Accessibility Programming Guidelines for Mac

Developer

Contents

Figures, Tables, and Listings

Introduction to Accessibility Programming Guidelines for Cocoa

All Cocoa applications can and should be accessible to users with disabilities. The process of making an application accessible is called access enabling. In Cocoa applications, accessibility is achieved by user interface classes adopting the NSAccessibility informal protocol. Because standard Cocoa controls and views automatically adopt the NSAccessibility protocol, there is very little you have to do to access enable your application if you rely only on standard control and view objects.

If your application implements custom controls or views, however, you need to provide additional accessibility information to make your application completely accessible.

This topic discusses how Cocoa implements accessibility and describes specific tasks you need to perform to access enable your application.

> **Note:** With App Sandbox, you can and should enable your app for accessibility, as described in this document. However, you cannot sandbox an assistive app such as a screen reader, and you cannot sandbox an app that controls another app.

Who Should Read This Document?

All Cocoa application developers should read this document to learn how to access enable their applications. Even if your application uses only standard Cocoa views and controls, there is some information you need to supply to ensure your application is both completely accessible and provides a good user experience. If you're new to accessibility or if you're unsure why your application should be accessible, you should read *Accessibility Overview for OS X* to learn how applications make themselves accessible to assistive technologies in OS X.

If you're an assistive application developer, you don't need to read this document. Instead, you should read *Accessibility Overview for OS X* to become familiar with the OS X accessibility architecture and learn about the attributes associated with each type of accessibility object.

Organization of This Document

The following articles describe how Cocoa implements accessibility:

- "Accessibility Objects and the Accessibility Hierarchy" (page 6) describes the accessibility object as it is implemented by the Application Kit. It also describes the accessibility hierarchy that represents an application and discusses how an assistive application interacts with the accessibility objects in your application.
- "Hit-Testing and Keyboard Focus" (page 13) discusses how an assistive application can access user interface objects by screen position and by keyboard focus.
- "Accessibility Notifications" (page 15) discusses how an application notifies assistive applications that some change has occurred in the user interface, such as a new window opening.

The following articles describe how to access enable your application:

- "Access Enabling a Cocoa Application" (page 16) provides guidance on which tasks you may need to perform to access enable your application.
- "Supporting Attributes" (page 21) describes how to add an attribute to a custom accessible object.
- "Supporting Actions" (page 24) describes how to add an action to a custom accessible object.
- "Manipulating the Accessibility Hierarchy" (page 26) describes how to change an accessibility object's status in the accessibility hierarchy.

See Also

The Accessibility Reference Library contains several documents that cover accessibility.

- *Getting Started with Accessibility* provides a brief introduction to accessibility and describes learning paths you might choose to follow.
- *Accessibility Overview for OS X* describes the OS X accessibility architecture.
- NSAccessibility describes the NSAccessibility protocol and its methods and constants.
- *Accessibility Programming Guidelines for Carbon* describes how to access-enable a Carbon application.
- *Carbon Accessibility Reference* describes the functions, data types, and constants used in accessible Carbon applications.

In addition to these documents, Apple maintains a website devoted to accessibility in OS X, with links to more information about compatible assistive technologies:

- http://www.apple.com/accessibility

Accessibility Objects and the Accessibility Hierarchy

At the heart of OS X accessibility is the accessibility object. This object represents a user-accessible element in your application, such as a button or a menu item, regardless of the application framework on which your application depends. An assistive application uses an accessibility object to make the user interface element it represents accessible to users and to manage the user's interaction with that element.

This article describes the accessibility object as it is implemented by the Application Kit. It describes the components of accessibility objects and the hierarchy in which they are arranged. It also describes the methods to which accessibility objects can respond.

Attributes

Accessibility objects are described by a set of attributes. These attributes include the type of object, the object's place in the accessibility hierarchy (described in "The Accessibility Hierarchy" (page 10)), its value, its size and position on the display screen, and so on. In Cocoa, attributes are identified by `NSString` values, such as `NSAccessibilityWindowAttribute` and `NSAccessibilityZoomButtonAttribute` (see *NSAccessibility Protocol Reference* for the complete list of attribute names).

Each attribute has a value that provides an assistive application with information about the user interface element the accessibility object represents. For example, the value of a button's role attribute is `NSAccessibilityButtonRole` and the value of its enabled attribute might be `YES` (if the button is enabled). In addition, many accessibility objects include a value attribute (`NSAccessibilityValueAttribute`) that contains the current value of a user interface object. For example, the value of a text field object is the contents of the text field and the value of a selected radio button is `true`.

Attribute values are always handled as objects. They must be either objects that conform to the `NSAccessibility` protocol or one of the following types of objects:

- `NSArray`
- `NSAttributedString`
- `NSData`
- `NSDate`
- `NSDictionary`
- `NSNumber` (for Boolean, integer, and floating point values)

- `NSString`
- `NSURL`
- `NSValue` (for point, size, rectangle, and range values)

> **Note:** When used as an attribute value, an `NSArray` or `NSDictionary` object can contain only the types of objects listed here (an `NSDictionary` object should use `NSString` objects for its keys).

