Text Editing Programming Guide for Cocoa

Cocoa > Text & Fonts

2008-02-08

Apple Inc.
© 2003, 2008 Apple Inc.
All rights reserved.

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

The Apple logo is a trademark of Apple Inc.

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

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

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

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

Apple, the Apple logo, Cocoa, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada.

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

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

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

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

Contents

Introduction to Text Editing 7

Who Should Read This Document 7
Organization of This Document 7
See Also 8

Overview of Text Editing 9

The Editing Environment 9
The Key-Input Message Sequence 9
Text View Delegation 11
Subclassing 12

Synchronizing Editing 13

Batch-Editing Mode 13
Forcing the End of Editing 14

Intercepting Key Events 17

Delegate Messages and Notifications 19

Subclassing NSTextView 21

Updating State 21
Custom Import Types 21
Altering Selection Behavior 22
Preparing to Change Text 22
Notifying About Changes to the Text 22
Smart Insert and Delete 23

Setting Focus and Selection Programmatically 25

Working With the Field Editor 27

What is the Field Editor? 27
How the Field Editor Works 27
Using Delegation and Notification With the Field Editor 28
    Changing Default Behavior 28
    Getting Newlines into an NSTextField Object 29
Using a Custom Field Editor 29

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Why Use a Custom Field Editor? 29
How to Substitute a Custom Field Editor 30
Field Editor–Related Methods 31

Handling Drops in a Text Field 35

Document Revision History 37

Index 39

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Figures, Tables, and Listings

Overview of Text Editing 9

Figure 1 Key-event processing 10
Figure 2 Text-input key event processing 11

Synchronizing Editing 13

Listing 1 Forcing layout 13
Listing 2 Forcing the end of editing 14

Delegate Messages and Notifications 19

Figure 1 Delegate of an NSTextView object 19

Working With the Field Editor 27

Figure 1 The field editor 28
Table 1 NSWindow field editor–related methods 31
Table 2 NSTextFieldCell field editor–related method 31
Table 3 NSCell field editor–related methods 31
Table 4 NSControl field editor–related methods 32
Table 5 NSResponder field editor–related methods 32
Table 6 NSText field editor–related methods 33
Listing 1 Forcing the field editor to enter a newline character 29
Listing 2 Substituting a custom field editor 30

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Introduction to Text Editing

Text Editing describes ways in which you can control the behavior of the Cocoa text system as it performs text editing. Text editing is the modification of text characters or attributes by interacting with a text view object, either programmatically or by direct user action.

Who Should Read This Document

You should read this programming topic if you need to understand how text editing works and how to modify that behavior.

To understand the information in this programming topic you should have prior general knowledge of the text system's capabilities and architecture, as well as basic Cocoa programming conventions.

Organization of This Document

This programming topic contains the following articles:

- "Overview of Text Editing" (page 9) 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.
- "Synchronizing Editing" (page 13) explains the batch editing concept and shows how to force the end of editing, which sends notifications and leaves the text backing store in a consistent state.
- "Intercepting Key Events" (page 17) explains how to catch key events received by an `NSTextView` object so that you can modify their effect.
- "Delegate Messages and Notifications" (page 19) describes the messages the text view delegate and registered observers of the text system can receive.
- "Subclassing NSTextView" (page 21) explains the responsibilities an `NSTextView` subclass must fulfill to interact successfully with the text system.
- "Setting Focus and Selection Programmatically" (page 25) explains how to make a text view the first responder and how to manipulate the selection programmatically.
- "Working With the Field Editor" (page 27) explains how the text system uses the field editor and how you can modify that behavior.
- "Handling Drops in a Text Field" (page 35) explains how to add drag-and-drop support to a text field, which includes providing a custom field editor for the text view.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

See Also

- *Text System User Interface Layer Programming Guide for Cocoa* provides more information about the primary interface to the text system, the `NSTextView` class.

The other programming topics in the text system area also have information related to text editing. In addition, please refer to the Cocoa text-related code samples on the Apple Developer Connection website and the Application Kit examples installed with Xcode Tools.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

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.

