might do this.

First, create an AppleScript Studio project in Project Builder, using the AppleScript Application template. Name the application "MyStudioLib" and, for this example, save it in your user folder (`/Users/yourname/`). In the main script file for the project, `MyStudioLib.applescript`, define handlers that return any scripts you want to implement. In this example, there is one handler, named `makeLibScript1`, which creates a script named `acknowledgeReceipt`. Although there is no return statement, `makeLibScript1` effectively returns the script `acknowledgeReceipt`.

```
on makeLibScript1()
    script myLibScript1

        -- Handlers
        on acknowledgeReceipt()
            display dialog "The acknowledgeReceipt script greets you."
        end acknowledgeReceipt

    end script
end makeLibScript1
```

Next, build the project, which causes a compiled script named `MyStudioLib.scpt` to be stored in the application bundle. You can define multiple handlers to return any scripts you want to make accessible from your script library, though this example supplies just one script.

Finally, you can add the following script statements to any AppleScript Studio project that needs to use any of the scripts in your MyStudioLib project. These statements:

- define properties (initialized to the constant `missing value`) to make scripts accessible throughout the file
- implement a `loadLibraryScripts` handler that loads the script file from the MyStudioLib application and extracts a specific script object from the script
- implement a `will finish launching` event handler that simply calls `loadLibraryScripts` when the application is launched
- implement a `clicked` handler to demonstrate how to call a loaded script; your application can make similar calls from throughout its script file

```
property libraryScript : missing value
property libScript1 : missing value

on loadLibraryScripts()
    set scriptPath to missing value
    set myLibBundle to call method "bundleWithPath:" of class "NSBundle" with
parameter "/Users/yourname/MyStudioLib/build/MyStudioLib.app"
    -- Log what we got for the bundle.
    log myLibBundle
    -- Use try, on error block to handle possible errors.
    try
        tell myLibBundle
            set scriptPath to path for script "MyStudioLib" extension "scpt"
        end tell
        set libraryScript to load script POSIX file (scriptPath)
        set libScript1 to makeLibScript1() of libraryScript
    on error
        log "Problem getting library script."
    end try
end loadLibraryScripts

on will finish launching theObject
    loadLibraryScripts()
end will finish launching

on clicked theObject
    tell libScript1 to acknowledgeReceipt()
```

```
end clicked
```

Version Notes

The `path` property in this class is not supported, through AppleScript Studio version 1.2. However, see the `path` property description for a workaround. This workaround is also demonstrated in one of the examples in the Examples section.

data

Plural: `data`
Inherits from: `item` (page 73)
Cocoa Class: NSData
Not supported (through AppleScript Studio version 1.2).

Version Notes

The `data` class was added in AppleScript Studio version 1.2, although it doesn't do anything in that version.

default entry

Plural: `default entries`
Inherits from: `item` (page 73)
Cocoa Class: ASKDefaultEntry
Specifies an entry in the Mac OS X user defaults system (a mechanism for storing default values as key-value pairs, where the key is simply a name string). You use this class in script statements to get, set, or remove the value for an entry in the application-specific defaults, which are typically used to store user preferences for the application.

For more information on the defaults system, see `user-defaults` (page 83), as well as the Cocoa Programming Topic User Defaults. You can also view man page information about the defaults system, using the Open Man Page menu from the Help menu in Project Builder (available starting with Mac OS X version 10.2) to display `defaults`, or by typing `man defaults` in a Terminal window (the Terminal application is located in `/Applications/Utilities`).

WARNING
You should not delete entries from the user defaults system in AppleScript Studio version 1.2. Doing so may cause your application to crash.

Properties of default entry objects

In addition to the properties it inherits from the `item` (page 73) class, a `default entry` object has these properties:

`content`

Access: read/write
Class: *item* (page 73)
the value of the entry; synonymous with `contents`

`contents`

Access: read/write
Class: *item* (page 73)
The value of the entry

Events supported by default entry objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The `application` (page 43) class has a `user defaults` property which you can use to manipulate user defaults entries. For example you could use the following to create a new entry in the user defaults. (An AppleScript Studio application script doesn't need to explicitly target the application—it's assumed within the script.)

```
make new default entry at end of default entries of user defaults with
properties {name:"defaultName", contents:"Testing"}
```

If you attempt to make a new entry for a key that already exists, no new entry is created and the value for the key is not changed. The assumption is that if the key already exists, it represents a saved value that you may want to preserve. However, you can change the value, if necessary, as shown below.

Defaults information for your application is stored in its plist file. For example, if the identifier for your application (which you set in Project builder) is "com.acme.application", the previous script statement results in a "defaultName" entry with value "Testing" in the file (where "~/" indicates the path to your user directory)

```
~/Library/Preferences/come.acme.application.plist
```

To get the value of any given entry in the user defaults, you simply refer to it by name. For example, given the previous `make new` statement, the following line returns the string value "Testing":

```
set myName to contents of default entry "defaultName" of user defaults
```

Important
The value of a `default entry` is Unicode text. Through AppleScript Studio version 1.2, you will have to convert a value to plain text if you want to coerce it to a number or boolean value. See the Discussion section below for more information.

Attempting to access an entry that doesn't exist will return an error, so you should enclose statements that access default entries within a try, on error block, as shown in the example in the Discussion section below.

To change a value, you use terminology like the following:

```
set contents of default entry "defaultName" of user defaults to "Check"
```

The following `awake from nib` (page 130) handler uses the statement `tell user defaults` to target the `user-defaults` (page 83) property of the `application` (page 43) object. After creating a new `default entry` object, it uses an another `tell` statement to log the contents of the entry, change the contents, and log the new contents.

```
on awake from nib theObject
    tell user defaults -- targets property of application
        make new default entry at end of default entries
            with properties {name:"test", contents:"testing"}

        tell default entry "test"
            log contents as string
```

```
            set contents to "completed"
            log contents as string
        end tell
    end tell
end awake from nib
```

The previous handler results in log entries like the following:

```
2002-08-12 13:46:32.260 Test3[477] "testing"
2002-08-12 13:46:32.340 Test3[477] "completed"
```

For related examples, see `user-defaults` (page 83).

Discussion

The contents of a `default entry` is Unicode text (as is the value returned by the `localized string` (page 116) command). You may need to convert the Unicode text to plain text—for example, to use in a command to another application that expects plain text, or to cast a retrieved string (such as “true” or “false”) to a boolean value. The following handler, from the SOAP Talk sample application distributed with AppleScript Studio, shows how to convert Unicode text to plain text. SOAP Talk convert strings to plain text because prior to AppleScript 1.9, Applescript’s `call xmlrpc` command won’t except Unicode text.

```
on getPlainText(fromUnicodeString)
    set styledText to fromUnicodeString as string
    set styledRecord to styledText as record
    return «class ktxt» of styledRecord
end getPlainText
```

The following fragment shows how to get the contents of a default entry, call `getPlainText` to convert it to plain text, and cast the result to boolean value. You could use similar statements to convert a numeric string to a number. The `try`, `on error` block deals with possible errors in getting the contents of the entry.

```
set tempString to contents of default entry "openFile" of user defaults
try
    set shouldOpen to getPlainText(tempString) as boolean
    if shouldOpen then
        -- Do whatever is needed to open.
    else
        -- Do what is needed when shouldOpen is false
```

```
    end
on error
    display dialog "Error getting should open value."
end
```

As an alternative to converting the Unicode text to plain text, you can compare the text directly, as in this example:

```
set shouldOpen to contents of default entry "openFile" of user defaults
try
    if shouldOpen is equal to "true" then
        -- Do whatever is needed to open.
    else
        -- Do what is needed when shouldOpen is false
    end
on error
    display dialog "Error getting should open value."
end
```

Version Notes

The `default entry` class was added in AppleScript Studio version 1.1.

The `content` property was added in AppleScript Studio version 1.2. For information on the difference between `content` and `contents`, see the Version Notes section for the `control` (page 270) class.

The `getPlainText` handler was added to the SOAP Talk sample application in AppleScript Studio version 1.2.

event

Plural: `events`
Inherits from: `item` (page 73)
Cocoa Class: NSEvent

Contains information about an input action, such as a mouse click or a key down. Each such user action is associated with a `window` (page 86), and is reported to the `application` (page 43) that created the window. The `event` object contains pertinent information about each event, such as where the mouse was located or which character was typed.

Several event handlers, such as `keyboard down` (page 141), `keyboard up` (page 141), `mouse down` (page 144), and `mouse up` (page 148), include an event parameter that refers to the `event` object associated with the handler. Within those handlers, you can use that parameter to access the properties described here.

Properties of event objects

In addition to the properties it inherits from the `item` (page 73) class, an `event` object has these properties:

`characters`

Access: read only
Class: *Unicode text*
the characters of the event; typically one typed character, such as "a"

`click count`

Access: read only
Class: *integer*
the click count of the event; one for a single-click, two for a double-click, and so on

`command key down`

Access: read only
Class: *boolean*
Is the Command key down?

`context`

Access: read only
Class: *item* (page 73)
the display context of the receiver (it is not recommended that you use this property)

`control key down`

Access: read only
Class: *boolean*
Is the Control key down?

`delta x`

Access: read only
Class: *real*
the x amount of a scroll wheel event

`delta y`

Access: read only
Class: *real*
the y amount of a scroll wheel event

`delta z`

Access: read only
Class: *real*
the z amount of a scroll wheel event; useful only for input devices that generate such a value

`event number`

Access: read only
Class: *integer*
the number of the event; this is a counter your application is unlikely to use; for more information, see the description for the `eventNumber` method in the NSEvent documentation

`event type`

Access: read only
Class: *enumerated constant* from "Event Type" (page 184)
the type of event

`key code`

Access: read only
Class: *integer*
the hardware-dependent value of the key pressed; you aren't likely to work with the key code and no constants are currently provided to specify key codes; however, you may want to experiment with possible values; (for example, for at least one keyboard, the key code for delete is 51 and for backspace is 117)

`location`

Access: read only
Class: *point*
the location in the window where the event happened; the location is returned as a two-item list of numbers {left, bottom}, where for example, `{0, 0}` would indicate the window was positioned at the bottom left of the display; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`option key down`

Access: read only
Class: *boolean*
Is the Option key down?

`pressure`

Access: read only
Class: *real*
a value between 0.0 and 1.0 representing the pressure of the input device for the event

`repeated`

Access: read only
Class: *boolean*
Is the event repeated?

`shift key down`

Access: read only
Class: *boolean*
Is the Shift key down?

`time stamp`

Access: read only
Class: *real*
the time that the event occurred, in seconds since system startup (for example, 2542.649003); by comparing their time stamp values, you can determine the elapsed time between two events; see also the Examples section

`unmodified characters`

Access: read only
Class: *Unicode text*
the unmodified characters of the event

`window`

Access: read only
Class: *window* (page 86)
the window associated with the event

Events supported by event objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The following `mouse down` (page 144) handler shows how to determine, from the passed `event` parameter, whether the Option key was pressed during the mouse down. Your handler can use such information to determine which actions to perform.

```
on mouse down theObject event theEvent
    if option key down of theEvent then
        log "the option key was used"
    else
        log "the option key wasn't used"
    end if
end mouse down
```

The following `mouse down` (page 144) handler uses the event's `time stamp` property to determine whether a user double-clicked. Of course, you could just connect a `double clicked` (page 338) handler to an object if you don't need this level of control. On the other hand, you may think of other uses for measuring the time between events.

This handler uses a script-global property to keep track of the previous timestamp, initializing it to the constant `missing value` to indicate that on startup it has not been set. It assumes that two clicks in less than a second constitutes a double click. Because the `time stamp` property is a real, you could measure for time in fractions of a second.

```
property lastTimeStamp : missing value

on mouse down theObject event theEvent
    if lastTimeStamp is missing value then
        set lastTimeStamp to time stamp of theEvent
    else
        if (time stamp of theEvent) - lastTimeStamp < 1 then
            display alert "You double clicked!"
        else
            set lastTimeStamp to time stamp of theEvent
        end if
```

```
    end if

end mouse down
```

font

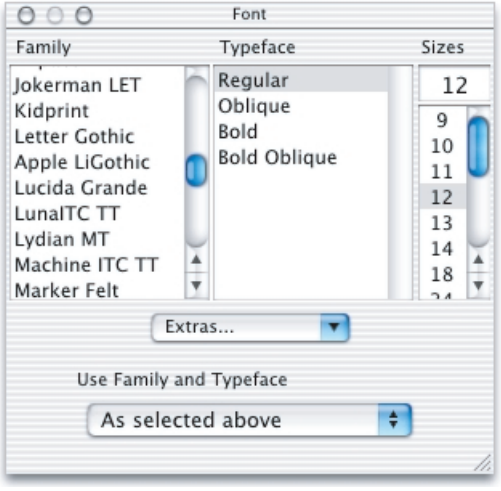
Plural: `fonts`
Inherits from: None.
Cocoa Class: NSFont

Not supported (through AppleScript Studio version 1.2). However, see the Examples section for information on setting fonts in Interface Builder.

Examples

You can set the font family, typeface, size, and color for `text field` (page 314), `text view` (page 520), and related classes in Interface Builder with the Font panel, shown in Figure 3-2.

Figure 3-2 The Font panel in Interface Builder



You can use the Extras pop-up to preview fonts, edit font sizes, open a color panel and choose a color for a font, and perform other operations. The Use Family and Typeface pop-up lets you choose from various preconfigured system fonts.

To make changes for a specific object, such as a text field, you select that object, open the Font panel by navigating to it from the Format menu (or by pressing Command-T), then make your choices.

formatter

Plural: formatters
Inherits from: None.
Cocoa Class: NSFormatter

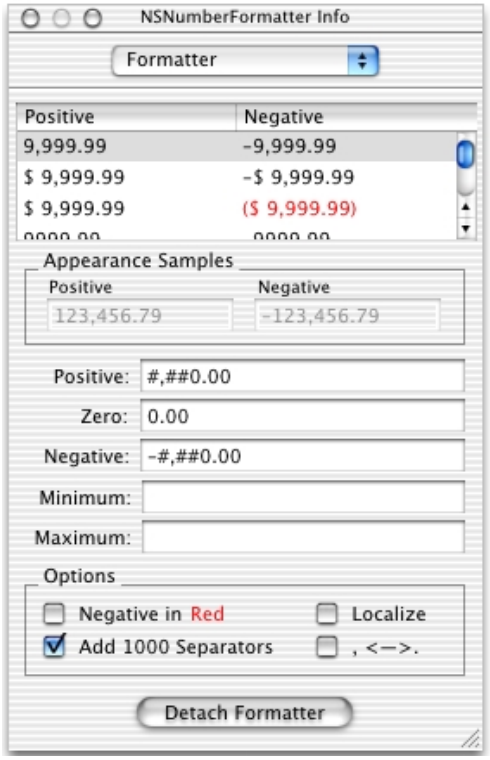
Controls the formatting of numbers or dates. A number formatter controls number formatting and a date formatters provide a similar function for dates. For information on specific formatter classes in Cocoa, see NSNumberFormatter and NSDateFormatter.

Figure 3-3 shows a number formatter you can drag from the Cocoa-Views pane of Interface Builder's Palette window. You can drag either a number formatter or a date formatter from the Palette window into a text field in your AppleScript Studio application to format the text in that field.

Figure 3-3 A number formatter in Interface Builder

Figure 3-3 shows the Formatter pane in Interface Builder's Info window, where you can adjust the format for a formatter. When you drag a formatter to a text field, Interface Builder automatically switches the Info window to the formatter pane, where you can adjust the format for that field. (You can open the Info window by typing Command-Shift-I.)

Figure 3-4 Interface Builder's Info window for a number formatter



Events supported by formatter objects

An `formatter` object supports handlers that can respond to the following events. You can use these steps to connect an event handler to a `formatter` object in Interface builder:

1. put the nib window for the `window` object that contains the formatter into outline mode by clicking the small outline icon (visible in Figure 3-1 (page 44)) above the right scroll bar.
2. Use the disclosure triangles to open the `window` and other objects until the `formatter` object is visible.
3. Select the formatter, then connect the event handler in the AppleScript pane of the Info window.

Nib

awake from nib (page 130)

Examples

Through AppleScript Studio version 1.2, formatter objects have no scriptable properties or elements. However you can use the call method (page 102) command to extract information from a formatter. You can also use call method to get a reference to a formatter from classes, such as control (page 270) and cell (page 256), that have a formatter property.

The text field (page 314) class inherits from the control class, so assuming you have added a formatter to a text field and given the text field the AppleScript name "formatted", you can use the following call to get a reference to the formatter object:

```
set theFormatter to call method "formatter"
    of (text field "formatted" of window 1)
```

Suppose the formatted text currently displayed is "$54.00" and you would like to get that exact string from the text field. The following clicked handler first gets a reference to the formatter, then uses call method again to call the stringForObjectValue: method of Cocoa's NSFormatter class to obtain the formatted text from the formatter. The handler uses a try, on error block to handle errors, and several log (page 118) statements to log various steps:

```
on clicked theObject
    tell (window of theObject)
        try
            set theValue to contents of text field "formatted"
            (* get formatter, then formatted text *)
            set theFormatter to call method "formatter"
                of object (text field "formatted")
            log "Got formatter"
            set theString to call method "stringForObjectValue:"
                of object theFormatter with parameter theValue
            log ("Got string: " & theString)
        on error
            log "Error getting formatted text."
        end try
        (* Perform any operations with the formatted text. *)
    end tell
end clicked
```

You can use the `call method` command to make other calls to methods of Cocoa's NSFormatter class

image

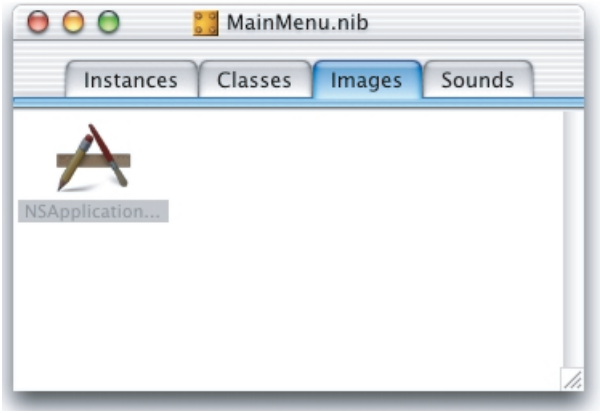
Plural: `images`
Inherits from: `item` (page 73)
Cocoa Class: NSImage

Represents an image. You don't typically script an image—instead you work with an `image view` (page 276) that contains the image.

The `application` (page 43) object has an `icon image` property to provide access to the application icon, as well as `image` elements. The `button` (page 246) and `button cell` (page 252) objects have `image` and `alternate image` properties, which can also be used to store icon images, or any other image. For related information, see the Cocoa Programming Topic Drawing and Images.

Figure 3-5 shows the Images tab in the default MainMenu.nib window for a new AppleScript Studio application in Interface Builder. The default application icon image is available automatically. You can insert an `image view` object in an application window by dragging it from the Cocoa-Other palette. You can then drag an image from the Images tab to the image view. You can also drag an image onto a button.

Figure 3-5 Application icon image in the Images tab in a main nib window in Interface Builder



You can add images to your application by dragging an image file into the Images pane of a nib window in Interface Builder, or by dragging it into the Files list in the Users & Groups pane in the Project Builder project for the application. You can then use the `load image` (page 108) command to load an image and the `image view` (page 276) class to display it. For information on how to free an `image` (page 72), see the Discussion section of the `load image` (page 108) command.

Events supported by image objects

Though you can drag an image into an image view in Interface Builder, you cannot connect any event handlers to an image.

Examples

For examples of working with an `image` object, see the `image view` (page 276) class.

`item`

Plural: `items`
Inherits from: None.
Cocoa Class: None.

Provides a parent class, with `name` and `ID` properties, for many other classes. The majority of AppleScript Studio classes descend from the `item` class, either directly, or through the `responder` (page 80), `view` (page 226), or other subclasses.

AppleScript Studio's `item` class is different than the `item` element of AppleScript's `list` class. You can use AppleScript Studio's `item` class to refer generically to an object that you know inherits from `item` or one of its subclasses. Suppose, for example, you have a handler that is always passed an object that is a subclass of `view` or `responder`. If the handler needs only to access the `name` or `ID` property of a passed object, it can treat passed objects as items. You will get an error, however, if you pass the handler an object that does not inherit from the `item` class.

Similarly, you can use AppleScript's `item` element to refer generically to any item in a list, though the list may contain different types of objects.

Properties of item objects

An `item` object has these properties:

`id`

Access: read only
Class: *integer*
the unique id of the object

`name`

Access: read/write
Class: *Unicode text*
the name of the object; you provide an AppleScript name for an object in Interface Builder, as described in the Examples section

Commands supported by item objects

Your script can send the following commands to an `item` object:

`log` (page 118)

Events supported by item objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

It is common to refer to objects by name in AppleScript Studio scripts. For example:

```
set userInput to contents of text field "input" of window "main"
```

You provide an AppleScript name for an object in Interface Builder with these steps:

1. With the object selected, open the Info window by choosing Show Info from the Tools menu or by typing Command-Shift-I.
2. Use the pop-up menu at the top of the Info window or (type Command-6) to display the AppleScript pane.
3. Type the name in the Name field.

You can use the following script in Script Editor to get the ids of every view in every open window of a simple document-based application. For testing purposes, each document window contains a number of buttons and text fields. Similar statements will work within an AppleScript Studio application script (though you won't need the `tell application` statement).

```
tell application "SimpleDocTest"
    id of every view of every window
end tell
```

Running this script with three open windows resulted in the following list of lists, containing one list of ids for each window:

```
{ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11},
{12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22},
{23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33} }
```

You can replace "id" with "name" in the script above to get the AppleScript name of every view (that has a name) in every open window.

For an example that uses `item` to access items in a list, see the sample script in the Discussion section for the `load image` (page 108) command.

`movie`

Plural: `movies`
Inherits from: `item` (page 73)
Cocoa Class: NSMovie

Provides a simple interface for loading a QuickTime movie into memory. You don't typically script a `movie` object itself. Instead, you work with the `movie view` (page 286) class.

Objects such as `application` (page 43) and `movie view` (page 286) have movie properties, but `movie` objects themselves currently have no scriptable properties or elements.

You can add a movie to your application by dragging a movie file into the Files list in the Users & Groups pane in the Project Builder project for the application. To use a movie, you load it with the `load movie` (page 112) command. For information on freeing `movie` objects, see the Discussion section for the `load image` (page 108) command.

Events supported by movie objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

For an example that loads a movie into a movie view, see the Examples section for the `movie view` (page 286) class.

pasteboard

Plural: `pasteboards`
Inherits from: `item` (page 73)
Cocoa Class: NSPasteBoard

Provides an interface to a pasteboard server that supports data transfer between applications, as in copy, cut, paste, or drag-and-drop operations. The data can be placed on the pasteboard in a variety of representations. A pasteboard is an element of the `application` (page 43) object and is analogous to the clipboard, except that there are multiple pasteboards available:

- general
- font
- ruler
- find

- drag

The font and ruler pasteboards are not used at this time, but may be available in a future release. The general pasteboard is the main pasteboard. To get the contents of the general pasteboard you can use `contents of pasteboard "general"`. You use this same format for the find pasteboard (which is used to set the value for find and replace operations in most applications). The drag pasteboard is used during drag-and-drop event handling.

A given pasteboard can contain a number of format types. The following types are directly supported by AppleScript Studio: “color”, “file”, “file names”, “font”, “html”, “image”, “pdf”, “pict image”, “postscript”, “rich text”, “rich text data”, “ruler”, “string”, “tabular text”, “url”, and “vcard”.

You may see additional pasteboard types that are defined by the system or by other applications. You can determine the types available at any given time for a pasteboard by looking at the types property. For example, if you use the phrase `types of pasteboard "general"` you might get a list result like `{"rich text", "string", "NeXT plain ascii pasteboard type", ...}`. You might also see other types not defined above (most of which will appear as `"CorePasteboardFlavorType 0x54455854"`).

When you want to get data from a pasteboard you need to set the `preferred type` property of the pasteboard (although by default the preferred type will be “string”).To get the data from the general pasteboard as a string, you can use the following:

```
set preferred type of pasteboard "general" to "string"
set myString to contents of pasteboard "general"
```

Note: In this version of AppleScript Studio (version 1.2), you must do some preparation before you can set the contents of a pasteboard directly with new data. To do so, you invoke the `call method` (page 102) command. The parameters to `call method` should be a list of types (described above) and the owner (usually 0 to represent nil).

The following example shows how to use the mechanism described in the previous note to put a string on the pasteboard.

```
call method "declareTypes:owner:" of pasteboard "general"
    with parameters {{"string"}, 0}
set contents of pasteboard "general" to "some new contents"
```

For an additional example, see the Drag and Drop sample application, distributed with AppleScript Studio (starting with version 1.2). For related information, see the "Drag and Drop Suite" (page 448), as well as the Cocoa Programming Topics Cutting and Pasting, Drag and Drop, and System Services.

Properties of pasteboard objects

In addition to the properties it inherits from the `item` (page 73) class, a `pasteboard` object has these properties:

`content`

Access: read/write
Class: *item* (page 73)
the contents of the pasteboard; see the Discussion section

`contents`

Access: read/write
Class: *item* (page 73)
the contents of the pasteboard; see the Discussion section

`name`

Access: read/write
Class: *Unicode text*
the name of the pasteboard; one of the values "general", "font", "ruler", "find", and "drag"

`preferred type`

Access: read/write
Class: *Unicode text*
the preferred data type when getting or setting the contents of the pasteboard; one of the type values listed above in the description for this class

`types`

Access: read
Class: *list*
a list of the data types supported by the pasteboard (consisting of type values listed above in the description for this class)

Events supported by pasteboard objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

You can list the pasteboards for an application with a script such as the following (you can use the same statements within an AppleScript Studio application, but you won't need the `tell` block):

```
tell application "myApp"
    pasteboards
end tell
```

The following will get the types for a pasteboard:

```
tell application "myApp"
    types of pasteboard "general"
end tell
```

The following `awake from nib` (page 130) handler (from the Drag and Drop sample application distributed with AppleScript Studio) uses the `register` (page 123) command to register the drag types an object can respond to.

```
on awake from nib theObject
    -- Enable support by registering the appropriate types.
    tell theObject to
        register drag types {"string", "rich text", "file names"}
end awake from nib
```

Discussion

You can use the `content` and `contents` properties interchangeably, with one exception. Within an event handler, `contents of theObject` returns a reference to an object, rather than the actual contents. To get the actual contents of an object (such as the text contents from a `text field` (page 314)) within an event handler, you can either use `contents of contents of theObject` or `content of theObject`.

For a sample script that shows the difference between `content` and `contents`, see the Version Notes section for the `control` (page 270) class.

Version Notes

The `pasteboard` class and the Drag and Drop sample application were added in AppleScript Studio version 1.2.

responder

Plural: `responders`
Inherits from: `item` (page 73)
Cocoa Class: NSResponder

Provides the basis for event and command processing. Any class that handles events must inherit from the `responder` class, as do the `application` (page 43), `document` (page 432), `window` (page 86), and `view` (page 226) classes.

Cocoa applications maintain a responder chain, which links together objects that can handle user-generated events and action messages. The first object in the chain is called the first responder. Events include key and mouse events, while action messages specify actions (or calls to methods) to be performed.

In an AppleScript Studio application, Cocoa events and action messages are translated into event handler calls, such as `keyboard down` (page 141), `mouse up` (page 148), `clicked` (page 337), or `should zoom` (page 162), to objects in the application. The default operation of the responder chain may be sufficient for many AppleScript Studio applications.

For objects of certain types, such as `application` (page 43) and `color well` (page 263) objects, key and mouse event handlers that you connect may never get called because they are handled by other objects before they get to the object. If your application really needs to deal with these events, consider connecting them to objects in the user interface that inherit from the `control` (page 270) class, such as `button` (page 246), `slider` (page 306), `stepper` (page 310), or `text field` (page 314) objects.

For related information, see the Cocoa Programming Topic Basic Event Handling.

Properties of responder objects

In addition to the properties it inherits from the `item` (page 73) class, a `responder` object has these properties:

`menu`

Access: read/write
Class: *menu* (page 463)
the menu for the responder; for a `window` (page 86) object, this is the same as the application's main menu property; for other objects, it will be undefined unless you have added a contextual menu for that object

Events supported by pasteboard objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The `responder` class is an abstract class that you don't typically target in your scripts. However, see the Examples section for the `window` (page 86) class for one situation where you might script a responder. See also the `first responder`, `key`, and `main` properties of the `window` class.

sound

Plural: `sounds`
Inherits from: `item` (page 73)
Cocoa Class: NSSound

Represents a sound that can be loaded and played. You don't typically script a sound directly, but you can use the `load sound` (page 115) command to load a `sound` and the `play` (page 327) command to play it. You can play any sound files supported by the NSSound class, including AIFF and WAV files. For related information, see the Cocoa Programming Topic Sound.

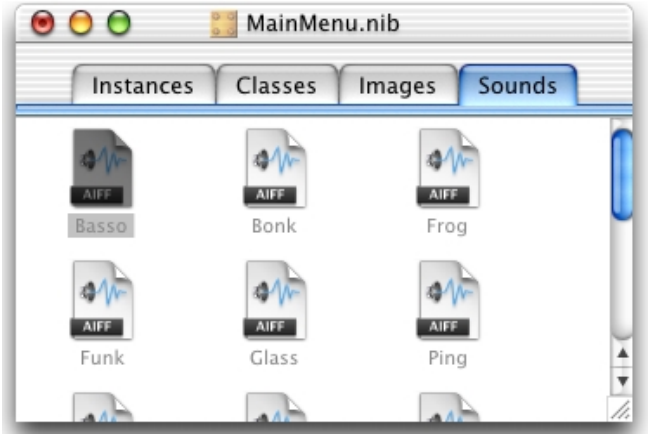
The `application` (page 43) object has `sound` elements, while `button` (page 246) and `button cell` (page 252) objects have `sound` properties.

Figure 3-6 shows sounds in a nib window in Interface Builder. You can drag a sound from the Sounds tab to an object that supports sounds, such as a `button` (page 246) object.

You can add sounds to your application by dragging a sound file into the Sounds pane of a nib window in Interface Builder, or by dragging it into the Files list in the Users & Groups pane in the Project Builder project for the application. To actually

play the sound, you will have to load it with the `load sound` command and play it with the `play` command. For information on freeing `sound` objects, see the Discussion section for the `load image` (page 108) command.

Figure 3-6 Sounds in the Sounds tab in a nib window in Interface Builder



Properties of sound objects

In addition to the properties it inherits from the `item` (page 73) class, a `sound` object has these properties:

`playing`

Access: read only
Class: *boolean*
Is the sound currently playing?

Commands supported by sound objects

Your script can send the following commands to a `sound` object:

`pause` (page 325)
`play` (page 327)
`resume` (page 327)
`start` (page 329)
`stop` (page 331)

Events supported by sound objects

Though you can drag a sound onto an object that supports sounds in Interface Builder, you cannot connect any event handlers to it.

Examples

The `load sound` (page 115) command provides an example that shows how an application can load and play a sound. The `slider` (page 306) class provides an example that uses a slider to let a user set the sound volume.

Version Notes

The `playing` property was added in AppleScript Studio version 1.1.

user-defaults

Plural: `user-defaults`
Inherits from: `item` (page 73)
Cocoa Class: NSUserDefaults

Provides access to the application's user defaults values (commonly used to store user preferences for the application).

The user defaults system in Mac OS X stores default values as key-value pairs, where the key is simply a name string. There are several domains for default values. 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. Default values in the application domain are commonly used to store preferences information for applications.

When an AppleScript Studio application is launched, it populates the application-specific defaults with all the defaults values it can obtain from both the application domain and the global domain, which contains defaults that apply to all of a user's applications. So an application can not only add its own defaults entries, it can make changes that override the loaded global defaults within its own domain. For example, the application could change the default date format (see the Examples section for more information). Such changes will not change global defaults for other applications.

Changes to the defaults system are automatically registered periodically, so that the next time the application is launched, the new values will be present.

To access user defaults definitions in your AppleScript Studio scripts, you can use the `user defaults` property that is associated with every `application` (page 43) object. Note that `user-defaults` is the class name, while `user defaults` specifies an object of that class.

WARNING
You should not delete entries from the user defaults system in AppleScript Studio version 1.2. Doing so may cause your application to crash.

For more information on the defaults system, see `default entry` (page 58), as well as the Cocoa Programming Topic User Defaults. You can also view man page information about the defaults system, using the Open Man Page menu from the Help menu in Project Builder (available starting with Mac OS X version 10.2) to display `defaults`, or by typing `man defaults` in a Terminal window (the Terminal application is located in `/Applications/Utilities`).

Elements of user-defaults objects

An `user-defaults` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`default entry` (page 58)
Specify by: "Standard Key Forms" (page 30)
the stored default value entries (key-value pairs)

Events supported by user-defaults objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

You can get a list of all of the named default entries with the following script statement. The list includes any entries you've added, as well as those provided by the Mac OS, such as "AppleKeyboardUIMode" and "NSDateFormatString":

```
set defaultsNames to name of every default entry of user defaults
```

Similarly, you can get the values for the entries with the following statement:

```
set defaultsContents to contents of every default entry of user defaults
```

The previous script statements access the `user defaults` property of the `application` (page 43) object. For related information, see the Example section for the `default entry` (page 58) class.

To use the user defaults system to store and retrieve preferences, your application should follow these guidelines:

1. Attempt to make new default entries for all preferences before attempting to retrieve the user's current settings from the defaults system. You make a default entry with a statement like the following:

   ```
   make new default entry at end of default entries
       of user defaults with properties
       {name:"defaultName", contents:"Testing"}
   ```

   You don't have to worry that this will replace any existing user preferences, because if you attempt to make a new entry for a key that already exists, no new entry is created and the value for the key is not changed. (See the `default entry` (page 58) class for information on how to change an entry when you need to.)

   A good place to perform this step is in a `will finish launching` (page 167) event handler connected to your `application` object. The application object is represented in Interface Builder by the File's Owner instance in the Instances pane of the MainMenu.nib window. This handler is called just before the application is launched. See the Discussion section for the `awake from nib` (page 130) event handler for information on the order in which event handlers are called on application start up.

2. Once you have set default values for preferences, you should attempt to read any existing user preferences from the user defaults system. You do so with statements like the following:

   ```
   set openWindowOnLaunch to
       contents of default entry "openWindowOnLaunch" as boolean
   ```

3. Your application can make new defaults entries or change existing ones as the user changes their preference settings.

4. Cocoa applications can call a method, `synchronize`, to specifically cause changes to be written out to the user defaults system. AppleScript Studio's `register` (page 123) command was originally intended to serve that purpose, but through AppleScript Studio version 1.2, the command does nothing to register defaults

(although it is used as part of drag-and-drop support). However, the Cocoa framework periodically calls the `synchronize` method, so user defaults in AppleScript Studio applications do get registered.

You can also use the `call method` (page 102) command to call Cocoa's `synchronize` method directly, as follows:

```
call method "synchronize" of object user defaults
```

5. The application should update its state to reflect any preference changes made by the user.

The Archive Maker application, distributed with AppleScript Studio, starting with version 1.1, provides a detailed example of how to use the user defaults system to work with user preferences.

Version Notes

The `user-defaults` class was added in AppleScript Studio version 1.1.

Starting with AppleScript Studio version 1.2, you can successfully use lists as a data type with default entries. In AppleScript Studio version 1.1, you could initially assign the contents of a default entry to a list and read it back, but trying to assign a new list to the contents of the default entry would not provide the correct result.

The Archive Maker application was first distributed with AppleScript Studio version 1.1.

Prior to AppleScript Studio version 1.2, this document listed `register` (page 123) as a command supported by the `user-defaults` class. In fact, using the `register` command with a `user-defaults` object does nothing.

window

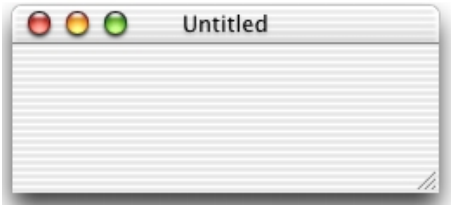
Plural: `windows`

Inherits from: `responder` (page 80)

Cocoa Class: NSWindow

Represents a window on screen. A `window` object manages an on-screen window, coordinating the display and event handling for its views. You can create and set up windows in Interface Builder, but you can also control many window properties directly in scripts. Figure 3-7 shows a simple window.

Figure 3-7 A simple window



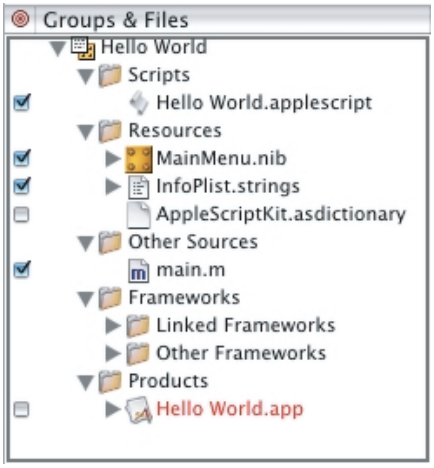
When you create an AppleScript Studio application from the AppleScript Application template in Project Builder, the application automatically contains a default window instance, stored in the `MainMenu.nib` nib file in the project's Resources group (shown in Figure 3-8). You use this application template for applications that don't need documents.

When you create an application from the AppleScript Document-based Application template, the application automatically contains a default window instance, stored in the `Document.nib` nib file. Document-based applications are set up to allow the user to create windows for multiple document instances.

Any application is free to define additional window nibs and to use them to create one or more window instances. You will find several predefined `window` objects (for windows, panels, and drawers) in the Cocoa-Windows pane of Interface Builder's Palette window, shown in Figure 4-3 (page 203).

In Interface Builder's Info window, you can set many attributes for windows, such as the kind of button controls it contains (Miniaturize, Close, and Resize), its size and resizing properties, and whether it is visible at launch time. To make a floating window (or utility window), for example, you use the window instance labeled "Panel" in Figure 4-3, then open the Attributes pane of the Info window and select the "Utility window" checkbox. In the version of Interface Builder distributed with Mac OS X version 10.2, you can even set the Textured Window attribute to specify the brushed-metal look for a window.

Figure 3-8 The Files list in the Groups & Files pane in a non-document AppleScript Studio project



In some cases, such as for a progress panel, you may only instantiate the window once, then show and hide it as needed (using either the `show` (page 125) and `hide` (page 107) commands, or by directly setting the window's `visible` property).

In other cases, you may want to instantiate a new window repeatedly and free it when the user is done with it (which you can do by setting the window's `released when closed` property in Interface Builder). The Mail Search application, distributed with AppleScript Studio, provides nibs and code for creating a one-time status panel, as well as a message window that is instantiated multiple times. (Prior to AppleScript Studio version 1.1, Mail Search was named Watson.)

For more information, see the Cocoa Programming Topic Windows and Panels.

Properties of window objects

In addition to the properties it inherits from the `responder` (page 80) class, a `window` object has these properties:

`alpha value`

Access: read/write
Class: *real*

the alpha value of the window; a value of 1.0 (the default) indicates the window is completely opaque, while 0.0 indicates the window is completely transparent; the following statement sets a value in the middle:

```
set alpha value of window "main" to 0.5
```

`associated file name`

Access: read/write
Class: *Unicode text*
the file name associated with the window; for a new, unsaved window, returns an empty string; for a window with an associated file, returns the full, POSIX-style (slash-delimited) path

`auto display`

Access: read/write
Class: *boolean*
not supported (through AppleScript Studio version 1.2); Automatically display the window?

`background color`

Access: read/write
Class: *RGB color*
the background color of the window; by default, {65535, 65535, 65535}, or white; not supported prior to AppleScript Studio version 1.2; after setting the background color, the new color will not become visible until the window is updated (via script or user interaction); see the Examples section for an example

`bounds`

Access: read/write
Class: *bounding rectangle*
the position and size of the window; the bounds are returned as a four-item list of numbers, {left, bottom, right, top}; for example, `{0, 0, 500, 250}` would indicate the window has its origin at the bottom left of the display, with a top right corner of 500, 250;
you can set a location or a bounds with real numbers (for example, {0.5, 0.5, 501.75, 250.1}), but the values returned from AppleScript Studio will always be rounded to whole numbers;
in this coordinate system, the origin is at the left, bottom and x, y values increase to the right and up, respectively; note that this is different than the Finder, which returns bounds as {left, top, right, bottom}, with the origin in the left, top and values increasing to the right and down

`can hide`

Access: read/write
Class: *boolean*
Can the window be hidden? default is `true`; overrides the `visible` property (that is, if `can hide` is `false`, setting the `visible` property to `false` will not hide the window)

`content view`

Access: read/write
Class: *anything*
the content view of the window; the superview of all other views in the window; the content view is inserted automatically; you don't typically interact with it in your scripts; you may be able to switch out the entire contents of a view by changing its content view—however, this is not recommended, and you can get a similar result by working with the `tab view` (page 219) and `tab view item` (page 224) classes

`document edited`

Access: read/write
Class: *boolean*
Has the document associated with the window been edited? (starting with AppleScript Studio version 1.2, the `window` class has a `document` (page 432) element); defaults to `false` if the window doesn't have an associated document; equivalent to the `modified` property of the `document` class

`excluded from windows menu`

Access: read/write
Class: *boolean*
Is the window excluded from the Windows menu? default is `false`

`first responder`

Access: read/write
Class: *responder* (page 80)
the first responder for the window (the first object in the responder chain to respond to user keystrokes or other actions); see also the Examples section for this class; as of AppleScript Studio version 1.2, you can effectively only set this property; getting it will not return a useful object

`has resize indicator`

Access: read/write
Class: *boolean*
Should the window have a resize indicator? default is `true`; you can set this property in the Info window in Interface Builder

`has shadow`

Access: read/write
Class: *boolean*
Should the window have a shadow?

`hides when deactivated`

Access: read/write
Class: *boolean*
Should the window be hidden when it is deactivated? if so, switching to another application will cause the window to be hidden; default is `false`; commonly used with utility windows (a special type of window described in the main description for this class); you can set this property in the Info window in Interface Builder

`key`

Access: read/write
Class: *boolean*
Is the window the key window? the key window is the current target for keystrokes; compare to the `first responder` and `main` properties

`level`

Access: read/write
Class: *integer*
the window's level; by default, set to 0; for more information, see the Discussion section for this class

`main`

Access: read/write
Class: *boolean*
Is the window the main window? the main window is the current focus of user activity; a window is often both key and main, but need not be; for example, a document window in a text editor may be the main window and key window, but when a user opens a find dialog, that dialog becomes the key window; after the user enters text and initiates a successful search, the document window again becomes both the main and key window; compare to the `key` property

`maximum size`

Access: read/write
Class: *point*
not supported (through AppleScript Studio version 1.2); the maximum size of the window as a two-item list {horizontal, vertical}

`miniaturized`

Access: read/write
Class: *boolean*
not supported (through AppleScript Studio version 1.2); Is the window miniaturized? (synonymous with minimized—reduced to an icon in the Dock)

`minimized image`

Access: read/write
Class: *image* (page 72)
not supported (through AppleScript Studio version 1.2); the image for the window when it is minimized

`minimized title`

Access: read/write
Class: *Unicode text*
the title of the window when it is minimized; this title shows up when you move the cursor over the minimized window icon in the Dock; by default, the same as the `title` property

`minimum size`

Access: read/write
Class: *point*
not supported (through AppleScript Studio version 1.2); the minimum size of the window as a two-item list {horizontal, vertical}

`needs display`

Access: read/write
Class: *boolean*
not supported in the `window` class (through AppleScript Studio version 1.2), but supported in the `view` (page 226) class; Should the window be displayed? setting this property to `true` causes the window to be redrawn; you can also use the `update` (page 126) command to update a view; see also the `update display` property of the `data source` (page 371) class

`opaque`

Access: read/write
Class: *boolean*
Is the window opaque? default is `true`; Cocoa considers opacity when drawing a window and its views (see the Cocoa Programming Topic Drawing and Images for more information), but most applications won't need to use this property; to make a window transparent, you use the `alpha value` property

`position`

Access: read/write
Class: *point*
the position of the window; the position is returned as a two-item list of numbers {left, bottom}; for example, `{0, 0}` would indicate the bottom, left corner of the window was positioned at the bottom left of the display; see the `bounds` property for more information on the coordinate system

`released when closed`

Access: read/write
Class: *boolean*
Should the window be released (equivalent to freed) when it is closed? default is `false`; you can set this value in Interface Builder; in some circumstances, you may prefer not to release a window but to hide it (with the `hide` (page 107) command or by setting its `visible` property to `false`), then show it again (with the `show` (page 125) command or by setting its `visible` property to `true`) when needed; once a window is released, you will have to create another instance from its nib file it to use it again

`sheet`

Access: read only
Class: *boolean*
Is the window a sheet? (that is, attached to another window)

`size`

Access: read/write
Class: *point*
the size of the window; the size is returned as a two-item list of numbers {horizontal, vertical}; for example, `{200, 100}` would indicate a width of 200 and a height of 100; see the `bounds` property for information on the coordinate system

`title`

Access: read/write
Class: *Unicode text*
the title of the window

`visible`

Access: read/write
Class: *boolean*
Is the window visible? default is `true` for the main window, but `false` for additional windows you add in Interface Builder; you can set this value in Interface Builder; see `released when closed` property for related information; setting the `hidden` property of an `application` (page 43) to `true` will set the `visible` property of all application windows to `false`, unless the `can hide` property is `false`, in which case it will have no effect

`zoomed`

Access: read/write
Class: *boolean*
Is the window zoomed?

Elements of window objects

A `window` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`box` (page 195)
Specify by: "Standard Key Forms" (page 30)
the window's boxes

`browser` (page 349)
Specify by: "Standard Key Forms" (page 30)
the window's browsers

`button` (page 246)
Specify by: "Standard Key Forms" (page 30)
the window's buttons

`clip view` (page 199)
Specify by: "Standard Key Forms" (page 30)

the window's clip views; not supported for windows (through AppleScript Studio version 1.2); a `scroll view` (page 211) uses a clip view, but typically without intervention by your application

`color well` (page 263)
Specify by: "Standard Key Forms" (page 30)
the window's color wells

`combo box` (page 265)
Specify by: "Standard Key Forms" (page 30)
the window's combo boxes

`control` (page 270)
Specify by: "Standard Key Forms" (page 30)
the window's controls

`document` (page 432)
Specify by: "Standard Key Forms" (page 30)
the window's document; provides access to a document from within the application's user interface

`drawer` (page 201)
Specify by: "Standard Key Forms" (page 30)
the window's drawers

`image view` (page 276)
Specify by: "Standard Key Forms" (page 30)
the window's image views

`matrix` (page 280)
Specify by: "Standard Key Forms" (page 30)
the window's matrixes

`movie view` (page 286)
Specify by: "Standard Key Forms" (page 30)
the window's movie views

`popup button` (page 292)
Specify by: "Standard Key Forms" (page 30)
the window's popup buttons

`progress indicator` (page 296)
Specify by: "Standard Key Forms" (page 30)

the window's progress indicators

`scroll view` (page 211)
Specify by: "Standard Key Forms" (page 30)
the window's scroll views

`secure text field` (page 301)
Specify by: "Standard Key Forms" (page 30)
the window's secure text fields

`slider` (page 306)
Specify by: "Standard Key Forms" (page 30)
the window's sliders

`split view` (page 216)
Specify by: "Standard Key Forms" (page 30)
the window's split views

`stepper` (page 310)
Specify by: "Standard Key Forms" (page 30)
the window's steppers

`tab view` (page 219)
Specify by: "Standard Key Forms" (page 30)
the window's tab views

`table header view` (page 385)
Specify by: "Standard Key Forms" (page 30)
the window's table header views

`table view` (page 386)
Specify by: "Standard Key Forms" (page 30)
the window's table views

`text field` (page 314)
Specify by: "Standard Key Forms" (page 30)
the window's text fields

`text view` (page 520)
Specify by: "Standard Key Forms" (page 30)
the window's text views

`view` (page 226)

Specify by: "Standard Key Forms" (page 30)
the window's views

Commands supported by window objects

Your script can send the following commands to a `window` object:

`center` (page 106)
`close` (from Cocoa's Core suite, described in Core Suites in the Cocoa Programming Topic Scriptable Applications)
`hide` (page 107)
`print` (from Cocoa's Core suite)
`register` (page 123)
`save` (from Cocoa's Core suite)
`show` (page 125)
`update` (page 126)

Events supported by window objects

A window object supports handlers that can respond to the following events:

Nib

`awake from nib` (page 130)

Panel

`alert ended` (page 510)
`dialog ended` (page 511)
`panel ended` (page 512)

Window

`became key` (page 135)
`became main` (page 136)
`deminiaturized` (page 137)
`exposed` (page 139)
`miniaturized` (page 143)
`moved` (page 149)
`opened` (page 150)
`resigned key` (page 152)
`resigned main` (page 153)

resized (page 153)
should close (page 157)
should zoom (page 162)
will close (page 166)
will miniaturize (page 169)
will move (page 170)
will open (page 170)
will resize (page 172)
will zoom (page 174)

Examples

Applications may need to perform additional initialization before showing the main window. One place you can do so is in the launched (page 142) event handler, which is called when the application is finished launching (and after the awake from nib (page 130) handler—another possible choice for performing additional initialization).

You can set a window's visible property to false in Interface Builder, then set it to true in the launched handler (as shown here) to show the window. For a more complete example, see the Assistant sample application, distributed with AppleScript Studio (starting with version 1.1). The startup-time calling order for application event handlers, including the launched handler, is listed in the description for the awake from nib (page 130) event handler.

This script assumes the window has the AppleScript name "main", which you set in the AppleScript pane of Interface Builder's Info window.

```
on launched theObject
    -- Perform any initialization before making window visible
    -- ...
    set visible of window "main" to true
end launched
```

Many common user interface classes (including subclasses of the control class) inherit from the view (page 226) class, which has a window element that identifies the window that contains the view. AppleScript Studio event handlers typically have a parameter that specifies the object for which the handler is called. If the object is an instance of a class that inherits from view (as it generally is), you can conveniently do the following, shown in a clicked (page 337) handler, to get access to the current window:

```
on clicked theObject
    set theWindow to window of theObject
    --Use the reference to the enclosing window as needed in the handler.
end clicked
```

To give the keyboard focus to an object such as a `text field` (page 314), you set the `first responder` property of its window to the object; for example, you could use the following line to give keyboard focus to a named text field:

```
set first responder of window 1 to text field "myText" of window 1
```

Note: As of AppleScript Studio version 1.2, you can effectively only set the `first responder` property; getting it will not return a useful object.

The following script statements set a window's background color to green and make the new color visible:

```
set background color of window "main" to {0, 65535, 0}
tell window "main" to update
```

Discussion

The stacking of windows depends on window level— windows at a higher level are shown in front of those at a lower level; windows at the same level can be displayed in front of or behind each other, but they cannot be displayed behind a window at a lower level.

Through version 1.2, AppleScript Studio does not define enumerated constants for setting window level, but as a convenience, Table 3-1 shows the current values for Cocoa's window level constants. You can use the values, but not the constants, in your scripts. Be aware that using these hard-coded values is not guaranteed to work in future versions of AppleScript Studio.

Table 3-1 Cocoa window-level constants

Cocoa window-level constant	Value
`NSNormalWindowLevel`	0
`NSFloatingWindowLevel`	3
`NSSubmenuWindowLevel`	3
`NSTornOffMenuWindowLevel`	3
`NSModalPanelWindowLevel`	8
`NSDockWindowLevel`	20
`NSMainMenuWindowLevel`	24
`NSPopUpMenuWindowLevel`	101
`NSScreenSaverWindowLevel`	1001

Version Notes

Support for the `background color` property was added in AppleScript Studio version 1.2.

Support for the `center` (page 106), `hide` (page 107), and `show` (page 125) commands was added in AppleScript Studio version 1.2.

Support for the `will open` (page 170) and `will zoom` (page 174) event handlers was added in AppleScript Studio version 1.2.

The following properties are not supported, through AppleScript Studio version 1.2:

- `auto display`
- `background color`

- `maximum size`
- `miniaturized`
- `minimized image`
- `minimum size`
- `needs display`

The `clip view` element is not supported, through AppleScript Studio version 1.2.

Support for the Textured Window attribute, which you can use to specify the brushed-metal look, was added to the version of Interface Builder distributed with Mac OS X version 10.2.

Commands

Objects based on classes in the Application suite support the following commands. (A command is a word or phrase a script can use to request an action.) To determine which classes support which commands, see the individual class descriptions.

`call method`

Provides a mechanism for calling methods of Objective-C objects from an application script. With the `call method` command, you can easily access Objective-C code you have written, or use Cocoa features that aren't currently exposed through AppleScript Studio's scripting terminology.

See the description for the `document` (page 432) class for details on how to use Project Builder (and an AppleScript Studio project) to find Cocoa class, method, and constant information you can use with the `call method` command.

Syntax

`call method`	*string*	required
`of`	*item*	optional
`of class`	*Unicode text*	optional
`of object`	*item*	optional
`with parameter`	*item*	optional
`with parameters`	*list*	optional

Parameters

string
The name of the method to call.

`of` *item* (page 73)
The object to send the method to. The `of` parameter was added in AppleScript Studio version 1.2, and can be used instead of the `of object` parameter if your application does not need to run with earlier versions.

You never use both `of` (or `of object`) and `of class`. If you don't specify either, the call goes to a method of the application's delegate object or, if the delegate doesn't support it, to the `application` (page 43) object itself.

Several classes in Cocoa use delegates, or helper objects, which can step in and perform operations for the class that uses the delegate. Delegates provide a convenient way to customize the behavior of a class without creating a new subclass. If you're not writing Cocoa code, you probably don't need to know anything more about delegates, but if you're interested, you will find more information in Using Window Notifications and Delegate Methods.

`of class` *Unicode text*
The class to send the method to. You never use both `of` (or `of object`) and `of class`.

`of object` *item* (page 73)
The object to call the method of. If your application needs to run with versions earlier than AppleScript Studio version 1.2, use `of object` instead of `of`.

`with parameter` *item* (page 73)
Specifies a parameter to be passed to the called method. Use this parameter for a method that takes a single parameter. You can use the parameter to pass an object or a simple value such as an integer. You can also pass a single list, which can

contain multiple items, but only if the called method expects a single parameter that encompasses multiple values, such as an array or dictionary. (Arrays and dictionaries are Cocoa types, based on the NSArray and NSDictionary classes.)

`with parameters` *list*

Specifies a list of parameters to be passed to the called method. Intended for use with methods that have more than one parameter, though you can also use it for a method with a single parameter. You specify a list with one item for each parameter of the specified method. An item within the list of parameters can be a list, if the called method expects a single parameter that encompasses multiple values in that position.

You never use both `with parameter` and `with parameters`. If you don't use either, it is assumed the method has no parameters.

You must use the `with parameters` parameter to pass a boolean value, even though it is a single parameter. You pass the boolean value as a single-item list. For example, to set the scrollable property of a matrix, you could use this statement:

```
call method "setScrollable:" of matrix 1 of window 1 with parameters {true}
```

Result

anything

The return value depends on the method that is called. The `call method` command can return the Cocoa types NSRect, NSPoint, NSSize, and NSRange, in addition to primitive types such as `int`, `double`, `char *`, as well as pointers to Cocoa objects and so on. You should use a `try`, `on error` block when working with the result (as shown in the Examples section for the `path for` (page 120) command).