The `NSValue` objects must be created with the appropriate convenience methods, such as `valueWithPoint:`, to be properly recognized and transmitted to an assistive application. An attribute value can also be `nil`, although this causes a "No Value" error to be seen by an assistive application.

An assistive application needs to know which attributes a given accessibility object supports to be able to accurately represent the user interface object to the user. This is because an assistive application uses the values of supported attributes to manipulate the object and give the user information about it. An accessibility object returns a list of its supported attributes in the `accessibilityAttributeNames` method. When an assistive application needs to get the value of a specific attribute, an accessibility object returns it in the `accessibilityAttributeValue:` method.

Some attributes are read-only, such as the role, but others, such as a slider's value, can be modified by the assistive application. An attribute whose value is modifiable by an assistive application is called settable. The modification of attribute values is one way an assistive application can manipulate the user interface of another application (the other way is through actions, described in "Actions" (page 9)).

An accessibility object indicates that a specific attribute's value can be modified by an assistive application by returning `YES` from the `accessibilityIsAttributeSettable:` method. An assistive application modifies the value of a specific attribute using the `accessibilitySetValue:forAttribute:` method.

> **Note:** Accessibility methods, such as the ones described in this section, are automatically invoked by the Cocoa framework when an assistive application requests the information. As a Cocoa application developer, you do not call these methods yourself.

In `NSAccessibility.h`, Cocoa defines a large number of constants for standard attributes, many of which are relevant to only a few classes. For example, there are a large set of attributes that are for use with objects that handle text. A few of the attributes, however, are required for every accessibility object. If you must create an accessibility object to represent a custom user interface element, you should at a minimum implement the following attributes. (See "Make Custom Classes and Subclasses Accessible" (page 19) for details.)

- `NSAccessibilityRoleAttribute`. This required attribute identifies the type of the accessibility object. Cocoa defines a large number of role values, such as `NSAccessibilityApplicationRole`, `NSAccessibilityRadioButtonRole`, and `NSAccessibilityScrollAreaRole`. The value of the role attribute is for programmatic identification purposes only (an assistive application will never present the value of the role attribute to a user) and therefore does not need to be localized. See the `NSAccessibility` protocol description for a list of all the roles defined in Cocoa.
- `NSAccessibilityRoleDescriptionAttribute`. This attribute holds a localized string describing the object's role. It is intended to be readable by (or speakable to) the user. For example, the role description string attribute for a button should be "button". This allows an assistive application accurately to identify the object's type to the user.
- `NSAccessibilityTitleAttribute`. This attribute is required for any user interface object that displays a visible text title as part of its visual interface. For example, the display of a Cancel button includes the title "Cancel". Therefore, the accessibility object representing this button must include the title attribute with the string value `cancel`. This enables an assistive application accurately to convey the purpose of the object to the user. If, for example, the Cancel button does not include the title attribute, an assistive application may be able to describe it to the user only as "button".

  Note that the title attribute is not intended to contain static text that serves as the title for a user interface element, but that is not part of its visual interface. For objects that are accompanied by separate static text titles and descriptions, use the `NSAccessibilityTitleUIElementAttribute` and `NSAccessibilityServesAsTitleForUIElementsAttribute` attributes (described in "Provide Descriptive Information for All Elements" (page 16)).
- `NSAccessibilityDescriptionAttribute`. Available in OS X version 10.4 and later, this attribute contains a localized string that describes the object's purpose. All user interface objects that do not display a text title (and therefore do not include the title attribute) should include the description attribute to allow an assistive application to describe the purpose of the object to the user. For example, an OK button does not have to include the description attribute because it already includes the title attribute with the string value `ok`. A print button that displays a printer icon instead of the title "Print", however, must include the description attribute with a string value, such as `print`. If such a button does not include the description attribute, an assistive application will be able to describe it to the user only as "button".
- `NSAccessibilityParentAttribute`. This attribute is required for all accessibility objects except the application-level object. The value of this attribute is the accessibility object that contains this one. For example, the value of the parent attribute in a button cell's accessibility object is the accessibility object that represents the button's control object. Frequently, the parent of an accessibility object is the ancestor of the user interface object it represents. Sometimes, however, the ancestor of an object might not be interesting to an assistive application. An example of this is the NSButton that contains an NSButtonCell. In such cases, the uninteresting parent object may be marked as "ignored", which makes it invisible to an assistive application. For more information about how the accessibility hierarchy can differ from the inheritance hierarchy, see "The Accessibility Hierarchy" (page 10).

When returning an object's parent, respect the ignored state of objects by using the `NSAccessibilityUnignoredAncestor` function to return the first unignored parent for the object. For more information on this function, see "Manipulating the Accessibility Hierarchy" (page 26).