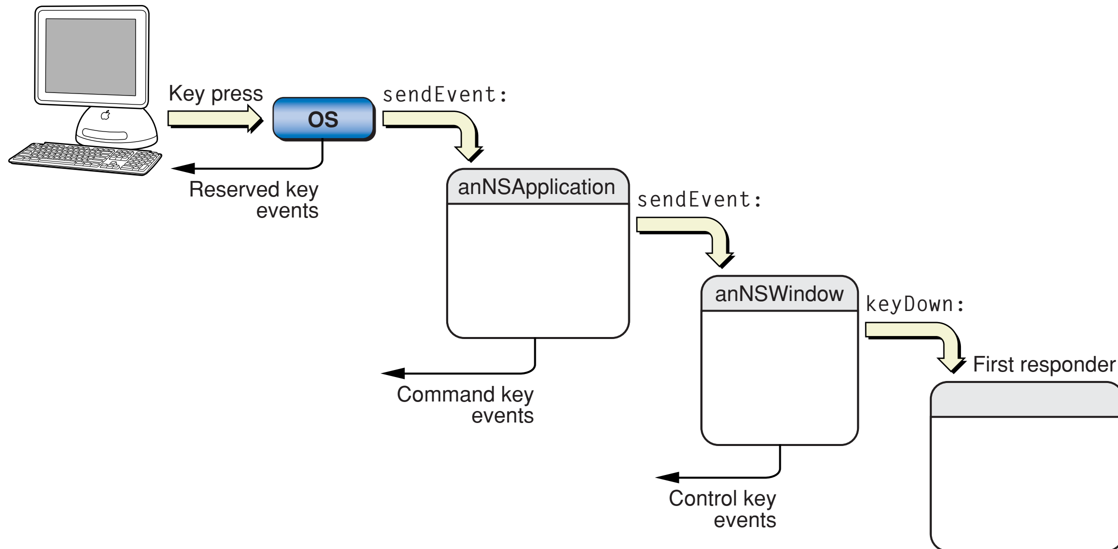
The Key-Input Message Sequence

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.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

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.

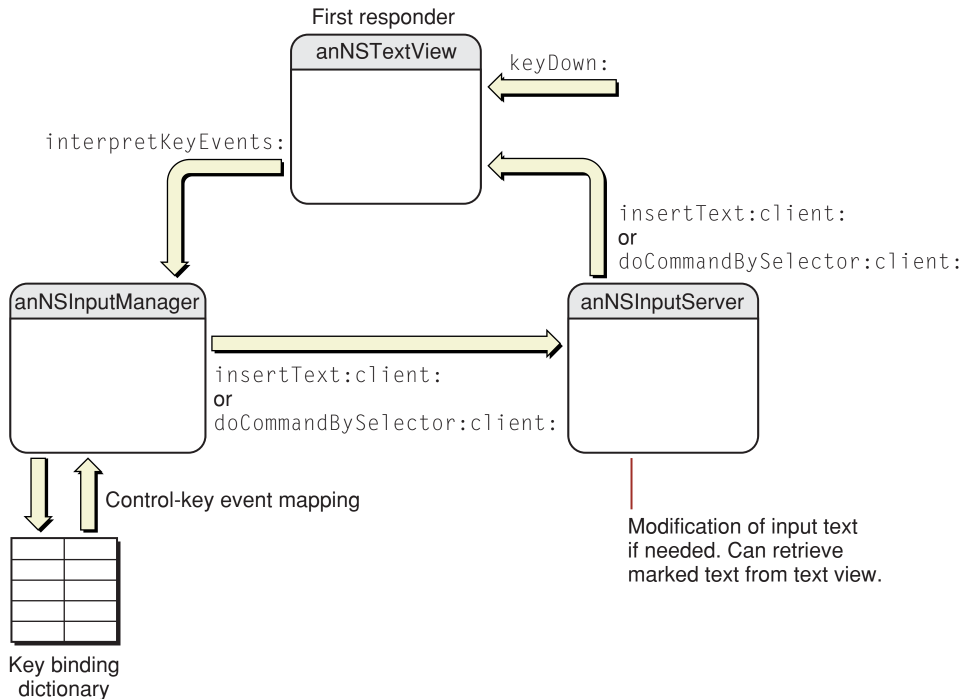
Figure 1 Key-event processing



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.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

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.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

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" (page 19).

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 21) 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.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Synchronizing Editing

The editing process involves careful synchronization of the complex interaction of various objects. The text system coordinates event processing, data modification, responder chain management, glyph generation, and layout to maintain consistency in the text data model.

The system provides a rich set of notifications to delegates and observers to enable your code to interact with this logic, as described in "Delegate Messages and Notifications" (page 19).