Examples

The following is a method declaration from Cocoa's NSDocument class:

```
- (Bool) readFromFile: (NSString *)
            fileName ofType: (NSString *) docType
```

This method has two parameters, so to call it with the `call method` command, you use the `with parameters` option. In the following example, the list consists of the two string variables (whose values you have set prior to the call) enclosed in curly brackets: `{myFilenameString, myDocTypeString}`.

```
call method "readFromFile:ofType:" of (document 1 of window 1)
    with parameters {myFilenameString, myDocTypeString}
```

The following example calls the `performClick:` method of a `button` object, passing as a parameter another `button` object (enclosed in parentheses because it is a multi-term reference).

```
call method "performClick:" of (button 1 of window 1)
    with parameter (button 2 of window 2)
```

If your application will run with versions of AppleScript Studio before version 1.2, you should use the `of object` parameter. Here is how you would do so for the previous example.

```
call method "performClick:" of object (button 1 of window 1)
    with parameter (button 2 of window 2)
```

The following example calls a class method of NSNumber to get back a number object initialized with an integer value. It passes a single value (the number 10) for its one parameter. In this case, the parameter is unambiguous, and does not require parentheses

```
set theResult to call method "numberWithInt:" of class "NSNumber"
    with parameter 10
```

To call the NSView method `- (void) setFrame: (NSRect) frameRect`, you use a script statement similar to the following (where the single parameter is a list that specifies the frame):

```
call method "setFrame:" of (view 1 of window 1)
    with parameter {20, 20, 120, 120}
```

For more examples that use the `call method` command, see the Examples sections of the `bundle` (page 51) class and the `will finish launching` (page 167) event handler.

Version Notes

The `of` parameter was added in AppleScript Studio in version 1.2 to take the place of the `of object` parameter. Both are supported, but `of` is preferred. This can lead to clearer scripts. For example, instead of `call method "title" of object (window 1)` you can now use `call method "title" of window 1`. However, if your application must run with versions of AppleScript Studio prior to version 1.2, you will have to use the `of object` form.

Starting with AppleScript Studio in version 1.2, the `call method` command supports the double data type. In previous versions a double would be interpreted as an integer.

The `call method` command had severe limitations in AppleScript Studio version 1.0, including misinterpreting objects specified in the `with parameter` and `with parameters` parameters, and an inability to correctly return Cocoa class objects.

center

Centers a window on screen.

For more information on windows, see the Cocoa Programming Topic Windows and Panels.

Syntax

`center` *reference* required

Parameters

reference
a reference to the `window` (page 86) object that receives the `center` command

Examples

The following `clicked` (page 337) handler for a `button` (page 246) object centers the `window` (page 86) on which the button resides. The window is centered with respect to the device on which it is currently displayed.

```
on clicked theObject
    tell window of theObject to center
end clicked
```

Version Notes

The `center` command was added in AppleScript Studio version 1.1.

hide

Hides the object, if it is visible. Only `window` (page 86) objects can hide. Hiding a window has the same effect as setting its `visible` property to `false`, unless the window's `can hide` property is set to `false`, in which case the `hide` command will have no effect.

You cannot connect the `was hidden` (page 164) or `will hide` (page 168) event handlers to a `window` (page 86) object. Those event handlers apply only to the `application` (page 43) object and are called only when the application is hidden as a result of a user choosing Hide from the application menu or pressing Command-h.

For more information on windows, see the Cocoa Programming Topic Windows and Panels.

Syntax

`hide`	*reference*	required

Parameters

reference
a reference to the `window` (page 86) object that receives the `hide` command

Examples

The following `clicked` (page 337) handler shows how to hide a window, specifying the window in one of two ways.

```
on clicked theObject
    --Next line hides the window that contains the clicked object.
    --If you hide the current window, be sure you have a reference
    --  to it so you can make it visible again!
    tell window of theObject to hide

    --Next line would hide a window specified by name.
    --  tell window "second" to hide
end clicked
```

Hiding a window is equivalent to setting its `visible` property to false. The following statement has the same result as telling the window to hide (unless the window's `can hide` property is set to `false`, in which case the `hide` command will have no effect):

```
set visible of window "second" to false
```

load image

Loads the specified image. You typically load an image as an `image` (page 72) object and display it in an `image view` (page 276). The `application` (page 43) object can contain image elements. Classes such as `button` (page 246), `cell` (page 256), `drag info` (page 449), `menu item` (page 465), and `slider` (page 306) can have associated images.

You can load images of any type supported by Cocoa's NSBitmapImageRep class. That includes JPEG, PNG, GIF, TIFF, BMP, PICT, EPS, and PDF. You can store images in your AppleScript Studio project with the Add Files… command from the Project menu. You can also drag image files from the Finder into one of the groups (typically the Resources group) in the Files list in Project Builder's Groups & Files pane. You can also drag images into the Images pane of a nib window in Interface Builder.

For related information, see the Cocoa Programming Topics Drawing and Images and Image Views.

Syntax

`load image` *string* required

Parameters

string
specifies the image to be loaded; see Examples for more information

Examples

If an image is part of your project, you can load it by specifying its name, excluding the file extension. For example, if the file `starryNights.tiff` is stored in your project and you have an image view with the AppleScript name "artImages" in a window with the name "artWindow", you can load the image and make it the current image with the following statements:

```
set artImage to load image "starryNights"
set image of image view "artImages" of window "artWindow" to artImage
```

You could also perform this operation in one statement:

```
set image of image view "artImages" of window "artWindow" to load image
"starryNights"
```

Note that to load a TIFF image from the project without specifying the extension, the extension currently (through AppleScript Studio version 1.2) must be "`tiff`" not "`tif`".

If an image is not part of your project, you can load it by specifying a POSIX path to the image file. For example, if `sunFlowers.png` is stored on disk in `/User/Me/Images`, you could load the image with the following statement:

```
set image of image view "artImages" of window "artWindow" to load image "/
User/Me/Images/sunFlowers.png"
```

For an example that deletes an image, see the following Discussion section.

Discussion

The `image` object returned by the `load image` command is retained. In Cocoa, all objects have a retain count. Retaining increases the count; releasing decreases it. When the count reaches 0, the object is disposed of. An object returned by one of the load commands has a retain count of 1.

For most objects you use in an AppleScript Studio application, you don't need to worry about retaining or releasing the object. However, if you make multiple calls to `load image` (or `load movie` (page 112) or `load sound` (page 115)) and don't release the image (or `movie` (page 75), or `sound` (page 81)), your application memory usage will increase. To avoid this problem, you can explicitly delete an image (or movie or sound if it isn't a system sound) when you are finished with it. Deleting an image (or movie or sound) deletes it from the list of images (and so on) kept by the

application and releases it, so that when the retain count reaches 0 the object will be released. Note that if you delete an image that is currently displayed in an image view, it will not actually be freed until the image view is done with it.

The following script sample shows how an application could find all the images of a certain type stored in the application, then use an `idle` handler to switch the image in an image view every two seconds. Each time an image is loaded, the previous image is deleted to free the memory it uses.

The `launched` handler uses the `call method` (page 102) command to call a method of the application's main `bundle` (page 51) and obtain a list (stored as a property) of all the JPEG images in the application bundle. The first parameter specifies the file extension to look for; the second parameter specifies the bundle directory to search—passing an empty string specifies a search of all directories. The handler stores the count of found images as a property.

If there are any images, the `idle` handler loads the next image (possibly the first), saves a reference to the old image, sets the new image in the image view (to display it), and frees the old image. If there is only one image, the `idle` handler doesn't bother to keep reloading it.

```
property imagePaths : {}
property imageCount : 0
property imageIndex : 0

on launched theObject
    -- Get the path to all of the JPEG images in the application
    set imagePaths to call method "pathsForResourcesOfType:inDirectory:"
         of main bundle with parameters {"JPG", ""}
    try
        set imageCount to count of imagePaths
        log imageCount
    end try
end launched

on idle theObject
    -- If we have some images
    if imageCount > 0 then
        -- Only load an image if this is the first,
        --  or if we have more than one to cycle through.
        if (imageCount > 1) or (imageIndex is equal to 0) then
            -- Adjust the count
```

```
            set imageIndex to imageIndex + 1
            if imageIndex > imageCount then
                set imageIndex to 1
            end if

            -- Load the new image
            set newImage to load image (item imageIndex of imagePaths)

            -- Get a reference to the old image, if there is one
            set oldImage to image of image view "image" of window "main"

            -- Set the new image
            set image of image view "image" of window "main" to newImage

            -- Delete the old image (use try block in case no image)
            try
                delete oldImage
            end try
        end if
    end if

    -- Return 2 to call idle routine again in 2 seconds.
    return 2
end idle
```

Version Notes

In AppleScript Studio version 1.0, the `load image` command would not load an image that was not part of the Project Builder project for the application. Starting with AppleScript Studio version 1.1, `load image` will load such an image, given a POSIX style (slash-delimited) path to the image. Paths that you obtain from the `bundle` (page 51) class are in this format.

You can obtain a POSIX style path to a file or alias object using the `path to` command and the `POSIX path` property provided by AppleScript's Standard Additions. For example:

```
set thePath to path to desktop
    --result: alias "MacOSX:Users:BigCat:Desktop:"
set POSIXpath to POSIX path of thePath
    --result: "/Users/BigCat/Desktop/"
```

You can examine the terminology for AppleScript's Standard Additions by opening the file `/System/Library/ScriptingAdditions/StandardAdditions.osax` with Project Builder, Script Editor, or another application that can display scripting dictionaries.

load movie

Loads the specified QuickTime movie. You typically load a movie as a `movie` (page 75) object and display it in a movie view. The `application` (page 43) object can contain movie elements.

See the `movie view` (page 286) class for a list of the commands you can use to control a movie. For information on how to free a `movie` (page 75), see the Discussion section of the `load image` (page 108) command.

Syntax

`load movie` *string* required

Parameters

string
specifies the movie to be loaded

Examples

If a movie is part of your project, you can load it by specifying its name, excluding the file extension. For example, if the file `bdayparty4.mov` is stored in your project and you have a movie view with the AppleScript name "movies" in a window with the name "homeMovies", you can load the movie and make it the current movie with the following statements:

```
set currentMovie to load movie "bdayparty4"
set movie of movie view "movies" of window "homeMovies" to currentMovie
```

You could also perform this operation in one statement:

```
set movie of movie view "movies" of window "homeMovies" to load movie
"bdayparty4"
```

If a movie is not part of your project, you can load it by specifying a POSIX path to the movie file. For example, if `bdayparty4.mov` is stored on disk in `/User/Me/Movies`, you could load the movie with the following statement, instead of the one shown above:

```
set movie of movie view "movies" of window "homeMovies" to load movie "/User/
Me/Movies/bdayparty4.mov"
```

Version Notes

In AppleScript Studio version 1.0, the `load movie` command would not load a movie that was not part of the Project Builder project for the application. Starting with AppleScript Studio version 1.1, `load movie` will load such a movie, given a POSIX style path to the movie.

load nib

Loads the specified nib (or user interface resource file). Starting with AppleScript Studio version 1.1, you should use the `load nib` command instead of the `load panel` (page 508) command to load a panel (as shown in the Examples section below).

You create nib files in Interface Builder. For more information on nib files, see `awake from nib` (page 130).

Syntax

`load nib` *string* required

Parameters

string
specifies the nib file to be loaded, without the `.nib` extension

Examples

A nib file stores a description of one or more user-interface objects, including their sizes, locations, and connections to other objects. Loading a nib file unarchives (or creates instances of) the user-interface objects described in the nib. For example, the Mail Search sample application distributed with AppleScript Studio defines a separate nib for a window to display a found mail message. To create a new message window, it makes the following call:

```
set messageWindow to makeMessageWindow()
```

The `makeMessageWindow` handler contains the following code to load the nib file. Loading the nib file creates a message window. The handler then sets the name of the window. This handler results in windows titled “message1”, “message2”, and so on.

```
on makeMessageWindow()
    load nib "Message"
    set windowCount to windowCount + 1
    set windowName to "message " & windowCount
    set name of window "message" to windowName
    return window windowName
end makeMessageWindow
```

The following statements are from the `clicked` (page 337) handler in the Display Panel sample application distributed with AppleScript Studio. The `property` definition occurs outside the handler.

This script shows how to load a panel with the `load nib` command. If the settings panel window doesn’t exist yet, as determined by checking the global property, the script creates it by calling `load nib`, passing the name of the nib file (`Settings.nib`). The script then gets a reference to the settings panel, using its AppleScript name `"settings"`, set in Interface Builder when the nib was built.

```
property panelWindow : missing value

    -- Following is extracted from clicked handler:

    if not (exists panelWindow) then
        load nib "SettingsPanel"
        set panelWIndow to window "settings"
    end if
```

Version Notes

Prior to AppleScript Studio version 1.1, the Mail Search sample application was named Watson.

Prior to AppleScript Studio version 1.1, the Display Panel sample application used the `load panel` command. That command is not recommended starting with AppleScript Studio version 1.1—use the `load nib` command instead, as shown in the Examples section above.

load sound

Loads the specified sound. You typically load a sound as a `sound` (page 81) object, and play it with the `play` (page 327) command. The `application` (page 43) object can contain `sound` elements, while the `button` (page 246) and `button cell` (page 252) classes have `sound` properties.

You can play any sound files supported by the NSSound class, including AIFF and WAV files. For information on how to free a `sound` (page 81), see the Discussion section of the `load image` (page 108) command.

Syntax

`load sound`	*string*	required

Parameters

string
specifies the sound to be loaded; the string can name a sound in the application's project or provide a POSIX (slash-delimited) path to a sound file; for more detail, see the Examples section

Examples

By default, an AppleScript Studio project provides access to a number of system sounds. You can view these sounds in the Sounds pane of Interface Builder's MainMenu.nib window, shown in Figure 3-6 (page 82).

To load a sound that is part of your project, the sound file must have the extension of a supported sound file format, such as `aif`, `aiff`, or `wav`, but you don't specify the extension (see example below). You can load a sound located outside of your project by specifying a full POSIX-style (slash-delimited) path; in that case you do need to include the extension of the sound file.

The following `clicked` (page 337) handler uses the scripting addition `set volume` to set a low volume level, then loads and plays a sound from the file `Sosumi.aiff` in `/System/Library/Sounds`. In this case, you shouldn't have to specify the full path because Interface Builder provides access to the sound in the Sounds tab of the nib window.

```
on clicked theObject
    set volume 1 -- volume level goes from 0 (silent) to 7 (full volume)
    set theSound to load sound "Sosumi"
    play theSound
end clicked
```

If you do want to specify the full path to a sound, you could use the following statement to specify the same sound file:

```
    set theSound to load sound "/System/Library/Sounds/Sosumi.aiff"
```

This `clicked` handler doesn't free the sound it loads. For information about freeing loaded objects, see the Discussion section of the `load image` (page 108) command.

Version Notes

The `load sound` command was added in AppleScript Studio version 1.1.

Prior to Mac OS X version 10.2 and AppleScript Studio version 1.2, you could only play a 16-bit sound, not an 8-bit sound, and could only specify sound files that had the extension `aiff`.

localized string

Loads the string for the specified key from a project strings file (a file with the extension "`.strings`").

Syntax

`localized string`	*string*	required
`from table`	*Unicode text*	optional
`in bundle`	*bundle*	optional

Parameters

string
the name of the key that specifies the localized string to get

`from table` *Unicode text*
the name of the strings file (each strings file is a represented by a separate table); if you do not specify a strings file, the default is the project's `localized.strings` file

`in bundle` *bundle* (page 51)
the `bundle` that contains the strings table; if you do not specify a bundle, the default is the application bundle

Result

Unicode text
The localized string for the specified key. If the command is unsuccessful, the result is undefined, so you should use a `try`, `on error` block when working with the result, as shown in the Discussion section below.

Examples

Assume you have two localized string files (stored in UTF-8 format) in your project, one for English and one for French. You can set the format for a strings file to UTF-8 by following these steps:

1. select the file in the Files list in Project Builder's Groups and Files pane
2. open the Info window by typing Command-I or choosing Show Info in the Projects menu
3. with the Text Settings pane visible, choose UTF-8 in the File Encoding pop-up

Assume the localized string files are organized as follows:

`English.lproj/Localized.strings`:

```
/* Text for the Open button */
```

```
"OPEN_KEY" = "Open";
/* Text for the Close button */
"CLOSE_KEY" = "Close";
```

French.lproj/Localized.strings:

```
/* Text for the Open button */
"OPEN_KEY" = "ouvrez-vouz";
/* Text for the Close button */
"CLOSE_KEY" = "étroit";
```

You might then use the `localized string` command as follows:

```
set theString to localized string "OPEN_KEY" from table "Localized"
```

This script statement will obtain the appropriate string based on the application's current locale.

A string returned from a call to `localized string` is Unicode text. You may need to convert the string value to plain text—for example, to use in a command to another application that expects plain text, or to cast a retrieved string (such as "true" or "false") as a boolean value. For an example that shows how to do that, see the Discussion section for the `default entry` (page 58) class.

For more information on getting string text, see the Discussion section below.

Version Notes

The `localized string` command was added in AppleScript Studio version 1.1.

`log`

Logs the specified object. The `log` command outputs a value to the Console pane of the Run tab if you run the application in Project Builder or to the Console application (in `/Applications/Utilities`) if you run it from the Finder.

The `log` command can be extremely useful in debugging scripts or just studying how AppleScript Studio works.

Syntax

`log`	*reference*	required

Parameters

reference
a reference to the object to log; you can optionally supply a string instead of a reference

Examples

The following sample statements show how to log a string and an object. The text after the comment delimiters (`--`) shows the result of the statements, when inserted at the beginning of the `clicked` handler in the Drawer sample application distributed with AppleScript Studio (though logging a string will produce the same result in any application).

```
on clicked theObject
log "just testing"
-- result: "just testing"
log theObject
-- result: 2002-07-23 11:42:09.274 Drawer(488) button id 2 of window id 1
-- Rest of handler not shown.
```

You can also log variables or properties, as in the following:

```
log someCountProperty
-- result: 2002-09-17 17:04:45.596 AppName[488] 7
--       (if the value of someCountProperty is 7)
```

To use the `log` command within a `tell` statement that targets an application, you can use syntax like this:

```
tell application "Finder"
    tell me to log "Entered Finder tell block."
end
```

Version Notes

The `log` command was added in AppleScript Studio version 1.1.

path for

Returns the full path for the specified resource in the targeted bundle, or if no bundle is targeted, in the application's main bundle. For related information on bundles, including examples that target external bundles, see bundle (page 51).

Syntax

path for	*reference*	required
column	*integer*	optional
directory	*Unicode text*	optional
extension	*Unicode text*	optional
localization	*Unicode text*	optional
resource	*Unicode text*	optional
script	*Unicode text*	optional

Parameters

reference
a reference to the bundle (page 51) from which to get the path; if no bundle is specified, the main bundle of the application (page 43) object is used

column *integer*
the zero-based column index of the browser view; when using a browser to display a file system, you can use path for to get a path to the directory that contains the files in that column

directory *Unicode text*
specifies a directory to search in within the bundle

extension *Unicode text*
the extension of the object to search for

localization *Unicode text*
the locale for the resource to search for

resource *Unicode text*
the type of resource to search for

script *Unicode text*
the script to search for

Result

Unicode text

The path to the specified resource. If the command is unsuccessful, the result is undefined, so you should use a `try`, `on error` block when working with the result (as shown in the example below.)

Examples

You can use the following script in Script Editor to get the full, slash-delimited path of the compiled main script in an AppleScript Studio application (in this case named "tester"). Similar statements will work within an AppleScript Studio application script (though you won't need the `tell application` block). The script specifies the `main bundle` property of the `application` (page 43) object as the target for the `path for` (page 120) command.

```
tell application "tester"
    tell main bundle
        set scriptPath to path for script "tester" extension "scpt"
    end tell
end tell
```

Depending on the location of the project, the results of the previous script would be something like the following:

```
"//Volumes/Projects/tester/build/tester.app/Contents/Resources/Scripts/
tester.scpt"
```

Because both `bundle` and `application` objects understand the `path for` command, you can simplify this script to the following. When no bundle is specified, the application automatically looks in the main bundle.

```
tell application "tester"
    set scriptPath to path for script "tester" extension "scpt"
end tell
```

The following `clicked` handler uses the `path for` command to get the path to a compiled script `Application.scpt` in an AppleScript Studio application. Because no bundle is specifically targeted, the command looks in the main bundle of the `application` (page 43) object. It stores the path in a global property and uses a `try`, `on error` block to handle the case where the `path for` command doesn't return a valid path.

If `path for` is successful, the script uses the `log` (page 118) command to the show the path. It then uses the `POSIX file` scripting addition to get the file for the path and the `load script` scripting addition to load the script, then assigns the script to a global script property. At that point, scripts in the application can call handlers in the loaded script.

If the script is unsuccessful, it displays the returned error number and error message. AppleScript supplies the `missing value` constant as a placeholder for missing information.

```
property mainScriptPath : missing value
property theScript : missing value

on clicked theObject
    set mainScriptPath to path for script "Application" extension "scpt"
    try
        log mainScriptPath -- log the result
        set theScript to load script POSIX file (mainScriptPath)
        -- Other statements here to work with the script.

    on error errMsg number errNum
        -- Deal with any error in getting path--first log to console:
        log "Error loading script. " & "Error: " & errNum & " Msg: " & errMsg

        -- For user-related error, can display a dialog:
        display dialog "Error: " & errNum & ". " & errMsg
    end try
end clicked
```

The following is a possible log message generated when an error occurs (in this case, the script file did not exist, so the variable `mainScriptPath` did not get set):

```
2002-10-30 16:56:44.697 on error test[512] "Error loading script. Error: -2753
Msg: The variable mainScriptPath is not defined."
```

If you are not interested in the error number or error message, or are not expecting values to be returned for them, you can just use `on error`.

For related examples, see the Examples section for the `bundle` (page 51) class.

register

Registers the specified object to receive drag operations. For an object to respond to any of the drag-and-drop event handlers (described in "Events" (page 452)), you must register the drag types that the object can accept. You do this with the `register` command, using the `drag type` parameter to supply a list of the pasteboard drag types desired. Possible pasteboard types are listed with the `pasteboard` (page 76) class.

Syntax

`register`	*reference*	required
`drag type`	*list*	optional

Parameters

reference
a reference to the object that is registered to receive drag operations

`drag type` *list*
the pasteboard drag types that the object will accept; must be present to register for drag operations; registering an empty list will clear the pasteboard and prevent drag operations

Examples

The following `awake from nib` (page 130) handler registers two drag types (`"string"` and `"file names"`) for the object it is connected to. You could, for example, use this handler to register these drag types for a `text field` (page 314) object.

```
on awake from nib theObject
    tell theObject to register drag types {"string", "file names"}
end awake from nib
```

For more examples, see the Drag and Drop sample application distributed with AppleScript Studio (starting with version 1.2).

Version Notes

The `register` command was added in AppleScript Studio version 1.1, but didn't do anything. The command was made useful for drag and drop with the addition of the `drag type` parameter in AppleScript Studio version 1.2.

The Drag and Drop sample application is first available with AppleScript Studio version 1.2.

select

Not supported (through AppleScript Studio version 1.2). Selects the specified object or objects.

Syntax

`select`	*reference*	required
`at index`	*integer*	optional
`item`	*item*	optional

Parameters

reference
a reference to the object or objects to select

`at index` *integer*
the index of the object to select

`item item` (page 73)
the object to select;

select all

Not supported (through AppleScript Studio version 1.2). Selects all of the contained objects within the specified object.

Syntax

`select all`	*reference*	required

Parameters

reference
a reference to the object whose handler is called

show

Shows the specified object, such as a window or a panel, by making it the main and also the key window. Showing a window has the same effect as setting its `visible` property to `true`.

The `show` command has two mutually exclusive optional parameters, `behind` and `in front of`. This gives you some control as to the front to back order of the windows. For more information on ordering, see the `level` property and the Discussion section for the `window` (page 86) class. For additional information on windows, see the Cocoa Programming Topic Windows and Panels.

Syntax

`show`	*reference*	required
`behind`	*window*	optional
`in front of`	*window*	optional

Parameters

reference
a reference to the object to show

`behind` `window` (page 86)
the window to show behind (do not use with `in front of`)

`in front of` `window` (page 86)
the window to show in front of (do not use with `behind`)

Examples

The following `launched` (page 142) handler is from the XMethods Service Finder sample application, distributed with AppleScript Studio (starting with version 1.2). The `launched` handler is called near the end of the launch sequence, after objects in the application's main nib file have been created and initialized. That's a good time to make the application's main window visible, which the handler does by calling the `show` command. In this application, the main window has the AppleScript name of `"main"`.

```
on launched theObject
    show window "main"
end launched
```

To specify window ordering, you use a statement like the following.

```
show window "main" in front of window "settings"
```

size to fit

Not supported (through AppleScript Studio version 1.2). Adjusts the size of the specified object to fit within its container.

Syntax

```
size to fit                reference                    required
```

Parameters

reference
a reference to the object to size

update

Updates display of the specified `window` or `view` object, causing it to be redrawn immediately.

Syntax

```
update                     reference                    required
```

Parameters

reference
a reference to the `window` (page 86) or `view` (page 226) object to update

Examples

The following is the `launched` handler, from the Browser sample application distributed with AppleScript Studio. This handler uses the Finder application to get a list of disk names for a `browser` object, sets a path separator for the browser, then uses the `update` command to update the browser's display.

```
on launched theObject
    tell application "Finder"
```

```
        set diskNames to name of every disk
    end tell

    set path separator of browser "browser" of window "main" to ":"

    tell browser "browser" of window "main" to update
end launched
```

Events

Objects based on classes in the Application suite support handlers for the following events (an event is an action, typically generated through interaction with an application's user interface, that causes a handler for the appropriate object to be executed). To determine which classes support which events, see the individual class descriptions.

- `activated` (page 130)
- `awake from nib` (page 130)
- `became key` (page 135)
- `became main` (page 136)
- `closed` (page 136)
- `deminiaturized` (page 137)
- `document nib name` (page 138)
- `exposed` (page 139)
- `idle` (page 139)
- `keyboard down` (page 141)
- `keyboard up` (page 141)
- `launched` (page 142)
- `miniaturized` (page 143)
- `mouse down` (page 144)
- `mouse dragged` (page 145)
- `mouse entered` (page 146)
- `mouse exited` (page 146)
- `mouse moved` (page 147)
- `mouse up` (page 148)
- `moved` (page 149)
- `opened` (page 150)
- `open untitled` (page 150)

- `resigned active` (page 151)
- `resigned key` (page 152)
- `resigned main` (page 153)
- `resized` (page 153)
- `right mouse down` (page 154)
- `right mouse dragged` (page 155)
- `right mouse up` (page 156)
- `scroll wheel` (page 156)
- `should close` (page 157)
- `should open` (page 158)
- `should open untitled` (page 159)
- `should quit` (page 160)
- `should quit after last window closed` (page 161)
- `should zoom` (page 162)
- `shown` (page 163)
- `updated` (page 163)
- `was hidden` (page 164)
- `was miniaturized` (page 165)
- `will become active` (page 165)
- `will close` (page 166)
- `will finish launching` (page 167)
- `will hide` (page 168)
- `will miniaturize` (page 169)
- `will move` (page 170)
- `will open` (page 170)
- `will quit` (page 171)
- `will resign active` (page 172)

- `will resize` (page 172)
- `will show` (page 174)
- `will zoom` (page 174)
- `zoomed` (page 176)

activated

Called after an `application` object has been activated. The handler can perform any operations needed on activation.

The startup-time calling order for application event handlers, including the `activated` handler, is listed in the description for the `awake from nib` (page 130) event handler.

Syntax

`activated` *reference* required

Parameters

reference
a reference to the `application` (page 43) object that was activated

Examples

When you connect an `activated` handler to an `application` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use this handler to perform any actions needed on activation, such as checking the state of items displayed in application windows.

```
on activated theObject
    (* Add script statements here to handle activation. *)
end activated
```

awake from nib

Called after an object is unarchived from its nib, which includes instantiating the object and restoring its values, including relationships to other objects in the nib. Archiving is the process of creating a detailed record of a collection of interrelated

objects and values, from which you can recreate the original collection (by unarchiving). For more information on archiving, see the Cocoa Programming Topic Archiving and Serialization.

A nib file is an archive of objects and connections created in Interface Builder. In the `awake from nib` handler, an object can perform any custom initialization, at a point when all objects in the nib have been unarchived and connected, but before the interface is made visible to the user. When a nib object is loaded, AppleScript Studio calls the `awake from nib` event handler for every object in the nib that attaches that handler.

All classes that inherit from `responder` (page 80), which includes many AppleScript Studio classes, theoretically support the `awake from nib` handler. However, in practice, `awake from nib` is only supported for a class if you can access that class in Interface Builder and connect the handler. To examine (or connect) the available handlers for a class in Interface builder, select an instance of an object of that type in the Instances pane of the nib window, then open the AppleScript pane of the Info window.

Figure 3-9 shows the AppleScript pane for a File's Owner that represents an `application` (page 43) object. (See the Discussion section below for more information on File's Owner objects.) This instance has one handler connected, the `should quit after last window closed` handler. The actual handler is in the project file `Application.applescript`.

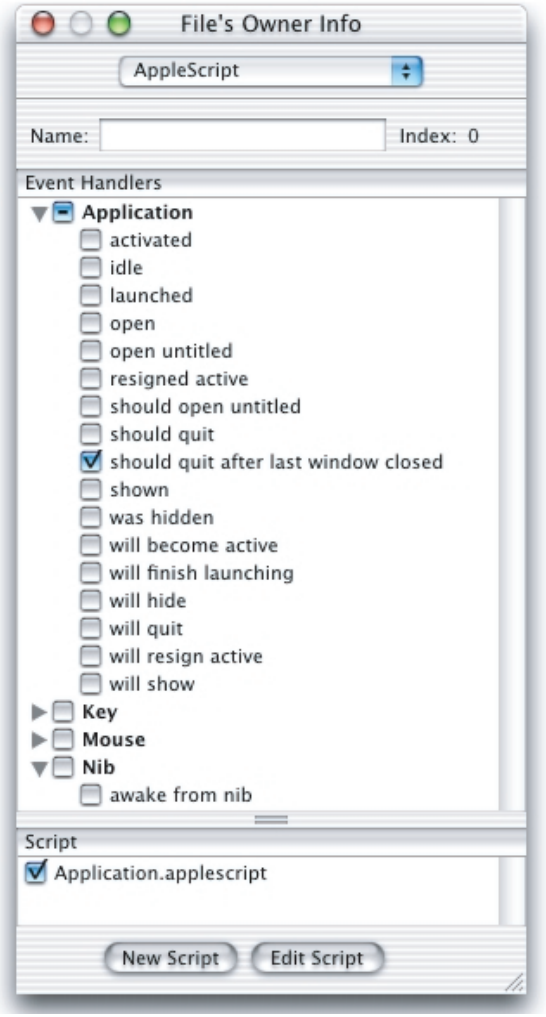
At application startup time, handlers connected to the `application` object are called in this order:

1. `will finish launching` (page 167)
2. `awake from nib` (page 130)
3. `launched` (page 142)
4. `will become active` (page 165)
5. `activated` (page 130)
6. `idle` (page 139)

Before any handlers are called for the `application` object (with the exception of the `will finish launching` handler, which is always called first, if present), the application's main nib file is loaded, all its objects unarchived, and `awake from nib` called for any objects that have that handler connected. So an `awake from nib`

handler connected to an object in the application's main nib file, such as the application's main window, will be called before any handlers connected to the `application` object itself.

Figure 3-9 The Info window in Interface Builder, showing information for an application's File's Owner instance



Syntax

`awake from nib` *reference* required

Parameters

reference
a reference to the object that was unarchived

Examples

When you connect an `awake from nib` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on awake from nib theObject
    (* Perform operations here after awaking from nib. *)
end awake from nib
```

You might connect the `awake from nib` handler to a window object and use it to make the window visible:

```
on awake from nib theObject
    set visible of theObject to true
end awake from nib
```

See the Examples section for the `default entry` (page 58) class for an `awake from nib` handler that creates a default entry.

Discussion

Interface Builder is Apple's graphical interface builder for Mac OS X. You use Interface Builder to lay out interface objects (including windows, controls, menus, and so on), resize them, set their attributes, and make connections to other objects. The resulting information is stored (or archived) in user interface resources, called nibs, which in turn are stored in nib files that become part of your application. (A nib file is an Interface Builder file—the "ib" in "nib" stands for Interface Builder.)

When the application is opened, it creates an interface containing the windows, buttons, and other user interface objects specified in its main nib file. An application contains a main nib file that is opened when the application is launched. It can also contain additional nib files and load them when needed, such as to create instances of windows. For related information, see `load nib` (page 113), and also the nib information in the description for the `document` (page 432) class.

When a nib is unarchived, it can restore connections among objects that were archived in the nib, but not to objects outside the archive. For that reason, an application must supply a File's Owner object in Interface Builder for each nib. For

the main nib, shown in Figure 3-1 (page 44), the File's Owner is created automatically and refers to the `application` (page 43) object. In a document-based AppleScript Studio application, the File's Owner for the `Document.nib` file is also created automatically, and refers to the `document` (page 432) object.

In Interface Builder, you can examine the class for a File's Owner object by selecting the object instance in the Instances pane of the nib window, opening the Info window, and using the pop-up menu to display the Custom Class pane. For example, you'll see that the class of File's Owner is NSApplication for an `application` object, but NSDocument for a `document` object. You can set the class of the File's Owner to another Cocoa class or to a custom class you have created, though most AppleScript Studio applications will not need to do so.

Version Notes

The `awake from nib` event handler was added in AppleScript Studio version 1.1.

If you are working with the version of Interface Builder distributed with Mac OS X version 10.2, see "Version Information" (page 22) for information on setting a Nib File Compatibility preference.

`became key`

Called after a window object has become the key window (or first recipient for keystrokes). See also `became main` (page 136) and `resigned key` (page 152). For related information, see the Cocoa Programming Topic Basic Event Handling

Syntax

`became key`	*reference*	required

Parameters

reference
a reference to the `window` (page 86) object that became key

Examples

When you connect a `became key` handler to a `window` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary when the window becomes key (or first recipient for keystrokes).

```
on became key theObject
    (* Perform operations here after becoming key. *)
end became key
```

became main

Called when a `window` object just became the main window—that is, the front window and principal focus of user action. The main window is not necessarily the key window. See also `became key` (page 135) and `resigned main` (page 153).

Syntax

`became main`	*reference*	required

Parameters

reference
a reference to the `window` (page 86) object that became main

Examples

When you connect a `became main` handler to a `window` (page 86) object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary upon becoming main.

```
on became main theObject
    (* Perform operations here after becoming main. *)
end became main
```

closed

Called after a `drawer` object closes. At this point the handler can perform any operations that should take place after the `drawer` (page 201) is closed.

Syntax

`closed`	*reference*	required

Parameters

reference
a reference to the `drawer` object that was closed

Examples

The following example of a `closed` handler is from the Drawer sample application distributed with AppleScript Studio.

```
on closed theObject
    set contents of text field "Date Field" of drawer "Drawer"
        of window "main" to "closed"
end closed
```

Because `theObject` is a reference to the object that was closed (the drawer), the following script statement is equivalent to the one shown above:

```
set contents of text field "Date Field" of theObject to "closed"
```

deminiaturized

Called after a window has been restored from its miniaturized state. The handler can perform any operations needed on deminiaturization (synonymous with minimization, or being put in the Dock) of the `window` (page 86).

Syntax

`deminiaturized`	*reference*	required

Parameters

reference
a reference to the `window` object that was deminiaturized

Examples

When you connect a `deminiaturized` handler to a `window` (page 86) object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary upon becoming deminiaturized, such as setting the `minimized title` property of the window.

```
on deminiaturized theObject
    (* Add script statements here to handle deminiaturizing. *)
end deminiaturized
```

document nib name

Returns the name of the document nib file. Your application doesn't need to attach this handler if its document nib is named `document.nib`. If you change the name of your document nib, you should add this handler to the application object and return the name of the document nib, without the ".nib" extension.

Syntax

`document nib name`	*reference*	required
`for document`	*document*	required

Parameters

reference
a reference to the `application` (page 43) object

`for document` `document` (page 432)
the document to obtain the nib file name for

Examples

When you connect a `document nib name` handler to an `application` object in Interface Builder, AppleScript Studio supplies an empty handler template. See the description for the `application` (page 43) class for information on how to connect an application handler.

You can use the `theObject` parameter to access properties or elements of the application and the `theDocument` parameter to access properties or elements of the document. Your handler should return the name for your document nib. For example, if your document nib file is named "MyDocument.nib", your handler might look something like the following:

```
on document nib name theObject for document theDocument
    (* If necessary, statements to determine name of document nib file. *)
    return "MyDocument"
end document nib name
```

Version Notes

The `document nib name` event handler was added in AppleScript Studio version 1.2.

exposed

Not supported (through AppleScript Studio version 1.2). Called after a window object has been exposed to view. The handler can perform any operations needed when the `window` (page 86) is exposed.

Syntax

`exposed`	*reference*	required

Parameters

reference
a reference to the `window` object that was exposed

idle

Called at specific intervals, as requested by the application. You typically use an `idle` handler to perform lengthy or recurring operations that take place outside the main flow of your application.

You connect an `idle` handler to the `application` object. See the description for the `application` (page 43) class for information on how to connect an application handler. The startup-time calling order for application event handlers, including the `idle` handler, is listed in the description for the `awake from nib` (page 130) event

handler. That order specifies when the `idle` handler is first called. The `idle` handler returns the number of seconds the application should wait before calling the handler again.

Syntax

`idle` *reference* required

Parameters

reference
a reference to the `application` (page 43) object whose `idle` handler is called

Result

integer
the number of seconds to delay before the next call to `idle`; to ensure that `idle` will be called again, always return a value of 1 or greater

Examples

When you connect an `idle` handler to an `application` object Interface Builder, AppleScript Studio supplies an empty handler template. You can use the handler to perform any required idle-time operations. Your handler should return the number of seconds to wait before calling the `idle` handler again.

```
on idle theObject
    (* Add script statements here to perform idle operations. *)
    return 1 -- call handler again in one second
end idle
```

AppleScript provides constants for the number of seconds in a minute, minutes in an hour, and so on. So to cause the handler to be called every five minutes, you can use `return 5 * minutes`. The constants `minutes`, `hours`, `days`, and `weeks` are described in *AppleScript Language Guide*, available in Project Builder Help and at Apple's Developer website.

Discussion

At application startup time, an `idle` handler will not be called for the first time until after certain other application handlers have been called, as listed in the description for the `awake from nib` (page 130) event handler.

keyboard down

Called when a key is pressed.

See the responder (page 80) class for information about how an application handles mouse and key events.

Syntax

keyboard down	*reference*	required
event	*event*	optional

Parameters

reference
a reference to the object whose keyboard down handler is called

event event (page 62)
the event information for the key down event

Examples

When you connect a keyboard down handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on keyboard down theObject event theEvent
    (* Add script statements here to handle the key down event. *)
end keyboard down
```

You can use the theEvent parameter to get information about the keyboard down event, such as the character(s), and whether the Command, Option, Shift, or Control keys were pressed. See the event (page 62) class for examples.

keyboard up

Called when a key is released.

Syntax

`keyboard up`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `keyboard up` handler is called

`event` `event` (page 62)
the event information for the key up event

Examples

When you connect a `keyboard up` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on keyboard up theObject event theEvent
    (* Add script statements here to handle the key up event. *)
end keyboard up
```

You can use the `theEvent` parameter to get information about the `keyboard up` event, such as the character(s), and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

Discussion

Due to terminology conflicts, this handler could not be named `key up`.

launched

Called after the application has been launched. You can only connect a `launched` handler to the `application` (page 43) object. The handler can perform any operations needed on launch.

The startup-time calling order for application handlers, including the `launched` handler, is listed in the description for the `awake from nib` (page 130) event handler.

Syntax

`launched`	*reference*	required

Parameters

reference
a reference to the `application` object that was launched

Examples

When you connect a `launched` handler to an `application` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary after being launched. For example, the Drawer sample application, distributed with AppleScript Studio, uses a `launched` handler to show its main window:

```
on launched theObject
    show window "main"
end launched
```

The Drawer application sets the `visible` property of its main `window` (page 86) to `false` in Interface Builder (in the Attributes pane of the Info window). It then sets up its user interface in an `awake from nib` (page 130) event handler, and finally shows the window in the `launched` handler. See the Discussion section for the `awake from nib` handler for information on the order in which event handlers are called on application start up.

For another example of a `launched` handler, see the Examples section for the `application` (page 43) class.

`miniaturized`

Called after a window has been changed to its miniaturized state. The handler can perform any operations needed when the object is miniaturized.

You should use `miniaturized`, rather than `was miniaturized` (page 165).

Syntax

`miniaturized`	*reference*	required

Parameters

reference
a reference to the `window` (page 86) object that was miniaturized

Examples

When you connect a `miniaturized` handler to a `window` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary after being miniaturized.

```
on miniaturized theObject
    (* Add script statements here to deal with miniaturizing. *)
end miniaturized
```

mouse down

Called when a mouse down event occurs.

Syntax

`mouse down`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `mouse down` handler is called

`event event` (page 62)
the event information for the mouse down event

Examples

When you connect a `mouse down` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on mouse down theObject event theEvent
```

```
    (* Add script statements here to handle the mouse down event. *)
end mouse down
```

You can use the `theEvent` parameter to get information about the `mouse down` event, such as the cursor location, click count, and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

mouse dragged

Called when a mouse dragged event occurs.

Syntax

`mouse dragged`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `mouse dragged` handler is called

`event` `event` (page 62)
the event information for the mouse dragged event

Examples

When you connect a `mouse dragged` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on mouse dragged theObject event theEvent
    (* Add script statements here to handle the mouse dragged event. *)
end mouse dragged
```

You can use the `theEvent` parameter to get information about the `mouse dragged` event, such as the cursor location, and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

mouse entered

Called when a mouse entered event occurs. That is, the cursor has entered the bounds of the object that is connected to the event handler. The many classes that inherit from the `control` (page 270)and `view` (page 226) classes support the `mouse entered` handler.

Syntax

`mouse entered`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `mouse entered` handler is called

`event event` (page 62)
the event information for the mouse entered event

Examples

When you connect a `mouse entered` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on mouse entered theObject event theEvent
    (* Add script statements here to handle the mouse entered event. *)
end mouse entered
```

You can use the `theEvent` parameter to get information about the `mouse entered` event, such as the cursor location, and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

mouse exited

Called when a mouse exited event occurs. That is, the cursor has exited the bounds of the object that is connected to the event handler. The many classes that inherit from the `control` (page 270)and `view` (page 226) classes support the `mouse exited` handler.

Syntax

`mouse exited`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `mouse exited` handler is called

`event event` (page 62)
the event information for the mouse exited event

Examples

When you connect a `mouse exited` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the `theEvent` parameter to get information about the mouse exited event, such as the location, and whether the Command, Option, Shift, or Control keys were pressed.

```
on mouse exited theObject event theEvent
    (* Add script statements here to handle the mouse exited event. *)
end mouse exited
```

You can use the `theEvent` parameter to get information about the `mouse exited` event, such as the cursor location, and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

mouse moved

Called when a mouse moved inside the bounds of the object. That is, the cursor has moved within the bounds of the object that is connected to the event handler. The many classes that inherit from the `control` (page 270)and `view` (page 226) classes support the `mouse moved` handler.

Syntax

`mouse moved`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `mouse moved` handler is called

`event event` (page 62)
the event information for the mouse moved event

Examples

When you connect a `mouse moved` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on mouse moved theObject event theEvent
    (* Add script statements here to handle the mouse moved event. *)
end mouse moved
```

You can use the `theEvent` parameter to get information about the `mouse moved` event, such as the cursor location, and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

`mouse up`

Called when a mouse up event occurs.

Syntax

`mouse up`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `mouse up` handler is called

`event event` (page 62)
the event information for the mouse up event

Examples

When you connect a `mouse up` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on mouse up theObject event theEvent
    (* Add script statements here to handle the mouse up event. *)
end mouse up
```

You can use the `theEvent` parameter to get information about the `mouse up` event, such as the cursor location, click count, and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

moved

Called after the object is moved.

Syntax

`moved`	*reference*	required

Parameters

reference
a reference to the object that moved

Examples

When you connect a `moved` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. The `theObject` parameter refers to the object that moved, typically a `view` (page 226) or subclass of a view. You can use the handler to perform any operations necessary after being moved. To determine how much the object has moved, you will have to store the old location of the object and compare it to the current location.

```
on moved theObject
    (* Add script statements here to handle operations after a move. *)
end moved
```

opened

Called after an object that supports this handler (such as a window, panel, or document) opens. At this point the handler can perform any operations that should take place after the object is open.

Syntax

`opened`	*reference*	required

Parameters

reference
a reference to the object that was opened

Examples

The following example of an `opened` handler is from the Drawer sample application distributed with AppleScript Studio.

```
on opened theObject
    set contents of text field "Date Field" of drawer "Drawer"
        of window "main" to "opened"
end opened
```

Because `theObject` is a reference to the object that was opened (the drawer), the following script statement is equivalent to the one shown above:

```
set contents of text field "Date Field" of theObject to "opened"
```

open untitled

Called when the application is about to open an untitled document. This handler is called only for a document-based application and only for the first document when the application is launched. It is called after `should open untitled` (page 159), and can prepare for the untitled window to be opened.

Syntax

`open untitled`	*reference*	required

Parameters

reference
a reference to the `application` (page 43) that is opening an untitled `document` (page 432)

Examples

You install an `open untitled` handler in Interface Builder by selecting the File's Owner instance in the application's MainMenu.nib window, then selecting `open untitled` in the Application handlers in the Info window. When you select a project script file, AppleScript Studio inserts a handler template like the one shown below.

The File's Owner instance in the main nib window represents `NSApp`, a global constant that references the NSApplication object for the application. (In a document nib file, the File's Owner instance typically represents the document.)

The `theObject` parameter of the `open untitled` handler refers to the application object for which an untitled document is about to be opened. You can use the parameter to access properties or elements of the application to prepare for opening the document.

```
on open untitled theObject
    (* Perform operations here before opening an untitled document. *)
end open untitled
```

resigned active

Called after the application object has resigned its active state. There is currently no event handler that allows the application to refuse to resign its active state. See also `will resign active` (page 172).

Syntax

`resigned active`	*reference*	required

Parameters

reference
a reference to the `application` (page 43) object that resigned its active state

Examples

When you connect a `resigned active` handler to an `application` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary upon resigning as the active object.

```
on resigned active theObject
    (* Perform operations here after resigning active state. *)
end resigned active
```

resigned key

Called after a window object has resigned its key state (as first recipient for keystrokes). See also `became key` (page 135) and `will resign active` (page 172).

Syntax

`resigned key`	*reference*	required

Parameters

reference
a reference to the `window` (page 86) object that resigned its key state

Examples

When you connect a `resigned key` handler to a `window` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary upon resigning as the key window.

```
on resigned key theObject
```

```
    (* Perform operations here after resigning key state. *)
end resigned key
```

resigned main

Called after a window object has resigned its main state (as the front window and principal focus of user action). An `window` (page 86) object may be main without being key (the first recipient for keystrokes). See also `became main` (page 136), `became key` (page 135), and `resigned key` (page 152).

Syntax

`resigned main` *reference* required

Parameters

reference
a reference to the `window` object that resigned its main state

Examples

When you connect a `resigned main` handler to a `window` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary upon resigning as the main window.

```
on resigned main theObject
    (* Perform operations here after resigning main. *)
end resigned main
```

resized

Called after an object is resized.

Syntax

`resized`	*reference*	required

Parameters

reference
a reference to the object that was resized

Examples

When you connect a `resized` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary after being resized. To determine the change in size, you will have to store the old size of the object and compare it to the current size. To do that, you might save the size in a `will resize` (page 172) handler, which is called before the `resized` handler.

```
on resized theObject
    (* Perform operations here after resizing the object. *)
end resized
```

right mouse down

Called when a right mouse down event occurs.

Syntax

`right mouse down`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `right mouse down` handler is called

`event event` (page 62)
the event information for the right mouse down event

Examples

When you connect a `right mouse down` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on right mouse down theObject event theEvent
    (* Add script statements here to handle the right mouse down event. *)
end right mouse down
```

You can use the `theEvent` parameter to get information about the `right mouse down` event, such as the cursor location, click count, and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

`right mouse dragged`

Called when a right mouse dragged event occurs.

Syntax

`right mouse dragged`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `right mouse dragged` handler is called

`event` `event` (page 62)
the event information for the `right mouse dragged` event

Examples

When you connect a `right mouse dragged` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on right mouse dragged theObject event theEvent
    (* Add script statements here to handle the right mouse dragged event. *)
end right mouse dragged
```

You can use the `theEvent` parameter to get information about the `right mouse dragged` event, such as the cursor location, and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

right mouse up

Called when a right mouse up event occurs.

Syntax

`right mouse up`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `right mouse up` handler is called

`event` `event` (page 62)
the event information for the right mouse up event

Examples

When you connect a `right mouse up` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following.

```
on right mouse up theObject event theEvent
    (* Add script statements here to handle the right mouse up event. *)
end right mouse up
```

You can use the `theEvent` parameter to get information about the `right mouse up` event, such as the cursor location, and whether the Command, Option, Shift, or Control keys were pressed. See the `event` (page 62) class for examples.

scroll wheel

Called when the scroll wheel moves.

Syntax

`scroll wheel`	*reference*	required
`event`	*event*	optional

Parameters

reference
a reference to the object whose `scroll wheel` handler is called

`event event` (page 62)
the event information for the scroll wheel event

Examples

The following `scroll wheel` handler responds to a scroll wheel by incrementing or decrementing a value in a `text field` (page 314) based on the passed `event` (page 62) parameter. The `awake from nib` (page 130) handler initializes the text field to a starting value of 100.

```
on scroll wheel theObject event theEvent
    set theValue to content of theObject as number
    set theValue to theValue + (delta y of theEvent)
    set content of theObject to theValue
end scroll wheel

on awake from nib theObject
    set content of theObject to 100
end awake from nib
```

should close

Called before an object that supports this handler closes. That includes classes such as `window` (page 86), `panel` (page 487), and `drawer` (page 201). The handler can return `false` to cancel the close operation.

Syntax

`should close`	*reference*	required

Parameters

reference
a reference to the `window` (page 86) or `panel` (page 487) object that may close

Result

boolean
Return `true` to allow closing; `false` to disallow it

Examples

When you connect a `should close` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template. Your handler should determine whether to allow the object to close, then return the appropriate value.

```
on should close theObject
    set allowClose to false
    --Check variable, perform test, or call handler to see if OK to close
    -- If so, set allowClose to true
    return allowClose
end should close
```

should open

Called before an object that supports this handler (such as a window, panel, or document) opens. The handler can return `false` to cancel the open.

Syntax

`should open`	*reference*	required

Parameters

reference
a reference to the object that may open

Result

boolean

Return `true` to allow opening; `false` to disallow it

Examples

When you connect a `should open` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template. Your handler should determine whether to allow the object to open, then return the appropriate value.

```
on should open theObject
    set allowOpen to false
    --Check variable, perform test, or call handler to see if OK to open
    -- If so, set allowOpen to true
    return allowOpen
end should open
```

should open untitled

Called before an object that supports this handler (typically the application object) opens an untitled window or document. The handler can return `false` to cancel the operation.

Syntax

`should open untitled` *reference* required

Parameters

reference

a reference to the object (typically the `application` (page 43)) that has the option of opening an untitled document or window

Result

boolean

Return `true` to allow opening an untitled document; `false` to disallow it

Examples

The following example of a `should open untitled` handler checks a global property `allowUntitled`, which is set elsewhere in the application, to determine whether to allow the object to open as untitled. You might instead call a separate handler you've written to determines whether to allow opening.

```
on should open untitled theObject
    if allowUntitled is equal to true then
        return true
    else
        return false
    end if
end should open untitled
```

should quit

Called to determine if the application should quit. The handler can return `false` to refuse to quit.

Syntax

`should quit`	*reference*	required

Parameters

reference
a reference to the `application` (page 43) object that may quit

Result

boolean
Return `true` to allow quitting; `false` to disallow it

Examples

The following example of a `should quit` handler calls an application handler `allowQuitting`, written by you, to determine whether to allow the application to quit, then returns the appropriate value. You might instead perform validation in the handler itself or check some global property.

```
on should quit theObject
```

```
    --Check property, perform test, or call handler to see if OK
    --to quit
    set allowQuit to allowQuitting(theObject)
    return allowQuit
end should quit
```

should quit after last window closed

Called to determine whether the application should quit when its last window is closed. The handler can return `false` to refuse to quit.

Syntax

`should quit after last window closed` *reference* required

Parameters

reference
a reference to the `application` (page 43) object that may quit when its last window is closed

Result

boolean
Return `true` to quit after last window is closed; `false` to disallow quitting

Examples

The following example of a `should quit after last window closed` handler calls an application handler `shouldQuit`, written by you, to determine whether to allow the application to quit, then returns the appropriate value. You might instead perform validation in the handler itself or check some global property.

```
on should quit after last window closed theObject
    --Check property, perform test, or call handler to see if OK
    --to quit after last window closed
    set allowQuit to shouldQuit(theObject)
    return allowQuit
end should quit after last window closed
```

should zoom

Called to determine if a window should be zoomed. The handler can examine the proposed bounds of the zoom, and can return `false` to refuse to zoom or `true` to allow the zoom. If you do not supply a handler, or if you supply a handler but do not return a value, by default zooming will be allowed.

If you wish to have control over the bounding rectangle for the zoom operation, use the `will zoom` (page 174) event handler. If you want access to the proposed bounds in the `will zoom` handler, you will have to implement `should zoom` and save the value of the `proposed bounds` parameter for later use by the `will zoom` handler.

Syntax

`should zoom`	*reference*	required
`proposed bounds`	*bounding rectangle*	optional

Parameters

reference
a reference to the `window` (page 86) object that may zoom

`proposed bounds` *bounding rectangle*
the proposed bounds of the object to be zoomed; a list of four numbers {left, bottom, right, top}; see the `bounds` property of the `window` (page 86) class for information on the coordinate system

Result

boolean
Return `false` to refuse to zoom or `true` to allow the zoom. If you implement this handler, you should always return a boolean value.

Examples

The following example of a `should zoom` event handler calls an application handler `isZoomable`, written by you, to determine whether to allow zooming, then returns the appropriate value. You might instead perform some kind of test in the `should zoom` handler itself, or check a global property. If you want access to the proposed bounds in the `will zoom` (page 174) handler, you should use this handler to save the value of the `proposed bounds` parameter for later use.

```
on should zoom theObject
    --Check property, perform test, or call handler to see if OK to edit
    set allowZooming to isZoomable(theObject)
    return allowZooming
end should zoom
```

Version Notes

The `proposed bounds` parameter was added in AppleScript Studio version 1.2.

In the dictionary file (`AppleScriptKit.asdictionary`) for AppleScript Studio version 1.2, the parameter type for the `proposed bounds` parameter is shown as `point`. The correct type is `bounding rectangle`, a list of four integer values {left, bottom, right, top}.

shown

Called after an application object is shown.

Syntax

`shown`	*reference*	required

Parameters

reference
a reference to the `application` (page 43) object that was shown

Examples

When you connect a `shown` handler to an `application` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary after being shown.

```
on shown theObject
    (* Perform operations here after the object is shown. *)
end shown
```

updated

Called after an object is updated.

Syntax

`updated`	*reference*	required

Parameters

reference
a reference to the object that was updated

Examples

When you connect an `updated` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary after being updated.

```
on updated theObject
    (* Perform operations here after the object is updated. *)
end updated
```

was hidden

Called after the application object is hidden as a result of a user choosing Hide from the application menu or pressing Command-h.

You cannot connect this event handler to a `window` (page 86) object, though you can use the `hide` (page 107) command to hide a window. (Your application can call commands such as `hide` explicitly, but when you connect event handlers such as `was hidden` to objects in Interface Builder, the handlers are called by AppleScript Studio at the appropriate times.)

Syntax

`was hidden`	*reference*	required

Parameters

reference
a reference to the `application` (page 43) object that was hidden

Examples

When you connect a `was hidden` handler yo the `application` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to perform any operations necessary after being hidden.

```
on was hidden theObject
    (* Perform operations here after the object was hidden. *)
end was hidden
```

was miniaturized

Called after a window has been changed to its miniaturized state. The handler can perform any operations needed on miniaturization.