- `NSAccessibilityChildrenAttribute`. This optional attribute identifies accessibility objects contained within this one. For example, the value of the children attribute in a control's accessibility object contains the control's cells. Similarly, the children attribute in an accessibility object representing a view contains the view's subviews. As with the parent attribute, not all children of an object are interesting to an assistive application and some may be marked as "ignored" in the accessibility hierarchy. For more information on this, see "The Accessibility Hierarchy" (page 10).

  When returning an object's children, respect the ignored state of objects by using the `NSAccessibilityUnignoredChildren` function to return the unignored children for the object. For more information on this function, see "Manipulating the Accessibility Hierarchy" (page 26).
- `NSAccessibilitySizeAttribute`. This required attribute contains an NSValue object giving the size of the user interface object in pixels. This attribute is required so that assistive applications can do such things as display a focus ring around a user interface object.
- `NSAccessibilityHelpAttribute`. This optional attribute is a localized string that describes the purpose of the object. The string should be a short phrase that provides a hint to the user for how to handle the object. For NSView objects, the default implementation returns the view's help tag (tool tip) string.

For more information on the required and optional attributes associated with each role, see the appendix "Roles and Associated Attributes" in *Accessibility Overview for OS X*.

Actions

Accessibility objects can have actions associated with them. These actions are the generic actions a user can take on the user interface object the accessibility object represents. This means that accessibility actions are not context-specific. Because an assistive application is driving the user interface of your application, actions typically correspond to a single mouse click or key press. For example, a Print button supports the generic press action, not a context-specific print action. Actions are identified by NSString values, such as `NSAccessibilityPressAction`.

An accessibility object returns a list of its supported actions in the `accessibilityActionNames` method. When an assistive application needs to cause your application to perform a specific action, it sends the `accessibilityPerformAction:` message. When your application receives the `accessibilityPerformAction:` message, it performs the requested action in the same way it does when the request comes directly from the mouse or keyboard.

In `NSAccessibility.h`, Cocoa defines a small number of constants for standard actions. If you are creating (or extending) an accessibility object, you must restrict yourself to these actions. If you do not, an assistive application is likely to be unable to handle your custom action.

- `NSAccessibilityConfirmAction`. This action simulates pressing the Enter key, such as in a text field.
- `NSAccessibilityDecrementAction`. This action decrements the value of an object, such as a slider.
- `NSAccessibilityIncrementAction`. This action increments the value of an object, such as a slider.
- `NSAccessibilityPickAction`. This action selects a menu item.
- `NSAccessibilityPressAction`. This action simulates a single mouse click, such as on a button.
- `NSAccessibilityCancelAction`. This action simulates pressing a Cancel button.
- `NSAccessibilityRaiseAction`. This action makes the application window frontmost (behind, perhaps, any floating or system-modal windows that are currently displayed).
- `NSAccessibilityDeleteAction`. Simulates a delete action for items that must otherwise be dragged to a destination (such as Trash) to be deleted.
- `NSAccessibilityShowMenuAction`. This action opens a contextual menu in the element represented by this accessibility object. (This action can also be used to display a menu that is pre-associated with an element, such as the menu that displays when a user clicks Safari's back button slowly.)

See "Supporting Actions" (page 24) for details on implementing an action.

The Accessibility Hierarchy

Accessibility objects are arranged into a hierarchy that represents your application's user interface. The application-level accessibility object (representing the NSApplication object) is at the top of the hierarchy and its first-order children are the accessibility objects that represent the main application windows and the menu bar. An assistive application can traverse the hierarchy using various attributes, primarily `NSAccessibilityParentAttribute` and `NSAccessibilityChildrenAttribute` (see "Attributes" (page 6) for more information on specific attributes).

Assistive applications can also move through the hierarchy using attributes that lead to accessibility objects not related by containment. If, for example, an application displays a list of documents in one view and the contents of a selected document in another view, the application can use the `NSAccessibilityLinkedUIElementsAttribute` attribute to link these two views. Using the value of this attribute, an assistive application can allow a user to jump easily between the related views. Other attributes, such as `NSAccessibilityMenuBarAttribute` and `NSAccessibilityTopLevelUIElementAttribute`, provide direct, convenient access to specific objects in the application's user interface.

The accessibility hierarchy is built into the object hierarchy that already exists in a Cocoa application. For example, the top-level NSApplication object manages a collection of windows. An assistive application accesses these windows by asking the accessibility object representing the NSApplication object for the value of its `NSAccessibilityWindowsAttribute` attribute. (The windows can also be obtained by requesting the value of the children attribute, but the menu bar is included in this value.) Each of these windows contains a view hierarchy wherein the window's top view, the content view, contains any number of subviews, each of which can contain even more subviews, and so on. This hierarchy can be traversed by asking for the value of each object's `NSAccessibilityChildrenAttribute`. At the bottom of the hierarchy are the objects with which the user usually interacts, such as buttons and text fields.