Batch-Editing Mode

If your code needs to modify the text backing store directly, you should bracket the changes between the `NSTextStorage` methods `beginEditing` and `endEditing`. Although this bracketing is not strictly necessary, it's good practice, and it's important for efficiency if you're making multiple changes in succession. `NSTextView` uses the `beginEditing` and `endEditing` methods to synchronize its editing activity, and you can use the methods directly to control the timing of notifications to delegates, observers, and associated layout managers. When the `NSTextStorage` object is in batch-editing mode, it refrains from informing its layout managers of any editing changes until it receives the `endEditing` message.

The "beginning of editing" means that a series of modifications to the text backing store (`NSTextStorage` for text views and cell values for cells) is about to occur. Bracketing editing between `beginEditing` and `endEditing` locks down the text storage to ensure that text modifications are atomic transactions.

The "end of editing" means that the backing store is in a consistent state after modification. In cells (such as `NSTextFieldCell` objects, which control text editing in text fields), the end of editing coincides with the field editor resigning first responder status, which triggers synchronization of the contents of the field editor and its parent cell.

In addition, the text view sends `NSTextDidEndEditingNotification`when it completes modifying its backing store, regardless of its first responder status. For example, it sends out this notification when the Replace All button is clicked in the Find window, even if the text view is not the first responder.

Important: Calling any of the layout manager's layout-causing methods between `beginEditing` and `endEditing` messages raises an exception. The `NSLayoutManager` reference documentation and the `NSLayoutManager.h` header file indicate which methods cause layout.

Listing 1 illustrates a situation in which the `NSTextView` method `scrollRangeToVisible:` forces layout to occur and raises an exception.

Listing 1 Forcing layout

```
[[myTextView textStorage] beginEditing];
[[myTextView textStorage] replaceCharactersInRange:NSMakeRange(0,0)
        withString:@"Hello to you!"];
[myTextView scrollrangeToVisible:NSMakeRange(0,13)]; //BOOM
```


2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

```
[[myTextView textStorage] endEditing];
```

Scrolling a character range into visibility requires layout to be complete through that range so the text view can know where the range is located. But in Listing 1, the text storage is in batch-editing mode. It is in an inconsistent state, so the layout manager has no way to do layout at this time. Moving the `scrollRangeToVisible:` call after `endEditing` would solve the problem.

There are additional actions that you should take if you implement new user actions in a text view, such as a menu action or key binding method that changes the text. For example, you can modify the selected range of characters using the `NSText` method `setSelectedRange`, depending on the type of change performed by the command, using the results of the `NSTextView` methods `rangeForUserTextChange`, `rangeForUserCharacterAttributeChange`, or `rangeForUserParagraphAttributeChange`. For example, `rangeForUserParagraphAttributeChange` returns the entire paragraph containing the original selection—that is the range affected if your action modifies paragraph attributes. Also, you should call `textView:shouldChangeTextInRange:replacementString:` before you make the change and `didChangeText` afterwards. These actions ensure that the correct text gets changed and the system sends the correct notifications and delegate messages to the text view's delegate. See "Subclassing NSTextView" (page 21) for more information.

Forcing the End of Editing

There may be situations in which you need to force the text system to end editing programmatically so you can take some action dependent on notifications being sent. In such a case, you don't need to modify the editing mechanism but simply stimulate its normal behavior.

To force the end of editing in a text view, which subsequently sends a `textDidEndEditing:` notification message to its delegate, you can observe the window's `NSWindowDidResignKey` notification. Then, in the observer method, send `makeFirstResponder:` to the window to finish any editing in progress while the window was active. Otherwise, the control that is currently being edited remains the first responder of the window and does not end editing.

Listing 2 presents an implementation of the `textDidEndEditing:` delegate method that ends editing in an `NSTableView` subclass. By default, when the user is editing a cell in a table view and presses Tab or Return, the field editor ends editing in the current cell and begins editing the next cell. In this case, you want to end editing altogether if the user presses Return. This method distinguishes which key the user pressed; for a Tab it does the normal behavior, and for Return it forces the end of editing completely by making the window first responder.