You should use `miniaturized` (page 143), rather than `was miniaturized`.

Syntax

`was miniaturized` *reference* required

Parameters

reference
a reference to the object that was miniaturized

will become active

Called when the object is about to become active. The handler cannot cancel activation, but can prepare for it.

The startup-time calling order for application event handlers, including the `will become active` handler, is listed in the description for the `awake from nib` (page 130) event handler.

Syntax

`will become active`	*reference*	required

Parameters

reference
a reference to the object that will become active

Examples

When you connect a `will become active` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for activation.

```
on will become active theObject
    (* Perform operations here before becoming active. *)
end will become active
```

will close

Called when the object is about to close. The handler cannot cancel the close operation, but can prepare for it. Classes such as `window` (page 86), `panel` (page 487), and `drawer` (page 201) support the `will close` handler.

Syntax

`will close`	*reference*	required

Parameters

reference
a reference to the `window`, `drawer`, or other object that is about to close

Examples

When you connect a `will close` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for closing. For example, you might want to update a controller object with the current information from a window's `text field`

(page 314) objects. Note that you should perform text field validation earlier, in the `should close` (page 157) handler, where the window can refuse to close if a field contains invalid data.

```
on will close theObject
    (* Perform operations here before closing. *)
end will close
```

will finish launching

Called when the application object is about to finish launching. The handler cannot cancel launching, but can prepare for it.

The startup-time calling order for application event handlers, including the `will finish launching` handler, is listed in the description for the `awake from nib` (page 130) event handler.

Syntax

`will finish launching` *reference* required

Parameters

reference
a reference to the `application` (page 43) object that will finish launching

Examples

When you connect a `will finish launching` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template. You can use the handler to perform operations after your main nib file has been loaded, but before the application finishes launching. For example, the following script makes the window from the main nib visible.

```
on will finish launching theObject
    (* Perform operations here before completion of launching. *)
    set visible of (window of theObject) to true
end will finish launching
```

You might also use the `will finish launching` handler to check for the presence of the minimum version of the AppleScript Studio runtime required by your application. (For information on runtime versions, see "Version Information" (page 22).)

The next listing shows a simple example of how the handler might check for the required version of AppleScript Studio. If the version is available, the handler registers a drag type, which is only supported starting with AppleScript Studio version 1.2. If version 1.2 is not available, the application puts up a message and then quits. Note that the handler doesn't check AppleScript Studio's version number directly. Instead, it checks for the corresponding AppleScript version, as shown in Table 2-1 (page 23).

```
on will finish launching theObject
    if AppleScript's version as string ≥ "1.9" then
        tell window 1 to register drag types {"file names"}
    else
        display dialog "This application requires AppleScript Studio 1.2 or later."
        quit
    end if
end will finish launching
```

will hide

Called when the `application` object is about to be hidden as a result of a user choosing Hide from the application menu or pressing Command-h. The handler cannot cancel hiding, but can prepare for it.

You cannot connect this event handler to a `window` (page 86) object, though you can use the `hide` (page 107) command to hide a window. (Your application can call commands such as `hide` explicitly, but when you connect event handlers such as `will hide` to objects, the handlers are called by AppleScript Studio at the appropriate times.)

Syntax

`will hide`	*reference*	required

Parameters

reference
a reference to the `application` (page 43) object

Examples

When you connect a `will hide` handler to an `application` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for being hidden.

```
on will hide theObject
    (* Perform operations here before the application hides. *)
end will hide
```

will miniaturize

Called when an object that supports this handler (such as a window or panel) is about to miniaturize. The handler cannot cancel miniaturizing, but it can prepare for it.

Syntax

`will miniaturize`	*reference*	required

Parameters

reference
a reference to the object that will miniaturize

Examples

When you connect a `will miniaturize` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for being miniaturized.

```
on will miniaturize theObject
    (* Perform operations here before the object miniaturizes. *)
```

```
end will miniaturize
```

will move

Called when an object is about to move. The handler cannot cancel the move operation, but can prepare for it.

Syntax

```
will move                reference                    required
```

Parameters

reference
a reference to the object that will move

Examples

When you connect a `will move` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for being moved.

```
on will move theObject
    (* Perform operations here before the object moves. *)
end will move
```

will open

Called when an object that supports this handler (such as a window or panel is about to open. The handler cannot cancel the open operation, but can prepare for it.

Syntax

```
will open                reference                    required
```

Parameters

reference
a reference to the object that will open, such as a `window` (page 86) or `panel` (page 487)

Examples

When you connect a `will open` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for being opened.

```
on will open theObject
    (* Perform operations here before the object opens. *)
end will open
```

Version Notes

Prior to AppleScript Studio version 1.2, you could connect a `will open` handler to a `document` (page 432) object.

Prior to AppleScript Studio version 1.2, a `will open` handler was called only once, when a window was loaded from its nib, not when it was actually opened. To simulate that behavior in version 1.2, you can replace calls to `will open` with calls to `awake from nib` (page 130).

will quit

Called when the application object is about to quit. The handler cannot cancel quitting, but can prepare for it.

Syntax

`will quit`	*reference*	required

Parameters

reference
a reference to the `application` (page 43) object

Examples

When you connect a `will quit` handler to an `application` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for quitting.

```
on will quit theObject
    (* Perform operations here before the application quits. *)
```

```
end will quit
```

will resign active

Called when an object is about to resign its active state. The handler cannot cancel resigning its active state, but can prepare for it. There is currently no event handler that allows the application to refuse to resign its active state

Syntax

```
will resign active      reference                    required
```

Parameters

reference
a reference to the `application` (page 43) object that will resign its active state

Examples

When you connect a `will resign active` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for resigning active state.

```
on will resign active theObject
    (* Perform operations before the object resigns its active state. *)
end will resign active
```

will resize

Called when a window object is about to be resized. The handler cannot cancel the resize operation, but can prepare for it, and can return a different size to specify the new size.

Syntax

`will resize`	*reference*	required
`proposed size`	*point*	optional

Parameters

reference
a reference to the `window` (page 86) object that will be resized

`proposed size` *point*
the proposed size of the window, consisting of a two-item list of numbers {horizontal, vertical}; see the `bounds` property of the `window` (page 86) class for information on the coordinate system

Result

point
You can return a different point to specify the size the window will be resized to. If you do not return a point, the value that was passed in the `proposed size` parameter is used.

Examples

When you connect a `will resize` handler to a window object in Interface Builder, AppleScript Studio supplies an empty handler template. You can use the handler to prepare for resizing, and can return a size value to specify the new size. If you need to know the previous size in a `resized` (page 153) handler (which is called after resizing takes place), you can use your `will resize` handler to save the current size.

```
on will resize theObject
    (* Perform operations here before the object resizes.
        Return a point to specify a different size.
        For example: *)
    return {200, 560}
end will resize
```

Version Notes

The `proposed size` parameter was added in AppleScript Studio version 1.2.

will show

Called when an object is about to be shown. The handler cannot cancel being shown, but can prepare for it.

Syntax

`will show`	*reference*	required

Parameters

reference
a reference to the object to be shown

Examples

When you connect a `will show` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for being shown.

```
on will show theObject
    (* Perform operations here before the object is shown. *)
end will show
```

will zoom

Called when a `window` object is about to be zoomed. The handler cannot cancel being zoomed, but can prepare for it.

Syntax

`will zoom`	*reference*	required
`screen bounds`	*bounding rectangle*	optional

Parameters

reference
a reference to the `window` (page 86) object that will be zoomed

`screen bounds`
: a list of four numbers {left, bottom, right, top} that specifies the bounds of the screen that contains the largest part of the window (minus dock and menu bar, if any); note that this is not the proposed bounds (or bounds the window will be zoomed to)—see the `should zoom` (page 162) event handler for how to obtain that information; see the `bounds` property of the `window` (page 86) class for information on the coordinate system

Result

bounding rectangle
: You can return a bounding rectangle (a list of four numbers {left, bottom, right, top}) to specify the bounds of the zoomed window. If you do not return a rectangle, the value that was originally passed to the `should zoom` (page 162) event handler is used.

Examples

When you connect a `will zoom` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template. The `theObject` parameter refers to the object that will be zoomed. You can use the `screenBounds` parameter, as well as the bounds of the window itself, and possibly the proposed bounds for the zoom operation (if you saved them in the `should zoom` (page 162) event handler) to determine whether to modify the bounds to which the window will be zoomed.

The following handler changes the bounds for the zoomed window to 70 pixels from the left side of the screen. It does so by getting the current bounds of the window, setting its left item value to 70, and returning those bounds for the zoom.

```
on will zoom theObject screen bounds screenBounds
    (* Perform operations here before the object is zoomed. *)
    set theBounds to bounds of theObject
    set item 1 of theBounds to 70
    return theBounds
end will zoom
```

Version Notes

The `will zoom` event was added in AppleScript Studio version 1.2.

zoomed

Called after an object (typically a window or panel) was zoomed. The handler cannot cancel being zoomed, but can prepare for it.

Syntax

`zoomed`	*reference*	required

Parameters

reference
a reference to the object that was zoomed

Examples

When you connect a `zoomed` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for being zoomed.

```
on zoomed theObject
    (* Perform operations here after the object zooms. *)
end zoomed
```

Enumerations

The Application suite provides the following enumerations, which define named constants you can use in AppleScript Studio applications.

Important
These enumerations are available to classes in all AppleScript Studio suites.

- Sort Case Sensitivity (page 189)
- Sort Order (page 190)
- Sort Type (page 190)
- Tab State (page 190)
- Tab View Type (page 191)
- Text Alignment (page 191)
- Tick Mark Position (page 192)
- Title Position (page 192)

Alert Return Values

Not used (through AppleScript Studio version 1.2).

Enumerated Values

`alternate return`
The alternate button was returned

`default return`
The default button was returned

`error return`
An error was returned

`other return`
The other button was returned

Alert Type

Specifies a type of alert. You pass one of these values to the `display alert` (page 500) command.

Enumerated Values

`critical`
Critical alert

`informational`
Informational alert

`warning`
Warning alert

Bezel Style

Specifies the appearance of a border. The bezel style property for a `button` (page 246) has a value equal to one of these constants.

Enumerated Values

`circular bezel`
Circular bezel (for a round button)

`regular square bezel`
Regular square bezel (for a square button)

`rounded bezel`
Rounded bezel (for a rounded-edge button

`shadowless square bezel`
Square bezel without a shadow (square, non-rounded button)

`thick square bezel`
Thick square bezel (square button)

`thicker square bezel`
Square bezel with a thicker border (square button)

Border Type

Specifies a border type. Objects such as `box` (page 195) and `scroll view` (page 211) objects have border properties. Border type can interact with "Box Type" (page 180).

Enumerated Values

`bezel border`
Bezel border

`groove border`
Groove border (when combined with `separator` box type, provides a separator line)

`line border`
Line border (depending on box type, causes border on various sides)

`no border`
No border

Box Type

Specifies the type of a box. The box property for a `box` (page 195) has a value equal to one of these constants. The box type can interact with "Border Type" (page 179). You can find illustrations of many types of controls, including group boxes, in *Inside Mac OS X: Aqua Human Interface Guidelines*, available in Project Builder Help (choose Developer Help Center from the Help menu, then click the link for Developer Essentials).

Enumerated Values

`old style type`
Old style box (if border type is `bezel border`, shaded, with border on top and left; if border type is `line border`, not shaded, with border on all sides)

`primary type`
Primary box (terminology for a type of group box described in older guidelines; the Aqua human interface guidelines discourage the use of group boxes)

`secondary type`
Secondary box (terminology for a type of group box described in older guidelines; the Aqua human interface guidelines discourage the use of group boxes)

`separator type`
Separator box (when combined with `groove border` border type, provides a separator line)

Button Type

Specifies the type for a button, which can affect both how the button looks and how it behaves. The button type property for a `button` (page 246) has a value equal to one of these constants.

Enumerated Values

`momentary change button`
Momentary change button

`momentary light button`
Button that is momentarily lighted

`momentary push in button`
Momentary push in button

`on off button`
On off button

`push on off button`
Button that is pushed on and then pushed off

`radio button`
Radio button

`switch button`
Checkbox button

`toggle button`
Button that toggles its state

Cell Image Position

Specifies the position for an image. Objects such as `button` (page 246) and `cell` (page 256) have both image and image position properties.

Enumerated Values

`image above`
Image above

`image below`
Image below

`image left`
Image to the left

`image only`
Only show the image

`image overlaps`
Image overlaps

`image right`
Image to the right

`no image`
No image

Cell State Value

Specifies a cell's state. The `cell` (page 256) class provides both a state property and a "supports mixed state" property. Subclasses can support either two states (on state and off state) or three states (on state, off state, and mixed state). A mixed state

is useful for a checkbox or radio button that reflects the status of a feature that's true only for some items. For example, an italic checkbox would be on if all text in the current selection is italic, off if none is, and mixed if only certain characters are italic.

Enumerated Values

`mixed state`
Mixed state

`off state`
Off state

`on state`
On state

Cell Type

Specifies a cell type. The `cell` (page 256) class provides cell type property that has a different value for subclasses such as `image cell` (page 274) and `text field cell` (page 319).

Enumerated Values

`image cell type`
Cell that contains image

`null cell type`
Cell that is empty

`text cell type`
Cell that contains text

Color Panel Mode

A color selection mode specifies how a user can select a color in a color-panel. This mask is set before you initialize a new instance of `color-panel` (page 476). You can set the color mode property of a color panel object to any of the values in this enumeration.

Although color panels can support many color modes, when you get or set a color property of an AppleScript Studio object, the color is represented as an RGB value. The RGB value is accessible as a three-item list that contains the values for each component of the color. For example, blue can be represented as {0,0,65535}.

Enumerated Values

`cmyk mode`
CMYK color mode

`color list mode`
Color list color mode

`color wheel mode`
Color wheel mode

`custom palette mode`
Custom palette color mode

`gray mode`
Gray color mode

`hsb mode`
HSB color mode

`rgb mode`
RGB color mode

Control Size

Specifies the control size. Classes such as `cell` (page 256), `progress indicator` (page 296), and `tab view` (page 219) include a control size property that has a value equal to one of these constants. The small size, which you can set in Interface Builder, provides smaller versions of these user-interface items, suitable for smaller windows or panels.

Enumerated Values

`regular size`
Regular size

`small size`
Small size

Control Tint

Specifies the control tint. Classes such as `cell` (page 256), `progress indicator` (page 296), and `tab view` (page 219) include a control tint property that has a value equal to one of these constants. For example, the current tab is aqua when the tint is default tint, but clear for clear tint.

Enumerated Values

`clear tint`
Clear tint

`default tint`
Default tint

Drawer State

Specifies the state of a drawer. For more information, see the `drawer` (page 201) class.

Enumerated Values

`drawer closed`
Drawer is closed

`drawer closing`
Drawer is closing

`drawer opened`
Drawer is opened

`drawer opening`
Drawer is opening

Event Type

Specifies an event type. For more information, see the `event` (page 62) class.

Enumerated Values

`appkit defined type`
An event defined by the AppKit (Cocoa is made up of two frameworks, the AppKit and the Foundation frameworks)

`application defined type`
An event defined by the `application` (page 43)

`cursor update type`
Cursor update event

`flags changed type`
Modifier keys changed event

`key down type`
Key pressed event

`key up type`
Key released event

`left mouse down type`
Left mouse button pressed event

`left mouse dragged type`
Left mouse button dragged event

`left mouse up type`
Left mouse button released event

`mouse entered type`
Mouse entered event

`mouse exited type`
Mouse exited event

`mouse moved type`
Mouse moved event

`other mouse down type`
Other mouse button pressed event

`other mouse dragged type`
Other mouse button dragged event

`other mouse up type`
Other mouse button released event

`periodic type`
Periodic event, such as an idle event or timer event

`right mouse down type`
Right mouse button pressed event

`right mouse dragged type`
Right mouse button dragged event

`right mouse up type`
Right mouse button released event

`scroll wheel type`
Scroll wheel event

`system defined type`
An event defined by the System

Go To

Specifies a movie location. You pass one of the values defined in this enumeration with the `go` (page 323) command. For related information, see the `movie view` (page 286) class.

Enumerated Values

`beginning frame`
Beginning

`end frame`
End

`poster frame`
Poster frame (a frame that represents the movie's content)

Image Alignment

Specifies the alignment for an image. Classes such as `image cell` (page 274) and `image view` (page 276) include an image alignment property.

Enumerated Values

`bottom alignment`
Align to the bottom

`bottom left alignment`
Align to the bottom left

`bottom right alignment`
Align to the bottom right

`center alignment`
Align to the center

`left alignment`
Align to the left

`right alignment`
Align to the right

`top alignment`
Align to the top

`top left alignment`
Align to the top left

`top right alignment`
Align to the top right

Image Frame Style

Specifies the frame style for an image. Classes such as `image cell` (page 274) and `image view` (page 276) include a frame style property.

Enumerated Values

`button frame`
Button frame

`gray bezel frame`
Gray bezel frame

`groove frame`
Grooved frame

`no frame`
No frame

`photo frame`
Photo frame

Image Scaling

Specifies the scaling for an image. Classes such as `image cell` (page 274) and `image view` (page 276) include a scaling property.

Enumerated Values

`no scaling`
No scaling

`scale proportionally`
Scale proportionally

`scale to fit`
Scale to fit

Matrix Mode

Specifies the mode for a matrix. A matrix is a group of cells that work together in various ways, such as a group of radio buttons.

Enumerated Values

`highlight mode`
A cell is highlighted before it is asked to track the mouse, then unhilighted when it is done tracking.

`list mode`
Cells are highlighted, but don't track the mouse.

`radio mode`
Selects no more than one cell at a time. When a cell is selected, the previous cell is deselected.

`track mode`
Cells are asked to track the mouse whenever the mouse is inside their bounds. No highlighting is performed.

QuickTime Movie Loop Mode

Specifies the playback behavior for a movie. You can get or set the loop mode property of a `movie view` (page 286).

Enumerated Values

`looping back and forth playback`
Loop the movie from front to back (playback runs forwards and backwards between both end points)

`looping playback`
Loop the movie (restart playback at beginning when end is reached)

`normal playback`
Normal playback (playback stops when end is reached)

Rectangle Edge

Specifies the edge a drawer should open on or that or a pop-up menu should open on if space is restricted. You can use these enumerated values to set the edge property of a `drawer` (page 201), or the preferred edge property of a `popup button` (page 292).

Enumerated Values

`bottom edge`
bottom edge

`left edge`
Left edge

`right edge`
Right edge

`top edge`
Top edge

Scroll To Location

Specifies location to scroll to. Intended for use with `scroll` (page 328) command. However, that command is not supported, through AppleScript Studio version 1.2.

Enumerated Values

`bottom`
Bottom

`top`
Top

`visible`
Visible

Sort Case Sensitivity

Specifies whether a sort should consider case, as when sorting columns in a data source.

Enumerated Values

`case insensitive`
Sort the data using case insensitivity

`case sensitive`
Sort the data using case sensitivity

Version Notes

The `sort case sensitivity` enumeration was added in AppleScript Studio version 1.2.

Sort Order

Specifies the order in which to sort, as when sorting columns in a data source.

Enumerated Values

`ascending`
Sort the data in an ascending fashion

`descending`
Sort the data in a descending fashion

Version Notes

The `sort order enumeration` was added in AppleScript Studio version 1.2.

Sort Type

Specifies a sort type, as when sorting columns in a data source.

Enumerated Values

`alphabetical`
Sort the data alphabetically

`numerical`
Sort the data numerically

Version Notes

The `sort type` enumeration was added in AppleScript Studio version 1.2.

Tab State

Specifies the current tab state of a tab view item. The `tab state` property of a `tab view item` (page 224) is read only.

Enumerated Values

`background`
Background tab

`pressed`
Pressed tab

`selected`
Selected tab

Tab View Type

Specifies the tab view type, by the position and format of the tab view.

Enumerated Values

`bottom tabs bezel border`
Tabs along the bottom with a bezel border

`left tabs bezel border`
Tabs along the left with a bezel border

`no tabs bezel border`
A bezel border without any tabs

`no tabs line border`
A line border without any tabs

`no tabs no border`
No border without any tabs

`right tabs bezel border`
Tabs along the right with a bezel border

`top tabs bezel border`
Tabs along the top with a bezel border

Version Notes

The constants for `bottom tabs bezel border`, `left tabs bezel border`, and `right tabs bezel border` work only with the version of Cocoa that became available with Mac OS X version 10.2.

Text Alignment

Specifies text alignment. See the `alignment` property of the `text` (page 517) class.

Enumerated Values

`center text alignment`
Align text to the center

`justified text alignment`
Justify the text to the left and right

`left text alignment`
Align text to the left

`natural text alignment`
Align text to the default alignment for the script

`right text alignment`
Align text to the right

Tick Mark Position

Specifies tick mark position. See the `tick mark position` property of the `slider` (page 306) class.

Enumerated Values

`tick mark above`
Tick mark above

`tick mark below`
Tick mark below

`tick mark left`
Tick mark to the left

`tick mark right`
Tick mark to the right

Title Position

Specifies title position. See the `title position` property of the `box` (page 195) class.

Enumerated Values

`above bottom`
Title at the bottom above the line

`above top`
Title at the top above the line

`at bottom`
Title at the bottom

`at top`
Title at the top

`below bottom`
Title at the bottom below the line

`below top`
: Title at the top below the line

`no title`
: No title

CHAPTER 4

Container View Suite

This chapter describes the terminology in AppleScript Studio's Container View suite. For other suites, see the following:

- "Application Suite" (page 42)
- "Control View Suite" (page 244)
- "Data View Suite" (page 348)
- "Document Suite" (page 430)
- "Drag and Drop Suite" (page 448)
- "Menu Suite" (page 462)
- "Panel Suite" (page 474)
- "Text View Suite" (page 516)

Container View Suite

The Container View suite defines the `view` class, as well as additional classes whose primary purpose is to contain other views. Most of the classes in the Container View suite inherit from the `responder` (page 80) class, either directly or through the `view` (page 226) class, so they are able to respond to keystrokes and mouse actions. The Container View suite also defines events for working with container views and the views they contain.

The classes, commands, and events in the Container View suite are described in the following sections:

For enumerated constants, see "Enumerations" (page 177).

Classes

The Container View suite contains the following classes:

box (page 195)

clip view (page 199)

drawer (page 201)

scroll view (page 211)

split view (page 216)

tab view (page 219)

tab view item (page 224)

view (page 226)

box

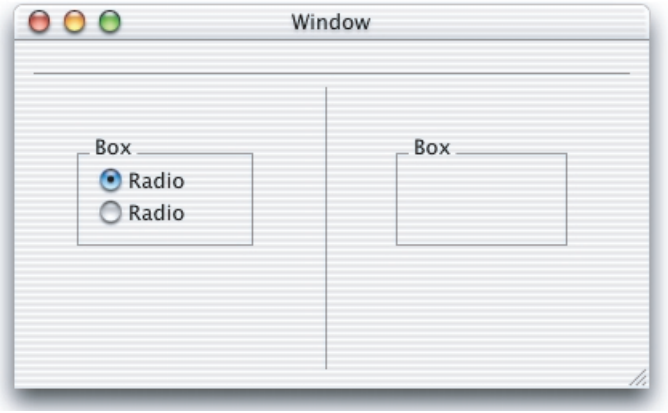
Plural: boxes
Inherits from: view (page 226)
Cocoa Class: NSBox

A simple view that can draw a border around itself and can title itself. You can use an NSBox to visually group other views or to serve as simple separators.

Figure 4-1 shows several boxes, including an empty box, a box that contains two radio buttons, and boxes used as horizontal and vertical separators. In Interface Builder, you will find separator boxes on the Cocoa Views palette (along with various buttons and text views), while container boxes are on the Container Views palette (along with tab and custom views).

See the scroll view (page 211) class for information on how to make objects into subviews of a box or other view in Interface Builder.

Figure 4-1 Boxes, including horizontal and vertical separators



Properties of box objects

In addition to the properties it inherits from the `view` (page 226) class, a `box` object has these properties:

`border rect`

Access: read only
Class: *bounding rectangle*
the bounds of the border; a four-item list of numbers, {left, bottom, right, top}; the box has its own coordinate system, so that the {left, bottom} value is always {0, 0}; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`border type`

Access: read/write
Class: *enumerated constant* from "Border Type" (page 179)
the border type

`box type`

Access: read/write
Class: *enumerated constant* from "Box Type" (page 180)
the box type

`content view`

Access: read/write
Class: *view*
the content view of the box, which contains all of its subviews; for related information, see the `content view` property of the `window` (page 86) class

`content view margins`

Access: read/write
Class: *point*
the margins of the content view, offset from the border of the box; a two item list of numbers {left, bottom}; default is {5.0, 5.0}; in order to see any change from setting this property, you will have to use the `call method` (page 102) command, as in the following example:

```
tell box 1
        set content view margins to {2.5, 2.5}
        call method "sizeToFit"
end tell
```

`title`

Access: read/write
Class: *Unicode text*
the title of the box

`title cell`

Access: read only
Class: *cell* (page 256)
the cell of the title

`title font`

Access: read only
Class: *font* (page 67)
not supported (through AppleScript Studio version 1.2); the font for the cell title

`title position`

Access: read/write
Class: *enumerated constant* from "Title Position" (page 192)
the position of the title

`title rect`

Access: read only
Class: *bounding rectangle*
the bounds of the title
the bounds of the border; a four-item list of numbers, {left, bottom, right, top}; offset within the coordinate system of the box; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

Elements of box objects

A `box` object can contain only the elements it inherits from `view` (page 226).

Events supported by box objects

A `box` object supports handlers that can respond to the following events:

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)

scroll wheel (page 156)

Nib

awake from nib (page 130)

View

bounds changed (page 238)

Examples

For a window "main" that contains a box with several text fields, you could set the text of one of the fields with the following statement:

```
set contents of text field "company" of box "info" of window "main" to "Acme
Nuts and Bolts, Ltd."
```

To access various properties of the same box, you could do the following:

```
tell window "main"
    tell box "info"
        set boxTitle to title
        set boxType to type
        -- and so on
    end tell
end tell
```

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

The `title font` property in this class is not supported, through AppleScript Studio version 1.2.

clip view

Plural: clip views
Inherits from: view (page 226)
Cocoa Class: NSClipView

A view that contains and scrolls the document view displayed by a scroll view. You normally don't need to script a clip view, as the `scroll view` (page 211) class handles most of the details of scrolling when changes in the document view's size or position require it. The `scroll view` class also provides access to many of the same properties listed for the `clip view` class.

Properties of clip view objects

In addition to the properties it inherits from the `view` (page 226) class, a `clip view` object has these properties:

`background color`

Access: read/write
Class: *RGB color*
the background color of the clip view; a three-item integer list that contains the values for each component of the color; for example, blue can be represented as {0,0,65535}; by default, {65535, 65535, 65535}, or white

`content view`

Access: read/write
Class: *view*
the content view of the clip view, which contains all of its subviews; for related information, see the `content view` property of the `window` (page 86) class

`copies on scroll`

Access: read/write
Class: *boolean*
Should the contents of the view be copied when scrolled?

`document rect`

Access: read only
Class: *bounding rectangle*
the bounds of the document view in the clip view; a four-item list of numbers, {left, bottom, right, top}; offset within the coordinate system of the clip view; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`document view`

Access: read/write
Class: *view*

the main subview of the clip view; for example, a `table view` (page 386) or `text view` (page 520)

`draws background`

Access: read/write
Class: *boolean*
Should the clip view draw its background?

`visible document rect`

Access: read only
Class: *bounding rectangle*
the visible bounds of the document view; a four-item list of numbers, {left, bottom, right, top}; offset within the coordinate system of the clip view; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

Elements of clip view objects

A `clip view` object can contain only the elements it inherits from `view` (page 226).

Events supported by clip view objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

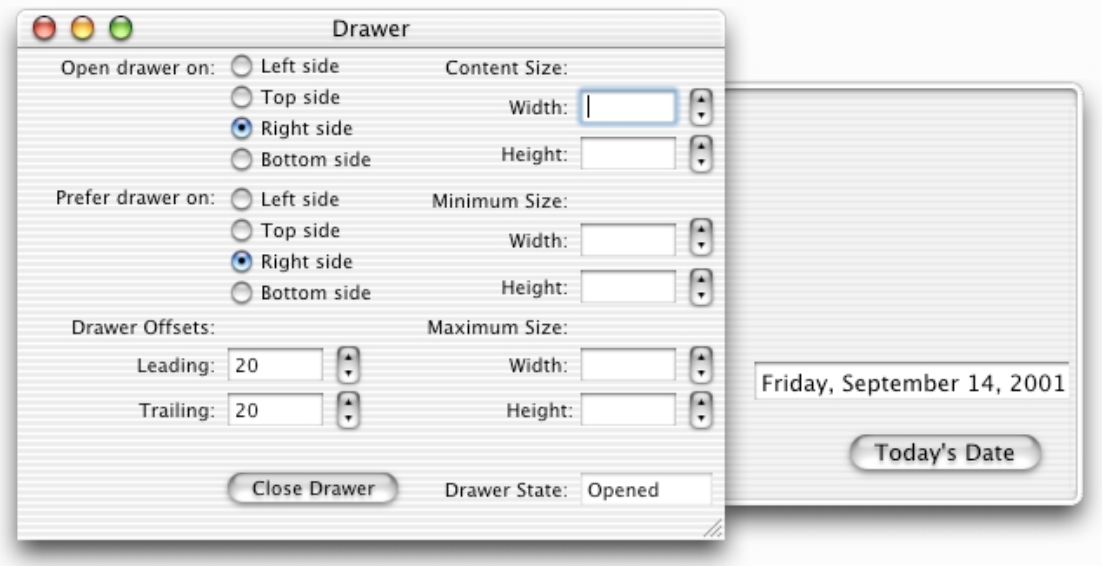
drawer

Plural: `drawers`
Inherits from: `responder` (page 80)
Cocoa Class: NSDrawer

A user interface element that contains and displays view objects. Drawers typically contain `text field` (page 314), `scroll view` (page 211), `browser` (page 349), and other objects based on classes that inherit from the `view` (page 226) class. Figure 4-2 shows a drawer that contains many user interface objects.

A drawer is associated with a `window` (page 86), called its parent, and can only appear while its parent is visible on screen. A drawer cannot be moved or ordered independently of a window, but is instead attached to one edge of its parent and moves along with it.

Figure 4-2 A window with an open drawer (from the Drawer sample application)



To add a drawer to your AppleScript Studio application, you drag it from the Cocoa-Windows palette, shown in Figure 4-3. You typically use the window item shown with an open drawer on the left. That's a convenience object that takes care of the overhead of creating and connecting a window, its drawer, and the content view for the drawer. If you drag that window to your main nib window, you will see the three instances visible in the bottom row in Figure 4-4. You use them as follows:

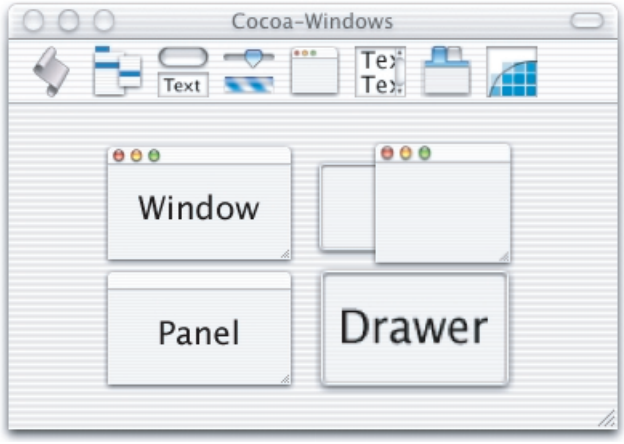
- NSDrawer instance: To connect event handlers for the drawer object, select the NSDrawer instance and open the Info window to the AppleScript pane.
- ParentWindow instance: To add interface items to the window that owns the drawer, double-click the ParentWindow instance to open the window. You can also use the Info window to connect event handlers for the window.
- DrawContentsView instance: To add interface items to the drawer, double-click the DrawContent… instance to open a window (shown in Figure 4-5). You can also use the Info window to connect event handlers for the window. You will see that the Info window identifies the instance as "NSView (Custom)".

Once you have added a window and drawer in Interface Builder, you will have to take steps to show the window in your application, and to allow a user to open and close the drawer. You can show the window by connecting a `launched` (page 142) handler to the File's Owner object (which represents the application) in the main nib window. See the `application` (page 43) class for more information about the File's Owner.

If you give your drawer window the AppleScript name "main" (in the AppleScript pane in the Info window in Interface Builder), your `launched` handler might look something like this one, from the Drawer sample application, distributed with AppleScript Studio:

```
on launched theObject
    show window "main"
end launched
```

Figure 4-3 Interface Builder's Cocoa-Windows palette, with drawers



To allow a user to open and close the drawer, you might add a button with the title "Open Drawer". You could then connect a `clicked` (page 337) handler to the button that

- opens or closes the drawer according to the current state of the drawer

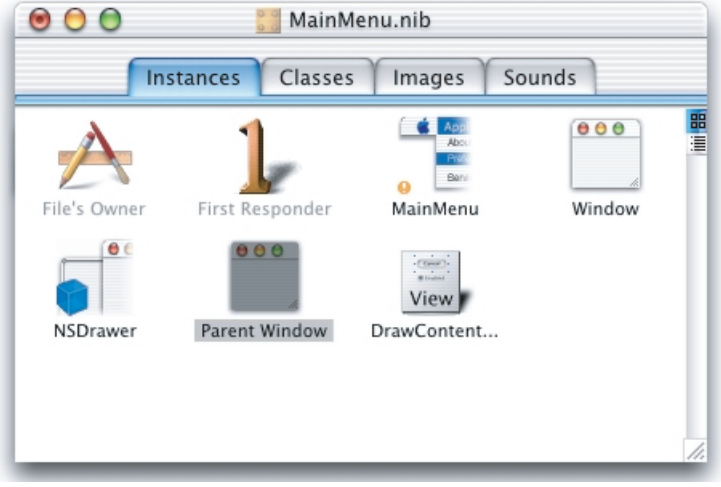
- sets the button title to reflect the state of the drawer (such as “Open Drawer” when the drawer is closed and “Close Drawer” when the drawer is open)

You can use the `open drawer` (page 236) command to open or close a drawer. For example, the following statement opens a drawer named “drawer”:

```
tell drawer "drawer" to open drawer
```

It is also possible to instantiate a drawer by itself (without a parent window or content view) by dragging the object labeled “Drawer” in Interface Builder’s Cocoa-Windows palette to a nib window. In that case, you will get just the NSDrawer instance, and you will have to hook it up to a window and content view yourself (the details of which are beyond the scope of this reference).

Figure 4-4 MainMenu.nib window after adding a window and drawer combination



For more information, see the Cocoa Programming Topic Drawers.

Figure 4-5 The content view for a drawer

Properties of drawer objects

In addition to the properties it inherits from the `responder` (page 80) class, a drawer object has these properties:

`content size`

Access: read/write
Class: *point*
the size of the content view of the drawer; the size is returned as a two-item list of numbers {horizontal, vertical}; for example, `{200, 100}` would indicate a width of 200 and a height of 100; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`content view`

Access: read/write
Class: *view* (page 226)
the content view of the drawer (described above), which contains all of its subviews; for related information, see the `content view` property of the `window` (page 86) class

`edge`

Access: read only
Class: *enumerated constant* from "Rectangle Edge" (page 188)
the edge of the window that the drawer is on

`leading offset`

Access: read/write
Class: *real*
for a drawer that opens on the left or right side, the distance from the top edge of the window to the top edge of the drawer
for a drawer that opens on the top or bottom side, the distance from the left edge of the window to the left edge of the drawer
if the leading offset is 0, the leading edge of the drawer is flush with the edge of the window; if you run the Drawer sample application (shown in Figure 4-2 (page 202)) you can adjust the leading offset to see how the drawer moves in relation to the window

`maximum content size`

Access: read/write
Class: *point*
the maximum size of the content view of the drawer; the size is returned as a two-item list of numbers {horizontal, vertical}, similar to the `content size` property above

`minimum content size`

Access: read/write
Class: *point*
the minimum size of the content of the drawer; the size is returned as a two-item list of numbers {horizontal, vertical}, similar to the `maximum content size` property above

`parent window`

Access: read/write
Class: *window* (page 86)
the window that the drawer belongs to

`preferred edge`

Access: read/write
Class: *enumerated constant* from "Rectangle Edge" (page 188)

the preferred edge (or side) on which to open the drawer; default is `left edge`; you can set this property in the Info window in Interface Builder

`state`

Access: read/write
Class: *enumerated constant* from "Drawer State" (page 184)
the open/closed state of the drawer

`trailing offset`

Access: read only
Class: *real*
for a drawer that opens on the left or right side, the distance from the bottom edge of the window to the bottom edge of the drawer
for a drawer that opens on the top or bottom side, the distance from the right edge of the window to the right edge of the drawer
if the trailing offset is 0, the trailing edge of the drawer is flush with the edge of the window; if you run the Drawer sample application (shown in Figure 4-2 (page 202)) you can adjust the trailing offset to see how the drawer moves in relation to the window

Elements of drawer objects

A `drawer` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`box` (page 195)
Specify by: "Standard Key Forms" (page 30)
the drawer's boxes

`browser` (page 349)
Specify by: "Standard Key Forms" (page 30)
the drawer's browsers

`button` (page 246)
Specify by: "Standard Key Forms" (page 30)
the drawer's buttons

`clip view` (page 199)
Specify by: "Standard Key Forms" (page 30)
the drawer's clip views

`color well` (page 263)

Specify by: "Standard Key Forms" (page 30)
the drawer's color wells

`combo box` (page 265)
Specify by: "Standard Key Forms" (page 30)
the drawer's combo boxes

`control` (page 270)
Specify by: "Standard Key Forms" (page 30)
the drawer's controls

`image view` (page 276)
Specify by: "Standard Key Forms" (page 30)
the drawer's image views

`matrix` (page 280)
Specify by: "Standard Key Forms" (page 30)
the drawer's matrixes

`movie view` (page 286)
Specify by: "Standard Key Forms" (page 30)
the drawer's movie views

`outline view` (page 377)
Specify by: "Standard Key Forms" (page 30)
the drawer's outline views

`popup button` (page 292)
Specify by: "Standard Key Forms" (page 30)
the drawer's popup buttons

`progress indicator` (page 296)
Specify by: "Standard Key Forms" (page 30)
the drawer's progress indicators

`scroll view` (page 211)
Specify by: "Standard Key Forms" (page 30)
the drawer's scroll views

`secure text field` (page 301)
Specify by: "Standard Key Forms" (page 30)
the drawer's secure text fields

`slider` (page 306)
Specify by: "Standard Key Forms" (page 30)
the drawer's sliders

`split view` (page 216)
Specify by: "Standard Key Forms" (page 30)
the drawer's split views

`stepper` (page 310)
Specify by: "Standard Key Forms" (page 30)
the drawer's steppers

`tab view` (page 219)
Specify by: "Standard Key Forms" (page 30)
the drawer's tab views

`table header view` (page 385)
Specify by: "Standard Key Forms" (page 30)
the drawer's table header views

`table view` (page 386)
Specify by: "Standard Key Forms" (page 30)
the drawer's table views

`text field` (page 314)
Specify by: "Standard Key Forms" (page 30)
the drawer's text fields

`text view` (page 520)
Specify by: "Standard Key Forms" (page 30)
the drawer's text views

`view` (page 226)
Specify by: "Standard Key Forms" (page 30)
the drawer's views

Commands supported by drawer objects

Your script can send the following commands to a `drawer` object:

`close drawer` (page 235)
`open drawer` (page 236)

Events supported by drawer objects

A drawer object supports handlers that can respond to the following events:

Drawer

`closed` (page 136)
`opened` (page 150)
`should close` (page 157)
`should open` (page 158)
`will close` (page 166)
`will open` (page 170)
`will resize` (page 172)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

Examples

For an overview of working with drawers, see the description above for this class. For a detailed programming example of working with drawers, see the Drawer sample application, distributed with AppleScript Studio.

scroll view

Plural: scroll views
Inherits from: view (page 226)
Cocoa Class: NSScrollView

Provides the ability to scroll a document view that's too large to display in its entirety. In addition to handling scrolling of the view, a scroll view can display horizontal and vertical scrollers and rulers (depending on how it is configured. Most scrolling is handled automatically by the scroll view, but see the scroll (page 328) command for a mechanism you can use to scroll in a text view.

A number of AppleScript Studio classes automatically include a scroll view, including the outline view (page 377), table view (page 386), and text view (page 520) classes. In Interface Builder, you can enclose any interface objects in a scroller by selecting the objects, choosing "Make subviews of" from the Layout menu, and then choosing Scroll View. You can use the same mechanism to make views be subviews of a box (page 195), a split view (page 216), or a tab view (page 219).

You can also use this mechanism to group objects on a custom view (defined by you) or some other kind of built-in view (such as a matrix (page 280)). To do so, follow these steps:

1. Select the objects to be grouped.
2. Choose "Make subviews of" from the Layout menu and choose "Custom View".
3. Select the resulting custom view. By default, it will be a view object (of class NSView).
4. In the Custom Class pane of the Info window, select the desired class type.

For related information, see the Cocoa Programming Topic Drawing and Views.

Properties of scroll view objects

In addition to the properties it inherits from the view (page 226) class, a scroll view object has these properties:

background color

Access: read/write
Class: *RGB color*

the color of the background; a three-item integer list that contains the values for each component of the color; for example, blue can be represented as {0,0,65535}; default is {65535, 65535, 65535} (or white)

`border type`

Access: read/write
Class: *enumerated constant* from "Border Type" (page 179)
the type of border for the scroll view

`content size`

Access: read only
Class: *point*
the size of the content view of the scroll view; the size is returned as a two-item list of numbers {horizontal, vertical}; for example, `{200, 100}` would indicate a width of 200 and a height of 100; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`content view`

Access: read/write
Class: *view*
the content view; a `clip view` (page 199) that clips the document view; for related information, see the `content view` property of the `window` (page 86) class

`document view`

Access: read/write
Class: *view* (page 226)
the document view that the scroller scrolls;

`draws background`

Access: read/write
Class: *boolean*
Should the scroll view draw the background? default is `true`;

`dynamically scrolls`

Access: read/write
Class: *boolean*
Should the view scroll dynamically? default is `true`;

`has horizontal ruler`

Access: read/write
Class: *boolean*
Does the scroll view have a horizontal ruler?

`has horizontal scroller`

Access: read/write
Class: *boolean*
Does the scroll view have a horizontal scroller?

`has vertical ruler`

Access: read/write
Class: *boolean*
Does the scroll view have a vertical ruler?

`has vertical scroller`

Access: read/write
Class: *boolean*
Does the scroll view have a vertical scroller?

`horizontal line scroll`

Access: read/write
Class: *real*
the horizontal amount to line scroll

`horizontal page scroll`

Access: read/write
Class: *real*
the horizontal amount to page scroll

`horizontal ruler view`

Access: read/write
Class: *anything*
the horizontal ruler view

`horizontal scroller`

Access: read/write
Class: *anything*
the horizontal scroller

`line scroll`

Access: read/write
Class: *real*
the amount to line scroll

`page scroll`

Access: read/write
Class: *real*
the amount to page scroll

`rulers visible`

Access: read/write
Class: *boolean*
Are the rulers visible?

`vertical line scroll`

Access: read/write
Class: *real*
the vertical amount to line scroll

`vertical page scroll`

Access: read/write
Class: *real*
the vertical amount to page scroll

`vertical ruler view`

Access: read/write
Class: *anything*
the vertical ruler view

`vertical scroller`

Access: read/write
Class: *anything*
the vertical scroller

`visible document rect`

Access: read only
Class: *bounding rectangle*
the visible bounds of the document

Elements of scroll view objects

A `scroll view` object can contain only the elements it inherits from `view` (page 226).

Events supported by scroll view objects

A `scroll view` object supports handlers that can respond to the following events:

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

View

`bounds changed` (page 238)

Examples

Because several kinds of objects, including outline, table, and text views automatically reside on a scroll view, AppleScript Studio applications often need to include a scroll view in specifying an object. The following line is from the Table sample application, distributed with AppleScript Studio.

```
tell table view "contacts" of scroll view "contacts"
    of window of theObject to update
```

You can access properties of a scroll view with statements like the following, which sets a boolean variable based on whether the scroll view has a vertical scroller:

```
set hasVertScroller to has vertical scroller of scroll view "contacts" of
window of theObject
```

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

split view

Plural: split views

Inherits from: view (page 226)

Cocoa Class: NSSplitView

Stacks several subviews within one view to coordinate changes in their relative sizes. The pane splitter bars between the views can be horizontal or vertical, depending on whether the views are arranged vertically or side by side. In Figure 4-6, an outline view appears above a table view.

See the scroll view (page 211) class for information on how to make objects into subviews of a split view or other view in Interface Builder.

Figure 4-6 A split view that contains an outline view and a table view

Properties of split view objects

In addition to the properties it inherits from the `view` (page 226) class, a split view object has these properties:

`pane splitter`

Access: read/write
Class: *boolean*
Is there a pane splitter? (see the horizontal bars in the middle of Figure 4-6)

`vertical`

Access: read/write
Class: *boolean*
Is the splitter vertical?

Elements of split view objects

A `split view` object can contain only the elements it inherits from `view` (page 226).

Events supported by split view objects

A split view object supports handlers that can respond to the following events:

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

Split View

`resized sub views` (page 239)
`will resize sub views` (page 241)

View

`bounds changed` (page 238)

Examples

The following script gets the `vertical` property of a split view in the Mail Search sample application, distributed with AppleScript Studio. This script is not part of the application, but you can run it from Script Editor to access the property in the Mail Search application. Similar statements will work within an AppleScript Studio application script (though you won't need the `tell application` statement).

```
tell application "Mail Search"
    tell front window
        set isVertical to vertical of first split view
        -- Do something based on result
    end tell
end tell
```

You may also need to refer to a split view to access another view in your application. The following fragment from Mail Search specifies a scroll view on a split view of a window:

```
tell scroll view "mailboxes" of split view 1 of theWindow
    -- Access properties or subviews of the scroll view.
end tell
```

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

Prior to AppleScript Studio version 1.1, the Mail Search sample application was named Watson.

tab view

Plural: `tab views`

Inherits from: `view` (page 226)

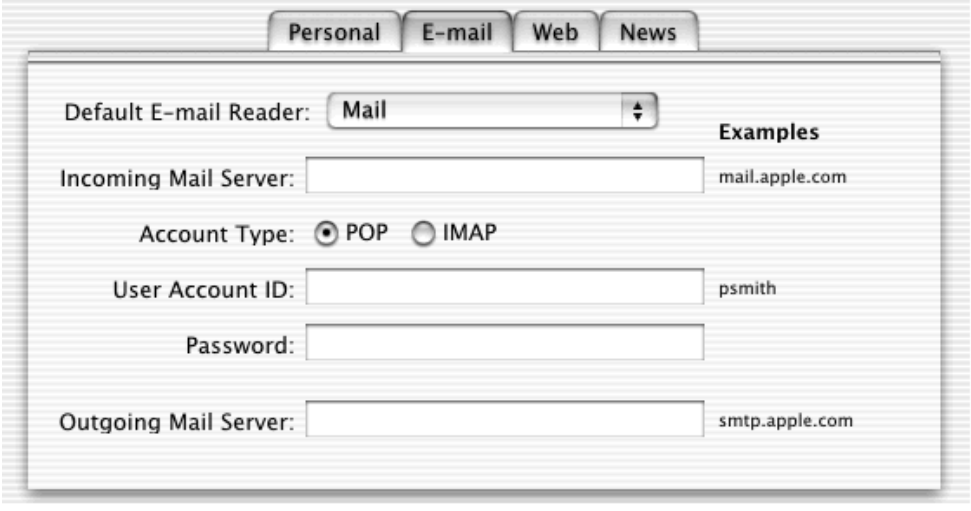
Cocoa Class: NSTabView

Provides a convenient way to present information in multiple pages. The view contains a row of tabs that give the appearance of folder tabs, as shown in Figure 4-7. The user selects the desired page by clicking the appropriate tab or using the arrow keys to move between pages. Each page displays a view hierarchy provided by your application. Each tab, and its associated view hierarchy, is represented by a `tab view item` (page 224).

The Examples section describes how to use a tab view to give the appearance of switching out the entire contents of a view.

See the `scroll view` (page 211) class for information on how to make objects into subviews of a tab view or other view in Interface Builder.

Figure 4-7 A tab view with four panes



Properties of tab view objects

In addition to the properties it inherits from the `view` (page 226) class, a `tab view` object has these properties:

`content rect`

Access: read only
Class: *bounding rectangle*
the bounds for contents of the tab view; a four-item list of numbers, {left, bottom, right, top}; the bounding rectangle is offset within the tab view's coordinate system; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`control size`

Access: read/write
Class: *enumerated constant* from "Control Size" (page 183)
the size of the tabs; default is `regular size`; you can set this property in Interface Builder (with the Small Tabs checkbox in the Attributes pane of the Info window)

`control tint`

Access: read/write
Class: *enumerated constant* from "Control Tint" (page 183)
the tint of the tabs; default is `default tint`

`current tab view item`

Access: read/write
Class: *tab view item* (page 224)
the current tab view item

`draws background`

Access: read/write
Class: *boolean*
Should the tab view draw its background? default is `true`; see the description above for information on when you can set this property in the Info window in Interface Builder

`tab type`

Access: read/write
Class: *enumerated constant* from "Tab View Type" (page 191)
the type of tab (such as a bezeled tab on the top of the tab view); you can set tab direction (Top, Left, Bottom, or Right) in Interface Builder, though Left, Bottom, and Right work only with the version of Cocoa that became available with Mac OS X version 10.2.

`truncated labels`

Access: read/write
Class: *boolean*
Should the labels be truncated, if necessary? you can set this property in the Info window in Interface Builder

Elements of tab view objects

In addition to the elements it inherits from the `view` (page 226) class, a `tab view` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`tab view item` (page 224)
Specify by: "Standard Key Forms" (page 30)
the view's tab view items, one per tab in the view

Events supported by tab view objects

A tab view object supports handlers that can respond to the following events.

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

Tab View

`selected tab view item` (page 239)
`should select tab view item` (page 240)
`will select tab view item` (page 242)

View

`bounds changed` (page 238)

Examples

Given appropriately-named objects, you can use terminology like the following to set the text in a text field on a tab view:

```
set contents of text field "textFieldName" of view of tab view item
"tabViewItemName" of tab view "tabViewName" of window "windowname"
```

Items on a tab view item are actually on a view that's on the tab view item, which accounts for the `view` in the phrase `of view of tab view item` in this example. However, starting in AppleScript Studio version 1.2, you no longer need to specify the view, and can use the following, slightly simpler form:

```
set contents of text field "textFieldName" of tab view item "tabViewItemName"
of tab view "tabViewName" of window "windowname"'
```

You can use a tab view to give the appearance of switching out the entire contents of a view. In Interface Builder, drag a tab view from the Cocoa-Containers pane of the Palette window to the target window. With the tab view selected, select the Tabless radio button. You can then choose styles between having a drop shadow or having no visible frame at all. If you choose no visible frame, you can also choose whether the tab view should draw its background.

You will still be able to put whatever user interface items you want on each tab view item. Because there are no tabs to select, the Info window provides a stepper control to navigate between tab view items. Since a user won't be able to select among tabs (presumably that's your goal), you will have to change the currently displayed tab view item programmatically, with statements like the following (if you've given the first tab view item the name `"tabViewItem1"`):

```
tell tab view "tabview" of window "main"
    set the current tab view item to tab view item "tabViewItem1"
end
```

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

Tab views in AppleScript Studio are implemented by Cocoa's NSTabView class. Prior to Mac OS X version 10.2, NSTabView only supported tab views with the tabs on the top.

Starting with AppleScript Studio version 1.2, a script can say `button 1 of tab view item 1 of tab view 1` instead of `button 1 of view of tab view item 1 of tab view 1`, though the longer version still works (and will run with any version of AppleScript Studio, not just version 1.2). See the Examples section above for another example.

tab view item

Plural: `tab view items`
Inherits from: None.
Cocoa Class: NSTabViewItem

Represents one tab in a tab view. When a user clicks on a tab, the `tab view` (page 219) displays a view page provided by your application. A tab view keeps a one-based array of tab view items, one for each tab in the view. The tab view in Figure 4-7 (page 220) has four tab view items.

When you drag a tab view from the Cocoa-Containers pane of Interface Builder's Palette window, it contains two tab view items. You can set the number of tab view items in Interface Builder's Info window.

Properties of tab view item objects

A tab view item has these properties:

`color`

Access: read/write
Class: *RGB color*

the RGB color of the tab view item; a three-item integer list that contains the values for each component of the color; for example, blue can be represented as {0,0,65535}; default is {65535, 65535, 65535} (or white)

`label`

Access: read/write
Class: *Unicode text*
the label of the tab view item; you can set the label in Interface Builder

`tab state`

Access: read only
Class: *enumerated constant* from "Tab State" (page 190)
the state of the tab view item

`tab view`

Access: read only
Class: *tab view* (page 219)
the tab view that contains this tab view item

`view`

Access: read/write
Class: *view* (page 226)
the view of the tab view item (on which you place the user-interface objects for the tab view item)

Events supported by tab view item objects

A tab view item supports handlers that can respond to the following events:

Nib

`awake from nib` (page 130)

Examples

The following handler is from the sample application Assistant, distributed with AppleScript Studio (starting with version 1.1). This handler is called when the script's properties need to be updated from the contents of user interface objects associated with a tab view item. As the first line of the `tell` statement shows, the

terminology to access a tab view can be quite complex. A common error prior to AppleScript Studio version 1.2 was to omit `view of` at the beginning of the statement. Starting with version 1.2, the `view of` is optional.

Within the `tell` block, the statements to gather information from individual text field objects are more straightforward. Each statement assigns a value to a property of the script.

```
on updateValues(theWindow)
    tell view of tab view item infoPanelName of tab view "info panels"
                of box "border" of theWindow
        set my company to contents of text field "company"
        set my name to contents of text field "name"
        set my address to contents of text field "address"
        set my city to contents of text field "city"
        set my state to contents of text field "state"
        set my zip to contents of text field "zip"
        set my email to contents of text field "email"
    end tell
end updateValues
```

The Examples section for the `tab view` (page 219) class describes how to use a tab view to give the appearance of switching out the entire contents of a view by switching tab view items.

`view`

Plural: `views`

Inherits from: `responder` (page 80)

Cocoa Class: NSView

An abstract class that defines the basic drawing, event-handling, and printing architecture of an application. Your scripts typically don't interact with `view` objects directly; rather, they interact with the many user interface classes that inherit from the `view` class.

You can create and access a `view` object in Interface Builder with the following step:

- Drag a CustomView instance from the Cocoa-Containers pane of the Palette window to the target window. This view is by default a `view` object of class NSView.

To change the class to a custom view class you have defined, or to another built-in class type, use these steps:

1. Select the custom view.
2. In the Custom Class pane of the Info window, select the new class type.

Properties of view objects

In addition to the properties it inherits from the `responder` (page 80) class, a `view` object has these properties:

`auto resizes`

Access: read/write
Class: *boolean*
Should the view auto resize?