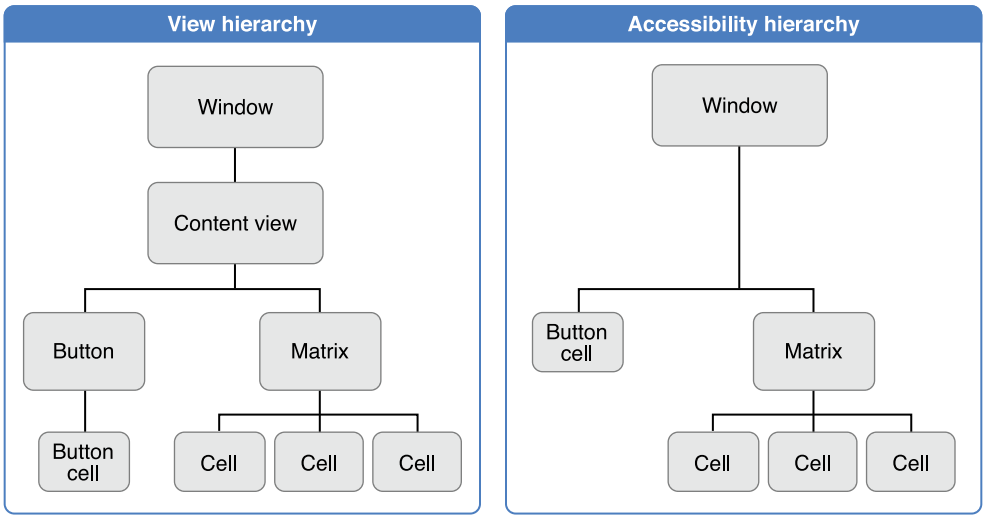
In some cases, however, an object needed in the Cocoa object hierarchy is not interesting to an assistive application. For example, the top-level view inside a window (the content view) is just a container for other views, which are the real objects that make up the user interface. Because a user never interacts directly with the content view itself, this implementation detail is hidden from the assistive application by marking the accessibility object representing the view as "ignored". Therefore, when a window is asked for its children, instead of returning its content view, the window should return the content view's children.

NSControl objects are also often hidden. An NSControl usually has a one-to-one relationship to an NSCell object, which implements the majority of the control's behavior. In these cases, the NSControl object is ignored and the accessibility hierarchy jumps from the view that contains the control to the control's cell. Controls that can represent more than one cell, however, such as NSMatrix or NSTableView, are not ignored.

It's important to note that ignored accessibility objects are not excised from an application's accessibility hierarchy; rather, they are not reported to an assistive application. An ignored object must remain in its place in the accessibility hierarchy to provide a bridge between its surrounding objects and to implement the functions and attributes needed by its children.

Figure 1 (page 12) compares the view hierarchy for a window and the accessibility hierarchy for the same window. The content view and the button control (an NSButton) are ignored objects, which are in the view hierarchy but not the accessibility hierarchy. The accessibility hierarchy skips the content view and goes directly to its children and the button is replaced by the button's cell (an NSButtonCell).

Figure 1 View hierarchy versus accessibility hierarchy



Objects indicate that they should be ignored by implementing the `accessibilityIsIgnored` method and returning `YES`. In fact, the default NSView implementation returns `YES`. To make themselves visible to an assistive application, therefore, NSView subclasses must override the `accessibilityIsIgnored` method and return `NO`. For details on working with ignored objects, see "Manipulating the Accessibility Hierarchy" (page 26).

Hit-Testing and Keyboard Focus

Assistive applications frequently need to be able to access the user interface object that is at a specific screen location or that currently has keyboard focus. The NSAccessibility protocol defines methods an assistive application uses to get this information from your application.

Hit-Testing

Hit-testing identifies the deepest member of the accessibility hierarchy that is located at a particular point on the screen. For example, an assistive application may want to identify the user-accessible object that lies beneath the current cursor position.

The NSAccessibility protocol defines the `accessibilityHitTest:` method which, similarly to the standard `hitTest:` view method, returns the farthest descendent of the accessibility hierarchy that contains the specified point (in bottom-left relative screen coordinates). Unlike the standard `hitTest:` view method, however, the `accessibilityHitTest:` method is never called unless the Cocoa accessibility implementation has already determined that the point lies within the receiver. This means that a class needs to override this method only when it contains child accessibility objects; childless accessibility objects never have to override this method.

When an accessibility object receives the `accessibilityHitTest:` method, therefore, it knows that the specified point lies somewhere within it. The accessibility object then has the opportunity to perform deeper hit-testing by identifying which child accessibility object, if any, contains the point. NSMatrix, for example, identifies which of its cells contains the point and propagates the `accessibilityHitTest:` message to it. Eventually, a childless accessibility object acknowledges that it is the object at the specified point and returns `self` from the method. An ignored accessibility object without any children does not return `self`; instead, it returns its first unignored ancestor.