Listing 2 Forcing the end of editing

```
- (void)textDidEndEditing:(NSNotification *)notification {
    if([[[notification userInfo] valueForKey:@"NSTextMovement"] intValue] ==
                        NSReturnTextMovement) {
    NSMutableDictionary *newUserInfo;
    newUserInfo = [[NSMutableDictionary alloc]
                            initWithDictionary:[notification userInfo]];
    [newUserInfo setObject:[NSNumber numberWithInt:NSIllegalTextMovement]
                                forKey:@"NSTextMovement"];
    notification = [NSNotification notificationWithName:[notification name]
                                object:[notification object]
                                userInfo:newUserInfo];
    [super textDidEndEditing:notification];
```


2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

```
    [newUserInfo release];
    [[self window] makeFirstResponder:self];
} else {
    [super textDidEndEditing:notification];
    }
}
```


2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Intercepting Key Events

This article explains how to catch key events received by a text view so that you can modify the result. It also explains the message sequence that occurs when a text view receives a key event.

You need to intercept key events, for example, if you want users to be able to insert a line-break character in a text field. By default, text fields hold only one line of text. Pressing either Enter or Return causes the text field to end editing and send its action message to its target, so you would need to modify the behavior.

You may also wish to intercept key events in a text view to do something different from simply entering characters in the text being displayed by the view, such as changing the contents of an in-memory buffer.

In both circumstances you need to deal with the text view object, which is obvious for the text view case but less so for a text field. Editing in a text field is performed by an `NSTextView` object, called the field editor, shared by all the text fields belonging to a window.

When a text view receives a key event, it 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, depending on whether the key event represents text to be inserted or a command to perform. (The message sequence invoked when a text view receives key events is described in more detail in "The Key-Input Message Sequence" (page 9).)

With the standard key bindings, an Enter or Return character causes the text view to receive `doCommandBySelector:` with a selector of `insertNewline:`, which can have one of two results. If the text view is not a field editor, the text view's `insertText:` method inserts a line-break character. If the text view is a field editor, as when the user is editing a text field, the text view ends editing instead. You can cause a text view to behave in either way by calling `setFieldEditor:`.

Although you could alter the text view's behavior by subclassing the text view and overriding `insertText:` and `doCommandBySelector:`, a better solution is to handle the event in the text view's delegate. The delegate can take control over user changes to text by implementing the `textView:shouldChangeTextInRange:replacementString:` method.

To handle keystrokes that don't insert text, the delegate can implement the `textView:doCommandBySelector:` method.

To distinguish between Enter and Return, for example, the delegate can test the selector passed with `doCommandBySelector:`. If it is `@selector(insertNewline:)`, you can send `currentEvent` to `NSApp` to make sure the event is a key event and, if so, which key was pressed.

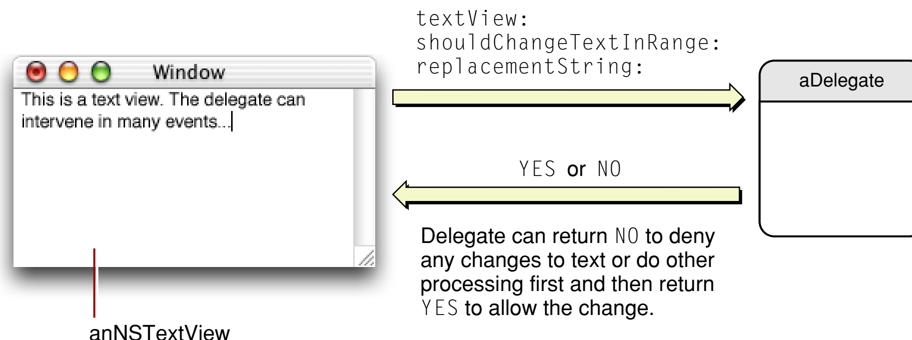
2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Delegate Messages and Notifications

An `NSTextView` object can have a delegate that it informs of certain actions or pending changes to the state of the text. The delegate can be any object you choose, and one delegate can control multiple `NSTextView` objects (or multiple series of connected `NSTextView` objects). Figure 1 illustrates the activity of the delegate of an `NSTextView` object receiving the delegate message `textView:shouldChangeTextInRange:replacementString:`.

Figure 1 Delegate of an NSTextView object



The `NSText` and `NSTextView` class reference documentation describes the delegate messages the delegate can receive. The delegating object sends a message only if the delegate implements the method.

All `NSTextView` objects attached to the same `NSLayoutManager` share the same delegate: Setting the delegate of one such text view sets the delegate for all the others. Delegate messages pass the `id` of the sender as an argument.