`bounds`

Access: read/write
Class: *bounding rectangle*
the position and size of the view (within its superview); the bounds are returned as a four-item list of numbers {left, bottom, right, top}; the bounding rectangle is offset from {0, 0} in the superview; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`bounds rotation`

Access: read/write
Class: *real*
the rotation of the bounds rectangle, in degrees; default is 0.0; positive values indicate counterclockwise rotation, negative clockwise; rotation is performed around the coordinate system origin, (0.0, 0.0), which need not coincide with that of the frame rectangle or the bounds rectangle; changing this value does not redisplay the view or mark it as needing display; you can do that by setting the `needs display` property to `true`, or by using the `update` (page 126) command

`can draw`

Access: read only
Class: *boolean*
Can the view be drawn? `true` if drawing commands will produce any result, `false` otherwise; this property is used when invoking drawing directly, along with the `lock focus` (page 235) and `unlock focus` (page 236) commands; however,

those commands are not supported through AppleScript Studio version 1.2; in addition, AppleScript Studio doesn't currently provide fine control over drawing, and most applications will not need to invoke drawing directly; if your application is an exception, see NSView, as well as the Cocoa Programming Topics pointed to in the documentation for that class

`enclosing scroll view`

Access: read/write
Class: *scroll view* (page 211)
the scroll view of the view (if any); see the description of the `scroll view` class for how to enclose a view in a scroll view in Interface Builder

`flipped`

Access: read only
Class: *boolean*
Is the coordinate system of the view flipped? by default the origin of a view's coordinate system is in the lower left; however, for some views, the default for this property is `true`, meaning the origin is in the top left; this is a read-only property, and it's unlikely your application will be concerned with it; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system, which is different than that used by the Mac OS Finder application

`needs display`

Access: read/write
Class: *boolean*
Does the view need to be displayed? setting this property to true will cause the view to be redrawn at the next opportunity; to cause an immediate redraw, use the `update` (page 126) command; note that through AppleScript Studio version 1.2, the `needs display` property is not supported for the `window` (page 86) class, but is supported for the `view` class

`opaque`

Access: read only
Class: *boolean*
Is the view opaque? see the description for the `opaque` property in the `window` (page 86) class

`position`

Access: read/write
Class: *point*
position of the view within its superview as a two-item list of numbers {left, bottom}; each view has its own coordinate system, with the origin in the lower left corner; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`size`

Access: read/write
Class: *point*
the size of the view; the size is returned as a two-item list of numbers {horizontal, vertical}; for example, `{200, 100}` would indicate a width of 200 and a height of 100; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`super view`

Access: read/write
Class: *view* (page 226)
the view that contains this view

`tag`

Access: read/write
Class: *integer*
the tag for the view; you can set the tag for some views, such as `text field` (page 314) views, in the Info window in Interface Builder

`tool tip`

Access: read/write
Class: *Unicode text*
the tool tip for the view (text to be displayed if the user lets the cursor hover over the view)

`visible`

Access: read/write
Class: *boolean*
Is the view visible?

`visible rect`

Access: read only
Class: *bounding rectangle*
the visible area of the view; a four-item list of numbers, {left, bottom, right, top}; the view has its own coordinate system; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`window`

Access: read/write
Class: *window* (page 86)
the window that contains this view

Elements of view objects

A `view` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`box` (page 195)
Specify by: "Standard Key Forms" (page 30)
the view's boxes

`browser` (page 349)
Specify by: "Standard Key Forms" (page 30)
the view's browsers

`button` (page 246)
Specify by: "Standard Key Forms" (page 30)
the view's buttons

`clip view` (page 199)
Specify by: "Standard Key Forms" (page 30)
the view's clip views

`color well` (page 263)
Specify by: "Standard Key Forms" (page 30)
the view's color wells

`combo box` (page 265)
Specify by: "Standard Key Forms" (page 30)
the view's combo boxes

`control` (page 270)

Specify by: "Standard Key Forms" (page 30)
the view's controls

`image view` (page 276)
Specify by: "Standard Key Forms" (page 30)
the view's image views

`matrix` (page 280)
Specify by: "Standard Key Forms" (page 30)
the view's matrixes

`movie view` (page 286)
Specify by: "Standard Key Forms" (page 30)
the view's movie views

`outline view` (page 377)
Specify by: "Standard Key Forms" (page 30)
the view's outline views

`popup button` (page 292)
Specify by: "Standard Key Forms" (page 30)
the view's popup buttons

`progress indicator` (page 296)
Specify by: "Standard Key Forms" (page 30)
the view's progress indicators

`scroll view` (page 211)
Specify by: "Standard Key Forms" (page 30)
the view's scroll views

`secure text field` (page 301)
Specify by: "Standard Key Forms" (page 30)
the view's secure text fields

`slider` (page 306)
Specify by: "Standard Key Forms" (page 30)
the view's sliders

`split view` (page 216)
Specify by: "Standard Key Forms" (page 30)
the view's split views

stepper (page 310)
Specify by: "Standard Key Forms" (page 30)
the view's steppers

tab view (page 219)
Specify by: "Standard Key Forms" (page 30)
the view's tab views

table header view (page 385)
Specify by: "Standard Key Forms" (page 30)
the view's table header views

table view (page 386)
Specify by: "Standard Key Forms" (page 30)
the view's table views

text field (page 314)
Specify by: "Standard Key Forms" (page 30)
the view's text fields

text view (page 520)
Specify by: "Standard Key Forms" (page 30)
the view's text views

view (page 226)
Specify by: "Standard Key Forms" (page 30)
the view's views

Commands supported by view objects

Your script can send the following commands to a view object:

lock focus (page 235) (not supported through AppleScript Studio version 1.2)
register (page 123)
unlock focus (page 236) (not supported through AppleScript Studio version 1.2)

Events supported by view objects

A view object view supports handlers that can respond to the following events. To connect an event handler to a view object in Interface builder, put the nib window for the window object that contains the view into outline mode by clicking the small

outline icon above the right scroll bar; use the disclosure triangles to open the `window` and other enclosed objects until the `view` object is visible; select it, then connect the event handler in the AppleScript pane of the Info window.

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

View

`bounds changed` (page 238)

Examples

The `view` class is an abstract class that you don't typically target in your scripts. More often you target subclasses, such as `box` (page 195) or `scroll view` (page 211) or `tab view` (page 219). The `control` (page 270) class also inherits from the `view` class, so all subclasses of control inherit view properties and elements (though view elements such as a movie view or progress indicator are not of much use to a control such as a button).

You can use the following script in Script Editor to rotates the text in a `text view` (page 520) by 50 degrees. Similar statements will work within an AppleScript Studio application script (though you won't need the `tell application` statement).

```
tell application "rotate"
    tell window 1
        tell scroll view 1
            tell text view 1
                set bounds rotation to 50.0
                set needs display to true
            end tell
        end tell
    end tell
end tell
```

Version Notes

Through AppleScript Studio version 1.2, the `needs display` property is not supported for the `window` (page 86) class, but is supported for the `view` class.

The `lock focus` (page 235) and `unlock focus` (page 236) commands are not supported through AppleScript Studio version 1.2.

Commands

Objects based on classes in the Container View suite support the following commands. (A command is a word or phrase a script can use to request an action.) To determine which classes support which commands, see the individual class descriptions.

- `close drawer` (page 235)
- `lock focus` (page 235)
- `open drawer` (page 236)
- `unlock focus` (page 236)

close drawer

Closes the specified drawer.

Syntax

`close drawer`	*reference*	required

Parameters

reference
a reference to the `drawer` (page 201) to close

Examples

For a window with AppleScript name “main” that contains a drawer named “drawer” you can close the drawer with a `tell` statement like the following:

```
tell window "main"
    tell drawer "drawer" to close
end tell
```

lock focus

Not supported (through AppleScript Studio version 1.2). Locks the focus on a view to prepare it for drawing.

Syntax

`lock focus`	*reference*	required

Parameters

reference
a reference to the view object for which to lock focus

open drawer

Opens the specified drawer.

Syntax

`open drawer`	*reference*	required
`on`	*enumerated constant*	optional

Parameters

reference
a reference to the `drawer` (page 201) to open

`on` *enumerated constant* from "Rectangle Edge" (page 188)
the edge of the window to open the drawer on

Examples

For a window with AppleScript name "main" that contains a drawer named "drawer" you can open the drawer on the left side of the window with a `tell` statement like the following:

```
tell window "main"
    tell drawer "drawer" to open drawer on left edge
end tell
```

unlock focus

Not supported (through AppleScript Studio version 1.2).

Syntax

```
unlock focus    reference    required
```

Parameters

reference
a reference to the view object for which to unlock focus

Events

Objects based on classes in the Container View suite support handlers for the following events (an event is an action, typically generated through interaction with an application's user interface, that causes a handler for the appropriate object to be executed). To determine which classes support which events, see the individual class descriptions.

- `bounds changed` (page 238)
- `resized sub views` (page 239)
- `selected tab view item` (page 239)
- `should select tab view item` (page 240)
- `will resize sub views` (page 241)
- `will select tab view item` (page 242)

`bounds changed`

Called after a `view` object's bounds are changed.

Syntax

`bounds changed` *reference* required

Parameters

reference
a reference to the `view` (page 226) whose bounds changed

Examples

When you connect a `bounds changed` handler to a `view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to react to a change in bounds. To determine the change in bounds, you will have to store the old bounds of the view and compare it to the current bounds.

```
on bounds changed theObject
    (* Perform operations here after bounds changed. *)
end bounds changed
```

resized sub views

Called after a view object's subviews are resized.

Syntax

`resized sub views`	*reference*	required

Parameters

reference
a reference to the object whose subviews were resized

Examples

When you connect a `resized sub views` handler to a `view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. To determine the change in the view's size, you will have to save the old size of the view (which you might do in a `will resize sub views` (page 241) handler) and compare it to the current size. You can use the handler to perform any operations necessary after subviews are resized.

```
on resized sub views theObject
    (* Perform operations here after sub views resized. *)
end resized sub views
```

selected tab view item

Called after a tab view item is selected, indicating the current tab view item has changed. A tab view item represents one tab on a `tab view` (page 219).

Syntax

`selected tab view item`	*tab view*	required
`tab view item`	*tab view item*	optional

Parameters

`tab view` (page 219)
the tab view whose tab view item was selected

`tab view item` `tab view item` (page 224)
the tab view item that was selected

Examples

The following `selected tab view item` handler is from the Assistant sample application, distributed with AppleScript Studio (starting with version 1.1). This handler is called when the current tab view item has been changed. That's a good place to perform any preparation before showing the contents of the tab.

```
on selected tab view item theObject tab view item tabViewItem
    -- We will give the new info panel a chance to
    --  prepare it's data values
    prepareValues(window of theObject)
        of infoPanelWithName(name of tabViewItem)
end selected tab view item
```

should select tab view item

Called to determine whether the object's tab view item should be selected, most likely because a user has clicked the associated tab. The handler can return `false` to refuse to allow the item to be selected (so there will be no switch of tab items) or `true` to allow selection. A tab view item represents one tab on a `tab view` (page 219).

Syntax

`should select tab view item`	*tab view*	required
`tab view item`	*tab view item*	optional

Parameters

`tab view` (page 219)
the tab view whose tab view item may be selected

`tab view item` `tab view item` (page 224)
the tab view item

Result

boolean

Return `true` to allow selection; `false` to disallow it. If you implement this handler, you should always return a boolean value.

Examples

The following example of a `should select tab view item` handler calls an application handler `shouldSelectTabViewItem`, written by you, to determine whether to allow the item to be selected, then returns the appropriate value. You might instead perform validation in the handler itself or check some global property.

```
on should select tab view item theObject
    --Check property, perform test, or call handler to see if OK
    --  to select tab view item specified by theObject
    set allowSelection to shouldSelectTabViewItem(theObject)
    return allowSelection
end should select tab view item
```

The Assistant sample application, available starting with AppleScript Studio version 1.2, includes a `should select tab view item` handler that examines each panel before making a decision on whether to allow a change in the selected tab.

`will resize sub views`

Called when a view object's subviews are about to be resized. The handler cannot cancel the resizing, but can prepare for it.

Syntax

`will resize sub views` *reference* required

Parameters

reference

a reference to the object whose subviews are about to be resized

Examples

When you connect a `will resize sub views` handler to a `view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for subviews being resized. For example, you might save the current size of the subviews

```
on will resize sub views theObject
    (* Perform any operations to prepare for resizing of subviews. *)
end will resize sub views
```

will select tab view item

Called when a tab view item is about to be selected. A tab view item represents one tab on a `tab view` (page 219). The handler cannot cancel the selection, but can prepare for it.

Syntax

`will select tab view item`	*tab view*	required
`tab view item`	*tab view item*	optional

Parameters

`tab view` (page 219)
the tab view whose tab view item is about to be selected

`tab view item` `tab view item` (page 224)
the tab view item that is about to be selected

Examples

When you connect a `will select tab view item` handler to a `tab view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for the tab view item being selected.

```
on will select tab view item theObject
    (* Perform any operations to prepare for selection of the item. *)
end will select tab view item
```

Control View Suite

This chapter describes the terminology in AppleScript Studio's Control View suite. For other suites, see the following:

Control View Suite

The Control View suite defines a number of classes for implementing or working with controls. Controls are graphic objects that cause instant actions or visible results when a user manipulates them with the mouse.

Most classes in this suite inherit from the `view` (page 226) class, either directly or through the `control` (page 270) class. The Control View suite also defines many events for working with user actions involving controls.

The classes, commands, and events in the Control View suite are described in the following sections:

"Classes" (page 245)

"Commands" (page 322)

"Events" (page 334)

For enumerated constants, see "Enumerations" (page 177).

Classes

The Control View suite contains the following classes:

action cell

Plural: action cells
Inherits from: cell (page 256)
Cocoa Class: NSActionCell

Defines an active area inside a control or one of its subclasses. As the active area of a `control` (page 270), an `action cell` does three things: it usually performs display of text or an icon; it provides the control with a target and an action; and it handles mouse (cursor) tracking by properly highlighting its area and sending action messages to its target based on cursor movement.

Events supported by action cell objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

See the examples for `cell` (page 256).

button

Plural: `buttons`

Inherits from: `control` (page 270)

Cocoa Class: NSButton

Control subclass that intercepts mouse-down events and initiates a handler call when it is clicked or pressed. A button contains a single `button cell` (page 252). Figure 5-1 shows a simple button.

Figure 5-1 A button

You will find an assortment of buttons on the Cocoa-Views pane of Interface Builder's Palette window. You can set many attributes for buttons in Interface Builder's Info window. You can also change button types in your scripts, using the constants defined in "Button Type" (page 180).

Properties of button objects

In addition to the properties it inherits from the `control` (page 270) class, a button object has these properties:

`allows mixed state`

Access: read/write
Class: *boolean*
Does the button allow a mixed state? (see also the `state` property); some buttons may reflect only two states, such as on or off; a mixed state reflects more than two states; suppose, for a simple-minded example, that a checkbox makes selected text bold; if all the selected text is bold, the checkbox is on; if none of the selected text is bold, it's off; if the selected text has a combination of bold and plain text, the state is mixed

`alternate image`

Access: read/write
Class: *image* (page 72)
the image for the button when it is in its alternate state; see the Discussion section

`alternate title`

Access: read/write
Class: *Unicode text*
the title of the button when it is in its alternate state; see the Discussion section

`bezel style`

Access: read/write
Class: *enumerated constant* from "Bezel Style" (page 179)
the bezel style of the button

`bordered`

Access: read/write
Class: *boolean*
Does the button have a border?

`button type`

Access: read/write
Class: *enumerated constant* from "Button Type" (page 180)
the type of button

`image`

Access: read/write
Class: *image* (page 72)
the image of the button; see the Discussion section

`image position`

Access: read/write
Class: *enumerated constant* from "Cell Image Position" (page 181)
the position of the image in the button, as a two-item list of numbers {left, bottom}; every window and view has its own coordinate system, with the origin in the lower left corner; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`key equivalent`

Access: read/write
Class: *Unicode text*
the key equivalent to clicking on the button; for example, the key equivalent might be the letter "U" or the combination Command-U; you can set this property on the attributes pane in the Info window in Interface Builder, where, for example, you can set a button to be the default button by choosing "Return" on the pop-up menu for the "Equiv." field; the default button automatically pulses and takes on the default color

`key equivalent modifier`

Access: read/write
Class: *number*
the modifier key for the key equivalent; you can set this property to either or both the Command and Option keys in the Info window in Interface Builder; however, the value returned by this property is a number; default is 0 (or no modifier)

`roll over`

Access: read/write
Class: *boolean*
Does the button act like a roll over?

`sound`

Access: read/write
Class: *sound* (page 81)
the sound of the button when it is clicked

`state`

Access: read/write
Class: *enumerated constant* from "Cell State Value" (page 181)
the state of the button; (see also the `allows mixed state` property)

`title`

Access: read/write
Class: *Unicode text*
the title of the button; see the Discussion section

`transparent`

Access: read/write
Class: *boolean*
Is the button transparent?

Elements of button objects

A `button` object can contain only the elements it inherits from `control` (page 270).

Commands supported by button objects

Your script can send the following command to a button object:

highlight (page 324)

Events supported by button objects

A button object supports handlers that can respond to the following events:

Action

clicked (page 337)

Drag and Drop

conclude drop (page 452)
drag (page 453)
drag entered (page 454)
drag exited (page 455)
drag updated (page 456)
drop (page 457)
prepare drop (page 459)

Key

keyboard down (page 141)
keyboard up (page 141)

Mouse

mouse down (page 144)
mouse dragged (page 145)
mouse entered (page 146)
mouse exited (page 146)
mouse up (page 148)
right mouse down (page 154)
right mouse dragged (page 155)
right mouse up (page 156)
scroll wheel (page 156)

Nib

awake from nib (page 130)

View

bounds changed (page 238)

Examples

AppleScript Studio provides access to many types of button objects and the sample applications distributed with AppleScript Studio include many examples of working with buttons. The following script statements demonstrate some of the basic terminology for using buttons.

In the most common case, you use a button to trigger an action in a clicked (page 337) handler. The following clicked handler, from the Currency Converter sample application, just performs the currency conversion, based on the values the user has supplied, and displays the result. The clicked handler's theObject parameter refers to the button. In this case, the handler uses the window of the button to get at other objects.

```
on clicked theObject
    tell window of theObject
        try
            set theRate to contents of text field "rate"
            set theAmount to contents of text field "amount" as number

            set contents of text field "total" to theRate * theAmount
        on error
            set contents of text field "total" to 0
        end try
    end tell
end clicked
```

The following statement disables a button with AppleScript name "someButton".

```
set enabled of button someButton to "false"
```

The following statements, from the clicked handler of the Unit Converter sample application, show how to determine if a particular button was the target for a view that has more than one button that may need to respond. In this case, the button of interest (the Convert button) is part of a box object.

```
on clicked theObject
    tell window "Main"
        if theObject is equal to button "Convert" of box 1 then
```

```
            my convert()
        else if ...
        ...
    end tell
end clicked
```

Discussion

Buttons provide a simple mechanism for switching the button title or image as a user toggles the button state. For example, to create a button in Interface Builder that toggles its text between "Start" and "Stop", you use these steps:

1. Drag a button object from the Cocoa-Views pane of the Palette window to the target window. You can use any button that shows "NSButton" when you rest the cursor over it in the Palette window.
2. With the button selected in the target window, open the Info window by choosing Show Info from the Tools menu or by typing Command-Shift-I.
3. If the Behavior pop-up on the attributes pane of the Info window isn't enabled, choose a button type in the Type pop-up that causes the Behavior pop-up to be enabled. For example, you might set the button type to Rounded Bevel Button, Square Button, or Round Button.
4. Set the Behavior pop-up to Toggle.
5. Type "Start" in the Title field and "Stop" in the Alt. Title field.
6. To test the button, choose Test Interface from the File menu (or type Command-R). You should be able to click the button and toggle its title between "Start" and "Stop".

The Attributes pane provides other button settings, including fields for setting an icon and an alternate icon for buttons that show an icon. Note that you can display any image in the button, not just an icon.

button cell

Plural: button cells

Inherits from: cell (page 256)

Cocoa Class: NSButtonCell

Subclass of cell that implements the user interface for push buttons, switches, and radio buttons. You don't typically need to access the properties of a button cell, as you can access the same properties through the button (page 246) class.

In Cocoa, a button cell can be used for any region of a view that is designed to send a message to a target when clicked, although again, this isn't a typical use for most AppleScript Studio applications. For related information, see the `action cell` (page 245), `cell` (page 256), and `button` (page 246) class descriptions.

You can create and access a button cell in Interface Builder with the following steps:

1. Drag a button from the Cocoa-Views pane of the Palette window to the target window.
2. Select the button.
3. Option-drag a resize handle of the button. As you drag, Interface Builder creates a `matrix` (page 280) containing multiple button cells.
4. Clicking once selects the matrix; double-clicking selects a button cell within the matrix.

Properties of button cell objects

In addition to the properties it inherits from the `cell` (page 256) class, a button cell object has these properties:

`alternate image`

Access: read/write
Class: *image* (page 72)
the image for the cell when it is in its alternate state; see the Discussion section of the `button` (page 246) class

`alternate title`

Access: read/write
Class: *Unicode text*
the title of the cell when it is in its alternate state see the Discussion section of the `button` (page 246) class

`bezel style`

Access: read/write
Class: *enumerated constant* from "Bezel Style" (page 179)
the bezel style of the cell

`button type`

Access: read/write
Class: *enumerated constant* from "Button Type" (page 180)
the button type of the cell

`highlights by`

Access: read/write
Class: *integer (enumeration; equivalent of Behavior for a button in IB*
the mechanism by which the button is highlighted; most applications shouldn't need to work with this property in their scripts and there are currently (through AppleScript Studio version 1.2) no AppleScript Studio enumerated constants defined for evaluating it; however, you can read about the Cocoa constants in the description for the NSCell class; you can set button behavior in Interface Builder using the Type and Behavior pop-ups in the Attributes pane of the Info window

`image dims when disabled`

Access: read/write
Class: *boolean*
Does the image dim when the button is disabled?

`key equivalent modifier`

Access: read/write
Class: *integer*
the modifier key for the key equivalent; see the description for this property in the `button` (page 246) class

`roll over`

Access: read/write
Class: *boolean*
Does the button cell act like a roll over?

`shows state by`

Access: read/write
Class: *integer*
not supported (through AppleScript Studio version 1.2); the way the button cell shows its state

`sound`

Access: read/write
Class: *sound* (page 81)

the sound played by the button cell when it is clicked

`transparent`

Access: read/write
Class: *boolean*
Is the button cell transparent?

Events supported by button cell objects

A button cell object supports handlers that can respond to the following events:

Action

`clicked` (page 337)

Nib

`awake from nib` (page 130)

Examples

AppleScript Studio applications typically script buttons, rather than scripting a button cell directly, and the `button` class has some of the same properties as the `button cell` class. In addition, the `button` (page 246) class inherits from the `control` (page 270) class, which has a `current cell` property, through which you can access properties of the button's button cell, if necessary. For example, you can use the following script in Script Editor to access the button cell of a push button in the main window of an AppleScript Studio application. Similar statements will work within an AppleScript Studio application script (though you won't need the `tell application` statement).

```
tell application "testApplication"
    -- check the "image dims when disabled" property:
    image dims when disabled of (current cell of first button of window 1)
        -- result: 1
    -- check transparency:
    transparent of (current cell of first button of window 1)
        -- result: 0
end tell
```

Version Notes

The `shows state by` property in this class is not supported, through AppleScript Studio version 1.2.

`cell`

Plural: `cells`
Inherits from: None.
Cocoa Class: NSCell

Provides a mechanism for displaying text or images in a view without the overhead of a full NSView subclass. Used by most `control` (page 270) classes to implement their internal workings. Cell subclasses include `action cell` (page 245), `button cell` (page 252), `image cell` (page 274), and `text field cell` (page 319).

For more information, see the Cocoa Programming Topic Controls and Cells.

Properties of cell objects

A `cell` object has these properties:

`alignment`

Access: read/write
Class: *enumerated constant* from "Text Alignment" (page 191)
the text alignment of the cell

`allows editing text attributes`

Access: read/write
Class: *boolean*
Can the text attributes be edited?

`allows mixed state`

Access: read/write
Class: *boolean*
Does the cell allow a mixed state? see the description for this property in the `button` (page 246) class

`associated object`

Access: read/write
Class: *item* (page 73)
the object associated with the cell

`bezeled`

Access: read/write
Class: *boolean*
Does the cell have a bezel?

`bordered`

Access: read/write
Class: *boolean*
Is the cell bordered?

`cell size`

Access: read only
Class: *point*
the size of the cell; the size is returned as a two-item list of numbers {horizontal, vertical}; for example, `{75, 19}` would indicate a width of 75 and a height of 19; see the `bounds` property of the `window` (page 86) class for information on the coordinate system

`cell type`

Access: read/write
Class: *enumerated constant* from "Cell Type" (page 182)
the type of cell

`content`

Access: read/write
Class: *item* (page 73)
the contents of the cell; synonymous with `contents`

`contents`

Access: read/write
Class: *item* (page 73)
the contents of the cell; synonymous with `content`

`continuous`

Access: read/write
Class: *boolean*
Does the cell generate actions as it is pressed?

`control size`

Access: read/write
Class: *enumerated constant* from "Control Size" (page 183)
the size of the control for the cell

`control tint`

Access: read/write
Class: *enumerated constant* from "Control Tint" (page 183)
the color of the control for the cell

`control view`

Access: read only
Class: *control* (page 270)
the control that the cell belongs to

`double value`

Access: read/write
Class: *real*
the value of the contents as a double; 0.0 if the contents cannot be interpreted as a double

`editable`

Access: read/write
Class: *boolean*
Is the cell editable?

`enabled`

Access: read/write
Class: *boolean*
Is the cell enabled?

`entry type`

Access: read/write
Class: *integer*
the entry type; the entry type is deprecated in Cocoa, so you should not use it in your scripts

`float value`

Access: read/write
Class: *real*
the value of the contents as a float; 0.0 if the contents cannot be interpreted as a float

`font`

Access: read/write
Class: *font* (page 67)
the font of the cell

`formatter`

Access: read/write
Class: *formatter* (page 68)
the formatter for the cell; this property is not supported (through AppleScript Studio version 1.2); however, see the Examples section of the `formatter` (page 68) class for a description of how to use the `call method` (page 102) command to get the formatter and extract information from it

`has valid object value`

Access: read only
Class: *boolean*
Does the cell contain a valid value? a valid object value is one that the cell's formatter (if one is present) can "understand"

`highlighted`

Access: read/write
Class: *boolean*
Is the cell highlighted?

`image`

Access: read/write
Class: *image* (page 72)
the image of the cell

`image position`

Access: read/write
Class: *enumerated constant* from "Cell Image Position" (page 181)
the position of the image in the cell

`imports graphics`

Access: read/write
Class: *boolean*
Should graphics be imported?

`integer value`

Access: read/write
Class: *integer*
the value of the contents of the cell as an integer; 0 if the contents cannot be interpreted as an integer

`key equivalent`

Access: read only
Class: *Unicode text*
the key equivalent for the cell; see the description for this property in the `button` (page 246) class

`menu`

Access: read/write
Class: *menu* (page 463)
the context menu for the cell, if any

`mouse down state`

Access: read only
Class: *integer*
the state of the mouse when it was clicked in the cell

`next state`

Access: read/write
Class: *integer*
the next state of the cell

`opaque`

Access: read only
Class: *boolean*
Is the cell opaque?

`scrollable`

Access: read/write
Class: *boolean*
Can the cell be scrolled?

`selectable`

Access: read/write
Class: *boolean*
Is the cell selectable?

`sends action when done editing`

Access: read/write
Class: *boolean*
Should the cell send its action when it is done editing? Cocoa applications typically connect user interface objects to action methods of a target object, but AppleScript Studio applications connect them to event handlers in a script; however, you cannot connect any event handlers to a `cell` object

`state`

Access: read/write
Class: *enumerated constant* from "Cell State Value" (page 181)
the state of the cell

`string value`

Access: read/write
Class: *Unicode text*
the value of the contents of the cell as text

`tag`

Access: read/write
Class: *integer*
the tag of the cell

`target`

Access: read/write
Class: *item* (page 73)
the target for the action of the cell; Cocoa applications typically connect user interface objects to action methods of a target object, but AppleScript Studio applications connect them to event handlers in a script; however, you cannot connect any event handlers to a `cell` object

`title`

Access: read/write
Class: *Unicode text*
the title of the cell

`wraps`

Access: read/write
Class: *boolean*
Does the cell wrap?

Commands supported by cell objects

Your script can send the following commands to a `cell` object:

`perform action` (page 326)

Events supported by cell objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

You typically script a subclass of `control` (page 270) or `view` (page 226) that contains a cell (or a subclass of cell), not the cell itself. One situation where you might access a cell itself is when working with a `matrix` (page 280). A matrix is used to create groups of cell objects, such as radio buttons.

The Assistant application, distributed with AppleScript Studio (starting with version 1.2) uses a matrix of radio buttons to specify the severity of a problem. It uses the following statement in its `updateValues` handler to get the title property of the currently selected radio button. This statement sets a property `severity` to the title of the current cell of the matrix (the matrix is also named "severity"):

```
set my severity to title of current cell of matrix "severity"
```

Version Notes

The `content` property was added in AppleScript Studio version 1.2. You can use `content` and `contents` interchangeably, with one exception. Within an event handler, `contents of theObject` returns a reference to an object, rather than the

actual contents. To get the actual contents of an object (such as the text contents from a `text field` (page 314)) within an event handler, you can either use `contents of contents of theObject` or `content of theObject`.

The `formatter` property in this class is not supported, through AppleScript Studio version 1.2.

For a sample script that shows the difference between `content` and `contents`, see the Version Notes section for the `control` (page 270) class.

`color well`

Plural: `color wells`
Inherits from: `control` (page 270)
Cocoa Class: NSColorWell

Supports selecting and displaying a single color value. A `color-panel` (page 476) uses a color well to display the current color selection. You typically work with colors through the `color panel` property of the `application` (page 43) class, not by working directly with a color well.

You will find the `color well` object on the Cocoa-Other pane of Interface Builder's Palette window. You can set attributes for color wells in Interface Builder's Info window.

For related information, see the Cocoa Programming Topic Using Color.

Properties of color well objects

In addition to the properties it inherits from the `control` (page 270) class, a color well object has these properties:

`active`

Access: read/write
Class: *boolean*
Is the color well active?

`bordered`

Access: read/write
Class: *boolean*
Does the color well have a border?

`color`

Access: read/write
Class: *RGB color*
the color in the well; a three-item integer list that contains the values for each component of the color; for example, red can be represented as {65535,0,0}

Events supported by color well objects

A color well object supports handlers that can respond to the following events:

Action

`clicked` (page 337)
Clicking a color well automatically opens a `color-panel` (page 476), and your `clicked` handler will only be called the first time the color well is clicked.

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)

scroll wheel (page 156)

Nib

awake from nib (page 130)

View

bounds changed (page 238)

Examples

You typically work with colors through the color panel property of the application (page 43) class, not by working directly with a color well. For examples, see color-panel (page 476).

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

combo box

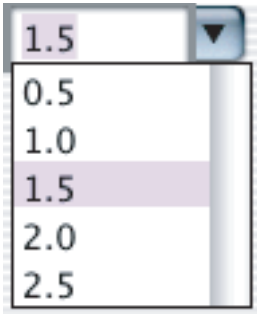
Plural: combo boxes
Inherits from: text field (page 314)
Cocoa Class: NSComboBox

A control that provides two ways to enter a value: through direct text entry (as with a text field), or by choosing from a pop-up list of pre-selected values. Figure 5-2 shows a combo box with its pop-up list hidden.

Figure 5-2 A combo box, with no pop-up list showing

Figure 5-2 shows the same combo box with its pop-up list displayed.

Figure 5-3 A combo box, with the pop-up list displayed



You will find the `combo box` object on the Cocoa-Other pane of Interface Builder's Palette window. You can set many attributes for combo boxes in Interface Builder's Info window. Note that through AppleScript Studio version 1.2, you cannot use a `data source` (page 371) object with a combo box.

For more information, see the Cocoa Programming Topic Combo Boxes.

Properties of combo box objects

In addition to the properties it inherits from the `text field` (page 314) class, a combo box object has these properties:

`auto completes`

Access: read/write
Class: *boolean*
Does the combo box use auto completion when typing? default is `false`; can be set in Info window in Interface Builder

`current item`

Access: read/write
Class: *integer*
the index of the current item, base 0

`data source`

Access: read/write
Class: *data source* (page 371)

not supported (through AppleScript Studio version 1.2); the data source for the combo box you can set this property in the Info window in Interface Builder, though the preferred mechanism is to set it in an application script, as shown in the Examples sections for the `append` (page 399) command and the `data item` (page 365) class

`has vertical scroller`

Access: read/write
Class: *boolean*
Does the combo box have a vertical scroll bar? default is `true`; can set Scrollable in Info window in Interface Builder

`intercell spacing`

Access: read/write
Class: *list*
the horizontal and vertical spacing between cells in the combo box's list; represented as a two-item list of numbers; default is {3,2}

`item height`

Access: read/write
Class: *real*
the height of an item

`uses data source`

Access: read/write
Class: *boolean*
not supported (through AppleScript Studio version 1.2); Does the combo box use a data source for its items? you can set this property in the Info window in Interface Builder

Elements of combo box objects

In addition to the elements it inherits from the `text field` (page 314) class, a `combo box` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`combo box item` (page 270)
Specify by: "Standard Key Forms" (page 30)
represent items in the combo box's pop-up list; stored as a simple list of text items

Commands supported by combo box objects

Your script can send the following commands to a combo box class object:

`scroll` (page 328)

Events supported by combo box objects

An object of class combo box supports handlers that can respond to the following events:

Action

`action` (page 334)

Combo Box

`selection changed` (page 339)
`selection changing` (page 341)
`will dismiss` (page 343)
`will pop up` (page 344)

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Editing

`begin editing` (page 335)
`changed` (page 336)
`end editing` (page 339)
`should begin editing` (page 341)
`should end editing` (page 342)

Key

`keyboard up` (page 141)

Mouse

mouse entered (page 146)
mouse exited (page 146)
scroll wheel (page 156)

Nib

awake from nib (page 130)

View

bounds changed (page 238)

Examples

You can access the items in the pop-up list of a combo box through its combo box item property. Given a combo box with AppleScript name "combo" in the front window, the following statement will return a simple list of text items, each representing one item in the pop-up list of the combo box:

```
every combo box item of combo box "combo" of window 1
```

The following lines will delete all the items in the combo box and add one new item:

```
delete every combo box item of combo box "combo" of window 1
make new combo box item at end of combo box items of combo box "combo" of
    window 1 with data "Test Item"
```

You can delete a combo box item by index with a statement like the following:

```
delete combo box item 2 of combo box 1 of window 1
```

See also the Examples section for the combo box item (page 270) class.

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

The following properties in this class are not supported, through AppleScript Studio version 1.2:

- data source

- uses data source

combo box item

Plural: combo box items
Inherits from: None.
Cocoa Class: ASKComboBoxItem

Represents one item in the pop-up list of a combo box. For related information, see combo box (page 265).

Events supported by combo box item objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

Given a combo box with AppleScript name "combo" in the front window, the following statement will return a simple list of text items, each representing one item in the pop-up list of the combo box:

```
every combo box item of combo box "combo"
```

See the Examples section for the combo box (page 265) class for information on creating and deleting combo box items.

Version Notes

The combo box item class was added in AppleScript Studio version 1.1.

control

Plural: controls
Inherits from: view (page 226)
Cocoa Class: NSControl

An abstract superclass that provides three fundamental features for implementing user-interface devices (such as buttons, scrollers, sliders, text fields, and so on): drawing devices onscreen, responding to user events, and sending action messages.

Works closely with `cell` (page 256) objects. Most applications won't often need to script a control directly—rather they script subclasses, such as `button` (page 246) or `text field` (page 314).

For more information, see the Cocoa Programming Topic Controls and Cells.

Properties of control objects

In addition to the properties it inherits from the `view` (page 226) class, a `control` object has these properties:

`alignment`

Access: read/write
Class: *enumerated constant* from "Text Alignment" (page 191)
the text alignment of the control

`cell`

Access: read/write
Class: *cell* (page 256)
the cell of the control

`content`

Access: read/write
Class: *item* (page 73)
the value of the control; synonymous with `contents`

`contents`

Access: read/write
Class: *item* (page 73)
the value of the control; synonymous with `content`

`continuous`

Access: read/write
Class: *boolean*
Does the control continuously generate actions?

`current cell`

Access: read only
Class: *cell* (page 256)
the current cell of the control

`current editor`

Access: read/write
Class: *text* (page 517) or *text view* (page 520)
not supported (through AppleScript Studio version 1.2); the current editor, if the control is being edited; if not, returns nil; see the `field editor` property of the `text` (page 517) class for a description of an editor

`double value`

Access: read/write
Class: *real*
the value of the control as a double; 0.0 if the contents cannot be interpreted as a double

`enabled`

Access: read/write
Class: *boolean*
Is the control enabled?

`float value`

Access: read/write
Class: *real*
the value of the control as a float; 0.0 if the contents cannot be interpreted as a float

`font`

Access: read/write
Class: *font* (page 67)
the font of the control

`formatter`

Access: read/write
Class: *formatter* (page 68)
the formatter of the control; this property is not supported (through AppleScript Studio version 1.2); however, see the Examples section of the `formatter` (page 68) class for a description of how to use the `call method` (page 102) command to get the formatter and extract information from it

`ignores multiple clicks`

Access: read/write
Class: *boolean*
Does the control ignore multiple clicks?

`integer value`

Access: read/write
Class: *integer*
the value of the control as an integer; 0 if the contents cannot be interpreted as a double

`string value`

Access: read/write
Class: *Unicode text*
the value of the control as text

`target`

Access: read/write
Class: *item* (page 73)
the target of the control's action

Elements of control objects

A `control` object can contain only the elements it inherits from `view` (page 226).

Commands supported by control objects

Your script can send the following commands to a `control` object:

`perform action` (page 326)

Events supported by control objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The `control` class is an abstract class that you don't typically target in your scripts. More often you target subclasses, such as `button` (page 246) or `slider` (page 306).

Version Notes

The `content` property was added in AppleScript Studio version 1.2. You can use `content` and `contents` interchangeably, with one exception. Within an event handler, `contents of theObject` returns a reference to an object, rather than the

actual contents. To get the actual contents of an object (such as the text contents from a `text field` (page 314)) within an event handler, you can either use `contents of contents of theObject` or `content of theObject`.

The `formatter` property in this class is not supported, through AppleScript Studio version 1.2.

The following listing of an `action` handler uses the log command to show the result of using `content` and `contents` on a text field that contains the text "Some text".

```
on action theObject
    log theObject -- result: text field 1 of window 1
    log contents of theObject -- result: text field 1 of window 1
    log contents of contents of theObject as string
        -- result: "Some text"
    log content of theObject as string
        -- result: "Some text"
end action
```

image cell

Plural: `image cells`
Inherits from: `cell` (page 256)
Cocoa Class: NSImageCell

Displays a single image in a frame. Provides properties for specifying the frame style and aligning and scaling the image. An image cell is usually associated with some kind of `control` (page 270) class, such as `image view` (page 276), `matrix` (page 280), or `table view` (page 386).

You can create and access an image cell in Interface Builder with the following steps:

1. Drag an image view from the Cocoa-Other pane of the Palette window to the target window.
2. Select the image view.
3. Option-drag a resize handle of the image view. As you drag, Interface Builder creates a `matrix` (page 280) containing multiple image cells.
4. Clicking once selects the matrix; double-clicking selects an image cell within the matrix.

For more information, see `image` (page 72), as well as the Cocoa topics Image Views, Matrices, and Table Views.

Properties of image cell objects

In addition to the properties it inherits from the `cell` (page 256) class, an image cell object has these properties:

`image alignment`

Access: read/write
Class: *enumerated constant* from "Image Alignment" (page 186)
the alignment of the image for the cell

`image frame style`

Access: read/write
Class: *enumerated constant* from "Image Frame Style" (page 187)
the frame style of the image

`image scaling`

Access: read/write
Class: *enumerated constant* from "Image Scaling" (page 187)
the scaling of the image

Events supported by image cell objects

An image cell object supports handlers that can respond to the following events:

Action

`clicked` (page 337)

Nib

`awake from nib` (page 130)

Examples

You don't typically script an image cell. You can script similar properties (`frame style`, `alignment`, `scaling`) of an `image view` (page 276) instead.

image view

Plural: image views
Inherits from: control (page 270)
Cocoa Class: NSImageView

Displays a single image in a frame, and can optionally allow a user to drag an image to it. Figure 5-4 shows an image displayed in an image view by the Image sample application, distributed with AppleScript studio.

You can store an image in your AppleScript Studio project by dragging an image file from the Finder into one of the groups in the Files list in Project Builder's Groups & Files pane, or by using the Add Files… command from the Project menu. You can also drag images into Images pane of a nib window in Interface Builder. You can display an image in an image view by loading it with load image (page 108) command. For related information, see image (page 72), as well as the Cocoa Programming Topic Image Views.

If you continually load images and don't free them, your application memory usage will increase. For information on how to free an image, movie (page 75), or sound (page 81), see the Discussion section of the load image (page 108) command.

Figure 5-4 A window displaying an image in an image view (from the Image sample application)

Properties of image view objects

In addition to the properties it inherits from the `control` (page 270) class, an `image view` object has these properties:

`editable`

Access: read/write
Class: *boolean*
Is the image view editable? default is `false`; you can set this property in the Info window in Interface Builder

`image`

Access: read/write
Class: *image* (page 72)
the image for the view; you can set this property in Interface Builder by dragging an image onto the image view

`image alignment`

Access: read/write
Class: *enumerated constant* from "Image Alignment" (page 186)
the alignment for the image; default is `center alignment`; you can set this property in the Info window in Interface Builder

`image frame style`

Access: read/write
Class: *enumerated constant* from "Image Frame Style" (page 187)
the frame style of the image; you can set this property in the Info window in Interface Builder

`image scaling`

Access: read/write
Class: *enumerated constant* from "Image Scaling" (page 187)
the scaling of the image; default is `scale proportionally`; you can set this property in the Info window in Interface Builder

Elements of image view objects

An `image view` object can contain only the elements it inherits from `control` (page 270).

Events supported by image view objects

An `image view` object supports handlers that can respond to the following events:

Action

`clicked` (page 337)

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

keyboard down (page 141)
keyboard up (page 141)

Mouse

mouse down (page 144)
mouse dragged (page 145)
mouse entered (page 146)
mouse exited (page 146)
mouse up (page 148)
right mouse down (page 154)
right mouse dragged (page 155)
right mouse up (page 156)
scroll wheel (page 156)

Nib

awake from nib (page 130)

View

bounds changed (page 238)

Examples

The following awake from nib (page 130) handler is from the Image sample application, distributed with AppleScript Studio. The handler simply uses the load image (page 108) command to load an image from the project. (The image is stored in the project's Resources group and is named AboutBox.tiff.)

```
on awake from nib theObject
    set image of image view "image" of window "main" to load image "AboutBox"
end awake from nib
```

If you continually load images and don't free them, your application memory usage will increase. For information on how to free an image (page 72), movie (page 75), or sound (page 81), see the Discussion section of the load image (page 108) command.

matrix

Plural: matrices
Inherits from: control (page 270)
Cocoa Class: NSMatrix

Used to create groups of cell objects, such as radio buttons, that work together in various ways. Figure 5-5 shows a matrix that contains three radio buttons.

Figure 5-5 A matrix with three radio buttons

You will find the matrix object (containing radio buttons) on the Cocoa-Views pane of Interface Builder's Palette window. You can set many attributes for matrix objects in Interface Builder's Info window. For information on creating matrix objects that contain other kinds of objects, see the scroll view (page 211) class, as well as the button cell (page 252), image cell (page 274), secure text field cell (page 304), and text field cell (page 319) classes.

For more information, see the Cocoa Programming Topic Matrices.

Properties of matrix objects

In addition to the properties it inherits from the control (page 270) class, a matrix object has these properties:

`allows empty selection`

Access: read/write
Class: *boolean*
Does the matrix allow an empty selection? default is false, for example, for a matrix of radio buttons; you can set this property in the Info window in Interface Builder

`auto scroll`

Access: read/write
Class: *boolean*
Should the matrix automatically scroll?

`auto sizes cells`

Access: read/write
Class: *boolean*
Should the matrix auto size its cells? default is `false` for a matrix of radio buttons; you can set this property in the Info window in Interface Builder

`background color`

Access: read/write
Class: *RGB color*
the background color of the matrix; a three-item integer list that contains the values for each component of the color; for example, red can be represented as {65535,0,0}; by default, {65535, 65535, 65535}, or white; you can set this property in the Info window in Interface Builder

`cell background color`

Access: read/write
Class: *RGB color*
the background color of the cells in the matrix; a three-item integer list that contains the values for each component of the color; by default, {65535, 65535, 65535}, or white

`cell size`

Access: read/write
Class: *point*
the size of each cell in the matrix; the size is returned as a two-item list of numbers {horizontal, vertical}; see the `bounds` property of the `window` (page 86) class for information on the coordinate system; not supported (through AppleScript Studio version 1.2); however, there is a workaround that uses the `call method` (page 102) command to get or set this property; the first statement below gets the size—the second statement sets it:

```
set cellSize to call method "cellSize" of matrix 1 of window 1
call method "setCellSize:" of matrix 1 of window 1 with parameter {200, 20}
```

`current cell`

Access: read/write
Class: *cell* (page 256)
the current cell

`current column`

Access: read/write
Class: *integer*
the one-based current column of the matrix

`current row`

Access: read/write
Class: *integer*
the one-based current row of the matrix

`draws background`

Access: read/write
Class: *boolean*
Should the matrix draw its background? default is `false` for a matrix of radio buttons; you can set this property in the Info window in Interface Builder

`draws cell background`

Access: read/write
Class: *boolean*
Should the cells of the matrix draw their background?

`intercell spacing`

Access: read/write
Class: *list*
the vertical and horizontal spacing between cells in the matrix; by default, both values are 1.0 in the coordinate system of the matrix (see the `bounds` property of the `window` (page 86) class for information on the coordinate system); not supported (through AppleScript Studio version 1.2); however, there is a workaround that uses the `call method` (page 102) command to get or set this property; the first statement below gets the spacing—the second statement sets it:

```
set spacing to call method "intercellSpacing" of matrix 1 of window 1
call method "setIntercellSpacing:" of matrix 1 of window 1 with parameter
{2.0, 2.0}
```

`key cell`

Access: read/write
Class: *cell* (page 256)
the key cell of the matrix

`matrix mode`

Access: read/write
Class: *enumerated constant* from "Matrix Mode" (page 187)
the mode of the matrix (for example, `radio mode`)

`next text`

Access: read/write
Class: *anything*
not supported (through AppleScript Studio version 1.2); use of the method in Cocoa's NSMatrix class that this property is based on is no longer encouraged, so this property is not likely to ever be supported; the next editor of the matrix; see the `field editor` property of the `text` (page 517) class for a description of an editor

`previous text`

Access: read/write
Class: *anything*
not supported (through AppleScript Studio version 1.2); use of the method in Cocoa's NSMatrix class that this property is based on is no longer encouraged, so this property is not likely to ever be supported; the previous editor of the matrix

`prototype cell`

Access: read/write
Class: *cell* (page 256)
the prototype cell of the matrix

`scrollable`

Access: read/write
Class: *boolean*
Is the matrix scrollable? not supported (through AppleScript Studio version 1.2); however, there is a workaround that uses the `call method` (page 102) command to set this property (though not to get it); note that you must pass a boolean value to the `call method` command as a one-item list:

```
call method "setScrollable:" of matrix 1 of window 1 with parameters {true}
```

`selection by rect`

Access: read/write
Class: *boolean*
Can cells be selected by rect? you can set this property in the Info window in Interface Builder

`tab key traverses cells`

Access: read/write
Class: *boolean*
Does the tab key traverse cells?

Elements of matrix objects

In addition to the properties it inherits from the `control` (page 270), a `matrix` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`cell` (page 256)
Specify by: "Standard Key Forms" (page 30)
the matrix's cells

Events supported by matrix objects

A `matrix` object supports handlers that can respond to the following events:

Action

`clicked` (page 337)

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

View

`bounds changed` (page 238)

Examples

The Drawer sample application distributed with AppleScript Studio uses a matrix that contains four radio buttons to specify the side on which its drawer should open (left, top, right, or bottom). The following statements, from the application's `awake from nib` (page 130) handler, set the current row of the matrix to indicate the currently selected radio button, based on edge information from the drawer.

```
on awake from nib theObject
    tell theObject
        set openOnEdge to edge of drawer "drawer"
        ...
        if openOnEdge is left edge then
            set current row of matrix "open on" to 1
        else if openOnEdge is top edge then
            set current row of matrix "open on" to 2
        else if openOnEdge is right edge then
```

```
            set current row of matrix "open on" to 3
        else if openOnEdge is bottom edge then
            set current row of matrix "open on" to 4
        end if
        ...
    end tell
end awake from nib
```

The following statements, from the same application's `clicked` (page 337) handler, show how to extract row information from the matrix, so that when the "drawer" button is clicked, the application knows on which side to open the drawer.

```
on clicked theObject
    tell window "main"
        if theObject is equal to button "drawer" then
            ...
            set openOnSide to current row of matrix "open on"
    ...
```

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

The following properties in this class are not supported, through AppleScript Studio version 1.2:

- `cell size`
- `intercell spacing`
- `next text`
- `previous text`
- `scrollable`

`movie view`

Plural: `movie views`
Inherits from: `view` (page 226)
Cocoa Class: NSMovieView

Displays a movie in a frame and provides properties associated with playing and editing the movie. You use the `load movie` (page 112) command to load a `movie` (page 75) for the movie view. For playing, stepping through, or jumping to a location in the movie, you use commands such as `start` (page 329), `stop` (page 331), `play` (page 327), `step forward` (page 330), `step back` (page 330), and `go` (page 323) (to jump to a specified location).

Figure 5-6 shows a movie displayed in a movie view in the Talking Head sample application, distributed with AppleScript studio.

You can store a movie in your AppleScript Studio project by dragging a movie file from the Finder into one of the groups in the Files list in Project Builder's Groups & Files pane, or by using the Add Files… command from the Project menu. You can display a movie in a movie view by loading it with `load movie` (page 112) command.

You can insert a `movie view` object into a nib file in Interface Builder by dragging the object represented by the QuickTime symbol from the Cocoa-GraphicsViews pane of the Palette window to the target window.

Figure 5-6 A movie view (from the Talking Head sample application)

Properties of movie view objects

In addition to the properties it inherits from the `view` (page 226) class, a `movie view` object has these properties:

`controller visible`

Access: read/write
Class: *boolean*
Is the controller visible? default is `true`; you can set this property in the Info window in Interface Builder

`editable`

Access: read/write
Class: *boolean*
Is the movie editable? default is `true`; you can set this property in the Info window in Interface Builder

`loop mode`

Access: read/write
Class: *enumerated constant* from "QuickTime Movie Loop Mode" (page 188)
the loop mode of the player; default is `normal`; you can set this property in the Info window in Interface Builder

`movie`

Access: read/write
Class: *movie* (page 75)
the movie for the view to play; you load the movie in your script with the `load movie` (page 112) command

`movie controller`

Access: read only
Class: *item* (page 73)
the QuickTime movie controller

`movie file`

Access: read/write
Class: *Unicode text*
the POSIX-style (slash-delimited) path to the movie file for the movie view; see the class description above for information on adding movies to your AppleScript Studio project

`movie rect`

Access: read/write
Class: *bounding rectangle*
the bounds of the movie in the view; a list of four numbers {left, bottom, right, top}; see the `bounds` property of the `window` (page 86) class for information on the coordinate system

`muted`

Access: read/write
Class: *boolean*
Is the movie muted? default is `false`

`playing`

Access: read/write
Class: *boolean*
Is the movie playing?

`plays every frame`

Access: read/write
Class: *boolean*
Should the movie play every frame? default is `false`; you can set this property in the Info window in Interface Builder

`plays selection only`

Access: read/write
Class: *boolean*
Should the player only play the selected portions of the movie? default is `false`; you can set this property in the Info window in Interface Builder

`rate`

Access: read/write
Class: *real*
the rate at which to play the movie

`volume`

Access: read/write
Class: *real*
the volume of the movie; for more information on volume, see the Examples section for the `slider` (page 306) class

Commands supported by movie view objects

Your script can send the following commands to a `movie view` object:

`copy` (from Cocoa's Core suite, described in Core Suites in the Cocoa Programming Topic Scriptable Applications)
`go` (page 323)
`play` (page 327)
`pause` (page 325)
`start` (page 329)
`step back` (page 330)
`step forward` (page 330)
`stop` (page 331)

Events supported by movie view objects

A `movie view` object supports handlers that can respond to the following events:

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)

drag updated (page 456)
drop (page 457)
prepare drop (page 459)

Key

keyboard down (page 141)
keyboard up (page 141)

Mouse

mouse down (page 144)
mouse dragged (page 145)
mouse entered (page 146)
mouse exited (page 146)
mouse up (page 148)
right mouse down (page 154)
right mouse dragged (page 155)
right mouse up (page 156)
scroll wheel (page 156)

Nib

awake from nib (page 130)

View

bounds changed (page 238)

Examples

The following will open (page 170) handler is from the Talking Head sample application, distributed with AppleScript Studio. The handler simply uses the load movie (page 112) command to load a movie from the project when the window that contains a movie view is opened. (The image is stored in the project's Resources group and is named jumps.mov.)

```
on will open theObject
    set movie of movie view "movie" of window "main" to load movie "jumps"
end will open
```

If you continually load movies and don't free them, your application memory usage will increase. For information on how to free an `image` (page 72), `movie` (page 75), or `sound` (page 81), see the Discussion section of the `load image` (page 108) command.

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

popup button

Plural: `popup buttons`
Inherits from: `button` (page 246)
Cocoa Class: NSPopUpButton

Provides access to a pop-up menu or a pull-down menu, from which a user can choose an item. Figure 5-7 shows a popup button displaying the first menu item in the pop-up menu.

Figure 5-7 A popup button

You will find the `popup button` object on the Cocoa-Other pane of Interface Builder's Palette window. You can set attributes for popup buttons in Interface Builder's Info window.

For related information, see the Cocoa Programming Topic Application Menus and Pop-up Lists.

Properties of popup button objects

In addition to the properties it inherits from the `button` (page 246) class, a `popup button` object has these properties:

`auto enables items`

Access: read/write
Class: *boolean*
Should the items in the menu be automatically enabled? default is `true`; you can set this property in the Info window in Interface Builder

`current menu item`

Access: read/write
Class: *menu item* (page 465)
the currently chosen menu item

`preferred edge`

Access: read/write
Class: *enumerated constant* from "Rectangle Edge" (page 188)
the preferred edge to present the menu under severe screen position restrictions

`pulls down`

Access: read/write
Class: *boolean*
Does the menu pull down? default is `false`; you can set this property in the Info window in Interface Builder

Elements of popup button objects

In addition to the elements it inherits from the `button` (page 246) class, a `popup button` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`menu` (page 463)
Specify by: "Standard Key Forms" (page 30)
the popup button's submenus

`menu item` (page 465)
Specify by: "Standard Key Forms" (page 30)
the popup button's menu items

Commands supported by popup button objects

Your script can send the following commands to a popup button:

`synchronize` (page 332)

Events supported by popup button objects

An popup button supports handlers that can respond to the following events:

Action

`action` (page 334)

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Menu

`choose menu item` (page 470)
`will pop up` (page 344)

Mouse

`mouse entered` (page 146)
`mouse exited` (page 146)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

View

`bounds changed` (page 238)

Examples

The following statement, from the `awake from nib` (page 130) handler of the Task List sample application (available starting with AppleScript Studio version 1.2) sets the title of a popup button with AppleScript name “priority” to “3”.

```
set title of popup button "priority" to "3"
```

You remove menu items from a popup button with the `delete` command and insert them with the `make` command. For example, in the file `Converter.applescript` of the Unit Converter sample application distributed with AppleScript Studio, you'll find the `updateUnitTypes` handler. This handler shows how to delete all the items in a popup menu and replace them with new items when you need to update the items in the menu.

```
on updateUnitTypes()
    tell box 1 of window "Main"
        -- Delete all of the menu items from the pop-ups
        delete every menu item of menu of popup button "From"
        delete every menu item of menu of popup button "To"

        -- Add each of the unit types as menu items to both of the pop-ups
        repeat with i in my unitTypes
            make new menu item at the end of menu items of menu of
                popup button "From"
                with properties {title:i, enabled:true}
            make new menu item at the end of menu items of menu
                of popup button "To"
                with properties {title:i, enabled:true}
        end repeat
    end tell
end updateUnitTypes
```

You can get the current menu item in a popup button with a statement like the following:

```
current menu item of popup button 1 of window 1
```

You can set the current menu item in a popup button with a statement like the following:

```
set current menu item of popup button 1 of window 1 to menu item 2 of menu of
popup button 1 of window 1
```

You can delete a named menu item from a named popup button with a statement like the following:

```
delete menu item "Shrink It" of menu of popup button "Do Laundry" of window
"Laundromat"
```

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

Starting with AppleScript Studio version 1.2, when referring to the menu of a popup button, a script can say `menu item 1 of popup button 1` instead of the longer `menu item 1 of menu of popup button 1`, although the longer form still works.