Determining Keyboard Focus

Focus-testing identifies the deepest member of the accessibility hierarchy that has the keyboard focus. The NSAccessibility protocol defines the `accessibilityFocusedUIElement` method an assistive application can send to determine which object has keyboard focus.

As with the `accessibilityHitTest:` method, the `accessibilityFocusedUIElement` method is never called unless the Cocoa accessibility implementation has already determined that the receiver has keyboard focus somewhere within it. This means that a class needs to override this method only when it contains child accessibility objects; childless accessibility objects never have to override this method.

When an accessibility object receives the `accessibilityFocusedUIElement` message, therefore, it knows that it (or one of its children) has keyboard focus. The accessibility object then has the opportunity to perform deeper focus-testing by determining which of its children has keyboard focus and sending to it the `accessibilityFocusedUIElement` message. NSMatrix, for example, identifies which of its cells has the focus and propagates the `accessibilityFocusedUIElement` message to it. Eventually, a childless accessibility object acknowledges that it has keyboard focus and returns `self` from the method. An ignored object without any children returns its first unignored ancestor instead of `self`.

Accessibility Notifications

Assistive applications need to be notified when something in the user interface of your application changes, such as a new window appearing or a control changing its value. An assistive application can then query an accessibility object in your application for its new state and present the information to the user.

NSAccessibility defines a number of notifications such as `NSAccessibilityWindowResizedNotification` and `NSAccessibilityFocusedUIElementChangedNotification`. The assistive application can register to receive notifications coming from a particular accessibility object or from any accessibility object in your application.

NSAccessibility notifications require special handling, so they cannot be sent using an NSNotificationCenter like regular notifications. Instead, notifications are sent with the following function:

```
void NSAccessibilityPostNotification( id element, NSString *notification );
```

The `element` argument is the affected accessibility object and `notification` is the notification name describing the event.

If you implement custom objects, you may need to send the `NSAccessibilityValueChangedNotification`; it is not likely you will ever need to send any others.

> **Note:** Cocoa sends notifications automatically for its user interface objects.

Access Enabling a Cocoa Application

This article describes the tasks you may have to perform to access enable a Cocoa application. Because many Cocoa classes support accessibility by adopting the NSAccessibility protocol, standard objects are already accessible to a great degree. For the most part, you have to add code for only the application-specific information that Cocoa cannot automatically supply.

This article is divided into sections that describe the tasks associated with specific scenarios in your application. Read this article to find out which of the tasks described in each section are necessary in your application. For example, the tasks in the first section, "Provide Descriptive Information for All Elements," are required for all applications. The tasks in "Make Custom Classes and Subclasses Accessible" (page 19), on the other hand, are required for only those applications that define custom classes or subclasses.

Provide Descriptive Information for All Elements

An assistive application needs to be able to describe all accessible user interface elements to the user. Often, an assistive application can present the title of the element to the user, but sometimes an element's title is either unavailable or not sufficiently descriptive. Therefore, you must examine your application and ensure that all accessibility objects supply descriptive information about themselves in either the title or description attributes.

First, determine if a given accessibility object already includes the title attribute. An object that displays a text title as part of its visual interface, such as an OK button, already includes the title attribute with the value of the displayed text. Such an object does not need a description attribute because its title is sufficient to convey its purpose to the user.

A button that displays an icon instead of a text title, however, does not have a title attribute. An example of such an object is a "back" button that displays a left-pointing arrow instead of the word "Back". An assistive application cannot describe such a button's purpose to a user unless the accessibility object representing the button includes the description attribute. If you have such an object in your application, you must supply an appropriate description in the description attribute.

Some user interface elements do not display a title, but are accompanied by a piece of static text that serves as a title. An example of this is a set of editable text fields accompanied by the string "Address:". To a sighted user, the proximity of the string to the text fields implies that "Address:" serves as the title for the entire set of

text fields. An assistive application, however, cannot make this determination. If you use static text objects to title user interface elements (or sets of user interface elements), you must make the relationship between them clear to assistive applications.

To do this, you create an accessibility object to represent the static text object. Then, in each accessibility object representing one of the titled user interface elements (each of the address fields, in this example), you add the `NSAccessibilityTitleUIElementAttribute` attribute. The value of this attribute is the accessibility object representing the static text title. Finally, in the static text title accessibility object, add the `NSAccessibilityServesAsTitleForUIElementsAttribute` attribute. The value of this attribute is an array of the accessibility objects for which this static text object serves as title (in this example, the array comprises the set of editable text fields).

The title and description attributes provide information about the layout of your application's user interface and so their values are static and not settable by an assistive application. This means you can choose to use a convenience method to provide these values instead of having to subclass the accessibility object and override the appropriate methods. See "Supporting Attributes" (page 21) for details on the two techniques you can use to support these descriptive attributes.