> **Note:** For multiple `NSTextView` objects attached to the same `NSLayoutManager` object, the argument `id` is that of the notifying text view, which is the first `NSTextView` object for the shared `NSLayoutManager` object. This `NSTextView` object is responsible for posting notifications at the appropriate times.

The notifications posted by `NSTextView` are:

```
NSTextDidBeginEditingNotification
NSTextDidEndEditingNotification
NSTextDidChangeNotification
NSTextViewDidChangeSelectionNotification
NSTextViewWillChangeNotifyingTextViewNotification
```

It is particularly important for observers to register for the last of these notifications. If a new `NSTextView` object is added at the beginning of a series of connected `NSTextView` objects, it becomes the new notifying text view. It doesn't have access to which objects are observing its group of text objects, so it posts an

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

NSTextViewWillChangeNotifyingTextViewNotification, which allows all those observers to unregister themselves from the old notifying text view and reregister themselves with the new one. For more information, see the description for this notification in the NSTextView reference documentation.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

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" (page 19).

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.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

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" (page 25). 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 NSNotFoundif 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

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

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.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Setting Focus and Selection Programmatically

Usually the user clicks a view object in a window to set the focus, or first responder status, so that subsequent keyboard events go to that object initially. Likewise, the user usually creates a selection by dragging the mouse in a view. However, you can set both the focus and the selection programmatically.

For example, if you have a window that contains a text view, and you want that text view to become the first responder with the insertion point located at the beginning of any text currently in the text view, you need a reference to the window and the text view. If those references are `theWindow` and `theTextView`, respectively, you can use the following code to set the focus and the insertion point:

```
[theWindow makeFirstResponder: theTextView];
[theTextView setSelectedRange:NSMakeRange(0,0)];
```

The insertion point is simply a zero-length selection range.

Whether the selection was set programmatically or by the user, you can get the range of characters currently selected using the `selectedRange` method. `NSTextView` indicates its selection by applying a special set of attributes to it. The `selectedTextAttributes` method returns these attributes, and `setSelectedTextAttributes:` sets them.

While changing the selection in response to user input, an `NSTextView` object invokes its `setSelectedRange:affinity:stillSelecting:` method. The first argument is the range to select. The second, called the selection affinity, determines which glyph the insertion point displays near when the two glyphs defining the selected range are not adjacent. It's typically used where the selected lines wrap to place the insertion point at the end of one line or the beginning of the following line. You can get the selection affinity currently in effect using the `selectionAffinity` method. The last argument indicates whether the selection is still in the process of changing; the delegate and any observers aren't notified of the change in the selection until the method is invoked with `NO` for this argument.

Another factor affecting selection behavior is the selection granularity: whether characters, words, or whole paragraphs are being selected. This is usually determined by number of initial mouse clicks; for example, a double click initiates word-level selection. `NSTextView` decides how much to change the selection during input tracking using its `selectionRangeForProposedRange:granularity:` method.

An additional aspect of selection, related to input management, is the range of marked text. As the input manager interprets keyboard input, it can mark incomplete input in a special way. The text view displays this marked text differently from the selection, using temporary attributes that affect only display, not layout or storage. For example, `NSTextView` uses marked text to display a combination key, such as Option-E, which places an acute accent character above the character entered next. When the user types Option-E, the text view displays an acute accent in a yellow highlight box, indicating that it is marked text, rather than final input. When the user types the next character, the text view displays it as a single accented character, and the marked text highlight disappears. The `markedRange` method returns the range of any marked text, and `markedTextAttributes` returns the attributes used to highlight the marked text. You can change these attributes using `setMarkedTextAttributes:`.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Working With the Field Editor

This article explains how the Cocoa text system uses the field editor and how you can modify that behavior. In most cases, you don't need to be concerned about the field editor because Cocoa handles its operation automatically, behind the scenes. However, it's good to know of its existence, and it's possible that in some circumstances you could want to change its behavior.

What is the Field Editor?

The field editor is a single `NSTextView` object that is shared among all the controls in a single window, including buttons, table views, and text fields. This text view object provides text entry and editing services for the currently active control. When the user clicks in a text field, for example, the field editor begins handling keystroke events and text display for that field.

