Text Views

Cocoa > User Experience

2004-02-09

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

Simultaneously published in the United States and Canada.

Contents

Figures

Introduction to Text Views

Text Views provides information about text views, the main user interface objects of the Cocoa text system. Text views handle user events to provide text entry and modification. Text views can display multiple lines of text laid out in paragraphs with all the characteristics of sophisticated typesetting.

Who Should Read This Document

You should read the information presented here if you need to understand what text views are and how they work.

Organization of This Document

The following documents describe text views:

- "Text Fields, Text Views, and the Field Editor" (page 9) introduces and compares the main user interface objects of the text system.
- "Overview of Text Editing" (page 17) provides a high-level view of the text editing mechanism and explains the message sequence that occurs when a text view receives a key event.
- "Subclassing NSTextView" (page 13) explains the responsibilities an NSTextView subclass must fulfill to interact successfully with the text system.

See Also

For more information, refer to the following documents:

- *Text System User Interface Layer Programming Guide for Cocoa* describes text views in the context of the Cocoa text system. This document includes multiple articles discussing text views.
- *Text System Overview* provides an overview of the Cocoa text system.
- *Text Input Management* describes the classes that interact with each other to transmit input from the user's keyboard or mouse to a text view.

See Also

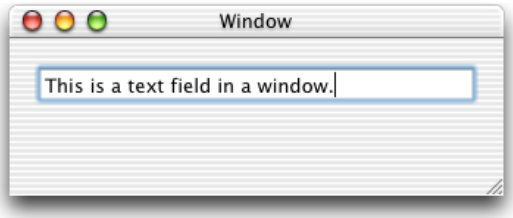
Text Fields, Text Views, and the Field Editor

Text fields, text views, and the field editor are important objects in the Cocoa text system because they are central to the user's interaction with the system. They provide text entry, manipulation, and display. If your application deals in any way with user-entered text, you should understand these objects.

Text Fields

A text field is a user interface control object instantiated from the NSTextField class. Figure 1 shows a text field. Text fields display small amounts of text, typically (although not necessarily) a single line. Text fields also provide places for users to enter text responses, such as search parameters. Like all controls, a text field has a target and an action. By default, text fields send their action message when editing ends—that is, when the user presses Return or moves focus to another control. You can also control a text field's shape and layout, the font and color of its text, background color, whether the text is editable or read-only, whether it is selectable or not (if read-only), and whether the text scrolls or wraps when the text exceeds the text field's visible area.

Figure 1 A text field



To create a secure text field for password entry, you use NSSecureTextField, a subclass of NSTextField. Secure text fields display bullets in place of characters entered by the user, and they do not allow cutting or copying of their contents. You can get the text field's value using the `stringValue` method, but users have no access to the value.

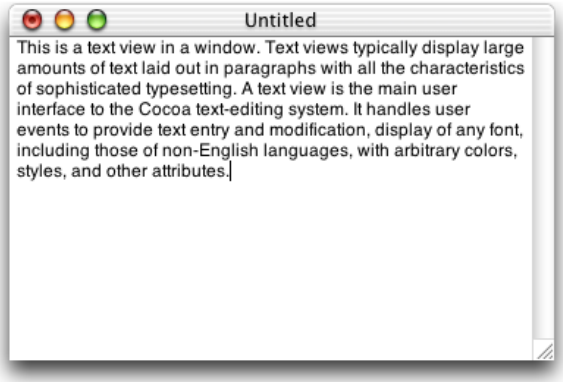
The usual way to instantiate a text field is to drag an NSTextField object from the the Cocoa-Views palette in Interface Builder and place it in a window of your application's user interface. Then, if you then want to convert the text field to a secure text field, you select it, open the Info window (Command-Shift-I), choose the Custom Class pane (Command-5), and select NSSecureTextField.

See *Text Fields* more information.

Text Views

Text views are user interface objects instantiated from the NSTextView class. Figure 2 shows a text view. Text views typically display multiple lines of text laid out in paragraphs with all the characteristics of sophisticated typesetting. A text view is the main user interface to the Cocoa text-editing system. It handles user events to provide text entry and modification, and to display any font, including those of non-English languages, with arbitrary colors, styles, and other attributes.