Link Conceptually Related Elements

In OS X version 10.4, the NSAccessibility protocol introduced two new attributes that allow you to describe conceptual links between accessibility objects. Conceptual links are those that are visually implied onscreen, but that are difficult for an assistive application to determine. An example of such a link is in the Mail application where the main window's upper view contains a list of message headers and the lower view displays the content of a selected message. It is obvious to a sighted user that the text in the lower view is the contents of the selected message header in the upper view, but an assistive application cannot make this determination. Another example is the relationship between a search field and the view containing the search results. An assistive application has no way to know where you display search results unless you make clear the relationship between the search field and the results view.

If your application includes elements that are linked conceptually, you should ensure that the accessibility objects representing those elements include the `NSAccessibilityLinkedUIElementsAttribute` attribute. The value of this attribute is an array of accessibility objects to which the element is linked. An assistive application can use the accessibility objects in the array to provide to the user a shortcut between the linked objects. Without this attribute, a user has to know which views are conceptually linked and must step through every intervening object to move between them.

Like the description and title attributes, the value of the linked-elements attribute is part of the layout of your application's user interface. An assistive application does not need to set the value of this attribute, so you have the option of using a convenience method to supply the value or of subclassing the accessibility objects and overriding the appropriate methods. For more details on both these techniques, see "Supporting Attributes" (page 21).

Make Substructure Accessible

Sometimes, your application uses a view object that contains substructure you need to make accessible. However, some NSView objects are implemented as monolithic objects; objects that, as far as Cocoa is concerned, contain no substructure. NSScrollBar is an example of such an object. Even though a sighted user can manipulate separately the parts of a scroll bar (the scroller, the page-up and page-down regions in the scroll track, and the scroll arrows), Cocoa does not represent these parts as separate objects. Therefore, an assistive application cannot make any of these subviews directly accessible to a user. This also affects the determination of mouse location and keyboard focus. A scroll bar can tell an assistive application that it has keyboard focus, for example, but not the precise part of itself that has keyboard focus.

If you use an object, such as an NSScrollBar, in your application and you need to make its substructure accessible, you must create an accessibility object to represent each part. The accessibility object must, of course, implement the NSAccessibility protocol, but it can be very lightweight. This is because it can ask its parent (the NSScrollBar object, in this example) to supply most of the information for which it might be asked, such as its containing window or position. This accessibility object's main responsibility is to allow an otherwise inaccessible part of the user interface to be represented in the application's accessibility hierarchy.

The most efficient way to create these accessibility objects is to define a utility class that implements the NSAccessibility protocol. Then, you create an instance of this class as needed and allow it to provide only the information that is specific to the subview it represents. In particular, the utility class should implement the hit-test and focus-test methods. This way, the accessibility object representing a subview can return itself when it receives a hit-test or focus-test message.

In most cases, it works well to create these accessibility objects as they are asked for; it's not necessary to create and store them in advance. This is because these accessibility objects will never be asked for (and therefore never created) if the user has not enabled access by assistive applications in the Universal Access System Preferences.

Your utility class should create an instance given the parent accessibility object of the subview and its role. This allows you easily to provide some of the required attributes for the new accessibility object. In the NSScrollBar example, the parent object is the accessibility object representing the NSScrollBar as a whole and the role is `NSAccessibilityScrollBarRole`.

Make Custom Classes and Subclasses Accessible

If you implement custom classes or custom subclasses of Cocoa classes in your application, you may need to create accessibility objects to represent them. Any custom subclass of NSObject, for example, does not inherit any accessibility support and must implement the methods to supply supported attributes and actions and to return hit-test and focus-test information.

Although NSObject does not adopt the NSAccessibility protocol, the following common base classes do:

- NSApplication
- NSWindow
- NSView
- NSControl
- NSCell

Although all the classes listed above implement the NSAccessibility protocol, they each support a different set of attributes, actions, and methods and each has its own default ignored status. It's important to be aware of the base set of functionality of these classes so you can determine what your subclass needs to override. Table 1 shows the default accessibility support of each class.

Table 1 AppKit classes and default accessibility support

Class	Supported attributes	Implemented methods	Ignored by default
NSApplication	role, role description, menu bar, windows, active, main window, key window, focused accessibility object (UIElement)	hit-testing, focus testing	No
NSWindow	role, role description, title, focused, parent, position, size, main, key	hit-testing, focus testing	No
NSView	role, role description, help, focused, parent, children, window, position, size	hit-testing, focus testing, keystroke handling	Yes
NSControl	role, role description, help, focused, parent, children, window, position, size, enabled	position and size of child methods	Yes, unless control has multiple child cells

Class	Supported attributes	Implemented methods	Ignored by default
NSCell	role, role description, help, focused, parent, children, window, position, size, enabled, value	hit-testing, focus-testing	No

When you need to make a custom class or subclass accessible, you create an accessibility object to represent it, as described in "Make Substructure Accessible" (page 18). Be sure to take into account the default ignored status of your chosen base class so you can override this value if appropriate for your subclass.