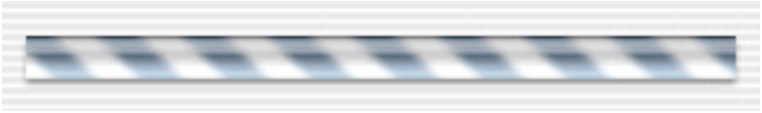
progress indicator

Plural: `progress indicators`
Inherits from: `view` (page 226)
Cocoa Class: NSProgressIndicator

A progress indicator (also known as a progress bar) provides a standard mechanism for user feedback. Combined with a `text field` (page 314), it can provide both determinate (the total time is known and the bar moves from left to right proportional to the percentage of the task completed) and indeterminate (the total time is not known, and a striped cylinder or circular image spins continually) progress bars, as well as text messages.

Figure 5-8 shows an indeterminate progress indicator.

Figure 5-8 An indeterminate progress indicator



You will find the progress indicator on the Cocoa-Other pane of Interface Builder's Palette window. Starting with the version of Interface Builder released with Mac OS X version 10.2, you can also choose the circular indeterminate progress indicator shown in Figure 5-8. This type of progress indicator is often used for activity such as connecting to a network.

You can set attributes for progress indicators in Interface Builder's Info window.

Figure 5-9 A circular indeterminate progress indicator

Properties of progress indicator objects

In addition to the properties it inherits from the `view` (page 226) class, a progress indicator object has these properties:

`animation delay`

Access: read/write
Class: *integer*
the amount to delay between animations default is 0

`bezeled`

Access: read/write
Class: *boolean*
Is the progress indicator bezeled? default is `false`

`content`

Access: read/write
Class: *real*

the value of the progress indicator; synonymous with `contents`

`contents`

Access: read/write
Class: *real*
the value of the progress indicator; synonymous with `content`

`control size`

Access: read/write
Class: *enumerated constant* from "Control Size" (page 183)
the size of the progress indicator; default is `regular size`; you can set this property in the Info window in Interface Builder

`control tint`

Access: read/write
Class: *enumerated constant* from "Control Tint" (page 183)
the tint of the progress indicator; default is `default tint`

`indeterminate`

Access: read/write
Class: *boolean*
Is the value of the progress indicator indeterminate? (an indeterminate progress indicator spins; a determinate progress indicator shows the percentage completed for the task); you can set this property in the Info window in Interface Builder

`maximum value`

Access: read/write
Class: *real*
the maximum value of the progress indicator; default is 100.0; you can set this property in the Info window in Interface Builder

`minimum value`

Access: read/write
Class: *real*
the minimum value of the progress indicator; default is 0.0; you can set this property in the Info window in Interface Builder

`uses threaded animation`

Access: read/write
Class: *boolean*
Should animation of this progress indicator be performed in a separate thread? if the application becomes multithreaded as a result of an invocation of this, the application's performance could become noticeably slower; default is `false`

Commands supported by progress indicator objects

Your script can send the following commands to a progress indicator object:

`animate` (page 322)
`increment` (page 324)
`start` (page 329)
`stop` (page 331)

Events supported by progress indicator objects

A progress indicator object supports handlers that can respond to the following events:

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)

scroll wheel (page 156)

Nib

awake from nib (page 130)

View

bounds changed (page 238)

Examples

For brief examples of commands you use with progress indicators, see the animate (page 322), increment (page 324), start (page 329), and stop (page 331) commands.

The following openPanel handler is from the Mail Search sample application, distributed with AppleScript Studio. (Prior to AppleScript Studio version 1.1, Mail Search was named Watson.) This handler is one of several handlers that are part of a script object defined in the Mail Search application to control the status panel window.

The handler has one parameter, statusMessage, which supplies a message to display in the status panel. It also relies on global and script properties, including initialized and statusPanelNibLoaded.

If the status panel window hasn't already been loaded, the handler loads it, sets the state of the window's progress indicator, tells the indicator to start, and sets a text message for the status panel. It then uses the display (page 496) command to display the status panel (and its progress indicator) attached to another window.

```
on openPanel(statusMessage)
    if initialized is false then
        if not statusPanelNibLoaded then
            load nib "StatusPanel"
            set statusPanelNibLoaded to true
        end if
        tell window "status"
            set indeterminate of progress indicator "progress"
                to true
            tell progress indicator "progress" to start
            set contents of text field "statusMessage"
                to statusMessage
```

```
            end tell
            set initialized to true
        end if
        display window "status" attached to theWindow
    end openPanel
```

The Mail Search application also demonstrates a determinate progress indicator. Look for the `makeStatusPanel` handler, which creates and returns a script object with handlers to load a status panel containing the progress indicator, set its minimum and maximum values, and tell it to increment. The Mail Search Application is described in detail in *Inside Mac OS X: Building Applications With AppleScript Studio* (see "Related Documentation" (page 19) for information on that document).

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

The `content` property was added in AppleScript Studio version 1.2. You can use `content` and `contents` interchangeably, with one exception. Within an event handler, `contents of theObject` returns a reference to an object, rather than the actual contents. To get the actual contents of an object (such as the text contents from a `text field` (page 314)) within an event handler, you can either use `contents of contents of theObject` or `content of theObject`.

For a sample script that shows the difference between `content` and `contents`, see the Version Notes section for the `control` (page 270) class.

Starting with AppleScript Studio version 1.2, the `display` command is preferred over the `display panel` command.

Starting with the version of Interface Builder released with Mac OS X version 10.2, you can use the circular indeterminate progress indicator shown in Figure 5-8.

Prior to AppleScript Studio version 1.1, the Mail Search sample application was named Watson.

secure text field

Plural: `secure text fields`
Inherits from: `text field` (page 314)
Cocoa Class: NSSecureTextField

Hides its text from display or other access via the user interface, and is thus suitable for use as a password-entry object, or for any item in which a secure value must be kept.

A user can drop text into a secure text field, but it will be displayed according to the current settings of the field—either as bullets as blank characters. The default is to display text as bullets. For information on how to specify blanks, see the Examples section of the `secure text field cell` (page 304) class.

Figure 5-7 shows a secure text field with several bullets visible. You can create a secure text field in Interface Builder with the following steps:

1. Drag a text field from the Cocoa-Views pane of the Palette window to the target window.
2. Select the text field.
3. In the Custom Class pane of the Info window, select NSSecureTextField as the class type.

Figure 5-10 A secure text field showing several bullets

For more information, see the class descriptions for `text field cell` (page 319) and `secure text field cell` (page 304), as well as the Cocoa Programming Topic Text Fields.

Properties of secure text field objects

A `secure text field` object has only the properties it inherits from `text field` (page 314).

Elements of secure text field objects

A `secure text field` object can contain only the elements it inherits from `text field` (page 314).

Events supported by secure text field objects

A `secure text field` object supports handlers that can respond to the following events:

Action

`action` (page 334)

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Editing

`begin editing` (page 335)
`changed` (page 336)
`end editing` (page 339)
`should begin editing` (page 341)
`should end editing` (page 342)

Key

`keyboard up` (page 141)

Mouse

`mouse entered` (page 146)
`mouse exited` (page 146)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

View

`bounds changed` (page 238)

Examples

A secure text field has no scriptable properties or elements of its own. However, it inherits all of the properties and elements of `text field` (page 314). See also `secure text field cell` (page 304).

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

`secure text field cell`

Plural: `secure text field cells`
Inherits from: `text field cell` (page 319)
Cocoa Class: NSSecureTextFieldCell

Works with a secure text field object to provide a text field whose value is guarded from user examination. Provides its own field editor, which doesn't display text or allow a user to Cut, Copy, or Paste its value. However, you can drag and drop text into a secure text field.

For more information, see `secure text field` (page 301), as well as the Cocoa Programming Topic Text Fields.

You can create and access a secure text field cell in Interface Builder by the following steps:

1. Use the steps described in the `secure text field` (page 301) class to create a secure text field.
2. Select the secure text field.
3. Option-drag a resize handle of the secure text field. As you drag, Interface Builder creates a `matrix` (page 280) containing multiple text field cells.
4. Clicking once selects the matrix; double-clicking selects a text field cell within the matrix. For each cell:
   a. select the cell
   b. open the Info window and use the pop-up menu to display the Custom Class pane
   c. change the class of the cell to NSSecureTextFieldCell

Properties of secure text field cell objects

In addition to the properties it inherits from the `text field cell` (page 319) class, a `secure text field cell` object has these properties:

`echos bullets`

Access: read/write
Class: *boolean*
Should the characters be echoed as bullets? default is `true`; if set to `false`, shows blank characters

Elements of secure text field cell objects

A `secure text field cell` object can contain only the elements it inherits from `text field cell` (page 319).

Examples

The following script statement changes the `echos bullets` property of a secure text field cell. By default, text characters are displayed as bullets. When bullets are turned off, characters are displayed as blanks.

```
set echos bullets of cell of secure text field 1 of window 1 to false
```

slider

Plural: sliders
Inherits from: control (page 270)
Cocoa Class: NSSlider

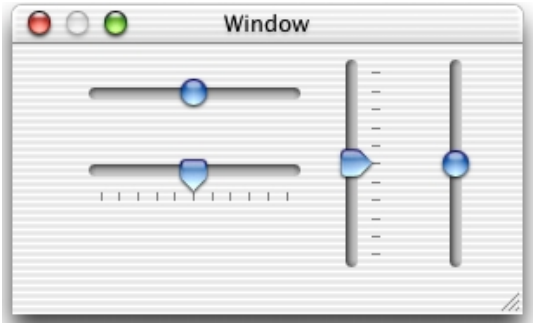
Displays a range of values and has an indicator, or knob, which indicates the current setting. A slider can optionally have tick marks at regularly spaced intervals. The user moves the knob along the slider's bar to change the setting.

The slider automatically determines whether it's horizontal or vertical by its size. If the slider is wider than it is tall, it's horizontal. Otherwise, it's vertical. A vertical slider has its minimum on the bottom; a horizontal slider has its minimum on the left.

You will find the several kinds of slider objects on the Cocoa-Other pane of Interface Builder's Palette window. You can set many attributes for sliders in Interface Builder's Info window.

Figure 5-11 shows vertical and horizontal sliders with different kinds of knobs, and with and without tick marks. You will find these types of sliders on the Cocoa-Other pane in Interface Builder's Palette window.

Figure 5-11 Horizontal and vertical sliders, with and without tick marks



For more information, see the Cocoa Programming Topic Sliders.

Properties of slider objects

In addition to the properties it inherits from the `control` (page 270) class, a `slider` object has these properties:

`alternate increment value`

Access: read/write
Class: *real*
the alternate increment value is the amount the slider will change its value when the user drags the knob with the Option key held down (or Alt key on some keyboards)

`image`

Access: read/write
Class: *image* (page 72)
the image for the slider; setting a slider's image in Cocoa is deprecated, so you shouldn't set the image in a script either

`knob thickness`

Access: read/write
Class: *real*
the thickness of the knob

`maximum value`

Access: read/write
Class: *real*
the maximum value of the slider; default is 100.0; you can set this property in the Info window in Interface Builder

`minimum value`

Access: read/write
Class: *real*
the minimum vale of the slider; default is 0.0; you can set this property in the Info window in Interface Builder

`number of tick marks`

Access: read/write
Class: *integer*
the number of tick marks for the slider; you can set this property in the Info window in Interface Builder; some sliders have no tick marks; for others, the default number is 11

`only tick mark values`

Access: read/write
Class: *boolean*
Should only values be allowed that correspond to the tick marks? you can set this property in the Info window in Interface Builder; for sliders that have no tick marks, the default value is `false`; for sliders with tick marks, the default value is `true`

`tick mark position`

Access: read/write
Class: *enumerated constant* from "Tick Mark Position" (page 192)
the position of the tick marks (above or below a horizontal slider, to the left or right of a vertical slider); you can set this property in the Info window in Interface Builder

`title`

Access: read/write
Class: *Unicode text*
the title of the slider; setting a slider's title in Cocoa is deprecated, so you shouldn't set the title in a script either

`title cell`

Access: read/write
Class: *text field cell* (page 319)
the cell of the title; as with the `title` property, you should not use this property in your scripts

`title color`

Access: read/write
Class: *RGB color*
the color of the title; as with the `title` property, you should not use this property in your scripts

`title font`

Access: read/write
Class: *font* (page 67)
the font of the title; as with the `title` property, you should not use this property in your scripts

`vertical`

Access: read/write
Class: *boolean*
Is the slider vertically oriented? you can choose vertical or horizontal sliders in Interface Builder

Elements of slider objects

A `slider` object can contain only the elements it inherits from `control` (page 270).

Events supported by slider objects

A `slider` object supports handlers that can respond to the following events:

Action

`action` (page 334)

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)

`right mouse dragged` (page 155)
`right mouse up` (page 156)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

View

`bounds changed` (page 238)

Examples

When you add a `slider` object to a window in Interface Builder, you can set various attributes of the slider, such as its minimum, maximum, and current (or start) values, and its number of markers (or tick marks). The following `action` (page 334) handler is from the Language Translator application distributed with AppleScript Studio. For that application, the slider is set to allow a range of 0.0 to 7.0, which corresponds to the volume level range you can set with the `set volume` scripting addition (where 0.0 is silent and 7.0 is full volume).

This `action` handler, which is called whenever a user changes the slider setting, gets the slider from the window of the passed `theObject` parameter. It then gets a volume value based on the current slider setting and calls the `set volume` scripting addition to adjust the volume at which translated text will be spoken.

```
on action theObject
    set enabled of slider of window "Language Translator" to true
    set volumevalue to contents of slider "volumeslider"
        of window "Language Translator" as integer
    set volume volumevalue
end action
```

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

stepper

Plural: `steppers`

Inherits from: `control` (page 270)
Cocoa Class: NSStepper
A control consisting of two small arrows that can increment and decrement a value that appears beside it, such as a date or time. Figure 5-12 shows a stepper to the right of a text field to display the stepper's value.

Figure 5-12 A stepper

You will find the `stepper` object on the Cocoa-Other pane in Interface Builder's Palette window. You can set many attributes for steppers in Interface Builder's Info window.

For more information, see the Cocoa Programming Topic Steppers.

Properties of stepper objects

In addition to the properties it inherits from the `control` (page 270) class, a `stepper` object has these properties:

`auto repeat`

Access: read/write
Class: *boolean*
Should the stepper auto repeat when clicked? default is `true`; you can set this property in the Info window in Interface Builder

`increment value`

Access: read/write
Class: *real*
the amount to increment the stepper by; default is 1.0; you can set this property in the Info window in Interface Builder

`maximum value`

Access: read/write
Class: *real*

the maximum value of the stepper; default is 59.0; you can set this property in the Info window in Interface Builder

`minimum value`

Access: read/write
Class: *real*
the minimum value of the stepper; default is 0.0; you can set this property in the Info window in Interface Builder

`value wraps`

Access: read/write
Class: *boolean*
Does the value of the stepper wrap? if `true`, then when incrementing or decrementing, the value will wrap around to the minimum or maximum value; if false, the value will stay pinned at the minimum or maximum; default is `true`; you can set this property in the Info window in Interface Builder

Elements of stepper objects

A `stepper` object can contain only the elements it inherits from `control` (page 270).

Events supported by stepper objects

A `stepper` object supports handlers that can respond to the following events:

Action

`clicked` (page 337)

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Key

`keyboard down` (page 141)
`keyboard up` (page 141)

Mouse

`mouse down` (page 144)
`mouse dragged` (page 145)
`mouse entered` (page 146)
`mouse exited` (page 146)
`mouse up` (page 148)
`right mouse down` (page 154)
`right mouse dragged` (page 155)
`right mouse up` (page 156)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

View

`bounds changed` (page 238)

Examples

When you add a `stepper` object to a window in Interface Builder, you can set various attributes of the stepper, such as its minimum, maximum, current (or start), and increment amount values. The following `clicked` (page 337) handler is from the Drawer application distributed with AppleScript Studio. For that application, the stepper that controls leading offset for the drawer is set to allow a range of 0.0 to 1000.0, with an increment amount of 1.0.

This `clicked` handler, which is called whenever a user clicks the stepper, checks the passed `theObject` parameter to determine which object it was called for. If it was the leading offset stepper, it gets a leading value based on the current stepper setting and sets the leading offset property of the drawer object. Finally, it sets the value displayed in its associated text field (note that it doesn't have to convert the value to a string to set the contents of the text field).

```
on clicked theObject
    tell window "main"
```

```
        if theObject is equal to button "drawer" then
    ...
        else if theObject is equal to stepper "leading offset" then
            set theValue to (contents of stepper "leading offset")
                as integer
            set leading offset of drawer "drawer" to theValue
            set contents of text field "leading offset" to theValue
    ...
```

The following script statements, from the `action` (page 334) handler of the Drawer application, show how to set the value of the stepper from a value entered in its associated text field.

```
on action theObject
    set textValue to contents of theObject
    ...
    if theObject is equal to text field "leading offset" then
        set leading offset of drawer "drawer" to textValue
        set contents of stepper "leading offset" to textValue
...
```

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

text field

Plural: `text fields`
Inherits from: `control` (page 270)
Cocoa Class: NSTextField

Provides the ability to input, display, and edit text, as well as to display non-editable text that can be used for labels.

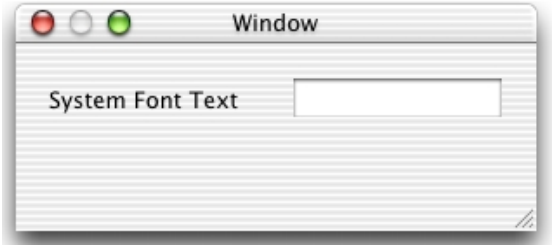
You will find the `text field` object on the Cocoa-Views pane in Interface Builder's Palette window. You can set many attributes for text fields in Interface Builder's Info window. For information on setting set font, color, and other attributes for a text field, see the Examples section for the `font` (page 67) class.

You can connect an `action` handler to a `text field` object in Interface Builder to gain control when a user finishes editing (either by tabbing to another field or by pressing the Enter key). In Interface Builder, you can also optionally set the text field to only call its `action` handler when the Enter key is pressed (and not when a user tabs out of the field).

AppleScript Studio applications automatically support tabbing between any text fields you add to a window. By default, tab order goes from left to right and down across the window, and is independent of the order in which you insert the text fields. See "Enabling Tabbing Between Objects" in Interface Builder help for a description of how to set tab order between text fields and other objects; you can specify the first object to receive keyboard events (or initial first responder), and each successive object in the responder chain.

Figure 5-13 shows a text field label ("System Font Text") and an input text field. For more information, see the Cocoa Programming Topic Text Fields.

Figure 5-13 Text fields used as labels and input fields



Properties of text field objects

In addition to the properties it inherits from the `control` (page 270) class, a `text field` object has these properties:

`allows editing text attributes`

Access: read/write
Class: *boolean*
Can the user edit the font attributes of the text field? default is `false`

`background color`

Access: read/write
Class: *RGB color*
the background color of the text field; a three-item integer list that contains the values for each component of the color; by default, {65535, 65535, 65535}, or white; you can set this property in the Info window in Interface Builder; for example, red can be represented as {65535,0,0}

`bezeled`

Access: read/write
Class: *boolean*
Is the text field bezeled? default is `true`

`bordered`

Access: read/write
Class: *boolean*
Is the text field bordered? default is `true`; you can set this property in the Info window in Interface Builder

`draws background`

Access: read/write
Class: *boolean*
Should the text field draw its background color behind the text? default is `true`; you can set this property in the Info window in Interface Builder, though you will first have to set the border type to no border

`editable`

Access: read/write
Class: *boolean*
Is the text field editable? default is `true`; you can set this property in the Info window in Interface Builder

`imports graphics`

Access: read/write
Class: *boolean*
Should the text field support dropping dragged images? default is `false`

`next text`

Access: read/write
Class: *text field* (page 314)

not supported (through AppleScript Studio version 1.2); use of the method in Cocoa's NSMatrix class that this property is based on is no longer encouraged, so this property is not likely to ever be supported; the next text editor

`previous text`

Access: read/write
Class: *text field* (page 314)
not supported (through AppleScript Studio version 1.2); use of the method in Cocoa's NSMatrix class that this property is based on is no longer encouraged, so this property is not likely to ever be supported; the previous text editor

`selectable`

Access: read/write
Class: *boolean*
Can the contents of the text field be selected? default is `true`; you can set this property in the Info window in Interface Builder

`text color`

Access: read/write
Class: *RGB color*
the color of the text; a three-item integer list that contains the values for each component of the color; for example, red can be represented as {65535,0,0}; by default, {0, 0, 0}, or black; you can set this property in the Info window in Interface Builder

Elements of text field objects

A `text field` object can contain only the elements it inherits from `control` (page 270).

Events supported by text field objects

A `text field` object supports handlers that can respond to the following events:

Action

`action` (page 334)

Drag and Drop

`conclude drop` (page 452)

`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Editing

`begin editing` (page 335)
`changed` (page 336)
`end editing` (page 339)
`should begin editing` (page 341)
`should end editing` (page 342)

Key

`keyboard up` (page 141)

Mouse

`mouse entered` (page 146)
`mouse exited` (page 146)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

View

`bounds changed` (page 238)

Examples

Many of the sample applications distributed with AppleScript Studio show how to work with text fields. For example, the Currency Converter application (available starting with AppleScript Studio version 1.1) uses the following statement to get text from a text field. In this statement, the text is stored in the variable `theRate`, while `"rate"` specifies the AppleScript name for the text field. (You supply the AppleScript name in the Name field on the AppleScript pane of the Info window in Interface Builder.)

```
set theRate to contents of text field "rate"
```

The same application uses the following statement to set the text in another text field to show the result of the currency conversion:

```
set contents of text field "total" to theRate * theAmount
```

Note that the text field automatically displays the result as text.

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2. See the "Drag and Drop Suite" (page 448) for details. In particular, the description for the `conclude drop` (page 452) event handler provides information on supporting drag and drop for `text view` (page 520) and `text field` (page 314) objects.

The `next text` and `previous text` properties in this class are not supported, through AppleScript Studio version 1.2. Support for these properties is not likely to be added.

text field cell

Plural: `text field cells`
Inherits from: `cell` (page 256)
Cocoa Class: NSTextFieldCell

Adds to text-display capabilities of a cell object by providing ability to set the color of both the text and its background, as well as to specify whether the cell draws its background at all.

For related information, see `cell` (page 256), as well as the Cocoa Programming Topic Text Fields.

You can create and access a text field cell in Interface Builder by the following steps:

1. Drag a text field from the Cocoa-Views pane of the Palette window to the target window.
2. Select the text field.
3. Option-drag a resize handle of the text field. As you drag, Interface Builder creates a `matrix` (page 280) containing multiple text field cells.

4. Clicking once selects the matrix; double-clicking selects a text field cell within the matrix.

Properties of text field cell objects

In addition to the properties it inherits from the `cell` (page 256) class, a `text field cell` object has these properties:

`background color`

Access: read/write
Class: *RGB color*
the background color of the cell; a three-item integer list that contains the values for each component of the color; for example, red can be represented as {65535,0,0}; by default, {65535, 65535, 65535}, or white; you can set this property in the Info window in Interface Builder

`draws background`

Access: read/write
Class: *boolean*
Should the cell draw its background? you can set this property in the Info window in Interface Builder, though you will first have to set the border type to no border

`text color`

Access: read/write
Class: *RGB color*
the color of the text; a three-item integer list that contains the values for each component of the color; by default, {0, 0, 0}, or black; you can set this property in the Info window in Interface Builder

Events supported by text field cell objects

A `text field cell` object supports handlers that can respond to the following events:

Action

`clicked` (page 337)

Nib

awake from nib (page 130)

Examples

You don't typically script a text field cell. You can script similar properties (background color, draws background, text color) of a text field (page 314) instead. However, if you do need to access the properties of a text field cell, you can do so with statements like the following:

```
set textFieldCell to cell 1 of matrix 1
set myColor to background color of textFieldCell
set draws background of textFieldCell to true
```

Commands

Objects based on classes in the Control View suite support the following commands. (A command is a word or phrase a script can use to request an action.) To determine which classes support which commands, see the individual class descriptions.

`animate`

Advances the progress animation of an indeterminate progress indicator by one step.

Syntax

`animate`	*reference*	required

Parameters

reference
a reference to the progress indicator to animate

Examples

Given a `window` (page 86) "main" with a `progress indicator` (page 296) "barber pole" the following statement causes the progress indicator to advance by one step. To animate the progress bar continually, you can either use this statement repeatedly in a loop, or use `start` (page 329).

```
tell progress indicator "barber pole" of window "main" to animate
```

The following statement is equivalent:

```
animate progress indicator "barber pole" of window "main"
```

go

Jumps to the specified location in a movie (the direct object). For related information, see the `movie view` (page 286) class.

Syntax

`go`	*reference*	required
`to`	*enumerated constant*	optional

Parameters

reference
a reference to the movie in which the jump takes place

`to` *enumerated constant* from "Go To" (page 186)
the place in the movie to jump to

Examples

The following `choose menu item` (page 470) handler is from the Talking Head sample application, distributed with AppleScript Studio. The handler uses the tag value from the chosen `menu item` (page 465) in the Movie menu to determine which command to execute.

For the Go to Beginning menu item, the handler uses the `go to` command to jump to the beginning frame. You can set the tag value for a menu item in Interface Builder in the Attributes pane of the Info window. The tag values in this example are from the Talking Head application.

```
on choose menu item theMenuItem
    tell window "main"
        set theCommand to tag of theMenuItem
        if theCommand is equal to 1001 then
    ...
        else if theCommand is equal to 1006 then
            tell movie view "movie" to go to beginning frame
    ...
```

highlight

Not supported (through AppleScript Studio version 1.2). Do not use this command.

Syntax

`highlight` *reference* required

Parameters

reference
a reference to the object to highlight

increment

Increments the specified object by the specified amount, or by one if no amount is specified.

Syntax

`increment`	*reference*	required
`by`	*real*	optional

Parameters

reference
a reference to the object to increment

`by` *real*
the amount to increment by

Examples

The following script statements are from the `incrementPanel` handler of the Mail Search sample application, distributed with AppleScript Studio. This handler is one of several handlers that are part of a script object defined in the Mail Search application to control the status panel window.

```
...
    tell window "status"
        tell progress indicator "progress" to increment by 1
        ...
        end tell
...
```

These script statements target a determinate progress indicator in a window and use the `increment by` command to increment the progress indicator.

Version Notes

Prior to AppleScript Studio version 1.1, the Mail Search sample application was named Watson.

`pause`

Not supported (through AppleScript Studio version 1.2). Pauses the current activity.

Syntax

`pause`	*reference*	required

Parameters

reference
a reference to the object to pause

perform action

Tells the receiving object to perform its action (causes a handler to be executed). For example, you can tell an interface object, such as a `button` (page 246), to perform its `clicked` (page 337) handler, thus providing a way to directly script the user interface. Note, however, that calling the `clicked` handler will not provide the visual feedback a user would see if they actually clicked the button. For other limitations on scripting the user interface, see *Inside Mac OS X: Building Applications With AppleScript Studio*, available in Project Builder help, or at http://developer.apple.com/techpubs/macosx/CoreTechnologies/AppleScriptStudio/index.html.

Syntax

`perform action`	*reference*	required

Parameters

reference
a reference to the object that should perform its action

Examples

You can use the following script statements in Script Editor to cause the "Drawer" button in the Drawer application (an AppleScript Studio sample application) to perform its `clicked` handler, which will either open or close the drawer, depending on its current state. Similar statements will work within an AppleScript Studio application script (though you won't need the `tell application` statement).

```
tell application "Drawer"
    set theButton to button "Drawer" of window "main"
    tell theButton to perform action
```

```
end tell
```

Discussion

The `perform action` command typically does nothing unless the specified object has an action handler—a handler such as a `clicked` or `double-clicked` handler in the Action group in the Interface Builder Info window for the object. You can, however, use the `perform action` command with menu items, using syntax such as the following:

```
tell menu item 1 of menu 1 of main menu to perform action
```

play

Plays the object.The `play` command is supported by the `movie view` (page 286) class.

Syntax

`play`	*reference*	required

Parameters

reference
a reference to the movie to play

Examples

You can tell a a movie view to play with statements like the following:

```
tell window "main"
    tell movie view "movie" to play
```

For a complete example of working with movies, see the Talking Head application, distributed with AppleScript Studio.

resume

Not supported (through AppleScript Studio version 1.2). Resumes the previous activity.

Syntax

`resume`	*reference*	required

Parameters

reference
a reference to the object that should resume its previous activity

`scroll`

Scrolls the specified object. Not supported (through AppleScript Studio version 1.2). However, see the Examples section below for information about how to scroll text in a text view.

Syntax

`scroll`	*reference*	required
`item at index`	*integer*	optional
`to`	*enumerated constant*	optional

Parameters

reference
a reference to the object that should be scrolled

`item at index` *integer*
the index of the item to scroll to

`to` *enumerated constant* from "Scroll To Location" (page 189)
the location to scroll to

Examples

Although the `scroll` command is not supported through AppleScript Studio version 1.2, you can use the `call method` (page 102) command to scroll. For a text view, for example, you can use `call method` with the `scrollRangeToVisible:` method of the NSText class (the `text view` (page 520) class inherits from `text` (page 517), as NSTextView inherits from NSText).

For a simple window (page 86) "main" with a text view (page 520) "myText", in a scroll view (page 211) "scroller", the following statements will scroll to the bottom of the text view.

```
tell text view "myText" of scroll view "scroller" of window "main"
    set theText to contents
    set theLength to (length of theText)
    call method "scrollRangeToVisible:" of object it
            with parameter {theLength, theLength}
end tell
```

Substituting the following version of the call method statement would instead scroll to the top of the view:

```
    call method "scrollRangeToVisible:" of object it with parameter {0, 0}
```

start

Starts an object.

Various classes support the start command. For example, you can use it to start the animation of an indeterminate progress indicator (page 296), which causes the barber pole to start spinning. The start command does nothing for a determinate progress indicator.

You can also use the start command to play a movie in a movie view (page 286).

Syntax

start	*reference*	required

Parameters

reference
a reference to the object to start

Examples

Given a window (page 86) "main" with an indeterminate progress indicator "barber pole" the following statement causes the progress indicator to start its animation.

```
tell progress indicator "barber pole" of window "main" to start
```

The following statement is equivalent:

```
start progress indicator "barber pole" of window "main"
```

step back

Repositions the movie's play position to one frame before the current frame. If the movie is playing, the it will stop at the new frame.

Syntax

`step back`	*reference*	required

Parameters

reference
a reference to the `movie view` (page 286) object that receives the `step back` command

Examples

You can tell a a movie view to step back with statements like the following:

```
tell window "main"
    tell movie view "movie" to step back
```

For a complete example of working with movies, see the Talking Head application, distributed with AppleScript Studio.

step forward

Repositions the movie's play position to one frame after the current frame. If the movie is playing, the it will stop at the new frame.

Syntax

`step forward`	*reference*	required

Parameters

reference
a reference to the `movie view` (page 286) object that receives the `step forward` command

Examples

You can tell a a movie view to step forward with statements like the following:

```
tell window "main"
    tell movie view "movie" to step forward
```

For a complete example of working with movies, see the Talking Head application, distributed with AppleScript Studio.

stop

Stops an object.

Various classes support the `stop` command. For example, you can use it to stop the animation of an indeterminate `progress indicator` (page 296), which causes the barber pole to stop spinning. The `stop` command does nothing for a determinate progress indicator.

You can also use the stop command to stop playing a movie in a `movie view` (page 286).

Syntax

`stop`	*reference*	required

Parameters

reference
a reference to the progress indicator

Examples

Given a `window` (page 86) "main" with an indeterminate progress indicator "barber pole" the following statement causes the progress indicator to stop its animation.

```
tell progress indicator "barber pole" of window "main" to stop
```

You can tell a a movie view to stop with statements like the following:

```
tell window "main"
    tell movie view "movie" to stop
```

For a complete example of working with movies, see the Talking Head application, distributed with AppleScript Studio.

synchronize

Saves to disk any modifications that have been made to user defaults and updates any unmodified values to what is on disk. Not supported (through AppleScript Studio version 1.2), but see Examples section for a workaround.

Syntax

`synchronize`	*reference*	required

Parameters

reference
a reference to the `user-defaults` (page 83) that should be synchronized

Result

boolean
Returns `false` if it could not save data to disk; `true` otherwise.

Examples

You can use the `call method` (page 102) command to call `synchronize`, as follows. These calls return `false` if the data could not be saved to disk and `true` otherwise.

```
-- Works in Jaguar as well as in earlier versions:
set succeeded to call method "synchronize" of object user defaults
```

```
-- Works in Jaguar only
set succeeded to call method "synchronize" of user defaults
```

Events

Objects based on classes in the Control View suite support handlers for the following events (an event is an action, typically generated through interaction with an application's user interface, that causes a handler for the appropriate object to be executed). To determine which classes support which events, see the individual class descriptions.

- `action` (page 334)
- `begin editing` (page 335)
- `changed` (page 336)
- `clicked` (page 337)
- `double clicked` (page 338)
- `end editing` (page 339)
- `selection changed` (page 339)
- `selection changing` (page 341)
- `should begin editing` (page 341)
- `should end editing` (page 342)
- `will dismiss` (page 343)
- `will pop up` (page 344)

`action`

Called when an action occurs for the object. This handler gets its name from the Cocoa concept of an action—a method that can be triggered by user interface objects. In AppleScript Studio, action handlers include `action`, `clicked` (page 337), and `double clicked` (page 338).

The `action` event handler itself is supported by classes such as `combo box` (page 265), `popup button` (page 292), `secure text field` (page 301), `slider` (page 306), `stepper` (page 310), `text field` (page 314), and `text field cell` (page 319). For example, the action handler for a text field is typically is triggered when a user attempts to tab out of the field or presses the Return key. (You can set a Send Action radio for a text field in the Attributes pane of the Info window in Interface builder.)

You can use the `perform action` (page 326) command to cause an `action` event handler to be called.

Syntax

`action`	*reference*	required

Parameters

reference
a reference to the object that received the action

Examples

The following `action` handler is from the file `Application.applescript` in the Unit Converter sample application distributed with AppleScript Studio. This handler just checks whether the object for which the handler was called is a particular text field. If so, it calls another handler to perform the unit conversion (such as converting a value from yards to meters).

```
on action theObject
    if theObject is equal to text field "Value" of box 1 of window "Main" then
        my convert()
    end if
end action
```

begin editing

Called before editing begins. You typically use this handler with a `text field` (page 314), `text view` (page 520), or related object. The handler cannot cancel the editing operation, but can prepare for it.

For example, when a user first presses a key to enter a character into a `text field`, AppleScript Studio calls the `should begin editing` (page 341) handler (if one is installed), which can refuse to allow editing. Then, if editing is allowed, the `begin editing` handler (if installed). The character is not entered into the field until the application returns from the `begin editing` handler.

Continuing with this example, AppleScript Studio calls the `changed` (page 336) handler (if installed) after the character is entered. When the user presses the Tab key or otherwise attempts to complete editing in that field, AppleScript Studio calls

the `should end editing` (page 342) handler, which can refuse to allow editing to end (if, for example, the field contains an invalid entry). Then, if editing is allowed to end, AppleScript Studio calls the `end editing` (page 339) handler (if installed), where the application can perform any preparation for editing to be completed.

Syntax

`begin editing`	*reference*	required

Parameters

reference
a reference to the `text field` (page 314), `text view` (page 520), or related object for which editing will begin

Examples

When you connect a `begin editing` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for editing.

```
on begin editing theObject
    (* Perform operations here before editing. *)
end begin editing
```

changed

Called after the contents of an object change. Typically used to indicate editing has caused a change in the text in a `text field` (page 314), `text view` (page 520), or related object. It is too late to do validation when this handler is called—use `should end editing` (page 342) instead. See also `begin editing` (page 335), `should begin editing` (page 341), and `end editing` (page 339).

Syntax

`changed`	*reference*	required

Parameters

reference
a reference to object in which editing caused a change

Examples

When you connect a `changed` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to respond to any changes due to editing.

```
on changed theObject
    (* Perform operations here after object changed due to editing. *)
end changed
```

clicked

Called when a user clicks the object. Nearly all subclasses of `control` (page 270) support the `clicked` handler.

Syntax

`clicked`	*reference*	required

Parameters

reference
a reference to the object that was clicked

Examples

The following `clicked` handler is for the "Convert" button in the Currency Converter application distributed with AppleScript Studio. Currency Converter provides text fields for entering an amount and a conversion rate, a field for displaying the result, as well as the "Convert" button for initiating the conversion.

```
on clicked theObject
    tell window of theObject
        try
            set theRate to contents of text field "rate"
            set theAmount to contents of text field "amount" as number
            set contents of text field "total" to theRate * theAmount
        on error
            set contents of text field "total" to 0
        end try
    end tell
end clicked
```

Only one object (the "Convert" button) in the Currency Converter application supports the `clicked` handler, so in that handler the application knows the `theObject` parameter is a reference to the button. The handler gets the window of that object so it can access text fields in the same window.

double clicked

Called when a user double-clicks the object. Subclasses of `control` (page 270) such `outline view` (page 377) and `table view` (page 386) support the `double clicked` handler.

Syntax

`double clicked` *reference* required

Parameters

reference
a reference to the object that was double-clicked

Examples

The following `double clicked` handler is for the "messages" table view in the Mail Search application distributed with AppleScript Studio. The table view displays email messages that were found in a search. Double-clicking a selected message should result in opening the message in a document window.

```
on double clicked theObject
    set theController to controllerForWindow(window of theObject)
    if theController is not equal to null then
        tell theController to openMessages()
    end if
end double clicked
```

This handler uses the `theObject` parameter to get the window of the table object so it can access the controller for that window. The controller is responsible for carrying out operations for the window. If the handler can successfully get the controller, it calls the `openMessages` handler of the controller to actually open the selected message (or messages).

Version Notes

Prior to AppleScript Studio version 1.1, the Mail Search sample application was named Watson.

end editing

Called before editing ends. You typically use this handler with a `text field` (page 314), `text view` (page 520), or related object. The handler cannot cancel the end of editing, but can prepare for it.

For additional information, see `begin editing` (page 335).

Syntax

`end editing` *reference* required

Parameters

reference
a reference to the `text field` (page 314), text view, or related object for which editing will end

Examples

When you connect a `end editing` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for editing ending.

```
on end editing theObject
    (* Perform operations here before editing stops. *)
end end editing
```

selection changed

Called after the selection of an object changes. The handler can perform actions in response to the selection change. Typically used with data views, such as `browser` (page 349), `outline view` (page 377), or `table view` (page 386). For example, when a user clicks to select a new row in an outline view, the `selection changed` handler is called (if one is connected).

Syntax

`selection changed`	*reference*	required

Parameters

reference
a reference to the object whose `selection changed` handler is called

Examples

The following `selection changed` handler is from the file `Application.applescript` in the Task List sample application distributed with AppleScript Studio (starting with version 1.2). This event handler is called whenever the selection in a table view changes. If there are any currently selected rows, this handler calls the `setUIValuesWithTaskValues` handler to update the user interface based on the selected row. If no row is selected (selection changed to no selection), the handler calls the `setDefaultUIValues` handler to update the user interface to its default values.

```
on selection changed theObject
    if name of theObject is "tasks" then
        -- If there is a selection then we'll update the UI;
        --  otherwise we set the UI to default values
        if (count of selected rows of theObject) > 0 then
            -- Get the selected data row of the table view
            set theTask to selected data row of theObject

            -- Update the UI using the selected task
            setUIValuesWithTaskValues(window of theObject, theTask)
        else
            -- Set the UI to default values
            setDefaultUIValues(window of theObject)
        end if
    end if
end selection changed
```

selection changing

Called repeatedly during a drag select operation (such as dragging to select multiple rows in a table or outline view) as items are added or removed from the selection. The handler can perform actions in response to the changing selection, though it should not perform lengthy operations. When the user concludes the selection (by releasing the mouse), the `selection changed` (page 339) handler is called (if one is connected).

Syntax

`selection changing` *reference* required

Parameters

reference
a reference to the object whose `selection changing` handler is called

Examples

When you connect a `selection changing` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. The `theObject` parameter refers to the object, such as a `browser` (page 349), `outline view` (page 377), or `table view` (page 386), for which selection is changing. You can use the handler to perform actions in response to the changing selection.

```
on selection changing theObject
    -- Perform operations here in response to changing selection.
end selection changing
```

should begin editing

Called before editing starts. The handler can return `false` to cancel editing. Classes such as `text field` (page 314) and `text view` (page 520) support this handler.

See also `begin editing` (page 335) and `should end editing` (page 342).

Syntax

`should begin editing`	*reference*	required
`object`	*anything*	optional

Parameters

reference
a reference to the object whose `should begin editing` handler is called

`object` *anything*
the text view that will do the editing

Result

boolean
Return `true` to allow editing to begin; `false` to prevent editing

Examples

The following example of a `should begin editing` handler calls an application handler `isItemEditable`, written by you, to determine whether to allow editing to begin, then returns the appropriate value. You might instead perform some kind of test in the handler itself or check some global property.

```
on should begin editing theObject
    --Check property, perform test, or call handler to see if OK to edit
    set allowEditing to isItemEditable(theObject)
    return allowEditing
end should begin editing
```

should end editing

Called before editing ends. The handler can return `false` to cancel ending. Classes such as `text field` (page 314) and `text view` (page 520) support this handler. A common use is to not allow editing to end if the current text is invalid.

See also `end editing` (page 339) and `should begin editing` (page 341).

Syntax

`should end editing`	*reference*	required
`object`	*anything*	optional

Parameters

reference
a reference to the object whose `should end editing` handler is called

`object` *anything*
the text view that is doing the editing

Result

boolean
Return `true` to allow editing to end; `false` to continue editing

Examples

The following example of a `should end editing` handler calls an application handler `isItemValid`, written by you, to determine whether to allow editing to end (if the edited item is valid), then returns the appropriate value. You might instead perform validation in the handler itself or check some global property.

```
on should end editing theObject
    --Check property, perform test, or call handler to see if OK
    --to stop editing (if edited item is valid)
    set allowStopEditing to isItemValid(theObject)
    return allowStopEditing
end should end editing
```

will dismiss

Called before the object is dismissed. This command is supported by `combo box` (page 265) objects. The handler cannot cancel dismissing, but can prepare for it.

Syntax

`will dismiss`	*reference*	required

Parameters

reference
a reference to the `combo box` (page 265) object whose `will dismiss` handler is called

Examples

When you connect a `will dismiss` handler to a `combo box` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. You can use the handler to prepare for being dismissed.

```
on will dismiss theObject
    (* Perform any operations to prepare for being dismissed. *)
end will dismiss
```

`will pop up`

Called before a pop-up menu pops up. This command is supported by `combo box` (page 265) and `popup button` (page 292) objects. The handler cannot cancel popping up, but can prepare for it.

Syntax

`will pop up`	*reference*	required

Parameters

reference
a reference to the `combo box` (page 265) or `popup button` (page 292) object whose `will pop up` handler is called

Examples

When you connect a `will pop up` handler to a `combo box` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. The `theObject` parameter refers to the combo box or popup button that is about to pop up. You can use the handler to prepare for popping up (such as checking the items in the combo box or popup button).

```
on will pop up theObject
    (* Perform any operations to prepare for popping up. *)
end will pop up
```

Control View Suite

CHAPTER 6

Data View Suite

This chapter describes the terminology in AppleScript Studio's Data View suite. For other suites, see the following:

- "Application Suite" (page 42)
- "Container View Suite" (page 194)
- "Control View Suite" (page 244)
- "Document Suite" (page 430)
- "Drag and Drop Suite" (page 448)
- "Menu Suite" (page 462)
- "Panel Suite" (page 474)
- "Text View Suite" (page 516)

Data View Suite

The Data View suite defines classes whose primary purpose is to display rows and columns of data. Many classes in the Data View suite inherit from the `view` (page 226) class or the `cell` (page 256) class. The Data View suite defines events for working with the items, cells, rows, and columns found in table and outline views.

The classes, commands, and events in the Data View suite are described in the following sections:

For enumerated constants, see "Enumerations" (page 177).

Classes

The Data View suite contains the following classes:

browser

Plural: `browsers`
Inherits from: `control` (page 270)
Cocoa Class: NSBrowser

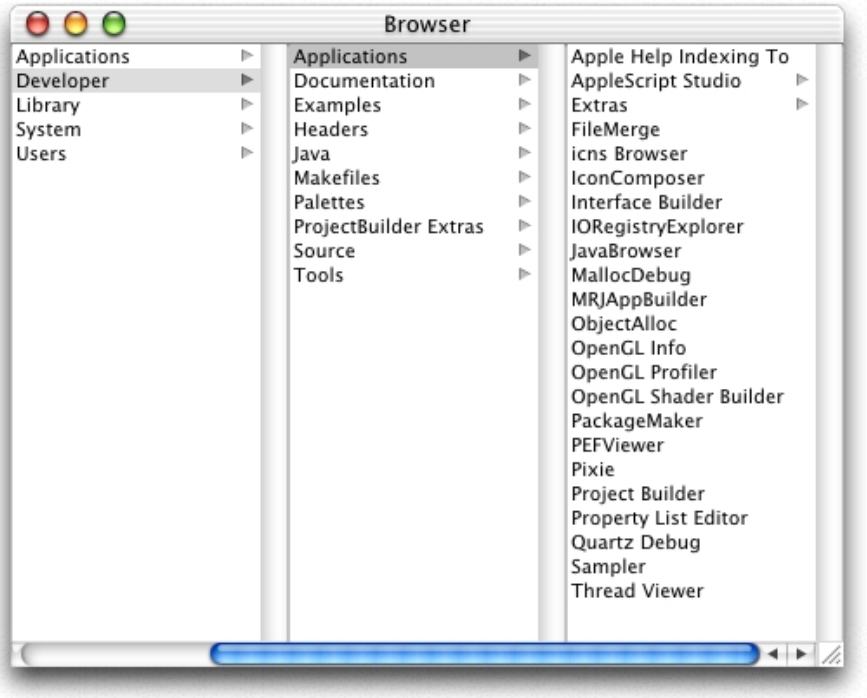
Provides a user interface for displaying and selecting items from a list of data, or from hierarchically organized lists of data such as directory paths. When working with a hierarchy of data, the levels are displayed in columns, which are numbered from left to right. Column numbers are zero-based.

You will find the `browser` object on the Cocoa-Data pane in Interface Builder's Palette window. You can set many attributes for browsers in Interface Builder's Info window.

Figure 6-1 shows a browser displaying the files in the `/Developer/Applications` directory. For more information, see `browser cell` (page 356) and `update` (page 126), as well as the Cocoa Programming Topic Browsers.

Note: Unlike other data views such as `outline view` (page 377) and `table view` (page 386), you cannot supply data to a browser view with a `data source` (page 371). As a result, performance may be inadequate for browser views that display more than a small number of items, and you should consider using one of the other data views, if suitable for your purpose.

Figure 6-1 A browser view displaying part of the file system



Properties of browser objects

In addition to the properties it inherits from the `control` (page 270) class, a `browser` object has these properties:

`accepts arrow keys`

Access: /write
Class: *boolean*
Does the browser accept input from arrow keys? default is `false`; you can set this property in the Info window in Interface Builder

`allows branch selection`

Access: read/write
Class: *boolean*
Does the browser allow selection of a branch item when multiple selection is enabled? each cell can be either a branch cell (such as a directory) or a leaf cell (such as a file); default is `false`; you can set this property in the Info window in Interface Builder

`allows empty selection`

Access: read/write
Class: *boolean*
Can there be an empty selection? default is `true`; you can set this property in the Info window in Interface Builder

`allows multiple selection`

Access: read/write
Class: *boolean*
Can there be multiple items selected? default is `false`; you can set this property in the Info window in Interface Builder

`cell prototype`

Access: read/write
Class: *browser cell* (page 356)
the prototype cell for the browser; this cell is copied to display items in the browser's columns

`displayed cell`

Access: read/write
Class: *browser cell* (page 356)
the cell being displayed in the browser

`first visible column`

Access: read/write
Class: *integer*

the zero-based index of the first visible column in the browser; see also the `maximum visible columns` property

`has horizontal scroller`

Access: read/write
Class: *boolean*
Does the browser have a horizontal scroll bar? default is `true`; you can set this property in the Info window in Interface Builder

`last column`

Access: read/write
Class: *integer*
the zero-based index of the last column in the browser

`last visible column`

Access: read/write
Class: *integer*
the zero-based index of the last currently visible column in the browser; see also the `maximum visible columns` property

`loaded`

Access: read only
Class: *boolean*
Is the data for the browser loaded? if `true`, all the information for the currently displayed columns has been acquired (a browser view calls the `will display browser cell` (page 425) event handler for each cell when it needs to display data in a column)

`maximum visible columns`

Access: read/write
Class: *integer*
the total number of visible columns; you can set the number of visible columns in the Info window in Interface Builder

`minimum column width`

Access: read/write
Class: *real*
the maximum width of a column

`path`

Access: read/write
Class: *Unicode text*
the path that represents the selected item (for example, when the browser is displaying files in a file system)

`path separator`

Access: read/write
Class: *Unicode text*
the string to use as the separator for the path; defaults to the slash character (/)

`reuses columns`

Access: read/write
Class: *boolean*
Should the browser reuse its columns? default is `true`, meaning the browser doesn't free underlying column objects when columns are unloaded

`selected cell`

Access: read/write
Class: *browser cell* (page 356)
the currently selected cell

`selected column`

Access: read/write
Class: *integer*
the zero-based index of the currently selected column

`send action on arrow key`

Access: read/write
Class: *boolean*
Should the browser perform actions when it receives arrow key input?

`separates columns`

Access: read/write
Class: *boolean*
Should the columns be separated by bezeled borders? default is `true`; you can set this property in the Info window in Interface Builder, but only if you have deselected the "Is titled" checkbox

`title height`

Access: read only
Class: *real*
the height of the title

`titled`

Access: read/write
Class: *boolean*
Does the browser use titles? you can set this property in the Info window in Interface Builder; not supported (through AppleScript Studio version 1.2); however, the following is a workaround that uses the `call method` (page 102) command:

```
set isTitled to call method "isTitled" of browser 1
```

`uses title from previous column`

Access: read/write
Class: *boolean*
Should the browser use the value of the previous column for the title of the next column? default is `true`

Elements of browser objects

In addition to the elements it inherits from the `control` (page 270) class, a `browser` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`cell` (page 256)
Specify by: "Standard Key Forms" (page 30)
the browser's data cells; may actually be of class `browser cell` (page 356)

Commands supported by browser objects

Your script can send the following commands to a `browser` object:

`path for` (page 120)
`update` (page 126)

Events supported by browser objects

A `browser` object supports handlers that can respond to the following events:

Action

clicked (page 337)

Browser View

number of browser rows (page 415)
will display browser cell (page 425)

Drag and Drop

conclude drop (page 452)
drag (page 453)
drag entered (page 454)
drag exited (page 455)
drag updated (page 456)
drop (page 457)
prepare drop (page 459)

Key

keyboard down (page 141)
keyboard up (page 141)

Mouse

mouse down (page 144)
mouse dragged (page 145)
mouse entered (page 146)
mouse exited (page 146)
mouse up (page 148)
right mouse down (page 154)
right mouse dragged (page 155)
right mouse up (page 156)
scroll wheel (page 156)

Nib

awake from nib (page 130)

View

`bounds changed` (page 238)

Examples

The following statement shows how to set the `path separator` property of a browser with AppleScript name “browser” in a window with name “main”.

```
set path separator of browser "browser" of window "main" to ":"
```

This statement is from the `launched` (page 142) handler of the Browser sample application distributed with AppleScript Studio. The full handler is listed in the example for the `update` (page 126) command.

Working with browser views is a complex task that can’t be covered fully here. The Browser application provides a complete but relatively simple example that browses the file system, displaying files and folders in a window similar to the Finder’s column view.

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

The `titled` property in this class is not supported, through AppleScript Studio version 1.2.

browser cell

Plural: `browser cells`
Inherits from: `cell` (page 256)
Cocoa Class: NSBrowserCell

Represents the default subclass of cell used to display data in the columns of a browser.

Each column in a browser contains a `matrix` (page 280) filled with browser cells. For more information, see `browser` (page 349), as well as the Cocoa Programming Topic Browsers.

Properties of browser cell objects

In addition to the properties it inherits from the `cell` (page 256) class, a browser cell has these properties:

`alternate image`

Access: read/write
Class: *image* (page 72)
an alternate image that should used for the highlighted state for the browser cell

`leaf`

Access: read/write
Class: *boolean*
Is this a leaf cell? each cell can be either a branch cell (such as a directory) or a leaf cell (such as a file); a branch browser cell has an image near its right edge indicating that more, nested information is available; a leaf browser cell has no image, indicating that the user has reached a terminal piece of information

`loaded`

Access: read/write
Class: *boolean*
Is the cell loaded? `true` if all the browser cell's state has been set and the cell is ready to display (a browser view calls the `will display browser cell` (page 425) event handler for each cell when it needs to display data in a column)

Events supported by browser cell objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The following is the `will display browser cell` (page 425) handler from the Browser sample application distributed with AppleScript Studio. The Browser application browses the file system, displaying files and folders in a window similar to the Finder's column view. This handler uses the Finder to get information about the browser cell to be displayed, then sets properties of the cell. See the Browser application for a complete example of working with a browser view, including the full text for this handler.

```
on will display browser cell theObject row theRow browser cell theCell in
column theColumn
```

```
    -- Code to set the values of the cellContents and isLeaf variables
    --      is not shown
    set string value of theCell to cellContents
    set leaf of theCell to isLeaf

end will display browser cell
```

Note that unlike other data views such as `outline view` (page 377) and `table view` (page 386), you cannot supply data to a browser view with a `data source` (page 371). As a result, performance may be inadequate for browser views that display more than a small number of items, and you should consider using one of the other data views, if suitable for your purpose.

data cell

Plural: `data cells`
Inherits from: None.
Cocoa Class: ASKDataCell

Represents one cell in a data row of a data source. The data cell stores contents for the cell, as well as other information for accessing its data.

You typically create a `data source` (page 371) to manage the data for an `outline view` (page 377) or `table view` (page 386). You then create each `data column` (page 361) and supply a name. This process is demonstrated in the Examples section for the `append` (page 399) command.

Next you create data rows (for a table view) or data items (for an outline view) for the data source. For each data row or data item you create, the data source automatically creates a `data cell` (page 358) for each column, by default giving each data cell the name of its column. After creating a row (or item), you can set the data for its data cells, typically by specifying the row (or item) and name for the cell. You can use the same information to get the contents of a data cell.