Figure 2 A text view

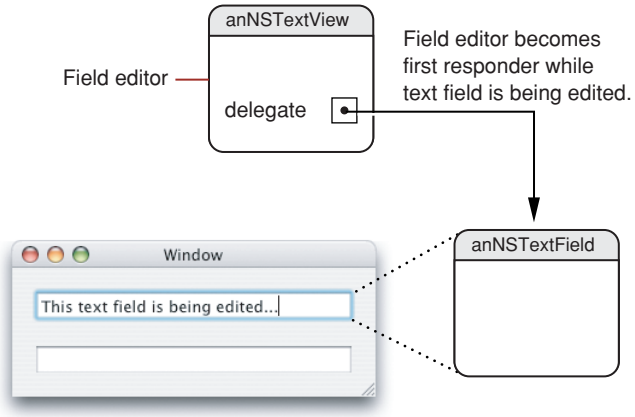


The Cocoa text system supports text views with many other underlying objects providing text storage, layout, font and attribute manipulation, spell checking, undo and redo, copy and paste, drag and drop, saving of text to files, and other features. NSTextView is a subclass of NSText, which is a separate class for historical reasons. You don't instantiate NSText, although it declares many of the methods you use with NSTextView. When you put an NSTextView object in an NSWindow object, you have a full-featured text editor whose capabilities are provided "for free" by the Cocoa text system. (See "Building a Text Editor in 15 Minutes" in *Text System Overview* for more information.)

The Field Editor

The field editor is a single NSTextView object that is shared among all the controls, including text fields, in a window. This text view object inserts itself into the view hierarchy to provide text entry and editing services for the currently active text field. When the user shifts focus to a text field, the field editor begins handling keystroke events and display for that field. The field editor designates the current text field as its delegate, enabling the text field to control changes to its contents. When the focus shifts to another text field, the field editor attaches itself to that field instead. Figure 3 illustrates the disposition of the field editor in relation to the text field it is editing.

Figure 3 The field editor



Because only one of the text fields in a window can be active at a time, the system needs only one NSTextView instance per window to be the field editor. Among its other duties, the field editor maintains the selection. Therefore, a text field that's not being edited typically does not have a selection at all. (However, developers can substitute custom field editors, in which case there could be more than one field editor.)

For more information about the field editor, see "Working With the Field Editor."

Subclassing NSTextView

This article explains how to subclass `NSTextView`. It describes the major areas where a subclass has obligations or where it can expect help in implementing new features.

Note: To modify editing behavior, your first resort should be to notification or delegation, rather than subclassing. It may be tempting to start by trying to subclass `NSTextView` and override `keyDown:` , but that's usually not appropriate, unless you really need to deal with raw key events before input management or key binding. In most cases it's more appropriate to work with one of the text view delegate methods or with text view notifications, as described in "Delegate Messages and Notifications" in *Text Editing Programming Guide for Cocoa*.

The text system requires `NSTextView` subclasses to abide by certain rules of behavior, and `NSTextView` provides many methods to help subclasses do so. Some of these methods are meant to be overridden to add information and behavior into the basic infrastructure. Some are meant to be invoked as part of that infrastructure when the subclass defines its own behavior.

Updating State

`NSTextView` automatically updates the Font window and ruler as its selection changes. If you add any new font or paragraph attributes to your subclass of `NSTextView`, you'll need to override the methods that perform this updating to account for the added information. The `updateFontPanel` method makes the Font window display the font of the first character in the selection. You could override this method to update the display of an accessory view in the Font window. Similarly, `updateRuler` causes the ruler to display the paragraph attributes for the first paragraph in the selection. You can also override this method to customize display of items in the ruler. Be sure to invoke the `super` implementation in your override to have the basic updating performed as well.

Custom Import Types

`NSTextView` supports pasteboard operations and the dragging of files and colors into its text. If you customize the ability of your subclass to handle pasteboard operations for new data types, you should override the `readablePasteboardTypes` and `writablePasteboardTypes` methods to reflect those types. Similarly, to support new types of data for dragging operations, you should override the `acceptableDragTypes` method. Your implementation of these methods should invoke the superclass implementation, add the new data types to the array returned from `super`, and return the modified array.

For dragging operations, if your subclass's ability to accept your custom dragging types varies over time, you can override `updateDragTypeRegistration` to register or unregister the custom types according to the text view's current status. By default this method enables dragging of all acceptable types if the receiver is editable and a rich text view.

To read and write custom pasteboard types, you must override the `readSelectionFromPasteboard:type:` and `writeSelectionToPasteboard:type:` methods. In your implementation of these methods, you should read the new data types your subclass supports and let the superclass handle any other types.