Supporting Attributes

This article describes two techniques you can use to support additional attributes and provide values for individual attributes. For more information on why you might need to do this, see "Access Enabling a Cocoa Application" (page 16).

Using a Convenience Method to Support Attributes

OS X version 10.4 introduced the `accessibilitySetOverrideValue:forAttribute:` convenience method, which allows you to add an attribute or change the value of an existing attribute without subclassing the accessibility object. Use this method only for adding attributes and supplying values that are unchanging and pertain to the layout of your application, such as titles, descriptions, and links between conceptually related elements.

Don't use this method to add attributes that should be settable by an assistive application. Attributes you add using the `accessibilitySetOverrideValue:forAttribute:` method are unsettable by default and will remain unsettable even if you subclass the accessibility object and override the `accessibilityIsAttributeSettable:` method. For information on the methods to override to support attributes that can be settable, see "Overriding Methods to Support Attributes."

Overriding Methods to Support Attributes

NSAccessibility defines four methods for accessing an object's attributes:

- `accessibilityAttributeNames`
- `accessibilityAttributeValue:`
- `accessibilityIsAttributeSettable:`
- `accessibilitySetValue:forAttribute:`

The first method returns an array of attribute names supported by the accessibility object. An assistive application uses this method to determine which attributes an accessibility object supports, so it can later get information about specific ones. The other three methods each operate on an individual attribute, querying or setting the attribute's value.

When supporting an additional attribute in a subclass, you must override the `accessibilityAttributeNames` method and append the attribute to the array of attributes already supported by the superclass. For example, if your accessibility object needs to support the `NSAccessibilityLinkedUIElementsAttribute`, you override this method and insert the attribute into the array. This method is likely to be called numerous times and there could be a large number of attributes, so it should be efficient.

Listing 1 shows an implementation that adds the `@"MyAttribute"` attribute, initializing a static variable with the array of attribute names the first time the method is invoked:

Listing 1 Supporting an additional attribute

```
static NSString *MyAttributeName = @"MyAttribute";

- (NSArray *)accessibilityAttributeNames
{
    static NSArray *attributes = nil;
    if (attributes == nil) {
        attributes = [[[super accessibilityAttributeNames]
                arrayByAddingObject:MyAttributeName] retain];
    }
    return attributes;
}
```

When a subclass supports an additional attribute, it must also override the methods that can get that attribute's value, determine if it is settable, and set its value. Listing 2 shows possible implementations for these methods, using the `@"MyAttribute"` attribute added in Listing 1 (page 22).

Listing 2 Supplying information about an added attribute

```
- (id)accessibilityAttributeValue:(NSString *)attribute
{
    // Determine if the attribute being asked about is the newly added one.
    if ( [attribute isEqualToString:MyAttributeName] )
        return [NSNumber numberWithInt:_MyAttribute];
    else
        // The attribute in question is not the added one, so let
        // the superclass handle it.
```

```
        return [super accessibilityAttributeValue:attribute];
}

- (BOOL)accessibilityIsAttributeSettable:(NSString *)attribute
{
    // Determine if the attribute being asked about is the newly added one.
    if ( [attribute isEqualToString:MyAttributeName] )
        return YES; // YES if MyAttribute is settable, NO if it is not.
    else
        // The attribute in question is not the added one, so let
        // the superclass handle it.
        return [super accessibilityIsAttributeSettable:attribute];
}

- (void)accessibilitySetValue:(id)value
                forAttribute:(NSString *)attribute
{
    // Determine if the attribute being asked about is the newly added one.
    if ( [attribute isEqualToString:MyAttributeName] )
        // Call the subclass's method to set the attribute's value.
        [self setMyAttributeValue:[value intValue]];
    else
        // The attribute in question is not the added one, so let
        // the superclass handle it.
        [super accessibilitySetValue:value forAttribute:attribute];
}
```

When setting an attribute, the implementation ideally should invoke the same methods that would have been invoked if the attribute was modified directly from the user interface.

Supporting Actions

NSAccessibility defines three methods for accessing an object's actions:

- `accessibilityActionNames`
- `accessibilityActionDescription:`
- `accessibilityPerformAction:`

The first method returns an array of action names supported by the accessibility object, the second returns a localized string describing a particular action, and the third performs a particular action.

When supporting an action in a subclass, you need to override all three methods. In the `accessibilityActionNames` method, you need to invoke the superclass's implementation and append your new action. This allows an assistive application to get an accurate list of all actions you support. In the other two methods, compare the action name to those your subclass supports; if no match is found, invoke the superclass's implementation. Listing 1 shows sample implementations of these methods that add a new action named `@"Boing"`.