The field editor provides significant optimization. Because only one control can be active at a time, the system needs only one `NSTextView` instance per window to be the field editor. This results in a performance gain because `NSTextView` is a relatively heavyweight object. Note, however, that you can substitute custom field editors, as described in "Using a Custom Field Editor" (page 29), in which case a window could have more than one field editor.

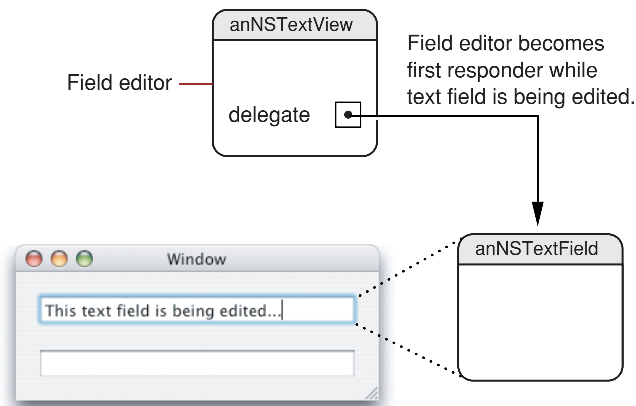
How the Field Editor Works

The text system automatically instantiates the field editor from the `NSTextView` class when the user begins editing text of an `NSControl` object such as a text field. While it is editing, the system inserts the field editor into the responder chain as first responder, so it receives keystroke events in place of the text field. This mechanism can be confusing if you're not familiar with the workings of the field editor, because the `NSWindow` method `firstResponder` returns the field editor, which is not visible, rather than the on-screen object that currently has keyboard focus.

The field editor designates the current text field as its delegate, which enables 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 1 illustrates the field editor in relation to the text field it is editing.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Figure 1 The field editor



Among its other duties, the field editor maintains the selection for the text fields it edits. Therefore, a text field that's not being edited does not have a selection (unless you cache it).

A field editor is defined by its treatment of certain characters during text input, which is different from an ordinary text view. An ordinary text view inserts a newline when the user types Return or Enter, it inserts a tab character when the user types Tab, and it ignores a Shift-Tab. In contrast, a field editor interprets these characters as cues to end editing and resign first responder status, shifting focus to the next object in the key view loop (or in the case of a Shift-Tab, the previous key view).

The end of editing triggers synchronization of the contents of the field editor and the `NSTextFieldCell` object that controls editing in the text field. At that point Cocoa detaches the field editor from the text field and restores the text field to its original place in the view hierarchy.

Using Delegation and Notification With the Field Editor

You can control the editing behavior of text fields by interacting with the field editor through delegation and notification. Because the field editor automatically designates any text field it is editing as its delegate, you can often encapsulate special editing behavior for a text field with the text field itself.

Changing Default Behavior

It's straightforward to change the default behavior of the field editor by implementing delegate methods. For example, the delegate can change the behavior that occurs when the user presses Return while editing a text view. By default, that action ends editing and selects the next control in the key view loop. If, for example, you want pressing Return to end editing but not select the next control, you can implement the `textDidEndEditing:` delegate method in the text field. The field editor automatically calls this method, if the delegate implements it, and passes `NSTextDidEndEditingNotification`. The implementation can examine this notification to discover the event that ended editing and respond appropriately.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Getting Newlines into an NSTextField Object

Users can easily put newline characters into a text field by pressing Option-Return or Option-Enter. However, there may be situations in which you want to allow users to enter newlines without taking any special action, and you can do so by implementing a delegate method.

The easiest approach is to call `setFieldEditor:NO` on the window's field editor. But, of course, this approach changes the behavior of the field editor for all controls. It looks promising to use the `NSControl` delegate message `control:textShouldBeginEditing:`, which is sent to a text view's delegate when the user enters a character into the text field. Because it passes references to both the text view and the field editor, you could test to see if the text view is one into which you want to enter newlines, then simply send `setFieldEditor:NO` to the field editor. However, this method is not called until after the user has entered one character into the text field, and if that character is a newline, it is rejected.

A better method is to implement another `NSControl` delegate method, `control:textView:doCommandBySelector:`, that enables the text field's delegate to check whether the user is attempting to insert a newline character and, if so, force to field editor to insert it. The implementation could appear as shown in Listing 1.