Altering Selection Behavior

Your subclass of `NSTextView` can customize the way selections are made for the various granularities (such as character, word, and paragraph) described in"Setting Focus and Selection Programmatically" in *Text Editing Programming Guide for Cocoa*. While tracking user changes to the selection, whether by the mouse or keyboard, an `NSTextView` object repeatedly invokes `selectionRangeForProposedRange:granularity:` to determine what range to actually select. When finished tracking changes, it sends the delegate a `textView:willChangeSelectionFromCharacterRange:toCharacterRange:` message. By overriding the `NSTextView` method or implementing the delegate method, you can alter the way the selection is extended or reduced. For example, in a code editor you can provide a delegate that extends a double click on a brace or parenthesis character to its matching delimiter.

These mechanisms aren't meant for changing language word definitions (such as what's selected on a double click). That detail of selection is handled at a lower (and currently private) level of the text system.

Preparing to Change Text

If you create a subclass of `NSTextView` to add new capabilities that will change the text in response to user actions, you may need to modify the range selected by the user before actually applying the change. For example, if the user is making a change to the ruler, the change must apply to whole paragraphs, so the selection may have to be extended to paragraph boundaries. Three methods calculate the range to which certain kinds of change should apply. The `rangeForUserTextChange` method returns the range to which any change to characters themselves—insertions and deletions—should apply. The `rangeForUserCharacterAttributeChange` method returns the range to which a character attribute change, such as a new font or color, should apply. Finally, `rangeForUserParagraphAttributeChange` returns the range for a paragraph-level change, such as a new or moved tab stop, or indent. These methods all return a range whose location is `NSNotFound`if a change isn't possible; you should check the returned range and abandon the change in this case.

Notifying About Changes to the Text

In actually making changes to the text, you must ensure that the changes are properly performed and recorded by different parts of the text system. You do this by bracketing each batch of potential changes with `shouldChangeTextInRange:replacementString:` and `didChangeText` messages. These methods ensure that the appropriate delegate messages are sent and notifications posted. The first method asks the delegate for permission to begin editing with a `textShouldBeginEditing:` message. If the delegate returns `NO`, `shouldChangeTextInRange:replacementString:` in turn returns `NO`, in which case your subclass should disallow the change. If the delegate returns `YES`, the text view posts an `NSTextDidBeginEditingNotification`, and `shouldChangeTextInRange:replacementString:` in

turn returns `YES`. In this case you can make your changes to the text, and follow up by invoking `didChangeText`. This method concludes the changes by posting an `NSTextDidChangeNotification`, which results in the delegate receiving a `textDidChange:` message.

The `textShouldBeginEditing:` and `textDidBeginEditing:` messages are sent only once during an editing session. More precisely, they're sent upon the first user input since the `NSTextView` became the first responder. Thereafter, these messages—and the `NSTextDidBeginEditingNotification`—are skipped in the sequence. The `textView:shouldChangeTextInRange:replacementString:` method, however, must be invoked for each individual change.

Smart Insert and Delete

`NSTextView` defines several methods to aid in "smart" insertion and deletion of text, so that spacing and punctuation are preserved after a change. Smart insertion and deletion typically applies when the user has selected whole words or other significant units of text. A smart deletion of a word before a comma, for example, also deletes the space that would otherwise be left before the comma (though not placing it on the pasteboard in a Cut operation). A smart insertion of a word between another word and a comma adds a space between the two words to protect that boundary. `NSTextView` automatically uses smart insertion and deletion by default; you can turn this behavior off using `setSmartInsertDeleteEnabled:`. Doing so causes only the selected text to be deleted, and inserted text to be added, with no addition of white space.

If your subclass of `NSTextView` defines any methods that insert or delete text, you can make them smart by taking advantage of two `NSTextView` methods. The `smartDeleteRangeForProposedRange:` method expands a proposed deletion range to include any white space that should also be deleted. If you need to save the deleted text, however, it's typically best to save only the text from the original range. For smart insertion, `smartInsertForString:replacingRange:beforeString:afterString:` returns by reference two strings that you can insert before and after a given string to preserve spacing and punctuation. See the method descriptions for more information.

Overview of Text Editing

The Cocoa text system implements a sophisticated editing mechanism that enables input of complex text character and style information. It is important to understand this mechanism if your code needs to hook into it.

The text system provides a number of control points where you can customize the editing behavior:

- Text system classes provide methods to control many of the ways in which they perform editing.
- You can implement more control through the Cocoa mechanisms of notification and delegation.
- In extreme cases where the capabilities of the text system are not suitable, you can replace the text view with a custom subclass.

The Editing Environment

Text editing is performed by a text view object. Typically, a text view is an instance of `NSTextView` or a subclass. A text view provides the front end to the text system. It displays the text, handles the user events that edit the text, and coordinates changes to the stored text required by the editing process. `NSTextView` implements methods that perform editing, manage the selection, and handle formatting attributes affecting the layout and display of the text.