Properties of data cell objects

A `data cell` object has these properties:

`content`

Access: read/write
Class: *item* (page 73)
the contents of the cell; synonymous with `contents`

`contents`

Access: read/write
Class: *item* (page 73)
the contents of the cell; synonymous with `content`

`name`

Access: read/write
Class: *Unicode text*
the name of the cell; when you create a data row, a data cell is created for each column and by default the name is set to the name of the column

Elements of data cell objects

A `data cell` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`data item` (page 365)
Specify by: "Standard Key Forms" (page 30)
data item for the row that contains the data cell

`data row` (page 369)
Specify by: "Standard Key Forms" (page 30)
the row that contains the data cell

Events supported by data cell objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The following `getContactInfo` handler is from the Table sample application distributed with AppleScript Studio. You will find it in the script file `WithDataSource.applescript`. The `getContactInfo` handler shows how to set the contents of a data cell from the contents of a `text field` (page 314). Each of the data cells is identified by its name, supplied earlier when the data source was initialized.

```
-- Get the values from the text fields and set the cells in the data row
--
on getContactInfo(theWindow, theRow)
    tell theWindow
```

```
        set contents of data cell "name" of theRow
            to contents of text field "name"
        set contents of data cell "address" of theRow
            to contents of text field "address"
        set contents of data cell "city" of theRow
            to contents of text field "city"
        set contents of data cell "state" of theRow
            to contents of text field "state"
        set contents of data cell "zip" of theRow
            to contents of text field "zip"
    end tell
end getContactInfo
```

Figure 6-2 (page 364) shows the running Table application, with one contact. Another handler from the Table application is shown in the Examples section for the `data column` (page 361) class.

To get information from a data cell, given the data row that contains the cell, you use a statement like the following, from the `setContactInfo` handler in the same Table application. As in the previous example, this statement appears within a `tell` block that specifies the window:

```
set contents of text field "name"
    to contents of data cell "name" of theRow
```

The Task List sample application (available starting with AppleScript Studio version 1.2), contains a `data representation` (page 439) handler that shows how to access every data cell of a data source.

```
on data representation theObject of type ofType
    -- Set some local variables to various objects in the UI
    set theWindow to window 1 of theObject
    set theDataSource to data source of table view "tasks"
        of scroll view "tasks" of theWindow
    set theTasks to contents of every data cell of every data row
        of theDataSource
    set theSortColumn to sort column of theDataSource

    -- Statements for working with data cells not shown.
end data representation
```

Instead of getting the contents of every data cell, you might get the contents of every data cell by name (that is, those in a particular column). The following statements show two ways to do so, first by columns, then by rows.

```
set theData to contents of every data cell of data column "address" of
theDataSource
set theData to contents of every data cell "address" of every data row of
theDataSource
```

For an example that uses data cells with data items in an outline view, see the Examples section for the `data item` (page 365) class. For an example that extracts a name from a data cell in the clicked data row, see the Examples section for the `table view` (page 386) class.

Version Notes

The `content` property was added in AppleScript Studio version 1.2. You can use `content` and `contents` interchangeably, with one exception. Within an event handler, `contents of theObject` returns a reference to an object, rather than the actual contents. To get the actual contents of an object (such as the text contents from a `text field` (page 314)) within an event handler, you can either use `contents of contents of theObject` or `content of theObject`.

For a sample script that shows the difference between `content` and `contents`, see the Version Notes section for the `control` (page 270) class.

data column

Plural: `data columns`

Inherits from: None.

Cocoa Class: ASKDataColumn

Represents a column in a data source. Stores the column name, data source, and other information for the column. You can use the data column's elements to access its rows or the individual cells that provide its data.

You typically create a `data source` (page 371) to manage the data for an `outline view` (page 377) or `table view` (page 386). You then create each `data column` (page 361) and supply a name. This process is demonstrated in the Examples section for the `append` (page 399) command.

For an outline view, one column is the outline column, which contains disclosure triangles for expanding or contracting its items. The first column is typically the outline column (though you can specify whether to allow users to reorder columns in the Info window in Interface Builder).

For related information, see `data cell` (page 358).

Properties of data column objects

A `data column` object has these properties (see the Version Notes section for this class for the AppleScript Studio version in which a particular property was added):

`data source`

Access: read only
Class: *data source* (page 371)
the data source the data column is associated with

`name`

Access: read/write
Class: *Unicode text*
the name of the column

`sort case sensitivity`

Access: read/write
Class: *enumerated constant* from "Sort Case Sensitivity" (page 189)
the case sensitivity of the sort (case sensitive, case insensitive)

`sort order`

Access: read/write
Class: *enumerated constant* from "Sort Order" (page 190)
the order of the sort (ascending, descending)

`sort type`

Access: read/write
Class: *enumerated constant* from "Sort Type" (page 190)
the type of sort (alphabetical, numerical)

Elements of data column objects

A `data column` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`data cell` (page 358)
Specify by: "Standard Key Forms" (page 30)
the data cells for the column, one per row; by default there is one data cell for each data column in a data row

`data row` (page 369)
Specify by: "Standard Key Forms" (page 30)
the data rows in the column

Events supported by data column objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The following `will open` (page 170) handler is from the Table sample application distributed with AppleScript Studio. It shows how to create and name the data columns of a `data source` (page 371). This handler does the following:

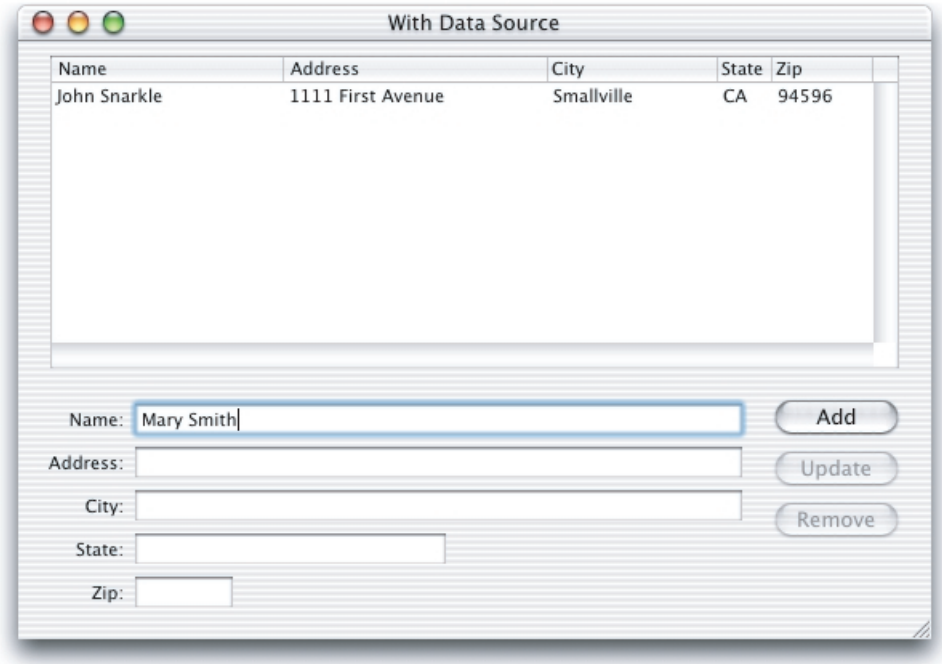
- Gets a reference to a data source from the `window` object passed to the handler. The data source is a property of a table view that resides on a scroll view on the window.
- It tells the `data source` object to make five new columns, each with the name of a different data field of a contact (name, address, city, and so on).

```
on will open theObject
    -- Set up reference variable to simplify later statements.
    set contactsDataSource to data source of table view "contacts"
        of scroll view "contacts" of theObject

    -- Add the data columns to the data source of the contacts table view.
    tell contactsDataSource
        make new data column at the end of the data columns
            with properties {name:"name"}
        make new data column at the end of the data columns
            with properties {name:"address"}
```

```
        make new data column at the end of the data columns
            with properties {name:"city"}
        make new data column at the end of the data columns
            with properties {name:"state"}
        make new data column at the end of the data columns
            with properties {name:"zip"}
    end tell
end will open
```

Figure 6-3 shows the running application, with one contact. For another example that uses data columns, see the Examples section for the `data item` (page 365) class.

Figure 6-2 The Table application



Version Notes

The `sort case sensitivity`, `sort order`, and `sort type` properties were added in AppleScript Studio version 1.2.

data item

Plural: data items
Inherits from: data row (page 369)
Cocoa Class: ASKDataItem

Represents one, possibly expandable, row of a data source. A data item can contain nested data items, supporting data storage for items in a hierarchical view, such as an outline view (page 377), where a user can open an item to display contained items. A data item's properties specify whether it has any nested data items, as well as whether it has a parent item, and if so, a reference to the parent item. Its elements store any nested data items.

You typically create a data source (page 371) to manage the data for an outline view (page 377) or table view (page 386). You then create each data column (page 361) and supply a name. This process is demonstrated in the Examples section below, as well as in the Examples section for the append (page 399) command.

Next, for a outline view, you create data item (page 365) objects for the data source. For each data item you create, the data source automatically creates a data cell (page 358) for each column, by default giving each data cell the name of its column. After creating a data item, you can set the data for its data cells, typically by specifying the data item and the name for the cell. You can use the same information to get the contents of a data cell.

For a table view, you create data row (page 369) objects instead of data item objects, as described in the Examples section for the data row class.

Properties of data item objects

In addition to the properties it inherits from the data row (page 369) class, a data item has these properties:

`has data items`

Access: read only
Class: *boolean*
Does this item have any data items?

`has parent data item`

Access: read only
Class: *boolean*
Does this item have a parent item?

`parent data item`

Access: read/write
Class: *data item*
the parent item for the item

Elements of data item objects

A `data item` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`data cell` (page 358)
Specify by: "Standard Key Forms" (page 30)
the data cells for the item, one per row

`data item` (page 365)
Specify by: "Standard Key Forms" (page 30)
the data item's nested data items

`data row` (page 369)
Specify by: "Standard Key Forms" (page 30)
the data rows in the column

Events supported by data item objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The following `launched` (page 142) handler is from a simple outline application. It demonstrates the common steps for working with data items and data columns in a data source, including:

- creating a `data source` (page 371) and storing it as an element of the `application` object
- creating and naming `data column` (page 361) objects for the data source
- creating parent `data item` objects and setting the contents of their `data cell` (page 358) objects

- creating new data items that are children of the parent data item and setting their contents
- creating additional child data items
- assigning the data source to a property of the `outline view` (page 377)

The `launched` event handler is called at the end of the startup sequence of an application, so it is a suitable place to create a data source for an outline view and populate it with data items. This handler adds the following information to the outline view:

```
- Things to do
    - Work on outline example
        - Make it plain and simple
        - Put it all in a "launched" event handler
    - Put it in my iDisk when done
```

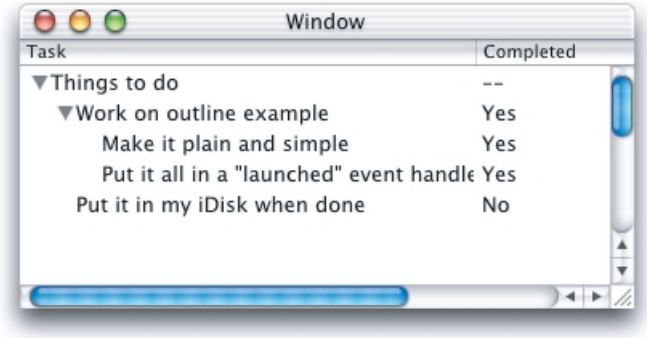
Here is the `launched` handler:

```
on launched theObject
    -- Create the data source; this places it in the application
    --  object's data source elements. (Assign it to outline view below.)
    set dataSource to make new data source at end of data sources
        with properties {name:"tasks"}

    -- Create the data columns
    tell dataSource
        make new data column at end of data columns
            with properties {name:"task"}
        make new data column at end of data columns
            with properties {name:"completed"}
    end tell

    -- Create the top-level parent data item "Things to do"
    set parentItem to make new data item at end of data items of dataSource
    set contents of data cell "task" of parentItem to "Things to do"
    set contents of data cell "completed" of parentItem to "--"

    -- Create the first child data item "Work on outline example", which
    --  will have its own children
    set childItem to make new data item at end of data items of parentItem
    set contents of data cell "task" of childItem
```

```
        to "Work on outline example"
    set contents of data cell "completed" of childItem to "Yes"

    -- Create first child data item of "Work on outline example"
    set childChildItem to make new data item at end of data items of childItem
    set contents of data cell "task" of childChildItem
        to "Make it plain and simple"
    set contents of data cell "completed" of childChildItem to "Yes"

    -- Create second child data item of "Work on outline example"
    set childChildItem to make new data item at end of data items
        of childItem
    set contents of data cell "task" of childChildItem
        to "Put it all in a \"launched\" event handler"
    set contents of data cell "completed" of childChildItem to "Yes"

    -- Create the second child data item of "Things to do"
    set childItem to make new data item at end of data items of parentItem
    set contents of data cell "task" of childItem
        to "Put it in my iDisk when done"
    set contents of data cell "completed" of childItem to "No"

    -- Assign the data source to the outline view
    set data source of outline view "tasks" of scroll view "scroll"
        of window "main" to dataSource
end launched
```

Figure 6-3 shows the running application, with all the data items expanded.

Figure 6-3 The To Do list application



data row

Plural: data rows
Inherits from: None.
Cocoa Class: ASKDataRow

Represents one row in a data source. Stores the row's data source and other information. You can use the data row's elements to access its columns or the individual cells that provide its data.

You typically create a data source (page 371) to manage the data for an outline view (page 377) or table view (page 386). You then create each data column (page 361) and supply a name. This process is demonstrated in the Examples section for the append (page 399) command.

Next, for a table view, you create data rows for the data source. For each row you create, the data source automatically creates a data cell (page 358) for each column, by default giving each data cell the name of its column. After creating a row, you can set the data for its data cells, typically by specifying the row and name for the cell. You can use the same information to get the contents of a data cell.

For an outline view, you create data item (page 365) objects instead of data row objects, as described in the Examples section for the data item class.

Properties of data row objects

A data row object has these properties:

`associated object`

Access: read/write
Class: *item* (page 73)
an object that can be associated with the row

`data source`

Access: read only
Class: *data source* (page 371)
the data source the data row is associated with

Elements of data row objects

A `data row` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`data cell` (page 358)
Specify by: "Standard Key Forms" (page 30)
the data row's data cells; each cell stores cell name, contents, and other information

`data column` (page 361)
Specify by: "Standard Key Forms" (page 30)
the data row's data columns; each column stores column name, data source, and other information

Events supported by data row objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The following statement shows how to create a data row. It is taken from the `clicked` (page 337) handler in the Table sample application distributed with AppleScript Studio.

```
set theRow to make new data row at the end of the data rows
    of contactsDataSource
```

For an example that extracts a name from a data cell in the clicked data row, see the Examples section for the `table view` (page 386) class. For an additional example that works with data rows, see the Examples section for `data cell` (page 358).

data source

Plural: data sources
Inherits from: None.
Cocoa Class: ASKDataSource

Stores the data for and provides the data to views that display row and column data. A data source object represents a form of backing store for a table and is based on a special class supplied by AppleScript Studio's AppleScriptKit framework.

For related information, see data cell (page 358), data column (page 361), data item (page 365), and data row (page 369), as well as the Cocoa Programming Topic Table Views.

Your application supplies a data source object with row and column data for a view such as a table view (page 386) or outline view (page 377). Once you have supplied the data, the data source object works with the view to automatically display the correct information as a user scrolls, resizes the window, reorders the columns, or otherwise changes the displayed rows and columns.

Using a data source is much more efficient that supplying data in script event handlers that must be queried for each bit of data. And to make use of a data source even more efficient, you can set its update views property to false before updating the data source, then set it to true afterwards, so that updating in the associated view happens all at once.

Each view that displays row and column data uses at most one data source. However, you can use multiple views with the same data source if, for example, you want to emphasize different aspects of the data. Then if you change the data in the data source, each view will automatically update to reflect the new values.

For examples that show how to create a data source in your application scripts, see the Examples sections for the append (page 399) command and the data item (page 365) class. You can also create a data source in Interface Builder by dragging it from the Cocoa-AppleScript pane of the Palette window. This mechanism isn't recommended and isn't described here, but you can read about it in the Mail Search tutorial in *Inside Mac OS X: Building Applications With AppleScript Studio*. See "Related Documentation" (page 19) for information on how to obtain that document.

Starting with AppleScript Studio version 1.2, data sources can be sorted. The data source still retains the data rows in the order that they were added to the data source, but it can present the data in sorted order. To enable sorting of a data source, you must do the following

- Add the following properties when creating data columns in the `make new` statement.
  - sort order: `ascending` or `descending`
  - sort type: `alphabetical` or `numerical`
  - sort case sensitivity: `case sensitive` or `case insensitive`

  For example:

  ```
  make new data column at end of data columns of theDataSource with
  properties {name: "name", sort order: ascending, sort type: alphabetical,
  sort case sensitivity: case sensitive}
  ```

- Set the sorted property of the data source to true.

  For example:

  ```
  set sorted of theDataSource to true
  ```

- Set the sort column property of the data source to the initial column to sort on.

  For example:

  ```
  set sort column of theDataSource to data column "name" of theDataSource
  ```

- Connect a `column clicked` event handler to your table view. This will provide the opportunity to change the current sort column, as well as the sort order of the data columns in your data source, when a user clicks in a column's `table header view` (page 385). The sample script shown in the Examples section below is fairly standard and can be used, as is, in your application.

  For full examples, see the Table Sort and Task List sample applications distributed with AppleScript Studio (starting with version 1.2).

If you designate the sort type of a column as `numerical`, you need to ensure that the contents of the data cells for that column are in fact numbers, not strings. Or if you designate the column as `alphabetical`, the contents of the data cells for that column must contain strings (text).

Properties of data source objects

A `data source` object has these properties (see the Version Notes section for this class for the AppleScript Studio version in which a particular property was added):

`localized sort`

Access: read/write
Class: *boolean*
Should the data be sorted using localization rules?

`sort column`

Access: read/write
Class: *data column* (page 361)
the data column to sort the data on; properties of the column control the sort

`sorted`

Access: read/write
Class: *boolean*
Should the data source be sorted?

`update views`

Access: read/write
Class: *boolean*
Should views of the data source be updated? for efficiency, you should avoid unnecessary updating by setting this property to `false` before updating the data source, then setting it to `true` afterwards, as shown in the Examples section for this class; see also the `update` (page 126) command

Elements of data source objects

A `data source` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`data cell` (page 358)
Specify by: "Standard Key Forms" (page 30)
the data source's data cells; each cell stores cell name, contents, and other information

`data column` (page 361)
Specify by: "Standard Key Forms" (page 30)

the data sources data columns; each column stores column name and other information

`data item` (page 365)
Specify by: "Standard Key Forms" (page 30)
the data source's data items

`data row` (page 369)
Specify by: "Standard Key Forms" (page 30)
the data source's data rows

`view` (page 226)
Specify by: "Standard Key Forms" (page 30)
the data source's view

Commands supported by data source objects

Your script can send the following commands to a `data source` object:

`append` (page 399)

Events supported by data source objects

A `data source` object supports handlers that can respond to the following events:

Nib

`awake from nib` (page 130)
You can only connect an `awake from nib` handler to a `data source` object if you create the data source in Interface Builder, not if you create the data source in an application script file (the recommended approach). However, it is unlikely you will need to connect this handler to a `data source` object. See the `data source` class description above for additional information about creating a data source in Interface Builder.

Examples

For examples that show how to create a data source, see the Examples sections for the `append` (page 399) command and the `data item` (page 365) class.

When you make changes to the data for a data source for a visible view, performance is likely to suffer as the data source continually updates the view. You can ensure optimum performance by turning off updating while you modify the data source, then turning it on again when finished. The following lines show how to do this.

```
-- Turn off updating
set update views of theDataSource to false
-- Add statements here that modify the data source
--
-- Turn updating back on
set update views of theDataSource to true
```

Sorting for data sources, added to AppleScript Studio version 1.2, is described in the description for this class above. You can also see sorting in the Table and Task List sample applications (available starting with AppleScript Studio version 1.2).

The following `column clicked` handler shows how to handle a change in the selected sort column. You connect the clicked handler to the `table view` (page 386) or `outline view` (page 377) that contains the columns to sort. This handler is fairly standard and can be used, as is, in most applications. It does the following:

- Gets the identifier of the clicked column.
- Gets the current sort column for the data source.
- If the columns are different, the user has chosen a new sort column, so it sets the data source's sort column to the new column.
- If the columns are the same, the user has chosen a new sort order, so it changes its sort order (from ascending to descending or vice versa).
- Calls the `update` (page 126) command to redraw the sorted data.

```
on column clicked theObject table column tableColumn
    -- Get the data source of the table view
    set theDataSource to data source of theObject

    -- Get the identifier of the clicked table column;
    --  you can instead use the name of the column
    set theColumnIdentifier to identifier of tableColumn

    -- Get the current sort column of the data source
    set theSortColumn to sort column of theDataSource
```

```
    -- If the current sort column is not the same as the clicked column
    --   then switch the sort column
    if (name of theSortColumn) is not equal to theColumnIdentifier then
        set the sort column of theDataSource to
            data column theColumnIdentifier of theDataSource
    else
        -- Otherwise change the sort order
        if sort order of theSortColumn is ascending then
            set sort order of theSortColumn to descending
        else
            set sort order of theSortColumn to ascending
        end if
    end if

    -- Update the table view (so it will be redrawn)
    update theObject
end column clicked
```

The Table Sort and Task List sample applications, first distributed with AppleScript Studio version 1.2, provide full examples of how to sort a data source.

Working with data sources is a complex task that can't be covered fully here. For an additional, relatively simple example, see the Outline sample application, which shows how to use an `outline view` (page 377) to display items in the file system. The Table sample application uses a data source with a `table view` (page 386).

For a more detailed example, see the chapters in *Inside Mac OS X: Building Applications With AppleScript Studio* that describe how to build the Mail Search application (which is also distributed with AppleScript Studio).

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

Sorting for data source objects was added in AppleScript Studio version 1.2.

The `localized sort`, `sort column`, and `sorted` properties were added in AppleScript Studio version 1.2.

The Table Sort and Task List sample applications were added in AppleScript Studio version 1.2.

The preferred (and most convenient) method for using a data source is to create it and assign it directly in an application script, as shown in the Examples sections for the `append` (page 399) command and the `data item` (page 365) class. This option was added in AppleScript Studio version 1.1. Prior to version 1.1, you had to use a more cumbersome process of adding and connecting a data source to your application in Interface Builder.

Note that if you wish to connect an `awake from nib` (page 130) handler to a data source, you will have to add and connect the data source in Interface Builder. The steps for doing so are described in *Inside Mac OS X: Building Applications With AppleScript Studio*, available in Project Builder help. See the chapter in the Mail Search tutorial on connecting the interface.

The `view` element was added in AppleScript Studio version 1.1.

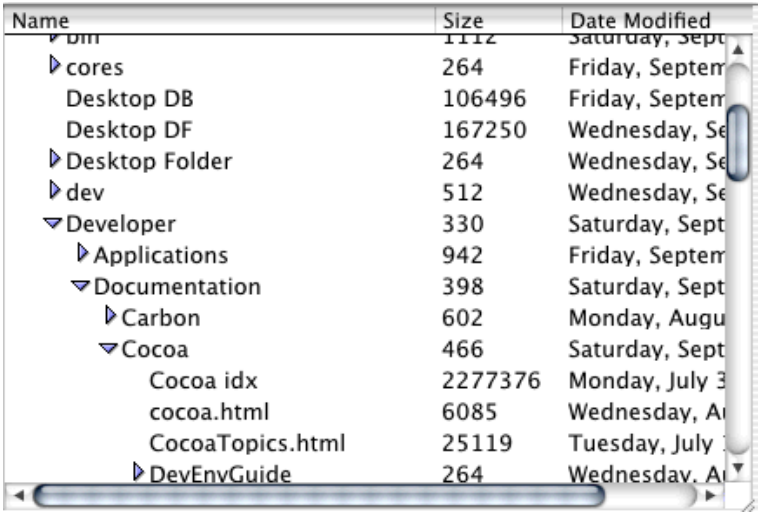
`outline view`

Plural: `outline views`
Inherits from: `table view` (page 386)
Cocoa Class: NSOutlineView

A view that uses a row-and-column format to display hierarchical data that can be expanded and collapsed, such as directories and files in a file system. A user can expand and collapse rows, edit values, and resize and rearrange columns.

Figure 5-13 shows an outline view that displays a hierarchy of files and folders.

Figure 6-4 An outline view



You will find the `outline view` object on the Cocoa-Data pane in Interface Builder's Palette window. You can set many attributes for outline views in Interface Builder's Info window.

Although AppleScript Studio provides event handlers for managing the data an outline view displays, the preferred and far more efficient approach is to use a `data source` (page 371) object.

For more information on outline views, see the Cocoa Programming Topic Outline Views.

Properties of outline view objects

In addition to the properties it inherits from the `table view` (page 386) class, an `outline view` object has these properties:

`auto resizes outline column`

Access: read/write
Class: *boolean*
Should the outline column be automatically resized? default is `false`; you can set this property in the Info window in Interface Builder

`auto save expanded items`

Access: read/write
Class: *boolean*
not supported (through AppleScript Studio version 1.2); Should the expanded state of the outline items be automatically save? default is `false`

`indentation per level`

Access: read/write
Class: *real*
the amount to indent per level; default is 16.0

`marker follows cell`

Access: read/write
Class: *boolean*
Should the marker follow the cells (that is, as the cells in the outline view are indented, should the disclosure triangle also be indented)? default is `true`;

`outline table column`

Access: read/write
Class: *table column* (page 382)
the table column that contains the outline

Elements of outline view objects

An `outline view` object can contain only the elements it inherits from `table view` (page 386).

Commands supported by outline view objects

Your script can send the following commands to an `outline view` object:

`item for` (page 401)
`update` (page 126)

Events supported by outline view objects

An `outline view` object supports handlers that can respond to the following events:

Action

`clicked` (page 337)

Data View

Drag and Drop

Key

Mouse

Nib

awake from nib (page 130)

Outline View

change item value (page 406)
child of item (page 407)
item expandable (page 412)
item value (page 413)
number of items (page 417)
should collapse item (page 419)
should expand item (page 420)
will display item cell (page 427)
will display outline cell (page 428)

Table View

cell value (page 404)
change cell value (page 405)
number of rows (page 418)

View

bounds changed (page 238)

Examples

The following script statement shows how to identify an outline view and send it the update (page 126) command.

```
tell outline view "outline" of scroll view "scroll" of window "main" to update
```

Working with outline views is a complex task that can't be covered in detail here. For a complete but relatively simple example, see the Examples section for the data item (page 365) class. For a more detailed example, see the chapters in *Inside Mac OS X: Building Applications With AppleScript Studio* that describe how to build the Mail Search application (which is also distributed with AppleScript Studio).

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

The auto save expanded items property in this class is not supported, through AppleScript Studio version 1.2.

See the Version Notes section for the table view (page 386) class (from which outline view inherits) for properties added in AppleScript Studio version 1.2 to support sorting.

Prior to AppleScript Studio version 1.1, the Mail Search application was named Watson.

table column

Plural: table columns
Inherits from: None.
Cocoa Class: NSTableColumn

Stores the display characteristics and identifier for a column in a table view or outline view. The table column determines the height and width limits, resizability, and editability of its column in the table view (page 386) or outline view (page 377). It also stores two cell objects: the header cell, which is used to draw the column header, and its data cell (page 358), used to draw the values for each row.

When you insert a table or outline view in Interface Builder, the view contains a table column object for each column. If the table or outline view will use a data source (page 371), you must specify either an identifier name or an AppleScript name for each column. For more information, see the identifier property and the Version Notes section for this class.

For more information, see table header cell (page 385) and table header view (page 385), as well as the Cocoa Programming Topic Table Views.

Properties of table column objects

A table column object has these properties:

data cell

Access: read/write
Class: *data cell* (page 358)
not supported (through AppleScript Studio version 1.2); the data cell for the column

`editable`

Access: read/write
Class: *boolean*
Is the column editable? default is `true`; you can set this property in the Info window in Interface Builder

`header cell`

Access: read/write
Class: *table header cell* (page 385)
the header cell used to draw the header for the column

`identifier`

Access: read/write
Class: *Unicode text*
the name used by the data source to identify a column; you can set this value in the Identifier field in the Attributes Pane of the Info window in Interface Builder; see the Version Notes section for this class for related information

`maximum width`

Access: read/write
Class: *real*
the maximum width of the column; default is 1000; you can set this property in the Info window in Interface Builder

`minimum width`

Access: read/write
Class: *real*
the minimum width of the column; you can set this property in the Info window in Interface Builder

`resizable`

Access: read/write
Class: *boolean*
Is the column resizable? default is `true`; you can set this property in the Info window in Interface Builder

`table view`

Access: read/write
Class: *table view* (page 386)
the table view or outline view that contains the table column

`width`

Access: read/write
Class: *real*
the width of the column

Elements of table column objects

A `table column` object has no elements.

Events supported by table column objects

A `table column` object supports handlers that can respond to the following events:

Nib

`awake from nib` (page 130)

Examples

For an example that shows how to access properties of a table column, see the `column clicked` (page 409) handler in the Examples section of the `data source` (page 371) class. The Examples section for the `table view` (page 386) class points to additional sample code and documentation for using table views, both with and without data sources.

Version Notes

Starting with AppleScript Studio version 1.2, and the version of Interface Builder released with Mac OS X version 10.2, you can name table columns in a `table view` (page 386) or `outline view` (page 377) using the Name field of the AppleScript pane in Interface Builder's Info window. It is no longer necessary to enter the value in the Identifier field of the Attributes pane (although that is still supported for backward compatibility).

If you do specify an identifier name (rather than an AppleScript name) for a table column, it must match the name of the `data column` (page 361) in your data source. Otherwise, starting in AppleScript Studio version 1.1, no data will be provided for that data column to draw.

The `data cell` property in this class is not supported, through AppleScript Studio version 1.2.

table header cell

Plural: table header cells
Inherits from: text field cell (page 319)
Cocoa Class: NSTableHeaderCell

Used by a table header view to draw its column headers.

For more information, see table header view (page 385), as well as the Cocoa Programming Topic Table Views.

Properties of table header cell objects

A table header cell object has only the properties it inherits from text field cell (page 319).

Events supported by table header cell objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

You don't typically script a text field cell, which adds no properties or elements to its superclass, text field cell (page 319).

table header view

Plural: table header views
Inherits from: view (page 226)
Cocoa Class: NSTableHeaderView

Used by a table view to draw headers over its columns and to handle mouse events in those headers.

For more information, see table view (page 386), as well as the Cocoa Programming Topic Table Views.

Properties of table header view objects

In addition to the properties it inherits from the view (page 226) class, a table header view object has these properties:

`dragged column`

Access: read only
Class: *integer*
if the user is dragging a column, the one-based index of that column; otherwise -1

`dragged distance`

Access: read only
Class: *real*
if the user is dragging a column, the column's horizontal distance from its original position, otherwise the value is meaningless

`resized column`

Access: read only
Class: *integer*
if the user is resizing a column, the one-based index of that column; otherwise -1

`table view`

Access: read/write
Class: *table view* (page 386)
the table view that contains the table header view

Elements of table header view objects

A `table header view` object can contain only the elements it inherits from `view` (page 226).

Events supported by table header view objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

You don't typically script a table header view.

`table view`

Plural: `table views`
Inherits from: `control` (page 270)
Cocoa Class: NSTableView

A view that displays record-oriented data in a table, and allows the user to edit values and resize and rearrange columns. A table view displays data for a set of related records, with rows representing individual records and columns representing the attributes of those records. If you use a `data source` (page 371) object to provide data for the table view, the data source works with the view to automatically display the correct information as a user scrolls, resizes the window, reorders the columns, or otherwise changes the displayed rows and columns.

You will find the `table view` object on the Cocoa-Data pane in Interface Builder's Palette window. When you insert a table view in a window, the table view is automatically enclosed in a `scroll view` (page 211). You can set many attributes for table views in Interface Builder's Info window, but to do so, make sure you've double-clicked to select the table view, not its enclosing scroll view (with Interface Builder's Info window open to the AppleScript pane, you should see "NSTableView" as the window title).

Figure 6-5 shows a table view in Interface Builder. The row and column information is temporary filler provided by Interface Builder. For more information, see the Cocoa Programming Topic Table Views.

Figure 6-5 A table view in Interface Builder

Although AppleScript Studio provides event handlers for managing the data a table view displays, the preferred and far more efficient approach is to use a `data source` (page 371) object to provide data. The Examples section for this class points to sample code and documentation for using table views both with and without data sources.

Properties of table view objects

In addition to the properties it inherits from the `control` (page 270) class, a `table view` object has these properties (see the Version Notes section for this class for the AppleScript Studio version in which a particular property was added):

`allows column reordering`

Access: read/write
Class: *boolean*
Can the columns be reordered? default is `true`; you can set this property in the Info window in Interface Builder; if you connect a `column clicked` (page 409) handler for a table view, the handler will not be called unless the value of this property is `true`

`allows column resizing`

Access: read/write
Class: *boolean*

Can the columns be resized? default is `true`; you can set this property in the Info window in Interface Builder

`allows column selection`

Access: read/write
Class: *boolean*
Can columns be selected? default is `true`; you can set this property in the Info window in Interface Builder

`allows empty selection`

Access: read/write
Class: *boolean*
Should the table view allow an empty selection? default is `true`; you can set this property in the Info window in Interface Builder

`allows multiple selection`

Access: read/write
Class: *boolean*
Should the table view allow multiple selection? default is `false`; you can set this property in the Info window in Interface Builder

`auto resizes all columns to fit`

Access: read/write
Class: *boolean*
Should the columns automatically be resized to fit? default is `false`; you can set this property in the Info window in Interface Builder

`auto save name`

Access: read/write
Class: *Unicode text*
the name used for automatically saving information about the table's columns (see `auto save table columns` property); default is no name; you can set this property in the Info window in Interface Builder

`auto save table columns`

Access: read/write
Class: *boolean*
not supported (through AppleScript Studio version 1.2); Should the order and width of the table columns be automatically saved? default is `false`; when you

supply a name for the `auto save name` property in the Info window in Interface Builder, this property is automatically set to `true`

`background color`

Access: read/write
Class: *RGB color*
the background color for the table view; a three-item integer list that contains the values for each component of the color; for example, red can be represented as {65535,0,0}; by default, {65535, 65535, 65535}, or white; you can set this property in the Info window in Interface Builder

`clicked column`

Access: read only
Class: *integer*
the one-based index of the column that was clicked to trigger an event handler; value is 0 if no event occurred; the return value of this method is meaningful only in the `clicked` (page 337) or `double clicked` (page 338) event handlers

`clicked data column`

Access: read only
Class: *data column* (page 361)
the data column that was clicked; allows you to get the clicked `data column` object directly and regardless of sorted state; returns nil if no column was clicked, so you should only access the value in a `try`, `on error` block (for an example of a `try`, `on error` block, see the Examples section of the `path for` (page 120) command)

`clicked data row`

Access: read only
Class: *data row* (page 369)
the data row that was clicked; allows you to get the `data row` object directly and regardless of sorted state; returns nil if no row was clicked, so you should only access the value in a `try`, `on error` block (for an example of a `try`, `on error` block, see the Examples section of the `path for` (page 120) command)

`clicked row`

Access: read only
Class: *integer*
the one-based index of the row that was clicked to trigger an event handler; value is 0 if no event occurred; the return value of this method is meaningful only in the `clicked` (page 337) or `double clicked` (page 338) event handlers

`corner view`

Access: read/write
Class: *anything*
the corner view (the view used to draw the area to the right of the column headers and above the vertical scroller of the enclosing `scroll view` (page 211); this is by default a simple `view` (page 226) that merely fills in its frame, but you can replace it with a custom view)

`draws grid`

Access: read/write
Class: *boolean*
Should the table view draw its grid? default is `false`; you can set this property in the Info window in Interface Builder

`edited column`

Access: read only
Class: *integer*
the one-based index of the column being edited; value is 0 if no column is being edited

`edited data column`

Access: read only
Class: *data column* (page 361)
the data column being edited; allows you to get the `data column` object directly and regardless of sorted state; value is `nil` if no column is being edited, so you should only access the value in a `try`, `on error` block (for an example of a `try`, `on error` block, see the Examples section of the `path for` (page 120) command)

`edited data row`

Access: read only
Class: *data row* (page 369)
the data row being edited; allows you to get the `data row` object directly and regardless of sorted state; value is `nil` if no row is being edited, so you should only access the value in a `try`, `on error` block (for an example of a `try`, `on error` block, see the Examples section of the `path for` (page 120) command)

`edited row`

Access: read only
Class: *integer*
the one-based index of the row being edited; value is 0 if no row is being edited

`grid color`

Access: read/write
Class: *RGB color*
the color of the grid; a three-item integer list that contains the values for each component of the color; for example, green can be represented as {0,65535,0}; default is {32767, 32767, 32767} (or gray); you can set this property in the Info window in Interface Builder

`header view`

Access: read/write
Class: *table header view* (page 385)
the table header view used to draw headers over columns; returns nil if the table has no header view, so you should only access the value in a `try`, `on error` block (for an example of a `try`, `on error` block, see the Examples section of the `path for` (page 120) command)

`intercell spacing`

Access: read/write
Class: *list*
the space between cells; represented as a two-item list of numbers

`row height`

Access: read/write
Class: *real*
the height of the row

`selected column`

Access: read/write
Class: *integer*
the one-based index of the selected column; 0 if no column is selected; if `allows column selection` is `true` but `allows multiple selection` is `false`, you can evaluate this property to obtain the index of the selected column, if any

`selected columns`

Access: read/write
Class: *list*
the one-based index of every selected column; an empty list if no column is selected; if `allows column selection` is `true` and `allows multiple selection` is `true`, you can use this property to determine the selected columns

`selected data column`

Access: read/write
Class: *data column* (page 361)
the data column that is selected; there will be no selected data column unless `allows multiple selection` is `true`; returns nil if no data column was selected, so you should only access the value in a `try`, `on error` block (for an example of a `try`, `on error` block, see the Examples section of the `path for` (page 120) command)

`selected data columns`

Access: read/write
Class: *list*
the data columns that are selected; returns an empty list if no data columns are selected; if `allows multiple selection` is `false`, the returned list will contain at most one data column

`selected data row`

Access: read/write
Class: *data row* (page 369)
the data row that is selected; if `allows multiple selection` is `false`, you can use this property to obtain the selected data row—otherwise use the `selected data rows` property; returns nil if no data row was selected, so you should only access the value in a `try`, `on error` block (for an example of a `try`, `on error` block, see the Examples section of the `path for` (page 120) command)

`selected data rows`

Access: read/write
Class: *list*
the data rows that are selected; returns an empty list if no data rows are selected; if `allows multiple selection` is `false`, the returned list will contain at most one data row

`selected row`

Access: read/write
Class: *integer*
the one-based index of the selected row; if `allows multiple selection` is `false`, you can check this property for the selected row

`selected rows`

Access: read/write
Class: *list*

the one-based index of every selected row; if `allows multiple selection` is `true`, you can check this property for the selected rows

Elements of table view objects

In addition to the properties it inherits from the `control` (page 270) class, a `table view` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`data source` (page 371)
Specify by: "Standard Key Forms" (page 30)
the data source that supplies data for the table; a table view can have either no data sources or one data source; you do not need an index to refer to the data source: `set theDataSource to data source of table view 1 of scroll view 1 of window 1`

`table column` (page 382)
Specify by: "Standard Key Forms" (page 30)
the tables columns, which store the display characteristics and identifier for each column

Commands supported by table view objects

Your script can send the following commands to a table view:

`update` (page 126)

Events supported by table view objects

A table view supports handlers that can respond to the following events:

Action

`clicked` (page 337)
`double clicked` (page 338)

Data View

`column clicked` (page 409)
`column moved` (page 410)
`column resized` (page 411)
`selection changed` (page 339)

Drag and Drop

Key

Mouse

Nib

Table View

`number of rows` (page 418)

View

`bounds changed` (page 238)

Examples

The following script statements show how to identify a table view and send it the `update` (page 126) command. The object names used in this example match those in the Table sample application distributed with AppleScript Studio.

```
set theTableView to table view "contacts" of scroll view "contacts" of window
"main"
tell theTableView to update
```

You can use statements like the following to set the selected rows in the table view. The first statement sets a property to allow multiple selection in the table view; the second statement selects the first and fourth rows of the table view:

```
set allows multiple selection of theTableView to true
set selected rows of theTableView to {1, 4}
```

You can use the similar statements to set the selected columns in the table view. In the following example, the second statement sets a property to allow column selection in the table view; the third statement selects the second and third columns in the table view:

```
set allows multiple selection of theTableView to true
set allows column selection of theTableView to true
set selected columns of theTableView to {2, 3}
```

To get information from a named data cell in a data row of a table view with a data source, you can use statements like those in the following `clicked` handler, connected to the table view.

```
on clicked theObject
    set rowIndex to clicked row of theObject
    if rowIndex is greater than 0 then
        set dataSource to data source of theObject
        set theRow to data row rowIndex of dataSource
        set theName to contents of data cell "name" of theRow
    end if
```

```
end clicked
```

Working with table views is a complex task that can't be covered fully here. For a complete example, see the Table sample application. The application demonstrates two mechanisms for working with table views. The preferred mechanism, which makes use of a `data source` (page 371) object to manage the table's data, is shown in the script file `WithDataSource.applescript`. A less efficient mechanism, which may be adequate for very simple tables, is shown in the file `WithoutDataSource.applescript`.

For a more detailed example, see the chapters in *Inside Mac OS X: Building Applications With AppleScript Studio* that describe how to build the Mail Search application (which is also distributed with AppleScript Studio).

Version Notes

Support for drag-and-drop commands was added in AppleScript Studio version 1.2.

The following properties were added to the `table view` class (and are therefore also available to its subclass, the `outline view` (page 377) class) in AppleScript Studio version 1.2:

- `edited data column`
- `clicked data column`
- `selected data column`
- `selected data columns`
- `clicked data row`
- `edited data row`
- `selected data row`
- `selected data rows`

These properties return the appropriate elements regardless of their sorted state and should be used in place of their non-sorted counterparts (`edited column`, `edited row`, and so on).

The `auto save table columns` property in this class is not supported, through AppleScript Studio version 1.2.

Because of a bug fix in Cocoa Scripting in Mac OS X version 10.2, it is now possible in AppleScript Studio version 1.2 to set a property that is a list to a new list. For instance you can now specify a list to select the rows in a table view, as shown in the Examples section for this class.

Starting with AppleScript Studio version 1.1, the behavior of table views and data sources was changed as follows: if the name you set for a table column in the Identifier field of the Attributes pane in Interface Builder does not match the name of the `data column` (page 361) in your data source, no data will be provided for that data column to draw.

However, starting with AppleScript Studio version 1.2, and the version of Interface Builder released with Mac OS X version 10.2, you can name table columns in a table view using the Name field of the AppleScript pane in Interface Builder's Info window. Use of an identifier name is still supported for backward compatibility.

Prior to AppleScript Studio version 1.1, the Mail Search application was named Watson.

Commands

Objects based on classes in the Data View suite support the following commands. (A command is a word or phrase a script can use to request an action.) To determine which classes support which commands, see the individual class descriptions.

append

Appends the provided list of lists or list of records to a data source. The data from each list or record provides the contents for the cells in one row of the data source. This command provides a simple, efficient mechanism for adding data to a data source associated with a view, such as an outline view (page 377) or a table view (page 386).

If you supply a list of records, the append command will attempt to match the labels for the fields of each record with the data column identifiers. For each label that matches a column identifier, it inserts the data from that field into the matching column. If no record label matches the identifier for a column, that column is left blank.

If you supply a list of lists, the append command will match the items in each list to the corresponding column, by index. That is, the data for the first item goes in the first column, and so on.

Syntax

append	*data source*	required
with	*list*	required

Parameters

data source (page 371)
the data source to append data to

with *list*
a list of lists or list of records to append to the specified data source

Examples

The following `awake from nib` (page 130) handler is from the Table Sort sample application distributed with AppleScript Studio (available starting with version 1.2). This handler, which is connected to a `table view` (page 386), does the following:

- Creates a data source named `"names"`
- Creates and adds four columns to the data source, one each for name, city, zip code, and age. The columns specify a sorting preferences, including sort type and sort order.
- Specifies that the data source should be sorted and the current sort column should be the name column.
- Assigns the data source to the table view whose `awake from nib` handler was called.
- Uses the append command to populate the data source with data from the application's `tableData` property (shown below).

```
on awake from nib theObject
    -- Create the data source; this places it in the application
    --  object's data source elements. (Assign it to table view below.)
    set theDataSource to make new data source at end of data sources
        with properties {name:"names"}

    -- Create each of the data columns, including the sort information
    --   for each column
    make new data column at end of data columns of theDataSource
        with properties {name:"name", sort order:ascending,
        sort type:alphabetical, sort case sensitivity:case sensitive}
    make new data column at end of data columns of theDataSource
        with properties {name:"city", sort order:ascending,
        sort type:alphabetical, sort case sensitivity:case sensitive}
    make new data column at end of data columns of theDataSource
        with properties {name:"zip", sort order:ascending,
        sort type:alphabetical, sort case sensitivity:case sensitive}
    make new data column at end of data columns of theDataSource
        with properties {name:"age", sort order:ascending,
        sort type:numerical, sort case sensitivity:case sensitive}

    -- Make this a sorted data source
    set sorted of theDataSource to true
```

```
    -- Set the "name" data column as the sort column
    set sort column of theDataSource to data column "name" of theDataSource

    -- Set the data source of the table view to the new data source
    set data source of theObject to theDataSource

    -- Add the table data (using the new "append" command)
    append theDataSource with tableData
end awake from nib
```

This is the `tableData` property defined in the Table Sort application. The "name" field is enclosed in vertical bars to differentiate it from any similarly-named key words:

```
property tableData : {{|name|:"Bart Simpson", city:"Springfield",
zip:"19542", age:12}, {|name|:"Ally McBeal", city:"Boston", zip:"91544",
age:28}, {|name|:"Joan of Ark", city:"Paris", zip:"53255", age:36},
{|name|:"King Tut", city:"Egypt", zip:"00245", age:45}, {|name|:"James
Taylor", city:"Atlanta", zip:"21769", age:42}}
```

You can also get the data back out of a data source (as a list of lists) with terminology like the following (where `theDataSource` specifies a data source):

```
set myList to contents of every data cell of every data row of theDataSource
```

Version Notes

The `append` command was added in AppleScript Studio version 1.2.

item for

Returns the data item for the specified row (base 1) of a data source. A `data item` (page 365) represents one, possibly expandable, row of the `data source` (page 371).

This command works only for an `outline view` (page 377).

Syntax

`item for`	*outline view*	required
`row`	*integer*	required

Parameters

`outline view` (page 377)
the outline view from which to get the item for the specified row

`row` *integer*
the one-based index of the row of the outline view from which to get the item

Result

data item
The `data item` (page 365) for the specified row of the outline view. Returns no result if the row is out of range.

Examples

The following script statements are from the `mailboxesForIndex` handler in the Mail Search sample application distributed with AppleScript Studio. Among other things, the Mail Search application uses an outline view to display account items, each of which may have multiple mailboxes. A mailbox, in turn, has a mailbox name and may contain nested mailboxes. The `mailboxesForIndex` handler uses the `item for` command to obtain a row at a specified index, then obtains data (the name of the mailbox at that row) from the first data cell of that item.

```
-- Determine if the selected item is an account or a mailbox
tell outline view "mailboxes" of scroll view "mailboxes"
    of split view 1 of theWindow
        set theItem to item for row mailboxIndex
        set theName to contents of data cell 1 of theItem
        -- some statements omitted
end tell
```

Events

Objects based on classes in the Data View suite support handlers for the following events (an event is an action, typically generated through interaction with an application's user interface, that causes a handler for the appropriate object to be executed). To determine which classes support which events, see the individual class descriptions.

cell value

Called for a table or outline view to get the value of a cell. The handler should return the value of the specified cell.

The preferred way to manipulate data for `table view` (page 386) and `outline view` (page 377) objects is to use a `data source` (page 371), in which case this handler is not needed (or called).

Syntax

`cell value`	*reference*	required
`row`	*integer*	required
`table column`	*table column*	required

Parameters

reference
a reference to the `table view` (page 386) or `outline view` (page 377) that contains the cell

`row` *integer*
the one-based row of the cell

`table column` *table column* (page 382)
the column of the cell

Result

anything
The value of the cell at the specified row and column. If you implement this handler, you should always return a value.

Examples

The following `cell value` handler is from the Table sample application distributed with AppleScript Studio. It's in the script file `WithoutDataSource.applescript`. (See the file `WithDataSource.applescript` for the preferred mechanism for working with tables, using a `data source` object.)

This handler:

- Sets the return value to an empty string.
- Checks for a valid row number.
- If the row number is in range, uses the column identifier to determine the field to get the cell value from.
- Returns the value.

```
on cell value theObject row theRow table column theColumn
    -- Set the value to an empty string for now
    set theValue to ""

    -- Make sure the row we're asked for is within the number of contacts
    if (count of contacts) ≥ theRow then
        set theContact to item theRow of contacts

        -- Get the column identifier to determine which field
        --  of the record to return
        set theID to identifier of theColumn
        if theID is "name" then
            set theValue to name of theContact
        else if theID is "address" then
            set theValue to address of theContact
        else if theID is "city" then
            set theValue to city of theContact
        else if theID is "state" then
            set theValue to state of theContact
        else if theID is "zip" then
            set theValue to zip of theContact
        end if
    end if

    -- Now return the value that we set
    return theValue
end cell value
```

change cell value

Called for a table or outline view to change the value of a cell.

The preferred way to manipulate data for `table view` (page 386) and `outline view` (page 377) objects is to use a `data source` (page 371), in which case this handler is not needed (or called).

Syntax

`change cell value`	*reference*	required
`row`	*integer*	required
`table column`	*table column*	required
`value`	*item*	required

Parameters

reference
a reference to the `table view` (page 386) or `outline view` (page 377) that contains the cell

`row` *integer*
the one-based row of the cell to change

`table column` *table column* (page 382)
the column of the cell to change

`value` *item* (page 73)
the new value

Examples

When you connect a `change cell value` handler to a `table view` or `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. Your handler should set the cell in the specified row and column to the specified value.

```
on change cell value theObject value theValue row theRow table column
tableColumn
    (*Set the specified cell to the passed value. *)
end change cell value
```

change item value

Called for an outline view to change the value of an item at a specified row.

The preferred way to manipulate data for an `outline view` (page 377) is to use a `data source` (page 371), in which case this handler is not needed (or called).

Syntax

`change item value`	*reference*	required
`outline item`	*item*	required
`table column`	*table column*	required
`value`	*item*	required

Parameters

reference
a reference to the `outline view` (page 377) that contains the item

`outline item` *item* (page 73)
The item to change the value for.

`table column` *table column* (page 382)
The column of the item to change.

`value` *item* (page 73)
The new value to change to.

Examples

When you connect a `change item value` handler to a `table view` or `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. Your handler should set the specified item to the specified value.

```
on change item value theObject value theValue outline item outlineItem table
column tableColumn
    (*Set the specified item to the passed value. *)
end change item value
```

child of item

Called for an outline view to get a specified child item from an item.

Although AppleScript Studio provides event handlers for managing the data an outline view displays, such as getting the child of an item, the preferred and far more efficient approach is to use a `data source` (page 371) object.

Syntax

`child of item`	*reference*	required
`outline item`	*item*	required
`child`	*integer*	optional

Parameters

reference
a reference to the `outline view` (page 377) that contains the items

`outline item` *item* (page 73)
the item that contains the child item; typically a one-based index or a string

`child` *integer*
the one-based index of the child of the given child

Result

anything
The handler should return the specified child of the specified outline item.

Examples

The following `child of item` event handler is from the Outline sample application distributed with AppleScript Studio. Outline uses an outline view to display items in the file system. This handler:

- Sets the variable `childItem` to an empty string.
- Calls on the Finder application to help it do the following:
  - If the passed outline item is outline item 0, representing a disk name at the highest level of the outline, it sets `childItem` to the disk name (which the application keeps track of separately in the `diskNames` property) of the disk specified by the `theChild` parameter, as a string. (In the Outline application, the `outline item` parameter to the `child of item` event handler is the numeric value 0 for disk names; for nested items, it's a colon-delimited path, such as "Hard Disk:", "Hard Disk:App Folder:", "Hard Disk:App Folder:SomeApp:", and so on.)
  - Otherwise, it sets `childItem` to the child item of the passed outline item, as a string.

Note that the Outline application is using the Finder to help it display items, which are things the Finder knows about, such as disks, files, and folders. The operations performed by the Finder in this handler are `(get item theChild)` and `(get item outlineItem)`, which ask it to get items at specified indices.

- Returns `childItem`.

```
on child of item theObject outline item outlineItem child theChild
    set childItem to ""

    tell application "Finder"
        if outlineItem is 0 then
            set childItem to disk (get item theChild
                of diskNames as string) as string
        else
            set childItem to item theChild of (get item outlineItem)
                as string
        end if
    end tell

    return childItem
end child of item
```

column clicked

Called for a table or outline view when a column is clicked. The handler can perform operations such as setting column-sorting properties of a `data source` (page 371) object.

This handler will not be called unless the table or outline view that contains the column allows column reordering (either the Allows Reordering checkbox for the table or outline view is selected on the Attributes pane of Interface Builder's Info window, or you set the `allows column reordering` property to `true` in an application script).

Syntax

`column clicked`	*reference*	required
`table column`	*table column*	required

Parameters

reference
a reference to the `table view` (page 386) or `outline view` (page 377)

`table column` *table column* (page 382)
the column that was clicked

Examples

For an example of a `column clicked` (page 409) handler, see the Examples section of the `data source` (page 371) class.

column moved

Called for a table or outline view after a column moves, as when a user shuffles the rows in a table or outline view.

Syntax

`column moved`	*reference*	required
`new column`	*integer*	required
`old column`	*integer*	required

Parameters

reference
a reference to the `table view` (page 386) or `outline view` (page 377)

`new column` *integer*
the zero-based index of the new column position

`old column` *integer*
the zero-based index of the old column position

Examples

When you connect a `column moved` handler to a `table view` or `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. The column parameters provide the indices of the new and old column positions. Your handler should perform any operations required by the changed column positions.

```
on column moved theObject new column newColumn old column oldColumn
    (* Respond to changed column position. *)
end column moved
```

column resized

Called for a table or outline view after a column is resized. The new size may be the same as the old size.

Syntax

`column resized`	*reference*	required
`old width`	*real*	required
`table column`	*table column*	required

Parameters

reference
a reference to the `table view` (page 386) or `outline view` (page 377)

`old width` *real*
the previous width of the column

`table column` *table column* (page 382)
the column that may have been resized

Examples

When you connect a `column resized` handler to a `table view` or `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template. Your handler should perform any operations required by the change in column size, possibly after first determining if the column width actually changed.

```
on column resized theObject table column tableColumn old width oldWidth
    if width of tableColumn is not equal to oldWidth then
```

```
    (* Add statements to perform any operations required after change
        of column width. *)
    end if
end column resized
```

item expandable

Called for an outline view to find out if the specified item is expandable. The handler returns `true` if the item is expandable, `false` if it is not.

The preferred way to provide data for an `outline view` (page 377) is to use a `data source` (page 371), in which case this handler is not needed (or called).

Syntax

`item expandable`	*reference*	required
`outline item`	*item*	optional

Parameters

reference
a reference to the `outline view` (page 377) that contains the item

`outline item` *item* (page 73)
the item that may be expandable

Result

boolean
Returns `false` if the item is not expandable or `true` if it is. If you implement this handler, you should always return a boolean value.

Examples

The following `item expandable` event handler is from the Outline sample application distributed with AppleScript Studio. Outline uses an outline view to display items in the file system. This handler:

- Sets the variable `isExpandable` to `false`.