Listing 1 Supporting a new action

```
static NSString *MyBoingActionName = @"Boing";

- (NSArray *)accessibilityActionNames
{
    static NSArray *actions = nil;
    if (actions == nil) {
        actions = [[[super accessibilityActionNames]
                arrayByAddingObject:MyBoingActionName] retain];
    }
    return actions;
}

- (NSString *)accessibilityActionDescription:(NSString *)action
{
```

```
    if ( [action isEqualToString:MyBoingActionName] )
        return NSLocalizedString(@"BoingDescription",
                    @"Performs the Boing action");
    else
        return [super accessibilityActionDescription:action];
}

- (void)accessibilityPerformAction:(NSString *)action
{
    if ( [action isEqualToString:MyBoingActionName] )
        [self doBoing];
    else
        [super accessibilityPerformAction:action];
}
```

When performing an action, the subclass's implementation ideally should invoke the same methods that are invoked if the action is performed directly from the user interface.

Manipulating the Accessibility Hierarchy

As described in "The Accessibility Hierarchy" (page 10), some accessibility objects are uninteresting to assistive applications and are designated as ignored. An ignored accessibility object is never exposed to an assistive application, but it does participate in the accessibility hierarchy. As you develop your Cocoa application, you may find that you need to adjust the accessibility hierarchy to present a cleaner representation to assistive applications. This article describes how to manipulate the accessibility hierarchy.

Any object indicates that it should not be visible to assistive applications by implementing the `accessibilityIsIgnored` method and returning `YES`. Because an ignored object still participates in the internal parent-child hierarchy, however, it must implement the `NSAccessibilityParentAttribute` and `NSAccessibilityChildrenAttribute` attributes to provide a link between the objects above and below it.

When an attribute value, hit-test, or focus test would otherwise return an ignored object, the object must be replaced by an unignored object. Depending on the situation, the object should be replaced either by an ancestor or by one or more descendants. The search for replacement objects iterates through the internal hierarchy until all ignored objects are eliminated. For example, when an object's children are requested, a child that is ignored is replaced by the child's children. If one of those children is also ignored, it is replaced by its children, and so on. Likewise, when asking for an object's parent, an ignored parent is skipped and the first unignored ancestor is returned.

To help manage ignored objects, Cocoa provides several functions that perform the necessary search-and-replace procedures. The following two functions test whether the given object is ignored and returns either that object, if not ignored, or the first unignored parent (ancestor) or child (descendant), if the object is ignored:

```
id NSAccessibilityUnignoredDescendant(id element);
id NSAccessibilityUnignoredAncestor(id element);
```

You use these utility functions to ensure that you return an unignored object in response to, for example, a request for the value of the `NSAccessibilityChildrenAttribute` attribute. The `NSAccessibilityUnignoredDescendant` function returns the first unignored object it finds as it checks the accessibility object passed to it and works its way down the object's descendant chain. If an ignored object has either no unignored children or multiple unignored children, this function fails and returns `nil`.

This does not mean that you should return `nil` in response to a hit-testing or focus-testing method, however. For hit-testing and focus-testing, you should return either an unignored child or `self`.

Similarly, the `NSAccessibilityUnignoredAncestor` function returns the first unignored object it finds as it checks the accessibility object passed to it and works its way up the object's ancestor chain.

Cocoa also provides the following two convenience functions that replace ignored objects with their unignored children:

```
NSArray *NSAccessibilityUnignoredChildren(NSArray *originalChildren);
NSArray *NSAccessibilityUnignoredChildrenForOnlyChild(id originalChild);
```

The `NSAccessibilityUnignoredChildren` function takes a set of accessibility objects and replaces any ignored objects with their children, recursing if necessary. The `NSAccessibilityUnignoredChildrenForOnlyChild` function replaces a single child with its unignored children. It is equivalent to the following usage of the `NSAccessibilityUnignoredChildren` function:

```
NSAccessibilityUnignoredChildren( [NSArray arrayWithObject:originalChild] )
```

You can use these convenience functions to make it easier to respect the ignored status of the accessibility objects you return to assistive applications. Listing 1 shows one way to do this.

Listing 1 Returning attribute values for unignored children

```
- (id)accessibilityAttributeValue:(NSString *)attribute
{
    if ( [attribute isEqualToString:NSAccessibilityChildrenAttribute] )
        return NSAccessibilityUnignoredChildren( _children );

    else if ( [attribute isEqualToString:NSAccessibilityParentAttribute] )
        return NSAccessibilityUnignoredAncestor( _parent );

    else
        return [super accessibilityAttributeValue:attribute];
}
```

In the code in Listing 1, `_children` and `_parent` represent the objects immediately above and below the given object, regardless of their ignored status. For an NSView object, for example, these are the subviews and superview, respectively.

Document Revision History

This table describes the changes to *Accessibility Programming Guidelines for Mac* .

Date	Notes
2013-09-18	Updated the list of object types that can represent attribute values.
2011-10-26	Added information about App Sandbox.
2007-02-08	Added recommended usage of the linked UI element attribute for search fields and results.
2005-04-29	Updated for OS X v10.4. Changed title from "Accessibility."
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.

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