`NSTextView` has a number of methods that control the editing behavior available to the user. For example, `NSTextView` allows you to grant or deny the user the ability to select or edit its text, using the `setSelectable:` and `setEditable:` methods. `NSTextView` also implements the distinction between plain and rich text defined by `NSText` with its `setRichText:` and `setImportsGraphics:` methods. See *Text System User Interface Layer Programming Guide for Cocoa* programming topic and the `NSTextView` and `NSText` class specifications for more information.

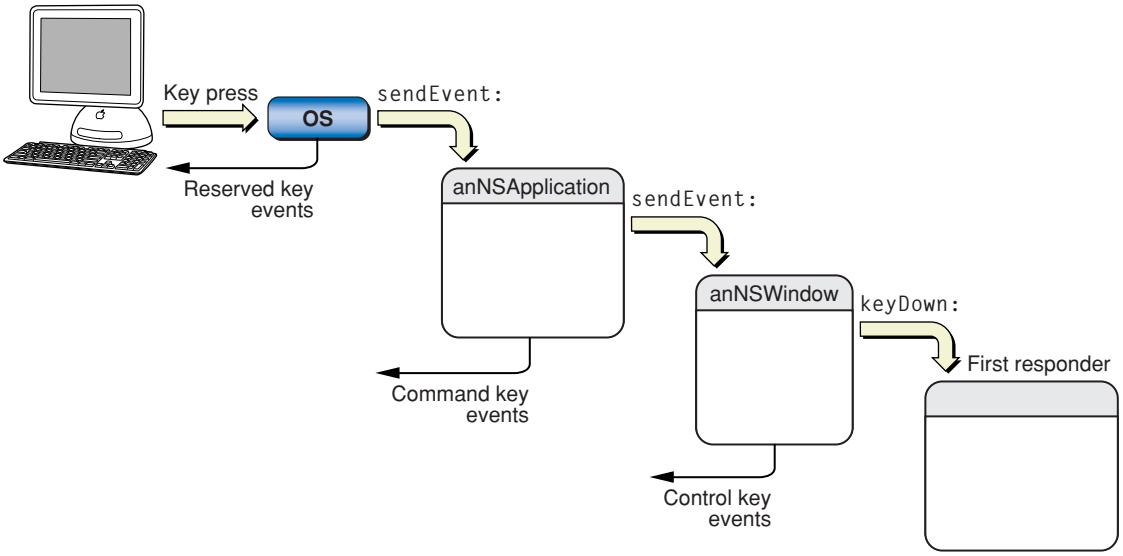
An editable text view can operate in either of two distinct editing modes: as a normal text editor or as a field editor. A field editor is a single text view instance shared among all the text fields belonging to a window in an application. This sharing results in a considerable performance gain because a text view is a heavyweight object. When a text field becomes the first responder, the window inserts the field editor in its place in the responder chain. A normal text editor accepts Tab and Return characters as input, whereas a field editor interprets Tab and Return as cues to end editing. The `NSTextView` method `setFieldEditor:` controls this behavior.

The Key-Input Message Sequence

When you want to modify the way in which Cocoa edits text, it's helpful to understand the message sequence that defines the editing mechanism, so you can select the most appropriate point at which to add your custom behavior.

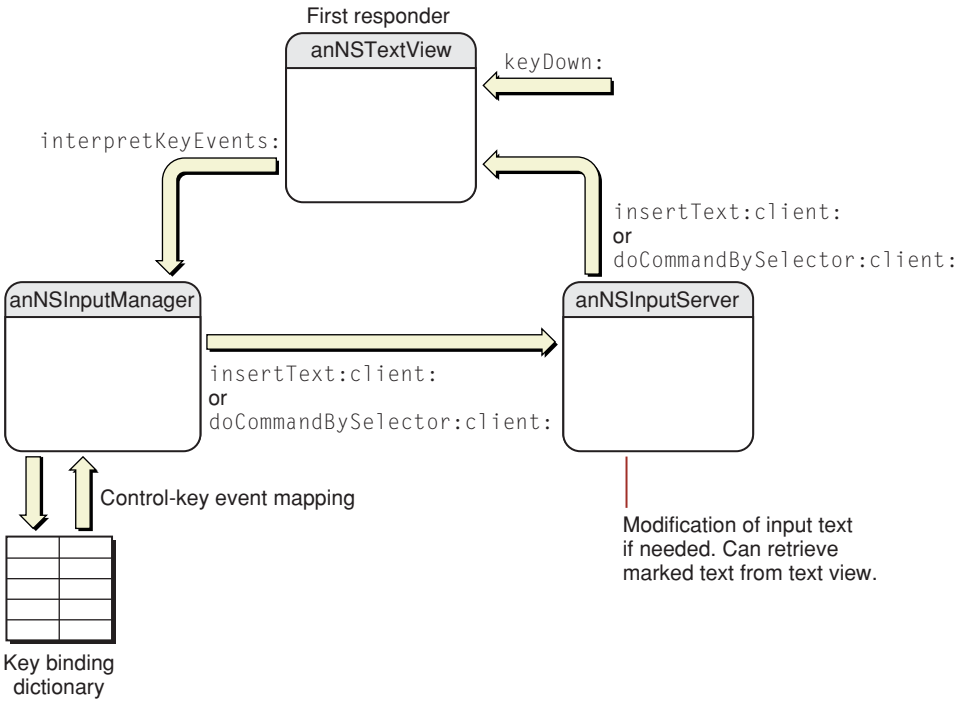
The message sequence invoked when a text view receives key events involves four methods declared by `NSResponder`. When the user presses a key, the operating system handles certain reserved key events and sends others to the `NSApplication` object, which handles Command-key events. The application object sends other key events to the key window, which handles Control-key events and sends other key events to the first responder. Figure 1 illustrates this sequence.