- If the outline item is outline item 0, representing disk names at the highest level of the outline, and if there is more than one disk name (which the application keeps track of separately in the `diskNames` property), it sets `isExpandable` to `true`.
- Otherwise, it calls on the Finder application to obtain the count of items for the outline item. If that count is greater than 1, it sets `isExpandable` to `true`.
- Returns `isExpandable`.

```
on item expandable theObject outline item outlineItem
    set isExpandable to false

    if outlineItem is 0 then
        if (count of diskNames) is greater than 1 then
            set isExpandable to true
        end if
    else
        tell application "Finder"
            if (count of items of (get item outlineItem))
                    is greater than 1 then
                set isExpandable to true
            end if
        end tell
    end if

    return isExpandable
end item expandable
```

item value

Called for an outline view to get the value for an item. The handler returns the value (typically as a string) to be displayed for the specified item.

The preferred way to provide data for an `outline view` (page 377) is to use a `data source` (page 371), in which case this handler is not needed (or called).

For outline views that do not use a data source, this event handler is called once for each currently displayed row in each column. For example, if an outline view displays a file system hierarchy in three columns, one each for the name, date modified, and size for each item, the outline view will call the `item value` handler three times for each displayed item (once for the name column, once for the modification date, and once for the size).

The handler is not called for collapsed rows that are not visible.

Syntax

`item value`	*reference*	required
`outline item`	*item*	optional
`table column`	*table column*	required

Parameters

reference
a reference to the `outline view` (page 377) that contains the item

`outline item` *item* (page 73)
the item to get the value for

`table column` *table column* (page 382)
the column of the item

Result

Anything
The value of the specified item; typically returned as a string.

Examples

The following `item value` event handler is from the Outline sample application distributed with AppleScript Studio. Outline uses an outline view to display items in the file system. This handler uses the column identifier to determine which kind of value to return for the item. It then calls on the Finder application to obtain the value (either a name, date, or a kind) and returns the value as a string.

```
on item value theObject outline item theItem table column theColumn
    set itemValue to ""

    if the identifier of theColumn is "name" then
        tell application "Finder"
            set itemValue to displayed name of (get item theItem)
                as string
        end tell
    else if the identifier of theColumn is "date" then
        tell application "Finder"
```

```
            set itemValue to modification date of
                (get item theItem) as string
        end tell
    else if the identifier of theColumn is "kind" then
        tell application "Finder"
            set itemValue to kind of (get item theItem) as string
        end tell
    end if

    return itemValue
end item value
```

number of browser rows

Called to obtain the number of rows in a browser view for the specified column.

Unlike other data views such as `outline view` (page 377) and `table view` (page 386), you currently cannot supply data to a browser view with a `data source` (page 371). As a result, performance may be inadequate for browser views that display more than a small number of items, and you should consider using one of the other data views, if suitable for your purpose.

Syntax

`number of browser rows`	*reference*	required
`in column`	*integer*	required

Parameters

reference
a reference to the `browser` (page 349) object for which to obtain the number of rows

`in column` *integer*
the one-based index of the column

Result

integer
Returns the number of rows in the specified column of the browser.

Examples

The following `number of browser rows` handler is from the Browser application distributed with AppleScript Studio. The Browser application browses the file system, displaying files and folders in a window similar to the Finder's column view. This handler:

- Sets the variable `rowCount` to 0.
- If there are no disk names displayed in the first column of the browser, it doesn't change the value of `rowCount`.
- If there are disk names displayed in the first column (`(count of diskNames) > 0`), and if the specified column is the first column, it sets `rowCount` to the count of disk names (which the application keeps track of separately in the `diskNames` property).
- If there are disk names displayed but the specified column is not the first column, it gets the file path for the column, then calls on the Finder application to obtain the count of items at that path. Then it sets `rowCount` to the count returned by the Finder.
- Returns `rowCount`.

```
on number of browser rows theObject in column theColumn
    set rowCount to 0

    if (count of diskNames) > 0 then
        if theColumn is 1 then
            set rowCount to count of diskNames
        else
            tell browser "browser" of window "main"
                set thePath to path for column theColumn - 1
            end tell

            tell application "Finder"
                set rowCount to count of items of item thePath
            end tell
        end if
    end if

    return rowCount
end number of browser rows
```

number of items

Called for an outline view to obtain the number of child items of the specified item.

The preferred way to provide data for an `outline view` (page 377) is to use a `data source` (page 371), in which case this handler is not needed (or called).

Syntax

`number of items`	*outline view*	required
`outline item`	*item*	optional

Parameters

reference
a reference to the `outline view` (page 377) that contains the items

`outline item` *item* (page 73)
the item for which to obtain the number of contained items

Examples

The following `number of items` handler is from the Outline application distributed with AppleScript Studio. Outline uses an outline view to display items in the file system. This handler uses the Finder application to count the number of file system items in the specified item. This process is described in more detail in the Examples sections for the `item expandable` (page 412) and `number of browser rows` (page 415) commands.

```
on number of items theObject outline item outlineItem
    set itemCount to 0

    tell application "Finder"
        if (count of diskNames) > 0 then
            if outlineItem is 0 then
                set itemCount to count of diskNames
            else
                set itemCount to count of items of (get item outlineItem)
            end if
        end if
    end tell
```

```
    return itemCount
end number of items
```

number of rows

Called for a table or outline view to get the number of rows.

The preferred way to manipulate data for `table view` (page 386) and `outline view` (page 377) objects is to use a `data source` (page 371), in which case this handler is not needed (or called).

Syntax

`number of rows` *reference* required

Parameters

reference
a reference to the `table view` (page 386) or `outline view` (page 377) for which to obtain the number of rows

Examples

The following `number of rows` handler is from the Table sample application distributed with AppleScript Studio. You will find it in the script file `WithoutDataSource.applescript`. Table demonstrates how to work with data displayed as rows and columns in a `table view` (page 386). Note, however, that although you can work with the data in a table view without a `data source` (page 371) object, as in this method, the preferred and far more efficient approach is to use a data source. The Table application demonstrates that approach, which doesn't require a `number of rows` handler, in the project script file `WithDataSource.applescript`.

The following handler just returns the count of contacts in a global script property, since there is one row per contact.

```
on number of rows theObject
    return count of contacts
end number of rows
```

should collapse item

Called for an outline view to determine if an item should be collapsed. The handler returns `true` to allow collapsing of the item, `false` to disallow it.

Syntax

`should collapse item`	*outline view*	required
`outline item`	*item*	optional

Parameters

reference
a reference to the `outline view` (page 377) that contains the items that may be collapsed

`outline item` *item* (page 73)
the item

Result

boolean
Return `true` to allow the item to collapse; `false` to prevent collapsing. If you implement this handler, you should always return a boolean value.

Examples

When you connect a `should collapse item` handler to an `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template. The `theObject` parameter refers to the outline view. Your handler should determine whether to allow the specified item to collapse, then return the appropriate value.

```
on should collapse item theObject outline item outlineItem
    set allowCollapse to false
    --Check variable, perform test, or call handler to see if OK to collapse
    -- If so, set allowCollapse to true
    return allowCollapse
end should collapse item
```

should expand item

Called for an outline view to determine if an item should be expanded. The handler returns `true` to allow expanding of the item, `false` to disallow it.

Syntax

`should expand item`	*outline view*	required
`outline item`	*item*	optional

Parameters

reference
a reference to the `outline view` (page 377) that contains the items that may be expanded

`outline item` *item* (page 73)
the item that may be expanded

Result

boolean
Return `true` to allow the item to expand; `false` to prevent expanding. If you implement this handler, you should always return a boolean value.

Examples

When you connect a `should expand item` handler to an `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template. The `theObject` parameter refers to the outline view. Your handler should determine whether to allow the specified item to expand, then return the appropriate value.

```
on should expand item theObject outline item outlineItem
    set allowExpand to false
    --Check variable, perform test, or call handler to see if OK to expand
    -- If so, set allowExpand to true
    return allowExpand
end should expand item
```

should select column

Called to determine if selection is allowed when a user clicks a column in a table or outline view (such as when a user attempts to drag a column to change its position). The handler should return `true` to allow selection or `false` to disallow it.

By default, column selection is turned on for `table view` (page 386) objects but not for `outline view` (page 377) objects, but you can change the setting in Interface Builder. You don't need to connect this handler unless you want to allow column selection in some cases but prevent it in others.

Syntax

`should select column`	*reference*	required
`table column`	*table column*	required

Parameters

reference
a reference to the `table view` (page 386) or `outline view` (page 377) that contains the column

`table column` *table column* (page 382)
the column to be selected

Result

boolean
Return `true` to allow the column to be selected; `false` to prevent selection. If you implement this handler, you should always return a boolean value.

Examples

When you connect a `should select column` handler to a `table view` or `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template. Your handler should determine whether to allow the specified column to be selected, then return the appropriate value.

```
on should select column theObject table column tableColumn
    set allowSelection to false
    --Check variable, perform test, or call handler to see if OK to select
    -- If so, set allowSelection to true
```

```
    return allowSelection
end should select column
```

should select item

Called to determine if selection is allowed when a user clicks an item in an outline view. The handler should return `true` to allow selection or `false` to disallow it. You don't need to connect this handler unless you want to allow item selection in some cases but prevent it in others.

Syntax

`should select item`	*reference*	required
`outline item`	*item*	optional

Parameters

reference
a reference to the `outline view` (page 377) that contains the item that may be selected

`outline item` *item* (page 73)
the item that may be selected

Result

boolean
Return `true` to allow the item to be selected; `false` to prevent selection. If you implement this handler, you should always return a boolean value.

Examples

When you connect a `should select item` handler to an `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template. Your handler should determine whether to allow the specified item to be selected, then return the appropriate value.

```
on should select item theObject outline item outlineItem
    set allowSelection to false
    --Check variable, perform test, or call handler to see if OK to select
    -- If so, set allowSelection to true
```

```
    return allowSelection
end should select item
```

should select row

Called to determine if selection is allowed when a user clicks a row in a table or outline view. The handler should return `true` to allow selection or `false` to disallow it. You don't need to connect this handler unless you want to allow row selection in some cases but prevent it in others.

Syntax

`should select row`	*reference*	required
`row`	*integer*	required

Parameters

reference
a reference to the `table view` (page 386) or `outline view` (page 377) that contains the row that may be selected

`row` *integer*
the one-based index of the row to be selected

Result

boolean
Return `true` to allow the row to be selected; `false` to prevent selection. If you implement this handler, you should always return a boolean value.

Examples

When you connect a `should select row` handler to a `table view` or `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template. Your handler should determine whether to allow the specified row to be selected, then return the appropriate value.

```
on should select row theObject row theRow
    set allowSelection to false
    --Check variable, perform test, or call handler to see if OK to select
    -- If so, set allowSelection to true
```

```
    return allowSelection
end should select row
```

should selection change

Called for a table or outline view to determine if the current selection should change. The handler returns `true` to allow selection to change, `false` to disallow it.

Syntax

`should selection change`	*reference*	required

Parameters

reference
a reference to the `table view` (page 386) or `outline view` (page 377) for which the selection may be changed

Result

boolean
Return `true` to allow the selection to change; `false` to prevent change. If you implement this handler, you should always return a boolean value.

Examples

When you connect a `should selection change` handler to a `table view` or `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template. Your handler should determine whether to allow the selection to change, then return the appropriate value.

```
on should selection change theObject
    set allowSelectionChange to false
    --Check variable, perform test, or call handler to see if OK to select
    -- If so, set allowSelectionChange to true
    return allowSelectionChange
end should selection change
```

will display browser cell

Called before a browser cell is displayed in a browser view. The handler cannot cancel the display operation, but can prepare for it.

Unlike other data views such as `outline view` (page 377) and `table view` (page 386), you cannot supply data to a `browser` (page 349) with a `data source` (page 371). As a result, performance may be inadequate for browser views that display more than a small number of items, and you should consider using one of the other data views, if suitable for your purpose.

Syntax

`will display browser cell`	*reference*	required
`browser cell`	*browser cell*	required
`in column`	*integer*	required
`row`	*integer*	required

Parameters

reference
a reference to the `browser` object that contains the cell that may be displayed

`browser cell` *browser cell* (page 356)
the cell that is to be displayed

`in column` *integer*
the one-based index of the column for the given cell

`row` *integer*
the one-based index of the row for the given cell

Examples

For an example of the `will display browser cell` handler, see the Examples section for the `browser cell` (page 356) class.

will display cell

Called before a cell is displayed for a table or outline view (thus displaying the data at a specified row and column). The handler cannot cancel the display operation, but can prepare for it.

The preferred way to manipulate data for table and outline views is to use a `data source` (page 371), in which case this handler is not needed (or called).

Syntax

`will display cell`	*reference*	required
`cell`	*anything*	required
`row`	*integer*	required
`table column`	*table column*	required

Parameters

reference
a reference to the object that contains the cell that will be displayed

`cell` *anything*
the cell that is about to be displayed; see `cell` (page 256), `image cell` (page 274), and `text field cell` (page 319)

`row` *integer*
the one-based index of the row of the cell to be displayed

`table column` *table column* (page 382)
the column of the cell to be displayed

Examples

When you connect a `will display cell` handler to a `table view` or `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. Your handler should do any required preparation for display of the specified cell.

```
on will display cell theObject row theRow cell theCell table column
tableColumn
    (* Prepare for cell to be displayed. *)
end will display cell
```

will display item cell

Called before a data cell is displayed for an outline view (thus displaying the data for a specified item and column). The handler cannot cancel the display operation, but can prepare for it.

The preferred way to manipulate data for table and outline views is to use a `data source` (page 371), in which case this handler is not needed (or called).

Syntax

`will display item cell`	*reference*	required
`cell`	*anything*	required
`outline item`	*item*	optional
`table column`	*table column*	required

Parameters

reference
a reference to the outline view that contains the item that will be displayed

`cell` *anything*
the cell that is about to be displayed; see `cell` (page 256), `image cell` (page 274), and `text field cell` (page 319)

`outline item` *item* (page 73)
the item that contains the cell that will be displayed

`table column` *table column* (page 382)
the column of the cell to be displayed

Examples

When you connect a `will display item cell` handler to an `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. Your handler should do any required preparation for display of the specified item cell.

```
on will display item cell theObject outline item outlineItem cell theCell
table column tableColumn
    (* Prepare for cell to be displayed. *)
end will display item cell
```

will display outline cell

Called before display of a cell that implements a disclosure triangle symbol in an outline view. The handler cannot cancel the display operation, but can prepare for it.

The preferred way to manipulate data for table and outline views is to use a `data source` (page 371), in which case this handler is not needed (or called).

Syntax

`will display outline cell`	*reference*	required
`cell`	*anything*	required
`outline item`	*item*	optional
`table column`	*table column*	required

Parameters

reference
a reference to the outline view that contains the outline cell that will be displayed

`cell` *anything*
the cell that is about to be displayed; see `cell` (page 256), `image cell` (page 274), and `text field cell` (page 319)

`outline item` *item* (page 73)
the item that contains the cell that will be displayed

`table column` *table column* (page 382)
the column of the cell to be displayed

Examples

When you connect a `will display outline cell` handler to an `outline view` object in Interface Builder, AppleScript Studio supplies an empty handler template like the following. The `theObject` parameter refers to the table or outline view. Your handler should do any required preparation for display of the specified outline cell.

```
on will display outline cell theObject outline item outlineItem cell theCell
table column tableColumn
    (* Prepare for cell to be displayed. *)
end will display outline cell
```

Document Suite

This chapter describes the terminology in AppleScript Studio's Document suite. For other suites, see the following:

- "Application Suite" (page 42)
- "Container View Suite" (page 194)
- "Control View Suite" (page 244)
- "Data View Suite" (page 348)
- "Drag and Drop Suite" (page 448)
- "Menu Suite" (page 462)
- "Panel Suite" (page 474)
- "Text View Suite" (page 516)

Document Suite

The **Document suite** provides terminology for working with documents in AppleScript Studio applications. This suite defines AppleScript Studio's version of the `document` (page 432) class, which takes the place of the one defined in Cocoa's Standard suite (described in "Terminology Supplied by the Cocoa Application Framework" (page 28)).

The Document suite was added in AppleScript Studio version 1.2 to make it easier to create document-based applications. The suite includes a pair of high-level event handlers (`data representation` (page 439) and `load data representation` (page 440)) and a pair of low-level handlers (`read from file` (page 442) and `write to file` (page 443)). In addition, the `window` (page 86) class now has a `document` element to provide access to a document from within the user interface.

Project Builder provides the AppleScript Document-based Application project template for applications that create and manage multiple documents. To support the Document suite, the settings for this project template were revised. The project's `Document.applescript` script file now includes empty versions of the `data representation` and `load data representation` event handlers. The Task List sample application (available with AppleScript Studio version 1.2) demonstrates how to read and write simple files with these high-level handlers. The Plain Text Editor sample application (also available starting with version 1.2) demonstrates how to read and write slightly more complex text files with the low-level handlers (`read from file` and `write to file`).

Whichever handlers you use, the great advantage of AppleScript Studio's document support is that your application doesn't need to do any of the work to put up the Open, Save, or Save as panels, or even to worry about the filenames chosen by the user. For the high-level handlers, it just reads or writes its data when the appropriate handler is called. For the low-level handlers, it uses the filename that is passed to the handler.

The classes, commands, and events in the Document suite are described in the following sections:

"Events" (page 439)

For enumerated constants, see "Enumerations" (page 177).

Classes

The Document suite contains the following classes:

document (page 432)

document

Plural: documents
Inherits from: responder (page 80)
Cocoa Class: NSDocument

Represents data displayed in windows that can typically be read from and written to files.

The Cocoa application framework provides a great deal of basic document support and, starting with AppleScript Studio version 1.2, you can take advantage of much of that support in AppleScript Studio document-based applications. For example, if you create a new AppleScript Document-based Application project in Project Builder, without making any changes, the application can open multiple untitled windows and even save them, though without any application data.

Figure 7-1 shows the Groups & Files pane for a new document-based project (named DefaultDocumentProject), with several groups expanded. Many of the items are common to all AppleScript Studio project templates, and are described in *Inside Mac OS X: Building Applications With AppleScript Studio* (see "Related Documentation" (page 19) for information on that document). The following is a description of the default items that are unique to a document-based project:

- Document.applescript is the default document script file. Starting with AppleScript Studio version 1.2, this file contains minimal data representation (page 439) and load data representation (page 440) handlers. You will have to fill in these handlers to supply data for saving to a document and to load data that has been read from a document.

  If you have other document-related script statements, you can add them to this file.

- Document.nib is the nib file for creating document-associated windows. Nib files are described with the awake from nib (page 130) command.

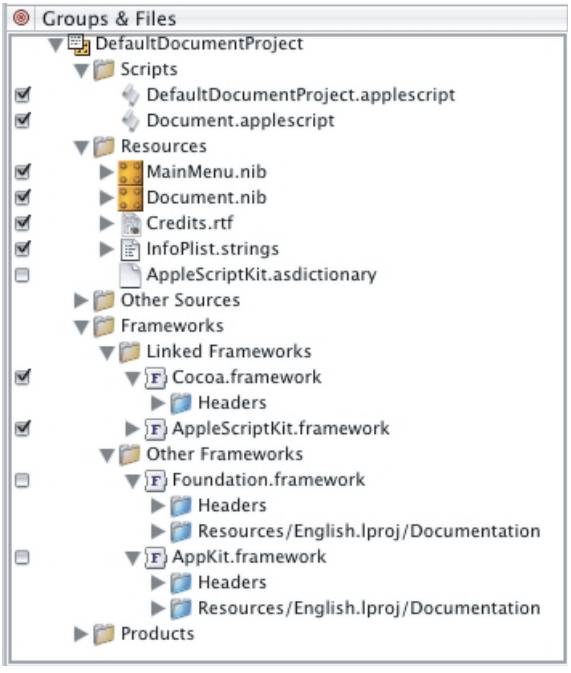
- `Credits.rtf` is a rich text format (`.rtf`) file that supplies text for the default About window in a Cocoa application. You edit this file to supply your own About window information.

The `Cocoa.framework` framework is listed in the Linked Frameworks group. Its target checkbox is checked, indicating it is part of the current target. The Headers group for Cocoa actually contains only one header file, Cocoa.h (not shown). That file imports the headers `Foundation.h` and `AppKit.h`, two files that in turn import all the header files for the two frameworks that make up Cocoa, `AppKit.framework` and `Foundation.framework`.

You can see in Figure 7-1 that the Other Frameworks group contains both `AppKit.framework` and `Foundation.framework` (this is true for all AppleScript Studio projects). These frameworks are listed in the Other Frameworks group to provide convenient, searchable access to header files that contain all the Cocoa classes, methods, and constants available to your AppleScript Studio application. Your application can make use of this information either in Objective-C or other code you write, or in application scripts that use the `call method` (page 102) command.

The folders named `Resources/English.lprog/Documentation` in the AppKit and Foundation frameworks contain documentation for Cocoa. It's the same documentation you can access by choosing Cocoa Help in Project Builder's Help menu.

Figure 7-1 Default contents of the Groups & Files pane in an AppleScript Document-based Application project



After creating a default document-based application project, you still have some significant work to do to take full advantage of AppleScript Studio's document support. That work falls into the following main categories:

- Adding user interface items to the document nib file.
- Providing your data for writing to a file and extracting your data when reading from a file:
  - For simple documents, you can use the high-level handlers `data representation` (page 439) and `load data representation` (page 440).
  - For more complex documents, you may want to use the low-level handlers, `write to file` (page 443) and `read from file` (page 442), instead.
- Supplying document type information in Project Builder. You can use the Target editor to specify:

- a named document type
- associated extensions (such as `"txt"`)
- associated OS types (four-character codes, such as `"TEXT"`)
- whether the application can edit the file type, or just view it
- an icon to associate with the document type

As a minimum for reading and writing files, you supply a document type and an extension, and specify that the application can edit the type.

Whether you use high-level or low-level handlers to read and write data, the great advantage of AppleScript Studio's document support is that your application doesn't need to do any of the work to put up the Open, Save, or Save as panels, or even to worry about the filenames chosen by the user. For the high-level handlers, it just reads or writes its data when the appropriate handler is called. For the low-level handlers, it uses the filename that is passed to the handler.

AppleScript Studio includes two sample applications, available starting with version 1.2, that demonstrate how to work with document-based applications. See the Examples section for more information.

Properties of document objects

In addition to the properties it inherits from the `responder` (page 80) class, a document object has these properties:

`file name`

Access: read/write
Class: *Unicode text*
the file name of the document; undefined for a new document until set (as when the document is saved); a POSIX-style (slash-delimited) string; for example, `"/Users/yourUser/Documents/someFile.ext"`

`file type`

Access: read/write
Class: *Unicode text*
the file type of the document; this is not the four-character type scripters may be familiar with; it is a string, such as "DocumentType", you set in Project Builder; (you can also, in Project Builder, define a four-character type, as well as an extension, for your documents)

`modified`

Access: read/write
Class: *boolean*
Has the document been modified?

`name`

Access: read only
Class: *Unicode text*
the name of the document; by default, "Untitled" for the first document; the only ways to change the name are to use Save As to save the document with a new name, or to separately change the name on disk

Elements of document objects

A `document` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30). See the Version Notes section for this class for the AppleScript Studio version in which a particular element was added.

`window` (page 86)
Specify by: "Standard Key Forms" (page 30)
the document's windows; see Version Notes section below

Commands supported by document objects

Your script can send the following commands to a document object:

`close` (from Cocoa's Core suite, described in Core Suites in the Cocoa Programming Topic Scriptable Applications)
`print` (from Cocoa's Core suite)
`save` (from Cocoa's Core suite)

Events supported by document objects

A document object supports handlers that can respond to the following events:

Document

`data representation` (page 439)
`load data representation` (page 440)
`read from file` (page 442)

write to file (page 443)

Nib

awake from nib (page 130)

Examples

The Plain Text Editor sample application (distributed with AppleScript Studio, starting with version 1.2) demonstrates how to read and write simple text files with the low-level handlers read from file (page 442) and write to file (page 443). The Examples sections for these handlers show the Plain Text Editor versions of these handlers. The Task List sample application (also available starting with version 1.2) demonstrates how to read and write simple files with the high-level handlers data representation (page 439) and load data representation (page 440). The Examples sections for these handlers show the Task List versions of these handlers.

You can use statements like the following in Script Editor to access document properties in an AppleScript Studio document-based application. Similar statements will work within an AppleScript Studio application script (though you won't need the tell application statement).

```
tell application "Document Application"
    set myName to name of the first document
    -- result: "Untitled 2"
end
```

Version Notes

The following changes were made for AppleScript Studio version 1.2:

- The document class was moved from the Application suite to its own Document suite.
- The file type property was added.
- Support was added for the following events: data representation (page 439), load data representation (page 440), read from file (page 442), and write to file (page 443).
- You can no longer connect a will open (page 170) handler to a document. However, you can connect that handler to a document's window (see the next item).

- The `window` element was added to the `document` class to provide access to interface elements that are associated with the document. Note that because `window` is an element (addressable by name, index, id, and so on), not a property, the statement `window of document 1` returns a list of 1 window (`{window id 1}`). That is, `window` is synonymous with `windows` for an element.
- Note also that the `document` element was added to the `window` (page 86) class, so that for windows that have an associated document, user interface elements can access the document.

Events

Objects in the Document Suite support handlers for the following events (an event is an action, typically generated through interaction with an application's user interface, that causes a handler for the appropriate object to be executed). You use these events with `document` (page 432) objects.

- `data representation` (page 439)
- `load data representation` (page 440)
- `read from file` (page 442)
- `write to file` (page 443)

`data representation`

Called when a document is about to be saved to supply the document's data. This handler is called as a result of a user opening the Save or Save as panels (or using the key equivalents) and choosing to save the document.

This is a high-level handler that you use when you want to create documents that are specific to your application. Your handler returns the document's data in any form you choose, such as a simple string, a list, a record, or other type of data. The application doesn't have to deal with opening a file and writing data—AppleScript Studio automatically saves the data in the document.

The counterpart to `data representation` is `load data representation` (page 440).

Syntax

`data representation`	*reference*	required
`of type`	*Unicode text*	required

Parameters

reference
a reference to the object whose `data representation` handler is called

`of type` *Unicode text*
the type (extension) of the document file

Examples

The Task List sample application (distributed with AppleScript Studio, starting with version 1.2), provides the following handler to demonstrate the high-level mechanism for writing data to files. In this case, the handler gets information from the `data source` (page 371) object for the `table view` (page 386) that displays the task list. The information includes the list of tasks, the name of the current sort column, and the sort order of the current sort column. The handler returns that information, which is all the application needs to recreate the current window state, in a record.

```
on data representation theObject of type ofType
    -- Set up local variables
    set theWindow to window 1 of theObject
    set theDataSource to data source of table view "tasks"
        of scroll view "tasks" of theWindow
    set theTasks to contents of every data cell of every data row
        of theDataSource
    set theSortColumn to sort column of theDataSource

    -- Create a record containing the list of tasks (just a list of lists),
    --  the name of the sort column, and the sort order.
    set theDataRecord to {tasks:theTasks,
        sortColumnName:name of theSortColumn,
        sortColumnOrder:sort order of theSortColumn}

    return theDataRecord
end data representation
```

Version Notes

The `data representation` event handler was added in AppleScript Studio version 1.2.

The Task List sample application was added in AppleScript Studio version 1.2.

load data representation

Called to load a document's data when the document is opened. This handler is called as a result of a user opening the Open panel and selecting one or more files to open, or of the user dragging an application document onto the application icon

or double-clicking an application document icon. The data provided to this event handler is the same data that the application supplied in the `data representation` (page 439) handler when the document was saved.

This is a high-level handler that you use for documents that are specific to your application. Your handler loads the supplied data in any form you choose (the same form in which it previously supplied the data in its `data representation` handler). The application doesn't have to deal with opening a file and reading data—AppleScript Studio automatically supplies the data from the document.

The counterpart to `load data representation` is `data representation`.

Syntax

`load data representation`	*reference*	required
`of type`	*Unicode text*	required
`with data`	*item*	required

Parameters

reference
a reference to the object whose `load data representation` handler is called

`of type` *Unicode text*
the type (extension) of the document file

`with data` *item*
the data to loaded from the document

Examples

The Task List sample application (distributed with AppleScript Studio, starting with version 1.2), provides the following handler to demonstrate the high-level mechanism for reading data from files. The parameter `theData` refers to an object of the same type as the one that was saved by the `data representation` (page 439) event handler—that is, a record that contains the list of tasks, the name of the current sort column, and the sort order of the current sort column. This handler extracts that information and inserts it into the `data source` (page 371) object for the `table view` (page 386) that displays the task list.

```
on load data representation theObject of type ofType with data theData
```

```
    -- Set up local variables
    set theWindow to window 1 of theObject
    set theDataSource to data source of table view "tasks"
        of scroll view "tasks" of theWindow

    -- Restore the sort column and sort order of the data source
    --  based on the information saved
    set sort column of theDataSource
        to data column (sortColumnName of theData) of theDataSource
    set sort order of sort column of theDataSource
        to (sortColumnOrder of theData)

    -- Use the "append" command to quickly populate the data source
    --  with the list of tasks
    append the theDataSource with (tasks of theData)

    -- Return true, signaling success. If you return "false",
    --  the document will fail to load and an alert will be presented.
    return true
end load data representation
```

Version Notes

The `load data representation` event handler was added in AppleScript Studio version 1.2.

The Task List sample application was added in AppleScript Studio version 1.2.

read from file

Called when the application needs to read a document's data. This handler is called as a result of a user opening the Open panel and selecting one or more files to open, or of the user dragging an application document onto the application icon or double-clicking an application document icon. The data in the document is the same data that was written by the `write to file` (page 443) handler when the document was saved.

This is a low-level handler that you use to work with more complicated documents or documents other applications might read, such as text files. The handler is responsible for opening the file specified by the passed `path name` parameter and for

closing it after reading is complete. The handler reads the supplied data according to the type supplied in the `of type` parameter. See the `write to file` handler for more information on document types.

The counterpart to `read from file` is `write to file` (page 443).

Syntax

`read from file`	*reference*	required
`of type`	*Unicode text*	required
`path name`	*Unicode text*	required

Parameters

reference
a reference to the object whose `read from file` handler is called

`of type` *Unicode text*
the type (extension) of the file

`path name` *Unicode text*
the path name (in POSIX, or slash-delimited, format) of the file to read from

Examples

The Plain Text Editor sample application (distributed with AppleScript Studio, starting with version 1.2) demonstrates how to read and write simple text files with the low-level handlers `read from file` (page 442) and `write to file` (page 443), including opening and closing a document file.

Version Notes

The `read from file` event handler was added in AppleScript Studio version 1.2.

`write to file`

Called when the application needs to write a document's data. This handler is called as a result of a user opening the Save or Save as panels (or using the key equivalents) and choosing to save the document. The data in the document is the same data that was written by the `write to file` (page 443) handler when the document was saved.

This is a low-level handler that you use to work with more complicated documents or documents other applications might read, such as text files. The handler is responsible for opening the file specified by the passed `path name` parameter and for closing it after writing. The handler writes the document's data according to the type supplied in the `of type` parameter.

By default, the document type for an AppleScript Studio document-based application is set to "DocumentType". You can change the document type for the active target in a Project builder project by modifying the Document Types section of the Target Editor. For example, in some versions of Project Builder, Document Types is in the Simple View section in the Info.plist Entries section. You can also make changes to other document-related information, including specifying the Extensions and OS types. If you specify more than one document type, the application will get a pop-up in the Save panel (when saving the document), which allows a user to specify the document type to save as. The value chosen in the pop-up is the value passed in the `of type` parameter.

The counterpart to `write to file` is `read from file` (page 442).

Syntax

`write to file`	*reference*	required
`of type`	*Unicode text*	required
`path name`	*Unicode text*	required

Parameters

reference
a reference to the object whose `write to file` handler is called

`of type` *Unicode text*
the type (extension) of the file

`path name` *Unicode text*
the path name (in POSIX, or slash-delimited, format) of the file to write to

Examples

The Plain Text Editor sample application (distributed with AppleScript Studio, starting with version 1.2) demonstrates how to read and write simple text files with the low-level handlers `read from file` (page 442) and `write to file` (page 443).

Version Notes

The `write to file` event handler was added in AppleScript Studio version 1.2.

Document Suite

CHAPTER 8

Drag and Drop Suite

This chapter describes the terminology in the Drag and Drop suite, which is available starting in AppleScript Studio version 1.2. For other suites, see the following:

- "Application Suite" (page 42)
- "Container View Suite" (page 194)
- "Control View Suite" (page 244)
- "Data View Suite" (page 348)
- "Document Suite" (page 430)
- "Menu Suite" (page 462)
- "Panel Suite" (page 474)
- "Text View Suite" (page 516)

Drag and Drop Suite

The **Drag and Drop suite** defines terms for working with drag and drop, including the `drag info` class to provide information about a drag, and events to drag, track, prepare for a drop, and conclude a drop. For related information, see the `pasteboard` (page 76) class.

Initial drag-and-drop support was added in AppleScript Studio 1.2. It provides the ability for various user interface elements to receive drag events. It does not, however, allow for initiating drag operations (other than those already supported by Cocoa classes).

Among the classes that support drag and drop are `button` (page 246), `clip view` (page 199), `color well` (page 263), `combo box` (page 265) `control` (page 270),`image view` (page 276), `matrix` (page 280), `movie view` (page 286), `popup button` (page 292), `progress indicator` (page 296), `scroll view` (page 211), `slider` (page 306), `stepper` (page 310), `tab view` (page 219), `text field` (page 314), `text view` (page 520), and `view` (page 226).

For an object to respond to any of the drag-and-drop event handlers (described in "Events" (page 452)), you must register the drag types that the object can accept. You do this with the `register` (page 123) command, using its `drag type` parameter to supply a list of the supported pasteboard drag types. Possible pasteboard types are listed with the `pasteboard` (page 76) class.

The `drop` (page 457) event handler is the only required handler for supporting dropped data in AppleScript Studio. However, see the `conclude drop` (page 452) handler for information on providing drag and drop for `text view` (page 520) and `text field` (page 314) objects.

The classes and events in the Drag and Drop suite are described in the following sections:

For enumerated constants, see "Enumerations" (page 177).

Classes

The Drag and Drop suite contains the following class:

`drag info` (page 449)

`drag info`

Plural: `drag infos`
Inherits from: `item` (page 73)
Cocoa Class: ASKDragInfo

Represents information and data for the current drag operation. A `drag info` object is passed to each of the drag-and-drop event handlers described in "Events" (page 452).

The most useful property of the `drag info` class is the `pasteboard` (page 76) property. It contains the data for the drag, which your application can extract and use in the `drop` (page 457) or other event handlers.

Properties of drag info objects

In addition to the properties it inherits from the `item` (page 73) class, a drag info object has these properties:

`destination window`

Access: read only
Class: *window* (page 86)
the destination window for the dragging operation

`image`

Access: read only
Class: *anything*
the image being dragged

`image location`

Access: read only
Class: *point*
not supported (through AppleScript Studio version 1.2); the location of the image being dragged; the location is returned as a two-item list of numbers {left,

bottom}; see the `bounds` property of the `window` (page 86) class for information on the coordinate system

`location`

Access: read only
Class: *point*
not supported (through AppleScript Studio version 1.2); the current location in the destination window for the dragging operation, as a two-item list of numbers {left, bottom}; each window has its own coordinate system, with the origin in the lower left corner; see the `bounds` property of the `window` (page 86) class for more information on the coordinate system

`pasteboard`

Access: read only
Class: *pasteboard* (page 76)
the pasteboard that contains the drag data

`sequence number`

Access: read only
Class: *integer*
not supported (through AppleScript Studio version 1.2); the unique identifier for the dragging operation

`source`

Access: read only
Class: *item* (page 73)
the source of the dragged data

`source mask`

Access: read only
Class: *integer*
not supported (through AppleScript Studio version 1.2); defines the type of drag (drag operation link, drag operation copy, drag operation generic)

Events supported by drag info objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The following example shows one way to examine a `drag info` object in a `drop` (page 457) event handler. This handler returns `false`, cancelling the drop, if the pasteboard of the drag info does not contain the "string" type. If it does contain that type, the handler uses the string data from the pasteboard to set the title of the object the data was dropped on and returns `true` to complete the drop operation.

```
on drop theObject drag info dragInfo
    set dropped to false
    if "string" is in types of pasteboard of dragInfo then
        set title of theObject to contents of pasteboard of dragInfo
        set dropped to true
    end if
    return dropped
end drop
```

Version Notes

The `drag info` class was added in AppleScript Studio version 1.2.

The `image location`, `location`, `sequence number`, and `source mask` properties in this class are not supported, through AppleScript Studio version 1.2.

Events

Objects in the Drag and Drop Suite support the following events (an event is an action, typically generated through interaction with an application's user interface, that causes a handler for the appropriate object to be executed). To determine which classes support which commands, see the individual class descriptions.

- `conclude drop` (page 452)
- `drag` (page 453)
- `drag entered` (page 454)
- `drag exited` (page 455)
- `drag updated` (page 456)
- `drop` (page 457)
- `prepare drop` (page 459)

`conclude drop`

Called to complete a successful drop operation. This handler is only called if both the `prepare drop` (page 459) and `drop` (page 457) event handlers were successful.You can use this handler, for example, to clean up any state that was set or changed in the `prepare drop` handler.

The `drop` handler is the only required handler for supporting dropped data in AppleScript Studio.

Both the `text view` (page 520) and `text field` (page 314) classes have built-in support for dropped text, provided automatically by the Cocoa classes they are based on (NSTextView and NSTextField). If that's all the support for dropped text your application needs for objects of these types, you don't need to do anything extra. However, an application that wants to handle data that is dropped on a text view or a text field, and not allow the default support to handle it, should take these steps:

- register the drag types the text view or text field can handle; see the `register` (page 123) command for details
- connect a `drop` handler to handle dropped text for the text view or text field

- connect an empty `conclude drop` handler for the text view or text field; this will prevent the text objects from providing their own support for dropped text

Syntax

`conclude drop`	*reference*	required
`drag info`	*drag info*	required

Parameters

reference
a reference to the object whose `conclude drop` handler is called

`drag info` *drag info* (page 449)
the information about the drag operation

Examples

When you connect a `conclude drop` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template. Your handler can take any necessary steps to deal with the conclusion of the drop that it didn't take care of in its `drop` (page 457) handler. For example, the handler might clean up any state that was set or changed in the `prepare drop` (page 459) handler.

```
on conclude drop theObject drag info dragInfo
    (* Statements to deal with the concluded drop. *)
end conclude drop
```

See also the description above for this class.

Version Notes

The `conclude drop` event handler was added in AppleScript Studio version 1.2.

drag

Not supported (through AppleScript Studio version 1.2). Support for this event handler is planned for a later version of AppleScript Studio.

Syntax

`drag`	*reference*	required
`drag info`	*drag info*	required

Parameters

reference
a reference to the object whose `drag` handler is called

`drag info` *drag info* (page 449)
the information about the drag operation

Version Notes

The `drag` event handler was added in AppleScript Studio version 1.2, although it does nothing in that version.

drag entered

Called when a user drags the registered type of data into the bounds of the object. Your application may not need a `drag entered` handler, as the `drop` (page 457) handler is the only required handler for supporting dropped data in AppleScript Studio.

Syntax

`drag entered`	*reference*	required
`drag info`	*drag info*	required

Parameters

reference
a reference to the object whose `drag entered` handler is called

`drag info` *drag info* (page 449)
the information about the drag operation

Examples

When you connect a `drag entered` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template. The `theObject` parameter refers to the object for which a potential drag has entered the object's bounds. The `dragInfo` parameter provides access to a `drag info` (page 449) object that contains all pertinent information for the drag operation. Your handler can take any necessary steps, such as providing visual feedback that it can accept information from the drag.

```
on drag entered theObject drag info dragInfo
    (* Statements to deal with the drag entering. *)
end drag entered
```

Version Notes

The `drag entered` event handler was added in AppleScript Studio version 1.2.

drag exited

Called when a user drags the registered type of data out of the bounds of the object. Your application may not need a `drag exited` handler, as the `drop` (page 457) handler is the only required handler for supporting dropped data in AppleScript Studio.

Syntax

`drag exited`	*reference*	required
`drag info`	*drag info*	required

Parameters

reference
a reference to the object whose `drag exited` handler is called

`drag info` *drag info* (page 449)
the information about the drag operation

Examples

When you connect a `drag exited` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template. The `theObject` parameter refers to the object for which a potential drag has exited the object's bounds. The `dragInfo` parameter provides access to a `drag info` (page 449) object that contains all pertinent information for the drag operation. Your handler can take any necessary steps, such as providing visual feedback that it is no longer waiting to accept information from the drag.

```
on drag exited theObject drag info dragInfo
    (* Statements to deal with the drag exiting. *)
end drag exited
```

Version Notes

The `drag exited` event handler was added in AppleScript Studio version 1.2.

drag updated

Called when a user moves a registered drag type within the bounds of the object. Your application may not need a `drag updated` handler, as the `drop` (page 457) handler is the only required handler for supporting dropped data in AppleScript Studio.

Syntax

`drag updated`	*reference*	required
`drag info`	*drag info*	required

Parameters

reference
a reference to the object whose `drag updated` handler is called

`drag info` *drag info* (page 449)
the information about the drag operation

Examples

When you connect a `drag updated` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template. The `theObject` parameter refers to the object in whose bounds a drag with a registered type has moved. The `dragInfo` parameter provides access to a `drag info` (page 449) object that contains all pertinent information for the drag operation. Your handler can take any necessary steps, such as providing visual feedback as to where the drag data might be inserted. Note however, that through AppleScript Studio version 1.2, you cannot access the `location` property of a `drag info` object, so the ability to determine an insertion point is limited.

```
on drag updated theObject drag info dragInfo
    (* Statements to deal with the drag updating. *)
end drag updated
```

Version Notes

The `drag updated` event handler was added in AppleScript Studio version 1.2.

drop

Called when a user has dropped data with a registered data type onto an object. You return `false` from this event handler to cancel the drop operation; otherwise you must return `true` to signal success for the drop operation.

You can examine the `pasteboard` (page 76) property of the `drag info` (page 449) parameter to obtain the data for the drop in the requested format. If you use the `prepare drop` (page 459) handler to verify that the data you need is present, you will know that data is available when the `drop` handler is called.

The `drop` handler is the only required handler for supporting dropped data in AppleScript Studio. However, see the `conclude drop` (page 452) event handler for information on providing drag and drop for `text view` (page 520) and `text field` (page 314) objects.

Syntax

`drop`	*reference*	required
`drag info`	*drag info*	required

Parameters

reference
a reference to the object whose handler is called to receive the dropped data

`drag info` *drag info* (page 449)
the information about the drag operation

Examples

The Drag and Drop sample application (distributed with AppleScript Studio, starting with version 1.2) demonstrates how various objects can accept dragged data. The following `drop` handler, from the file `Text.applescript` in that application, shows how to accept dragged text data in a `text field` (page 314).

The `drop` event handler is called when the appropriate type of data is dropped onto the object. All of the pertinent information about the drop is contained in the `drag info` (page 449) object (passed in the `drag info` (page 449) parameter). In this case, the handler just checks the `pasteboard` (page 76) of the `drag info` object for the "string" data type. If it is present, the handler uses it to set the contents of the passed text field (from the `theObject` parameter).

```
on drop theObject drag info dragInfo
    -- We are only interested in the "string" data type
        -- If that type is present, set the contents of the text
        --  field to the contents of the pasteboard
    if "string" is in types of pasteboard of dragInfo then
        set string value of theObject to contents of pasteboard of dragInfo
    end if

    return true
end drop
```

By default, the `preferred type` property for a pasteboard is "string", which is why the handler above doesn't set the preferred type. To explicitly set the preferred type of a pasteboard (in this example, the "general" pasteboard) to "string" before getting data from the pasteboard, you could use the following statements:

```
set preferred type of pasteboard "general" to "string"
if "string" is in types of pasteboard "general" then
    set myString to contents of pasteboard "general"
```

Version Notes

The `drop` event handler was added in AppleScript Studio version 1.2.

prepare drop

Called when a user has dropped the dragged data onto the object. You return `false` from this event handler to cancel the drop operation; otherwise you must return `true` to continue the drop operation.

You can examine the `pasteboard` (page 76) property of the `drag info` parameter to determine, for example, whether it contains data in the format you want.

Your application may not need a `prepare drop` handler, as the `drop` (page 457) handler is the only required handler for supporting dropped data in AppleScript Studio.

Syntax

`prepare drop`	*reference*	required
`drag info`	*drag info*	required

Parameters

reference
a reference to the object whose handler is called

`drag info` *drag info* (page 449)
the information about the drag operation

Examples

When you connect a `prepare drop` handler to an object in Interface Builder, AppleScript Studio supplies an empty handler template. The `theObject` parameter refers to the object for which a drop is about to take place. The `dragInfo` parameter provides access to a `drag info` (page 449) object that contains all pertinent information for the drag operation. Your handler can do any necessary preparation,

such as making sure the necessary data is present for the drop operation to take place. The handler should return `false` to cancel the drop operation or `true` to allow it.

```
on prepare drop theObject drag info dragInfo
    (* Statements to prepare for a drop.
        In this example, only allow drop if string info available. *)
    if "string" is in types of pasteboard of dragInfo then
        return true
    else
        return false
    end if
end prepare drop
```

Version Notes

The `prepare drop` event handler was added in AppleScript Studio version 1.2.

Menu Suite

This chapter describes the terminology in AppleScript Studio's Menu suite. For other suites, see the following:

- "Application Suite" (page 42)
- "Container View Suite" (page 194)
- "Control View Suite" (page 244)
- "Data View Suite" (page 348)
- "Document Suite" (page 430)
- "Drag and Drop Suite" (page 448)
- "Panel Suite" (page 474)
- "Text View Suite" (page 516)

Menu Suite

The Menu suite defines a small number of classes and event handlers for working with menus. It includes the `menu` (page 463) and `menu item` (page 465) classes and the `choose menu item` (page 470) and `update menu item` (page 470) event handlers, described in the following sections:

For enumerated constants, see "Enumerations" (page 177).

Classes

The Menu suite contains the following classes:

`menu` (page 463)

`menu item` (page 465)

`menu`

Plural: `menus`
Inherits from: `item` (page 73)
Cocoa Class: NSMenu

Represents a menu. For every menu in the menu bar (such as Application, File, Edit, and so on) there is a `menu` object. For every menu item in a menu (such as New or Open), there is a `menu item` (page 465) object.

Figure 9-1 The File menu in Interface Builder

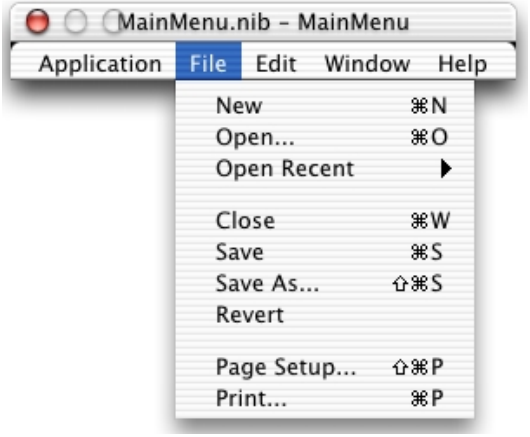


Figure 9-2 shows the default menus for an AppleScript Studio application (when you create the project by choosing either the AppleScript or the AppleScript Document-based application template in Project Builder). The menus are shown in an Interface Builder nib window, with the File menu open.

You can add a menu in Interface Builder by dragging a Submenu item from the Cocoa-Menus pane of the Palette window to the main menu. You can set various attributes for menu in Interface Builder's Info window.

Although it is possible to change a menu dynamically by adding and deleting items, all the added menu items will be disabled and there is no way (through AppleScript Studio version 1.2) to connect a handler to respond to a user choosing one of the added items. However, you can dynamically populate the menu for a popup button, as shown in the Examples section for the `popup button` (page 292) class.

For more information on menus, see the Cocoa Programming Topic Application Menus and Pop-up Lists.

Properties of menu objects

A `menu` object has these properties:

`auto enables items`

Access: read/write
Class: *boolean*
Should the menu auto enable its items? default is `true`; you can set this value in Interface Builder; however, your application has no convenient way of enabling and disabling menu items on standard application menus (such as the File and Edit menus), so setting this property to `false` for those menus is not recommended; to read more about Cocoa's underlying support for menu enabling, see NSMenu and NSMenuValidation

`super menu`

Access: read/write
Class: *menu* (page 463)
the menu that contains this menu

`title`

Access: read/write
Class: *Unicode text*
the title of the menu; you can set this value in Interface Builder

Elements of menu objects

A `menu` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

menu (page 463)
Specify by: "Standard Key Forms" (page 30)
the menu's submenus

menu item (page 465)
Specify by: "Standard Key Forms" (page 30)
the menu's menu items

Events supported by menu objects

A menu object supports handlers that can respond to the following events:

Nib

awake from nib (page 130)

Examples

The following script statements, taken from a script running in the Script Editor application, target the menus of a simple AppleScript Studio application. You can use similar statements within an AppleScript Studio application script, though you won't need the tell application statement.

```
tell application "TestApp"
    first menu -- result: main menu of application "TestApp"
    title of main menu -- result: "MainMenu"
    menu items of main menu -- result: a list of menu items
    title of menu items of main menu
    -- result: {"", "File", "Edit", "Window", "Help"}
    menus of main menu -- result: long list
    -- {sub menu of menu item id 1 of main menu of application
         "TestApp", etc. }
    menu items of sub menu of menu item id 1 of main menu
        -- result: a long list
end tell
```

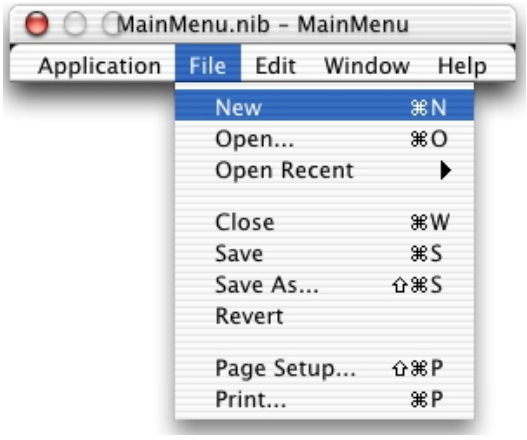
The Examples section for the menu item (page 465) class shows how to access additional properties.

menu item

Plural: menu items

Inherits from: item (page 73).
Cocoa Class: NSMenuItem
Represents an item in a menu. Every menu item is associated with a menu (page 463) object. Figure 9-2 shows the New menu item in the File menu in an Interface Builder nib window.

Figure 9-2 The New menu item in the File menu in Interface Builder



You can add a menu item to a menu in Interface Builder by dragging an Item from the Cocoa-Menus pane of the Palette window to the menu. You can set various attributes for menu items in Interface Builder's Info window.

For more information on menus, see the Cocoa Programming Topic Application Menus and Pop-up Lists.

Properties of menu item objects

A menu item object has these properties:

`associated object`

Access: read/write
Class: *item* (page 73)
the object associated with the menu item

`enabled`

Access: read/write
Class: *boolean*
Is the menu item enabled? you can connect a `update menu item` (page 470) handler to a menu item to gain control over whether it should be enabled or disabled

`has sub menu`

Access: read only
Class: *boolean*
Does the menu item have a sub menu?

`image`

Access: read/write
Class: *image* (page 72)
the image for the menu item; by default, no image is assigned to a menu

`key equivalent`

Access: read/write
Class: *Unicode text*
the key equivalent to select the menu item; default is no key equivalent for menu items you add in Interface Builder, but you can specify a key equivalent in the Attributes pane of the Info window; default menu items have default key equivalents (such as Command-N for the New menu item in the File menu)

`key equivalent modifier`

Access: read/write
Class: *number*
not supported (through AppleScript Studio version 1.2); the modifier key for the key equivalent; default is no key equivalent modifier for menu items you add in Interface Builder, but you can set a modifier in the Attributes pane of the Info window; default menu items have default key equivalents modifies (such as Shift-Command-P for the Page Setup… menu item in the File menu)

`menu`

Access: read/write
Class: *menu* (page 463)
the menu that contains the menu item; when you create and modify menus in Interface Builder, this property is set automatically

`separator item`

Access: read/write
Class: *boolean*
Is the menu item a separator item? you can add separator items in Interface Builder by dragging the empty menu from the Cocoa-Menus pane in the Palette window (you can use tool tips to find this item)

`state`

Access: read/write
Class: *enumerated constant* from "Cell State Value" (page 181)
the state of the menu item

`sub menu`

Access: read/write
Class: *menu* (page 463)
the sub menu of the menu item (if any)

`tag`

Access: read/write
Class: *integer*
the tag of the menu item (an arbitrary item associated with the menu item); default is 0; you can set this property in the Info window in Interface Builder; you might use a tag, for example, as a way to identify a particular menu item

`title`

Access: read/write
Class: *Unicode text*
the title of the menu item; you can set this property in the Info window in Interface Builder

Commands supported by menu item objects

Your script can send the following commands to a menu item object:

`perform action` (page 326)

Events supported by menu item objects

A menu item object supports handlers that can respond to the following events:

Menu

choose menu item (page 470)
update menu item (page 470)

Nib

awake from nib (page 130)

Examples

The following script statements, taken from a script running in the Script Editor application, show how to access properties of the "New" menu item in the "File" menu of a simple AppleScript Studio application. You can use similar statements within an AppleScript Studio application script, though you won't need the `tell application` statement.

```
tell application "TestApp"
    set menuItem to second menu item of main menu
    title of menuItem -- result: "File"
    set item1 to first menu item of sub menu of menuItem
    title of item1 -- result: "New"
    key equivalent of item1 -- result: "n"
end tell
```

The Examples section for the `menu` (page 463) class shows how to access additional properties.

Version Notes

The `key equivalent modifier` property in this class is not supported, through AppleScript Studio version 1.2.

Events

Objects based on classes in the Menu suite support handlers for the following events (an event is an action, typically generated through interaction with an application's user interface, that causes a handler for the appropriate object to be executed). To determine which classes support which events, see the individual class descriptions.

`choose menu item`

Called when a menu item is chosen.

Syntax

`choose menu item` *reference* required

Parameters

reference
a reference to the menu item object whose `choose menu item` handler was called

Examples

When you connect a `choose menu item` handler to a `menu item` object in Interface Builder, AppleScript Studio supplies an empty handler template. This handler is where your application deals with the user's menu choice.

```
on choose menu item theObject
    (* Add script statements here to handle the chosen menu item. *)
end choose menu item
```

`update menu item`

Called periodically when the state of a menu item may need to be updated. The handler should return `true` to enable the menu item or `false` to disable it.

Syntax

`update menu item` *reference* required

Parameters

reference
a reference to the menu item object whose `update menu item` handler was called

Examples

When you connect an `update menu item` handler to a `menu item` object in Interface Builder, AppleScript Studio supplies an empty handler template. The `theObject` parameter refers to the chosen menu item and you can use the parameter to access properties or elements of the item. If, for example, your browser application has a menu item to display the modified date for a selected file, you might use an `update menu item` handler to disable the menu item when no file is selected.

```
on update menu item theObject
    (* if the menu item should be enabled… *)
    return true
    (* other statements *)
    (* if the menu item should be disabled *)
    return false
end update menu item
```

CHAPTER 10

Panel Suite

This chapter describes the terminology in AppleScript Studio's Panel suite. For other suites, see the following:

- "Application Suite" (page 42)
- "Container View Suite" (page 194)
- "Control View Suite" (page 244)
- "Data View Suite" (page 348)
- "Document Suite" (page 430)
- "Drag and Drop Suite" (page 448)
- "Menu Suite" (page 462)
- "Text View Suite" (page 516)

Panel Suite

The Panel suite defines classes and events for dealing with dialogs, alerts, and panels. Most classes in this suite inherit from `window` (page 86), either directly or through one of the panel classes. A panel is a special kind of `window` object that can optionally be displayed as a dialog or utility window. For related information on terms used in the panel suite, see "Panels Versus Dialogs and Windows" (page 37).

The classes and commands in the Panel suite are described in the following sections:

For enumerated constants, see "Enumerations" (page 177).

Classes

The Panel suite contains the following classes:

alert reply

Plural: alert replies
Inherits from: None.
Cocoa Class: ASKAlertReply

Reply record for the display alert command. An alert reply is similar in concept to the dialog alert class defined as part of AppleScript's Standard Additions (in the file StandardAdditions.osax in /System/Library/ScriptingAdditions).

See also display alert (page 500) and alert ended (page 510).

Properties of alert reply objects

An alert reply object has these properties. This class is not accessible in Interface Builder, so you can't set its properties there. You can only set them in a script.

button returned

Access: read only
Class: *Unicode text*
the button that was clicked to end the alert; for example, "Cancel" if the user clicked the Cancel button

Events supported by alert reply objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

For an example of an alert reply, see the Examples section for the `alert ended` (page 510) event handler. The Display Alert application distributed with AppleScript Studio also uses an alert reply.

color-panel

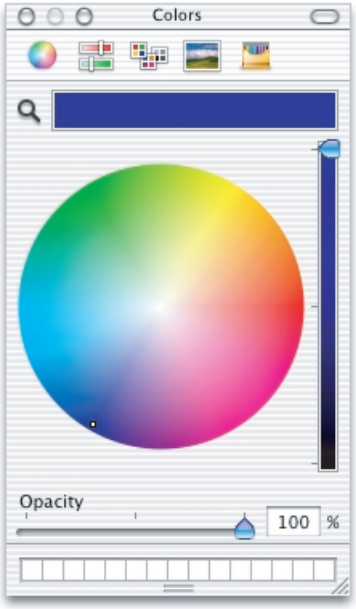
Plural: `color-panels`
Inherits from: `panel` (page 487)
Cocoa Class: NSColorPanel

Provides a standard user interface for selecting color in an application. A color panel uses a `color well` (page 263) to select an individual color. To use a color panel in your AppleScript Studio scripts, you can access the `color panel` property that is associated with every `application` (page 43) object. Note that `color-panel` is the class name, while `color panel` specifies an object of that class.

A color panel provides a number of standard color selection modes, which you can set using the constants in "Color Panel Mode" (page 182). However, when you get or set a color property of an AppleScript Studio object, the color is represented as an RGB value, accessible as a three-item list that contains the values for each component of the color. For example, green can be represented as {0,65535,0}.

Figure 10-1 shows a color panel. The Opacity slider is not shown unless you set the `shows alpha` property of the color panel to `true`. For more information on colors and color panels, see the Cocoa Programming Topic Using Color.

Figure 10-1 A color panel



Properties of color panel objects

In addition to the properties it inherits from the `panel` (page 487) class, a `color panel` object has these properties. This class is not accessible in Interface Builder, so you can't set any of the properties there—you can only set them in a script.

`alpha`

Access: read/write
Class: *real*
the alpha value for the color; ranges from 0.0 (transparent) to 1.0 (opaque); default is 1.0

`color`

Access: read/write
Class: *RGB color*
the color; returned as a three-item list of integers {red value, green value, blue value}; for example, {0, 0, 0} represents black, while {0, 65535, 0} represents green

`color mode`

Access: read/write
Class: *enumerated constant* from "Color Panel Mode" (page 182)
the color mode for the panel

`continuous`

Access: read/write
Class: *boolean*
Should the color panel send color changes continuously as the user manipulates the color in the panel? default is `true`

`shows alpha`

Access: read/write
Class: *boolean*
Should the panel show the alpha value? default is `false`; if you set this value to `true`, the panel displays the Opacity slider (visible in Figure 10-1)—otherwise it does not display the slider (and returns an alpha value of 1.0)

Elements of color panel objects

A `color-panel` object can contain only the elements it inherits from the `panel` (page 487) class.

Events supported by color panel objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

A script handler in an AppleScript Studio application can make the application's color panel visible (or hide it when it is visible) by setting its `visible` property. AppleScript Studio scripts implicitly target the application, so you don't need a `tell application "MyApplication"` statement to access an application property.

```
set visible of color panel to true
```

You can use the following statement to get the currently selected color from the `color panel`:

```
set myColor to color of color panel
```

```
    -- returns an RGB color value, as a three item list: {int, int, int}
```

The following `on launched` handler, connected to the `application` object through the File's Owner instance in the MainMenu.nib window in Interface Builder, sets the color of the application's color panel to red and makes the panel visible when the application is launched.

```
on launched theObject
    set color of color panel to {65535, 0, 0}
    set visible of color panel to true
end launched
```

Version Notes

Starting in AppleScript Studio version 1.1, the name of the `color panel` class was changed to `color-panel`. This is to make it clear which refers to the class (`color-panel`) and which to the property (`color panel`) of the `application` (page 43) object.

Prior to AppleScript Studio version 1.1, this class had limited functionality.

dialog reply

Plural: `dialog replies`
Inherits from: None.
Cocoa Class: ASKDialogReply

Reply record for the `display dialog` command. A dialog reply is similar in concept to the `dialog reply` class defined as part of AppleScript's Standard Additions (in the file `StandardAdditions.osax` in `/System/Library/ScriptingAdditions`).

See also `display dialog` (page 503) and `dialog ended` (page 511).

Properties of dialog reply objects

A dialog reply object has these properties. This class is not accessible in Interface Builder, so you can't set its properties there. You can only set them in a script.

`button returned`

Access: read only
Class: *Unicode text*

name of button chosen (empty if `giving up after` was supplied and dialog timed out)

`gave up`

Access: read only
Class: *boolean*
Did the dialog time out? (present only if `giving up after` parameter was used in the call to the `display dialog` command)

`text returned`

Access: read only
Class: *Unicode text*
text entered (present only if `default answer` was supplied)

Events supported by dialog reply objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

The following is the `dialog ended` (page 511) handler from the Display Dialog sample application distributed with AppleScript Studio. Display Dialog shows how to display a dialog and obtain information when it is dismissed. This handler gets called when the dialog is dismissed after it was called with the `attached to` optional parameter, so that it is shown as a sheet. The handler extracts information from the `dialog reply` object provided by `theReply` parameter and displays information in the application's window.

```
on dialog ended theObject with reply theReply
    -- Set the values returned in "theReply"
    set contents of text field "text returned" of window "main" to
        text returned of theReply
    set contents of text field "button returned" of window "main" to
        button returned of theReply
    set state of button "gave up" of window "main" to gave up of theReply
end dialog ended
```

font-panel

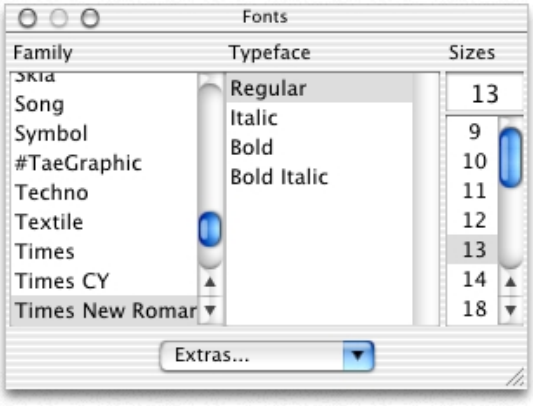
Plural: `font-panels`

Inherits from: `panel` (page 487)
Cocoa Class: NSFontPanel

Displays a list of available fonts, allowing for preview and selection of the text display font. There is one font panel per application, accessible through the `font panel` property of the `application` (page 43) object. Note that `font-panel` is the class name, while `font panel` specifies an object of that class.

Figure 10-2 shows a font panel. For AppleScript Studio limitations in font handling, see the `font` (page 67) class. For additional information on fonts, see the Cocoa topics Font Panels and Font Handling

Figure 10-2 A font panel



Properties of font panel objects

In addition to the properties it inherits from the `panel` (page 487) class, a `font panel` object has these properties. This class is not accessible in Interface Builder, so you can't set its properties there. You can only set them in a script.

`enabled`

Access: read/write
Class: *boolean*
Is the panel enabled?

font

Access: read/write
Class: *font* (page 67)
the current font

Elements of font panel objects

A font-panel object can contain only the elements it inherits from panel (page 487).

Events supported by font panel objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

A script handler in an AppleScript Studio application can refer to the font panel property of the application object without having to specify the application, as in the following statement, which shows the font panel:

```
set visible of font panel to true
```

Version Notes

Starting in AppleScript Studio version 1.1, the name of the font panel class was changed to font-panel. This is to make it clear which refers to the class (font-panel) and which to the property (font panel) of the application (page 43) object.

Prior to AppleScript Studio version 1.1, this class had limited functionality.

open-panel

Plural: open-panels
Inherits from: save-panel (page 490)
Cocoa Class: NSOpenPanel

Provides a standard dialog applications can use to query a user for the name of a file to open. An open panel can be run as application modal or document modal (as a sheet attached to a window). Note that open-panel is the class name, while open panel specifies an object of that class.

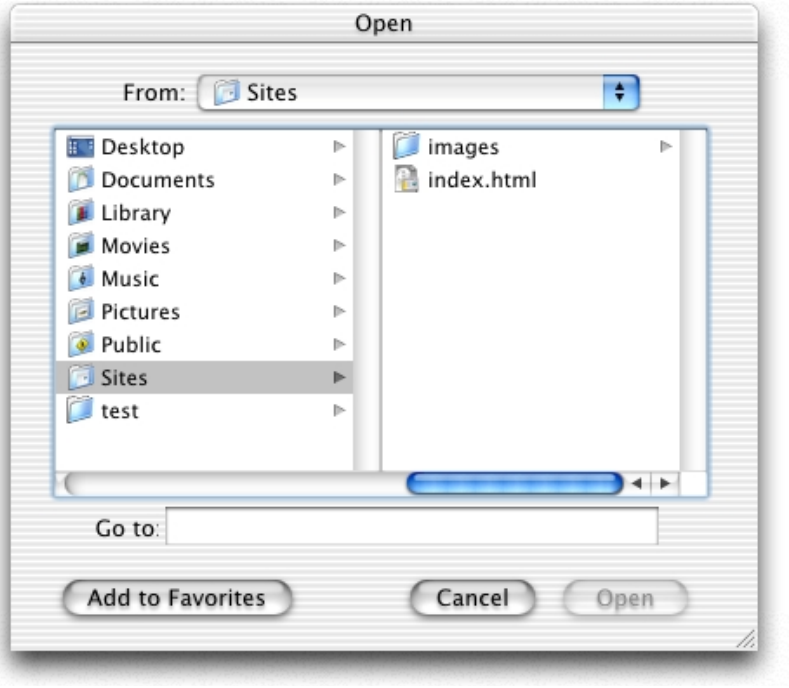
To use an open panel in an AppleScript Studio application, you use the `display` (page 496) command to display the `open panel` property that is associated with every `application` (page 43) object. If you display the panel as a sheet (attached to a window), you will also need to connect a `panel ended` (page 512) event handler. When the user closes the panel, you can get information, such as a list of paths to the files the user has selected, by accessing properties of the `open-panel` object.

An `open panel` object only understands and returns only POSIX style (slash-delimited) paths. It does not understand `file` or `alias` types. You can, however, use AppleScript's `POSIX file` and `POSIX path` scripting addition commands to convert between path types. These commands are defined as part of AppleScript's Standard Additions (in the file `StandardAdditions.osax` in `/System/Library/ScriptingAdditions`).

For more information, see `save-panel` (page 490), as well as the Cocoa topics File Management and Windows and Panels.

Figure 10-3 shows an open panel.

Figure 10-3 An open panel



Properties of open panel objects

In addition to the properties it inherits from the `save-panel` (page 490) class, an `open panel` object has these properties (see the Version Notes section for this class for the AppleScript Studio version in which a particular property was added). This class is not accessible in Interface Builder, so you can't set its properties there. The Open Panel sample application, distributed with AppleScript Studio starting in version 1.1, demonstrates how to set many open panel properties in a script.

`allows multiple selection`

Access: read/write
Class: *boolean*
Can multiple items be selected?

`can choose directories`

Access: read/write
Class: *boolean*
Can directories be selected?

`can choose files`

Access: read/write
Class: *boolean*
Can files be selected?

`path names`

Access: read only
Class: *list*
the list of files selected to be opened; each name is a POSIX (slash-delimited) path

Elements of open panel objects

An `open-panel` object can contain only the elements it inherits from `panel` (page 487).

Commands supported by open panel objects

Your script can send the following commands to an `open panel` object:

`display` (page 496)
`display panel` (page 507)

Events supported by open panel objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

A script handler in an AppleScript Studio application can refer to the `open panel` property of the `application` object without having to specify the application, as in the following statement, which obtains the list of full POSIX style (slash-delimited) paths for the files chosen in the open panel:

```
set fileList to path names of open panel
```

An application could use the following clicked handler to prompt a user to choose a folder. This handler first sets properties of the `open panel` property of the application object to specify a folder (or directory) to search. Note that you don't need to implicitly specify the open panel as belonging to the application object.

```
on clicked theObject
    set can choose directories of open panel to true
    set can choose files of open panel to false
    display open panel attached to window "main"
end clicked
```

Because the handler displays the panel as a sheet, the panel is document modal, which means that application execution continues after the panel is displayed. So to get the user's choice from the panel, the application needs to use Interface Builder to connect a `panel ended` handler to the `window` (page 86) to which the panel is attached. The `panel ended` handler is called when the panel is dismissed. In the example shown here, the handler checks the result and if it is 1 (a folder was chosen; 0 indicates the panel was cancelled), extracts the folder from the returned list. As mentioned above, the folder path is a POSIX style path.

```
on panel ended theObject with result withResult
    if withResult is 1 then
        set theFolder to item 1 of (path names of open panel as list)
        -- do something with the supplied folder path
    end if
end panel ended
```

If a script does not attach the panel to a window, the panel is displayed as application modal, and execution halts until the panel is dismissed. In that case, a `panel ended` event handler is not needed.

The Open Panel sample application, distributed with AppleScript Studio starting in version 1.1), demonstrates how to use an open panel, both as a separate panel and as a sheet. See the "Document Suite" (page 430) for information on how to read and write files.

Version Notes

The `path names` property was added in AppleScript Studio version 1.1.

Starting in AppleScript Studio version 1.1, the name of the `open panel` class was changed to `open-panel`. This is to make it clear which refers to the class (`open-panel`) and which to the property (`open panel`) of the `application` (page 43) object.

Prior to AppleScript Studio version 1.1, this class had limited functionality.

The Open Panel sample application was first distributed with AppleScript Studio version 1.1.

panel

Plural: `panels`
Inherits from: `window` (page 86)
Cocoa Class: NSPanel

A type of window that typically serves an auxiliary function in an application. For example, a panel can optionally be displayed as a utility `window` (page 86), which can float above other windows.

You create a panel in Interface Builder by dragging a panel object from the Cocoa Windows pane of the Palette window.

For more information, see the `display` (page 496) command, as well as the Cocoa Programming Topic Windows and Panels.

Properties of panel objects

In addition to the properties it inherits from the `window` (page 86) class, a panel object has these properties:

`floating`

Access: read/write
Class: *boolean*
Is the panel a floating panel?

Elements of panel objects

An `panel` object can contain only the elements it inherits from `window` (page 86).

Commands supported by panel objects

Your script can send the following commands to a panel object:

close panel (page 495)
display (page 496)
display panel (page 507)

Events supported by panel objects

A panel object supports handlers that can respond to the following events:

Key

keyboard down (page 141)
keyboard up (page 141)

Mouse

mouse down (page 144)
mouse dragged (page 145)
mouse entered (page 146)
mouse exited (page 146)
mouse up (page 148)
right mouse down (page 154)
right mouse dragged (page 155)
right mouse up (page 156)
scroll wheel (page 156)

Nib

awake from nib (page 130)

Panel

alert ended (page 510)
dialog ended (page 511)
panel ended (page 512)

Window

became key (page 135)
became main (page 136)
exposed (page 139)
miniaturized (page 143)
moved (page 149)

opened (page 150)
resigned key (page 152)
resigned main (page 153)
resized (page 153)
should close (page 157)
should zoom (page 162)
was miniaturized (page 165)
will close (page 166)
will miniaturize (page 169)
will move (page 170)
will open (page 170)
will resize (page 172)
will zoom (page 174)

Examples

The following is a partial listing of the `clicked` handler for the main window in the Display Panel sample application distributed with AppleScript Studio. This handler is in the file `Window.applescript`. The Display Panel application shows how to display a panel and obtain information when it is dismissed. Because this handler uses the `attached to` parameter to specify that the panel should be displayed attached to the window "main", the application supplies a `panel ended` (page 512) handler that is called when the panel is dismissed. No `panel ended` handler is needed if the panel is not attached, because then it is displayed in document-modal fashion and control continues in the statement after the panel is displayed.

```
on clicked theObject
    -- Some statements not shown
    -- Make sure panel has been loaded from nib into global property
    if not (exists panelWIndow) then
        load nib "SettingsPanel"
        set panelWIndow to window "settings"
    end if

    -- Statements for setting state of panel items not shown
    -- Now display the panel
    display panelWIndow attached to window "main"
    -- Other statements not shown
end clicked
```

The Display Panel sample application uses the following statement to display an application-modal panel. The call to `display` returns the number of the button used to dismiss the panel, and the result is used here in an `if` statement test. A return value of 0 indicates the panel was cancelled; 1 indicates it was accepted.

```
if (display panelWIndow) is 1 then
```

Note: The Display Panel application uses the `display` (page 496) command, which is the preferred way to display a panel, rather than the `display panel` (page 507) command.

save-panel

Plural: `save-panels`

Inherits from: `panel` (page 487)

Cocoa Class: NSSavePanel

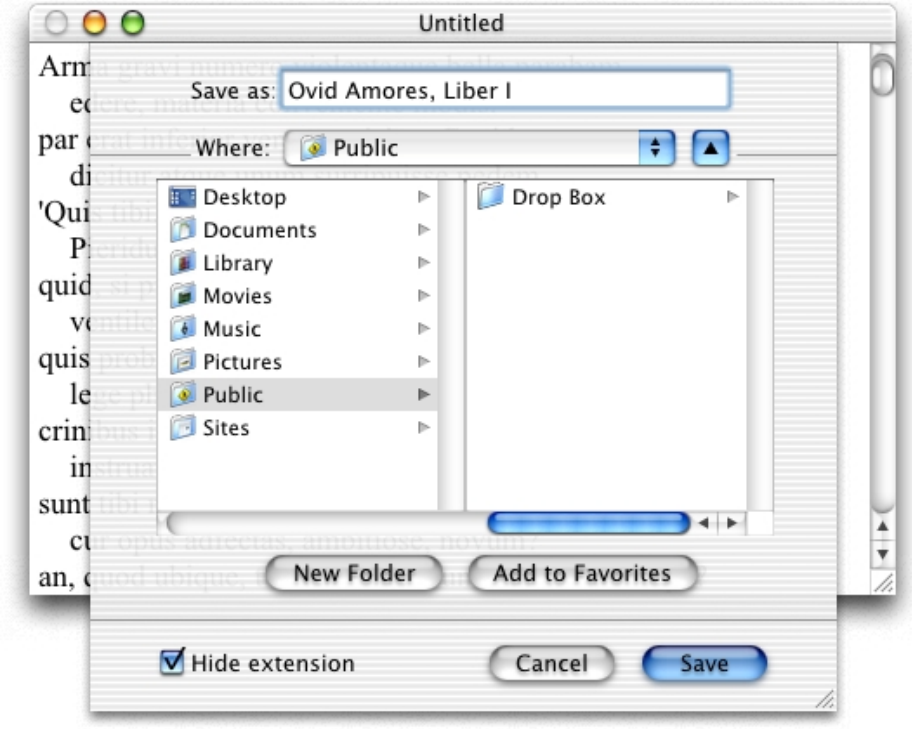
Allows users to specify the directory and name under which a file is saved. The `save-panel` class supports browsing of the file system and it accommodates custom accessory views. To use a save panel in your AppleScript Studio scripts, you can access the `save panel` property that is associated with every `application` (page 43) object. Note that `save-panel` is the class name, while `save panel` specifies an object of that class.

To use a save panel in an AppleScript Studio application, you use the `display` (page 496) command to display the `save panel` property that is associated with every `application` object. If you display the panel as a sheet (attached to a window), you will also need to connect a `panel ended` (page 512) event handler. When the user closes the panel, you can get information, such as path name the user has chosen for saving a file, by accessing properties of the `save-panel` object.

A `save panel` object only understands and works with POSIX style (slash-delimited) paths. It does not understand `file` or `alias` types. You can however use AppleScript's `POSIX file` and `POSIX path`, defined as part of AppleScript's Standard Additions (in the file `StandardAdditions.osax` in `/System/Library/ScriptingAdditions`), to convert between path types.

Figure 10-4 shows a save panel. For more information on panels, see `open-panel` (page 482), as well as the Cocoa Programming Topic Windows and Panels.

Figure 10-4 A save panel



Properties of save panel objects

In addition to the properties it inherits from the `panel` (page 487) class, a `save panel` object has these properties. This class is not accessible in Interface Builder, so you can't set its properties there. The Save Panel sample application, distributed with AppleScript Studio starting in version 1.1, demonstrates how to set many save panel properties in a script.

`directory`

Access: read/write
Class: *Unicode text*
the directory to use in the panel; must be a POSIX (slash-delimited) path

`expanded`

Access: read only
Class: *boolean*
Is the panel expanded? if so, the column browser is visible, as in Figure 10-4

`path name`

Access: read only
Class: *Unicode text*
the path name returned from the panel; a POSIX path; this property was called `file name` prior to AppleScript Studio version 1.2

`prompt`

Access: read/write
Class: *Unicode text*
the prompt to display for the default button (by default "Save", as in Figure 10-4)

`required file type`

Access: read/write
Class: *Unicode text*
specifies an extension to be appended to any selected files that don't already have that extension; an extension, such as "rtf" or "txt"; do not include the period that begins the extension

`title`

Access: read/write
Class: *Unicode text*
the title of the panel ("Save as" in Figure 10-4)

`treat packages as directories`

Access: read/write
Class: *boolean*
Should the panel treat packages as directories? in Mac OS X, an applications is packaged as a bundle, or directory in the file system that stores executable code and the software resources related to that code; "package" is sometimes used as a synonym for "bundle"; if this property is set to false, the save panel will show the contents of a bundle, rather than display it as if it were one file

Elements of save panel objects

A `save-panel` object can contain only the elements it inherits from `panel` (page 487).

Commands supported by save panel objects

Your script can send the following commands to a `save panel` object:

`display` (page 496)
`display panel` (page 507)

Events supported by save panel objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

A script handler in an AppleScript Studio application can refer to the `save panel` property of the `application` object without having to specify the application, as in the following statement, which displays the save panel (attached to the window with AppleScript name "main"):

```
display save panel attached to window "main"
```

Because the statement displays the panel as a sheet, the panel will be document modal, which means that application execution continues after the panel is displayed. So to get the user's choice from the panel, the application needs to use Interface Builder to connect a `panel ended` handler to the `window` (page 86) to which the panel is attached. The `panel ended` handler is called when the panel is dismissed.

If a script does not attach the panel to a window, the panel is displayed as application modal, and execution halts until the panel is dismissed. In that case, a `panel ended` event handler is not needed.

For a summary of how to use the save panel, see the Examples section of the `display` (page 496) command and the Save Panel application distributed with AppleScript Studio. For an example that shows how to use an open panel, see the Examples section of the `open-panel` (page 482) class. See the "Document Suite" (page 430) for information on how to read and write files.

Version Notes

Starting in AppleScript Studio version 1.1, the name of the `save panel` class was changed to `save-panel`. This is to make it clear which refers to the class (`save-panel`) and which to the property (`save panel`) of the `application` (page 43) object.

Prior to AppleScript Studio version 1.1, this class had limited functionality.

Prior to AppleScript Studio version 1.1, the name of the `path name` property was `file name`.

The Save Panel sample application was first distributed with AppleScript Studio version 1.1.

Commands

Objects based on classes in the Panel suite support the following commands. (A command is a word or phrase a script can use to request an action.) To determine which classes support which commands, see the individual class descriptions.

`close panel`

Closes the specified panel, optionally supplying a return value. A panel is a special kind of `window` (page 86) object that can optionally be displayed as a utility window. If the panel

Syntax

`close panel`	*reference*	required
`with result`	*anything*	optional

Parameters

reference
a reference to the panel to close

`with result` *anything*
the result of the panel; for example, an integer indicating the button pressed

Examples

The following is the `clicked` handler for the buttons in the panel window in the Display Panel sample application distributed with AppleScript Studio. This handler is in the file `Settings.applescript`. The Display Panel application shows how to display a panel and obtain information when it is dismissed.

If the panel is displayed as a sheet (document modal), application execution continues after the panel is displayed. To get the user's choice from the panel, the application connects a `panel ended` (page 512) handler to the `window` (page 86) to which the panel is attached (as described in the Examples section for the `panel` (page 487) class). The `panel ended` handler is called when the panel is dismissed. The value returned by the `close panel` command is passed to `panel ended` handler, which does nothing unless the value is 1, indicating the user clicked the Change button (not the Cancel button).

If the panel was displayed as a separate window (application modal), execution halts until the panel is dismissed. In that case, a `panel ended` event handler is not needed. The value returned by the `close panel` command is returned directly to the calling script, which again does nothing unless the value is 1, indicating the user clicked the Change button (not the Cancel button).

```
on clicked theObject
    if name of theObject is "cancel" then
        close panel (window of theObject)
    else if name of theObject is "change" then
        close panel (window of theObject) with result 1
    end if
end clicked
```

display

Presents a panel in an application-modal fashion, or in document-modal fashion (as a sheet) when the optional `attached to` parameter is used.

You should use display rather than `display panel` (page 507) to display `open-panel` (page 482), `save-panel` (page 490), or any other panels. For examples that show the correct terminology, see the Examples section below.

The `display` command allows your application to display the open and save panels that have been added as properties of the `application` (page 43) object in AppleScript Studio version 1.1.

Syntax

`display`	*reference*	required
`afterwards calling`	*anything*	optional
`attached to`	*window*	optional
`for file types`	*list*	optional
`in directory`	*Unicode text*	optional
`with file name`	*Unicode text*	optional

Parameters

reference
a reference to the window to display (in which case, you use a statement like `display panelWindow`; for a save or open panel, use `display save panel` or `display open panel`)

`afterwards calling` *anything*
not supported (through AppleScript Studio version 1.2); a reference to a script to run when the displayed window is dismissed

`attached to` `window` (page 86)
the window to attach the displayed window to; attaching the window makes it document modal (attached as a sheet to the specified window)

`for file types` *list*
a list of file extension that are allowed (for open/save panels), such as “rtf” or “txt”; do not include the period that begins the extension

`in directory` *Unicode text*
the starting directory (for open/save panels)

`with file name` *Unicode text*
the default file name (for open/save panels)

Result

integer
A value indicating which button was used to dismiss the panel. For AppleScript Studio’s Save and Open panels, 0 represents the Cancel button and 1 represents the Save or Open button. If you use the display command with your own panel, you call the `close panel` (page 495) command, passing an integer value that is in

turn passed on to your `panel ended` handler, and represents how your panel was dismissed. In that case, return values are arbitrary, and it's up to your application to assign them and interpret them.

Examples

You typically use steps like the following, based on the Save Panel sample application distributed with AppleScript Studio, to display and respond to a save or open panel (in this case, a save panel):

- Set up your application in Interface Builder.
  - You may want to display the panel when a user clicks a button in a window. To do so, use a `clicked` (page 337) handler, as in Save Panel.
  - Or you may want to display the panel when a user chooses the Save item in the File menu. In that case, use a `choose menu item` (page 470) handler.
- If you will show the panel as a sheet, connect a `panel ended` (page 512) handler to the window to which you will attach the panel.
- To display the panel as a sheet (document modal), use the `attached to` parameter and don't expect to get an immediate response. The `panel ended` handler will be called when the user dismisses the panel, and you can get the information you need about the panel there, as described in the Examples section for `panel ended` (page 512).

  Here's how the Save Panel application displays a panel as a sheet:

  ```
  display save panel in directory theDirectory with file name theFileName
  attached to window of theObject
  ```

  Note that when you display AppleScript Studio's built-in save panel, shown in Figure 10-4 (page 491), you're actually using the `save panel` property of the `application` (page 43) object.
- If you display the panel in the standard (non-sheet) way, the result is application modal (nothing else can happen until the user responds). Control returns to the script statement following the `display save panel` statement. Here is how the Save Panel application does it:

  ```
  set theResult to display save panel in directory theDirectory with file
  name theFileName
  ```

  The result is an integer, where 0 represents the Cancel button and 1 represents the Save button.

You can also use display to display a panel you have constructed in Interface builder (instead of the standard Save panel provided by AppleScript Studio). In that case, you will need to call the `close panel` (page 495) command when your panel is dismissed. For details on how to do so, see the Display Panel application distributed with AppleScript Studio.

Given a window "main" and a panel window "preferences" in a nib named "preferences.nib", the following script displays the panel window attached to the window:

```
load nib "preferences"
set preferencesPanel to window "preferences"
display preferencesPanel attached to window "main"
```

Discussion

When you display a window in application-modal fashion, your script waits for the user to dismiss the window. If you display the window in document-modal fashion by specifying the optional `attached to` parameter, script execution continues. To perform any actions when the user dismisses the window, you have to supply a `panel ended` (page 512) handler. You can see an example of this handler in the Display Panel sample application distributed with AppleScript Studio.

Version Notes

The `display` command was added in AppleScript Studio version 1.1. Prior to version 1.1, you can use the `display panel` (page 507) command. (A different `display` command was available in the Application suite in AppleScript Studio version 1.0.)

Starting in AppleScript Studio version 1.2, the `display` command is recommended as a replacement for the `display panel` command.

The `afterwards calling` parameter to this command is not supported, through AppleScript Studio version 1.2.

The Save Panel application was first distributed with AppleScript Studio version 1.1.

display alert

Displays the specified alert. The many optional parameters allow you to control how the alert is displayed. You typically display an alert to provide information that the user should reply to immediately. However, you can display an alert as a sheet (attached to a window), allowing a user to continue to work with other windows before responding to the alert.

Figure 10-5 shows an alert displayed as a sheet by the Display Alert sample application distributed with AppleScript studio.

Figure 10-5 An alert displayed as a sheet by the display alert command

Syntax

`display alert`	*Unicode text*	required
`afterwards calling`	*anything*	optional
`alternate button`	*Unicode text*	optional
`as`	*enumerated constant*	optional
`attached to`	*window*	optional
`default button`	*Unicode text*	optional
`message`	*Unicode text*	optional
`other button`	*Unicode text*	optional

Parameters

Unicode text
text to display in the alert; can be an empty string

`afterwards calling` *anything*
not supported (through AppleScript Studio version 1.2); a reference to a script to run when the alert is finished

`alternate button` *Unicode text*
the title of the alternate button

`as` *enumerated constant* from "Alert Type" (page 178)
the type of alert

`attached to window` (page 86)
the window to attach the alert to

`default button` *Unicode text*
the title of the default button

`message` *Unicode text*
the message for the alert

`other button` *Unicode text*
the title of the other button

Result

alert reply
An `alert reply` (page 475) that contains information on the dismissed alert. When the alert is displayed as a sheet (attached to a window), there is no immediate result, and you must install an `alert ended` (page 510) handler to respond to the alert being dismissed.

Examples

The following statement shows the syntax for displaying an alert as a sheet (which results in a document-modal alert). It is taken from the `clicked` handler of the Display Alert application distributed with AppleScript Studio. Most of the parameters are variables that are set from text fields in the application's main window (not shown).

```
display alert dialogText as dialogType message dialogMessage
    default button defaultButtonTitle alternate button alternateButtonTitle
    other button otherButtonTitle attached to window "main"
```

When you display an alert attached to a window, you must install an `alert ended` handler for the window. The handler is called when the alert is dismissed. For an example of an `alert ended` handler, see the Examples section of the `alert ended` (page 510) handler.

The following statement shows the syntax for displaying an alert that not attached to a window (and is therefore application modal). Again, most of the parameters are variables that are set from text fields in the application's main window (not shown). In this case, though, execution stops until the alert is dismissed, and `display alert` returns an `alert reply` (page 475), from which you can get information about how the alert was dismissed (as shown the Examples section of the `alert ended` (page 510) handler).

```
set theReply to display alert dialogText as dialogType
    message dialogMessage default button defaultButtonTitle
    alternate button alternateButtonTitle other button otherButtonTitle
```

Version Notes

The `afterwards calling` parameter to this command is not supported, through AppleScript Studio version 1.2.

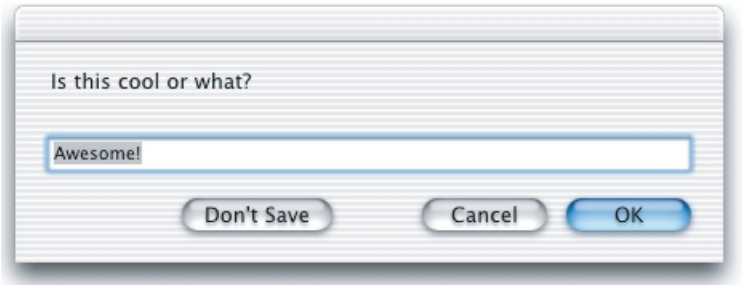
display dialog

Displays the specified dialog. AppleScript Studio overrides the `display dialog` scripting addition command (provided in the file `/System/Library/ScriptingAdditions/StandardAdditions.osax`) to implement its own version.

This command has many optional parameters to allow you to control how the dialog is displayed. For example, you can present the window in an application-modal fashion, or in document-modal fashion (as a sheet). Displaying a dialog as a sheet allows a user to continue to work with other windows before responding.

Figure 10-5 shows a dialog displayed by the Display Dialog sample application distributed with AppleScript studio. Display Dialog shows how to display a dialog and obtain information when it is dismissed.

Figure 10-6 A dialog displayed by the display dialog command



Syntax

`display dialog`	*reference*	required
`attached to`	*window*	optional
`buttons`	*list*	optional
`default answer`	*Unicode text*	optional
`default button`	*anything*	optional
`giving up after`	*integer*	optional
`with icon`	*anything*	optional

Parameters

reference
a reference to the dialog window to display

`attached to window` (page 86)
the window to attach the dialog to; supplying this parameter causes the dialog to be displayed in document-modal fashion, attached to the specified window

`buttons` *list*
a list of up to three button names

`default answer` *Unicode text*
the default editable text

`default button` *anything*
the name or number of the default button

`giving up after` *integer*
number of seconds to wait before automatically dismissing the dialog

`with icon` *anything*
a string or integer that specifies the icon to display; for a string, the string must specify a TIFF file (without the `.tiff` extension) in the project that contains an icon image; for an integer, you can use the values 0, 1, or 2; see the Examples section below for more information.

Result

dialog reply
A `dialog reply` (page 479) that contains information on the dismissed dialog. When the dialog is displayed as a sheet (attached to a window), there is no immediate result, and you must install a `dialog ended` (page 511) handler to respond to the dialog being dismissed.

Examples

The following example displays a dialog with text and three buttons. It specifies a timeout value (`giving up after 10`), a default button (`default button "Goodbye"`), and a type of warning icon (`with icon 0`). The example then displays a second dialog to show which button was clicked:

```
on clicked theObject
    set theReply to display dialog "Please click a button."
        buttons {"Hello", "Goodbye", "OK"} default button "Goodbye"
        giving up after 10 with icon 0
    display dialog button returned of theReply
end clicked
```

In this example, control does not return to the second `display dialog` statement until either the user clicks a button (or presses the Return key for the default), or waits more than 10 seconds, causing a timeout.

You can display an application-defined icon by specifying the TIFF file for the icon. For example, if the file is named `myIcon.tiff` and you've dragged it into the Project Builder project for the application, you could replace `with icon 0` with `with icon myIcon` in the `clicked` handler above.

The AppleScript constants `stop`, `note`, and `caution` will not work to specify an icon in a `display dialog` command for your application. Use the values 0, 1 and 2, instead. AppleScript Studio relies on the icons supplied by Cocoa for those values. Through Mac OS X version 10.2, passing 0 results in the display of an icon with an exclamation mark, while passing 1 or 2 results in the generic application icon.

You can, however, use the constants `stop`, `note`, and `caution` to display a dialog for another, non-AppleScript Studio application, inside a `tell` block, as in the following example.

```
on clicked theObject
    tell application "Finder"
        set theReply to display dialog "Please click a button."
            buttons {"Hello", "Goodbye", "OK"} default button "Goodbye"
            giving up after 10 with icon stop
        display dialog button returned of theReply
    end tell
end clicked
```

See the Discussion section for information about displaying a dialog as a sheet (attached to a window). For a more complete example, see the Display Dialog sample application distributed with AppleScript Studio, as well as the Examples section of the `dialog reply` (page 479) class.

Discussion

When you display a dialog independently, script execution stops until the dialog is dismissed, at which point it continues with the statement following the `display dialog` statement.

When you display a dialog as a sheet (attached to a window), script execution continues, and the following statement is executed immediately. This can lead to some confusion. To get control when the user dismisses the dialog, you install a `dialog ended` handler, as demonstrated in the Display Dialog sample application distributed with AppleScript Studio.

When you display a dialog independently, the `display dialog` command generates a “user canceled” error when the cancel button is pressed. Your script should use a `try on error` block (also demonstrated in the Display Dialog application) to deal with the error.

When you display a dialog attached to a window, your `dialog ended` handler can treat cancel like any other button.

Version Notes

Starting with AppleScript Studio version 1.1, you can pass a number to the `display dialog` command without having to coerce it to a string, as shown in the following statement (where `amount` and `rate` are previously defined numeric variables):

```
display dialog amount * rate
```

For another simple example, the following statement displays a minimal dialog (with Cancel and OK buttons) that contains displays the string "10" and dismisses itself after 3 seconds if the user doesn't respond:

```
display dialog 10 giving up after 3
```

display panel

Displays the specified panel window. Not recommended, starting in AppleScript Studio version 1.2. Use `display` (page 496) instead. The optional parameters allow you to control how the panel window is displayed.

Syntax

`display panel`	*reference*	required
`afterwards calling`	*anything*	optional
`attached to`	*window*	optional
`for file types`	*list*	optional
`in directory`	*Unicode text*	optional
`with file name`	*Unicode text*	optional

Parameters

reference
a reference to the window to display

`afterwards calling` *anything*
the script to run when the display is finished

`attached to` `window` (page 86)
the window to attach the panel window to

`for file types` *list*
a list of file extension that are allowed (for open/save panels)

`in directory` *Unicode text*
the starting directory (for open/save panels)

`with file name` *Unicode text*
the default file name (for open/save panels)

Result

integer
an integer value representing the button that dismissed the panel; a return value of 0 indicates the panel was cancelled; 1 indicates it was accepted

Examples

For examples that show how to load and display a panel, see the Examples sections of the `panel ended` (page 512) and `load nib` (page 113) event handlers.

Version Notes

Starting in AppleScript Studio version 1.2, this command is not recommended—use `display` (page 496) instead.

load panel

Not recommended, starting with AppleScript Studio version 1.1. Loads the specified panel from the specified nib file.

Syntax

`load panel`	*reference*	required
`from nib`	*Unicode text*	optional

Parameters

reference
a reference to the panel window to load

`from nib` *Unicode text*
the name of the nib to get the panel window from

Examples

To use this command, you insert a panel window in your nib file and supply a window title on the Attributes pane of the Info window (not an AppleScript name, which you specify on the AppleScript pane). You then load the panel in a script by specifying the window title. For example, given a window with title “myWindow” in a nib file “myNib”, you could load an instance of the window for use as a panel with the following statement:

```
load panel "myWindow" from nib "myNib"
```

Version Notes

Starting in AppleScript Studio version 1.2, this command is not recommended—use `load nib` (page 113) instead.

Events

Objects based on classes in the Panel suite support handlers for the following events (an event is an action, typically generated through interaction with an application's user interface, that causes a handler for the appropriate object to be executed). To determine which classes support which events, see the individual class descriptions.

`alert ended`

Called after an alert ends if the alert panel was displayed attached to a window.

When you display an alert in application-modal fashion, your script waits for the user to dismiss the alert. If you display the alert in document-modal fashion by specifying the optional `attached to` parameter, script execution continues. To perform any actions when the user dismisses the alert, you have to supply an `alert ended` handler.

Syntax

`alert ended`	*reference*	required
`with reply`	*alert reply*	optional

Parameters

reference
a reference to the `window` (page 86) to which the panel was attached

`with reply` *alert reply*
the reply returned from the alert

Examples

When you connect an `alert ended` handler to a `window` object in Interface Builder, AppleScript Studio supplies an empty handler template. The following handler is from the Display Alert sample application distributed with AppleScript studio. The handler obtains the name of the button returned from the `with reply` parameter and uses it to set the contents of a text field (also named "button returned").

```
on alert ended theObject with reply theReply
    set contents of text field "button returned" of window "main"
        to button returned of the reply
end alert ended
```

dialog ended

Called after a dialog ends if the dialog was displayed attached to a window.

When you display a dialog in application-modal fashion, your script waits for the user to dismiss the dialog. If you display the dialog in document-modal fashion by specifying the optional `attached to` parameter, script execution continues. To perform any actions when the user dismisses the alert, you have to supply a `dialog ended` handler.

Syntax

`dialog ended`	*reference*	required
`with reply`	*dialog reply*	optional

Parameters

reference
a reference to the panel for which the handler is called

`with reply` *dialog reply*
the reply returned from the dialog

Examples

For an example of a dialog ended handler, see the Examples section of the `dialog reply` (page 479) command.

panel ended

Called after a panel ends if the panel was displayed attached to a window.

When you display a panel in application-modal fashion, your script waits for the user to dismiss the panel. If you display the panel in document-modal fashion by specifying the optional `attached to` parameter, script execution continues. To perform any actions when the user dismisses the panel, you have to supply a `panel ended` handler.

The `panel ended` event handler is called in the script that is connected to the window that the `open-panel` (page 482) or `save-panel` (page 490) is connected to. For example, if your window event handlers are connected to a script named `Window.applescript` and your script displays a panel with a statement like `display open panel attached to window "main"` , then the `panel ended` event handler should also be connected to `Window.applescript`. The handler is called when the panel is dismissed.

Syntax

`panel ended`	*reference*	required
`with result`	*anything*	optional

Parameters

reference
a reference to the panel for which the handler is called

`with result` *anything*
the result returned from the panel

Examples

The following `panel ended` handler is from the Save Panel sample application, distributed with AppleScript Studio (available starting in version 1.1). You connect the handler to a window in Interface Builder. Then your application calls the `display panel` (page 507) command, using the optional `attached to window` parameter to attach the panel to the window with the `panel ended` handler. The handler is called when the panel is dismissed.

```
on panel ended theObject with result withResult
    if withResult is 1 then
```

```
        set contents of text field "path name" of window "main"
            to path name of save panel
    else
        set contents of text field "path name" of window "main" to ""
    end if
end panel ended
```

The Save Panel application uses the following statement to display a panel attached to a window that uses the `panel ended` handler shown above. The `in directory` and `with file name` parameters are optional.

```
display save panel in directory theDirectory
    with file name theFileName attached to window of theObject
```

For more information, see the Examples section of the `display` (page 496) command.

Version Notes

The Save Panel application was first distributed with AppleScript Studio version 1.1.

CHAPTER 10

Panel Suite

Text View Suite

This chapter describes the terminology in AppleScript Studio's Text View suite. For other suites, see the following:

- "Application Suite" (page 42)
- "Container View Suite" (page 194)
- "Control View Suite" (page 244)
- "Data View Suite" (page 348)
- "Document Suite" (page 430)
- "Drag and Drop Suite" (page 448)
- "Menu Suite" (page 462)
- "Panel Suite" (page 474)

Text View Suite

The Text View suite defines two classes for displaying and manipulating text. The `text` (page 517) class inherits from the `view` (page 226) class and `text view` (page 520) inherits from `text`. The classes in the Text View suite are described in the following section:

"Classes" (page 517)

For enumerated constants, see "Enumerations" (page 177).

Classes

The Text View suite contains the following classes:

text

Plural: text
Inherits from: view (page 226)
Cocoa Class: NSText

Provides high-level text management. Your application typically works with the text view (page 520) subclass, rather than with objects based on the text class itself.

For an overview of Cocoa's text-handling system, see the Cocoa Programming Topic Text System Architecture.

Properties of text objects

In addition to the properties it inherits from the view (page 226) class, a text object has these properties:

alignment

Access: read/write
Class: *enumerated constant* from "Text Alignment" (page 191)
the alignment of the text

background color

Access: read/write
Class: *RGB color*
the background color of the view; a three-item integer list that contains the values for each component of the color; for example, green can be represented as {0,65535,0}

content

Access: read/write
Class: *Unicode text*
the contents of the view

`contents`

Access: read/write
Class: *Unicode text*
the contents of the view

`draws background`

Access: read/write
Class: *boolean*
Should the view draw its background?

`editable`

Access: read/write
Class: *boolean*
Is the view editable?

`field editor`

Access: read/write
Class: *boolean*
Is this a field editor? a field editor is used by simple text-bearing objects; for example, a `text field` (page 314) object uses its window's field editor to display and manipulate text; the field editor can be shared by any number of objects and so its state may be constantly changing; for additional information, see the description for the `fieldEditor:forObject:` method in the NSWindow class

`font`

Access: read/write
Class: *font* (page 67)
the font for the view

`horizontally resizable`

Access: read/write
Class: *boolean*
Is the view horizontally resizable

`imports graphics`

Access: read/write
Class: *boolean*
Should the view import graphics?

`maximum size`

Access: read/write
Class: *point*
the maximum size of the view; the size is returned as a two-item list of numbers {horizontal, vertical}; see the `bounds` property of the `window` (page 86) class for information on the coordinate system

`minimum size`

Access: read/write
Class: *point*
the minimum size of the view; the size is returned as a two-item list of numbers {horizontal, vertical}; see the `bounds` property of the `window` (page 86) class for information on the coordinate system

`rich text`

Access: read/write
Class: *boolean*
Does the text support Rich Text Format? default is `true`

`selectable`

Access: read/write
Class: *boolean*
Is the view selectable?

`text color`

Access: read/write
Class: *RGB color*
the color of the text; a three-item integer list that contains the values for each component of the color; for example, green can be represented as {0,65535,0}

`uses font panel`

Access: read/write
Class: *boolean*
Can the view use the font panel?

`vertically resizable`

Access: read/write
Class: *boolean*
Is the view vertically resizable?

Elements of text objects

A `text` object can contain only the elements it inherits from `view` (page 226).

Events supported by text objects

This class is not accessible in Interface Builder, and therefore you cannot connect any event handlers to it.

Examples

See the `text view` (page 520) class for scripting samples that work with text.

Version Notes

The `content` property was added in AppleScript Studio version 1.2. You can use `content` and `contents` interchangeably, with one exception. Within an event handler, `contents of theObject` returns a reference to an object, rather than the actual contents. To get the actual contents of an object (such as the text contents from a `text field` (page 314)) within an event handler, you can either use `contents of contents of theObject` or `content of theObject`.

For a sample script that shows the difference between `content` and `contents`, see the Version Notes section for the `control` (page 270) class.

See the Examples section for the `scroll` (page 328) command for information about how to scroll text in a text view.

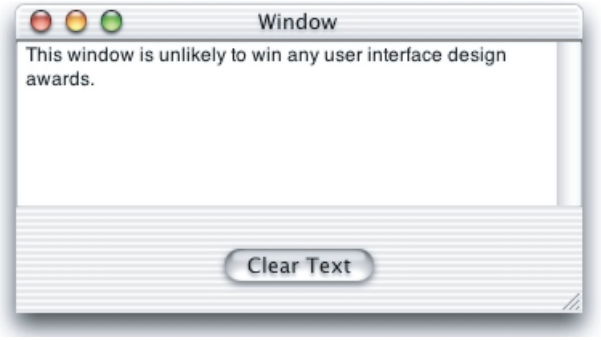
`text view`

Plural: `text views`
Inherits from: `text` (page 517)
Cocoa Class: NSTextView

Displays and manipulates text laid out in an area defined by a text object, and adds many features to those defined by its superclass. Many of the properties that you'll use most frequently are declared by the superclass, `text` (page 517).

Figure 11-1 shows a window that contains a text view.

Figure 11-1 A text view that contains some text



You will find the `text view` object on the Cocoa-Data pane of Interface Builder's Palette window. You can set many attributes for text views in Interface Builder's Info window.

For an overview of Cocoa's text-handling system, see the Cocoa Programming Topic Text System Architecture.

Properties of text view objects

In addition to the properties it inherits from the `text` (page 517) class, a `text view` object has these properties:

`allows undo`

Access: read/write
Class: *boolean*
Should the text view allow undo?

`ruler visible`

Access: read/write
Class: *boolean*
Is the ruler visible?

`smart insert delete enabled`

Access: read/write
Class: *boolean*

Is the smart insertion and deletion of text enabled? default is `true`; controls whether the view inserts or deletes space around inserted and deleted words so as to preserve proper spacing and punctuation

`spell checking enabled`

Access: read/write
Class: *boolean*
Is spell checking enabled for the text view?

`text container inset`

Access: read/write
Class: *point*
the amount of the empty space the view leaves around its associated text container; the text container represents the region where text is laid out; the inset is returned as a two-item list of numbers {width, height}; default is {0,0}

`text container origin`

Access: read only
Class: *point*
the origin of the text container within the text view, which is calculated from the view's bounds rectangle, container inset, and the container's used rect; the text container represents the region where text is laid out; the origin is returned as a two-item list of numbers {left, bottom}; see the `bounds` property of the `window` (page 86) class for information on the coordinate system

`uses ruler`

Access: read/write
Class: *boolean*
Should the text view use rulers?

Elements of text view objects

In addition to the elements it inherits from the `text` (page 517)class, a `text view` object can contain the elements listed below. Your script can access most elements with any of the key forms described in "Standard Key Forms" (page 30).

`text` (page 517)
Specify by: "Standard Key Forms" (page 30)
the view's text

Events supported by text view objects

A `text view` object supports handlers that can respond to the following events:

Drag and Drop

`conclude drop` (page 452)
`drag` (page 453)
`drag entered` (page 454)
`drag exited` (page 455)
`drag updated` (page 456)
`drop` (page 457)
`prepare drop` (page 459)

Editing

`begin editing` (page 335)
`changed` (page 336)
`end editing` (page 339)
`should begin editing` (page 341)
`should end editing` (page 342)

Key

`keyboard up` (page 141)

Mouse

`mouse entered` (page 146)
`mouse exited` (page 146)
`scroll wheel` (page 156)

Nib

`awake from nib` (page 130)

View

`bounds changed` (page 238)

Examples

The following statement, from the Open Panel sample application (first distributed with AppleScript Studio version 1.1), shows how to set the text of a text view. The statement actually clears the text view by setting its text to an empty string.

```
set contents of text view "path names" of scroll view "path names" of window
"main" to ""
```

You can use the following script in Script Editor to set the text color in a text view to green. Similar script statements will work within an AppleScript Studio application script (though you won't need the `tell application` block).

```
tell application "myTextViewApp"
    tell text view "text" of scroll view "scroller" of window "main"
        set text color to {0, 65535, 0}
    end tell
end tell
```

Text-handling terminology can be a bit confusing. AppleScript Studio's Text View suite defines the `text view` class, which inherits from the `text` (page 517) class. In addition, Cocoa defines the Text suite, which defines classes such as `character`, `paragraph`, `text`, and `word`, which in turn have elements such as `character`, `paragraph`, and `word`, and properties such as `color`, `font`, and `size`. The Text suite is a default suite that is available to all Cocoa applications that support scripting.

To further complicate matters, AppleScript classes such as `string` and `Unicode text` have `character`, `paragraph`, `text`, and `word` elements. In addition, if a class and a property (such as `text`) have the same name, use of the name within a `tell` statement defaults to the class.

The following script shows various operations you can perform on the text from a text view in a window. A text view is automatically enclosed in a scroll view, so the script accesses the text view through the scroll view. This script was tested with AppleScript Studio version 1.2 in Script Editor, but you can use similar statements in an application script (where you won't need the `tell application` block). This script may include statements that do not run with earlier versions of AppleScript Studio.

For this example, the text in the text view was "This is the only sentence." Note that you will get evaluation errors for some lines in this script if there is no text in the text view, or, for example, less than 7 characters (causing the statement `character 7 of text of text view 1` to generate an error).

Text View Suite

```
tell application "simple"
    tell window 1
        tell scroll view 1
            class of text of text view 1 -- text
            word 1 of text of text view 1 -- result: "This"
            set myTextObject to text of text view 1
                -- result: "This is the only sentence."
            class of myTextObject
                -- result: Unicode text
                -- In Studio version 1.1, the result is string.
            -- Next line generates error because Unicode text doesn't
            --  have a color property
            --color of myTextObject
                -- result: Can't get color of "This is the only sentence."
            word 1 of myTextObject -- result: "This"
            character 7 of myTextObject -- result: "s"
            character 1 of word 1 of myTextObject -- result: "T"
            set myText to contents of text view 1
                -- result: "This is the only sentence."
            class of myText -- result: Unicode text
            word 3 of myText -- result: "the"
            character 13 of myText -- result:"o"
            editable of text view 1 -- result: true (inherited from text)
            background color of text view 1 -- result: {65535, 65535, 65535}
            set myTextRef to a reference to (text of text view 1)
                -- result: every text of text view 1 of scroll view 1
                --           of window 1 of application "simple"
            class of myTextRef -- result: text
            color of myTextRef -- result: {0, 0, 0}
            --font of myTextRef -- NSCannotCreateScriptCommandError
            contents of myTextRef
                -- result: "This is the only sentence."
            size of myTextRef -- result: 12.0
            word 1 of myTextRef -- result: "This"
            color of word 1 of myTextRef -- result: {0, 0, 0}
            set color of word 1 of myTextRef to {65535, 0, 0}
                -- result: color of first worth ("This") is red
        end tell
    end tell
end tell
```

See the Examples section for the `scroll` (page 328) command for information about how to scroll text in a text view.

Version Notes

Starting with AppleScript Studio version 1.2, a script can say `word 1 of text view 1` instead of `word 1 of text of text view 1` (the default `of text` is assumed), though the longer version still works.

Support for drag-and-drop commands was added in AppleScript Studio version 1.2. See the "Drag and Drop Suite" (page 448) for details. In particular, the description for the `conclude drop` (page 452) event handler provides information on supporting drag and drop for `text view` (page 520) and `text field` (page 314) objects.

Document Revision History

Table A-1 describes revisions to *Inside Mac OS X: AppleScript Studio Terminology Reference*.

Table A-1 Document revision history .

Date	Notes
October, 2002	Updated for AppleScript Studio version 1.2 terminology.
	Completed descriptions for all AppleScript Studio 1.2 terminology and added many examples.
	Updated Version Information to flag differences between AppleScript Studio versions.
	Added the Document suite.
	Added the Drag and Drop suite.
	Moved suites into separate chapters.
	Divided front matter into "About This Document" (page 17) and "Terminology Fundamentals" (page 21) chapters.

Table A-1 Document revision history (continued).

Date	Notes
	Added following sections to "Terminology Fundamentals" (page 21) chapter: "Building AppleScript Studio Applications" (page 24) "Naming Conventions for Methods and Handlers" (page 31) "Accessing Properties and Elements" (page 32) "Event Handler Parameters" (page 34) "Connecting Key and Mouse Event Handlers" (page 35) "Scripting Error Messages" (page 35) "Using the Sample Scripts" (page 37) "Panels Versus Dialogs and Windows" (page 37) "A Word on Unicode Text" (page 39)
	Added Index (PDF only).
June, 2002	Preliminary version to ship with Mac OS X version 10.2. Documents only AppleScript Studio version 1.1 terminology.
	Many descriptions added.
	Added following sections to front matter: "Related Documentation" (page 19) "Version Information" (page 22)
April, 2002	First release of preliminary draft.

Index

This index provides links to the AppleScript Studio suites, classes, properties, elements, commands, events, and enumerated constants in this document. A term may appear more than once—for example, "box" appears as an entry for the box class and as an entry for the box element (with links to the box elements of the drawer, view, and window classes).

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y

Z