Figure 1 Key-event processing



If the first responder is a text view, the key event enters the text system. The key window sends the text view a `keyDown:` message with the event as its argument. The `keyDown:` method passes the event to `interpretKeyEvents:`, which sends the character input to the input manager for key binding and interpretation. In response, the input manager sends either `insertText:` or `doCommandBySelector:` to the text view. Figure 2 illustrates the sequence of text-input event processing.

Figure 2 Text-input key event processing



For more information about text-input key event processing, see *Text Input Management* and "Text System Defaults and Key Bindings."

When the text view has enough information to specify an actual change to its text, it sends an editing message to its `NSTextStorage` object to effect the change. The methods that change character and attribute information in the text storage object are declared in the `NSTextStorage` superclass `NSMutableAttributedString`, and they depend on the two primitive methods `replaceCharactersInRange:withString:` and `setAttributes:range:`. The text storage object then informs its layout managers of the change to initiate glyph generation and layout when necessary, and it posts notifications and sends delegate messages before and after processing the edits. For more information about the interaction of text view, text storage, and layout manager objects, see*Text Layout Programming Guide for Cocoa*.

Text View Delegation

Delegation provides a powerful mechanism for modifying editing behavior because you can implement methods in the delegate that can then perform editing commands in place of the text view, a technique called delegation of implementation. `NSTextView` gives its delegate this opportunity to handle a command by sending it a `textView:doCommandBySelector:` message whenever it receives a `doCommandBySelector:` message from the input manager. If the delegate implements this method and returns `YES`, the text view does nothing further; if the delegate returns `NO`, the text view must try to perform the command itself.

Before a text view makes any change to its text, it sends its delegate a `textView:shouldChangeTextInRange:replacementString:` message, which returns a Boolean value. (As with all delegate messages, it sends the message only if the delegate implements the method.) This mechanism provides the delegate with an opportunity to control all editing of the character and attribute data in the text storage object associated with the text view.

For more information about text view delegation, see "Delegate Messages and Notifications" in *Text Editing Programming Guide for Cocoa*.

Subclassing

Using `NSTextView` directly is the easiest way to interact with the text system, and its delegate mechanism provides an extremely flexible way to modify its behavior. In cases where delegation does not provide required behavior, you can subclass `NSTextView`. See "Subclassing NSTextView" (page 13) for more information on how to implement a subclass of `NSTextView`.

Note: To modify editing behavior, your first resort should be to notification or delegation, rather than subclassing. It may be tempting to start by trying to subclass `NSTextView` and override `keyDown:`, but that's usually not appropriate, unless you really need to deal with raw key events before input management or key binding. In most cases it's more appropriate to work with one of the text view delegate methods or with text view notifications.

A strategy even more complicated than subclassing `NSTextView` is to create your own custom text view object. If you need more sophisticated text handling than `NSTextView` provides, for example in a word processing application, it is possible to create a text view by subclassing `NSView`, implementing the `NSTextInput` protocol, and interacting directly with the input management system. For information on creating custom text views, see "Creating Custom Views." Also refer to the reference documentation for `NSText`, `NSTextView`, `NSView`, and the `NSTextInput` protocol.

Document Revision History

This table describes the changes to *Text Views*.

Date	Notes
2004-02-09	Revised introduction and added an index.
2003-08-04	First content added: introduction and links to existing Cocoa documents describing text views.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.

Index