Listing 1 Forcing the field editor to enter a newline character

```
- (BOOL)control:(NSControl *)control textView:(NSTextView *)fieldEditor
                                doCommandBySelector:(SEL)commandSelector {
     BOOL retval = NO;
     if (commandSelector == @selector(insertNewline:)) {
         retval = YES;
         [fieldEditor insertNewlineIgnoringFieldEditor:nil];
     }
     return retval;
}
```

This method returns `YES` to indicate that it handles this particular command and `NO` for other commands that it doesn't handle. This approach has the advantage that it doesn't change the setup of the field editor but handles just the special case of interest. Because the delegate message includes a reference to the control being edited, you could add a check to restrict the behavior to a particular class, such as `NSTextField`, or an individual subclass.

Using a Custom Field Editor

To customize behavior in ways that go beyond what the delegate can do, you need to define a subclass of `NSTextView` that incorporates your specialized behavior and substitute it for the window's default field editor.

Why Use a Custom Field Editor?

It's not necessary to use a custom field editor if you simply need to validate, interpret, format, or even edit the contents of text fields as the user types. You can attach an `NSFormatter`, such as `NSNumberFormatter`, `NSDateFormatter`, or a custom formatter, for that purpose. See *Data Formatting Programming Guide for*

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Cocoa for more information about using formatters. Delegation and notification also provide many opportunities for you to intervene, as described previously in "Using Delegation and Notification With the Field Editor" (page 28).

A secure text field is an example of truly specialized handling of data that goes beyond what can be reasonably handled by formatters or delegates. A secure text field must accept text data entered by the user and validate the entries, which are easily done with a regular text field and a formatter. But it must display some bogus characters to keep the real data secret while it preserves the real data for an authentication process or other purpose. Moreover, a secure text field must keep its data safe from unauthorized access by disabling features such as copy and cut, and possibly encrypting the data. To implement these specialized requirements, it is easiest to deploy a custom field editor. In fact, Cocoa implements a custom field editor in the `NSSecureTextField` class.

As another example, you must use a custom field editor to support drag and drop in a text field that has keyboard focus. You can add support for drag and drop in a subclass of `NSTextField` itself, and it works fine as long as the text field is not currently being edited. During editing, however, the field editor becomes the first responder, so it is the target of a drop in the text field. Therefore, to handle drag and drop while the text field is being edited, you must implement support in a subclass of the field editor. This procedure is described in "Handling Drops in a Text Field." (page 35)

Any situation requiring unusual processing of data entered into a text field, or other individualized behavior not available through the standard Cocoa mechanisms, is a good candidate for a custom field editor.

How to Substitute a Custom Field Editor

You can substitute your custom field editor in place of the window's default version by implementing the `NSWindow` delegate method `windowWillReturnFieldEditor:toObject:`. You implement this method in the window's delegate, which could be, for example, the window controller object. The window sends this message to its delegate with itself and the object requesting the field editor as arguments. So, you can test the object and make substitution of your custom field editor dependent on the result. The window continues to use its default field editor for other controls.

For example, the implementation shown in Listing 2 tests whether or not the requesting object is an `NSTextField`, and, if it is, returns a custom field editor.

Listing 2 Substituting a custom field editor

```
(id)windowWillReturnFieldEditor:(NSWindow *)sender toObject:(id)anObject
{
    if ([anObject isKindOfClass:[NSTextField class]])
    {
        return [[[myCustomFieldEditor alloc] init] autorelease];
    }
    return nil;
}
```

If the requesting object is not a text field or subclass, the delegate method returns `nil` and the window uses its default field editor. This arrangement has the advantage that it does not instantiate the custom field editor unless it is needed. The custom field editor should be released in an appropriate place, such as the `dealloc` method of the window delegate object (if the field editor was never instantiated, the `release` message has no effect and is harmless).

You can find more information about subclassing `NSTextView` in "Subclassing NSTextView (page 21)."

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Field Editor–Related Methods

This section lists the Application Kit methods most directly related to the field editor. You can peruse these tables to understand where Cocoa provides opportunities for you to interact with the field editor. Refer to the Application Kit reference documentation for details. The `NSWindow` methods related to the field editor are listed in Table 1.

Table 1 NSWindow field editor–related methods

Method	Description
`fieldEditor: forObject:`	Returns the receiver's field editor, creating it if needed.
`endEditingFor:`	Forces the field editor to give up its first responder status and prepares it for its next assignment.
`windowWillReturnFieldEditor: toObject:`	Delegate method invoked when the field editor of sender is requested by an object. If the delegate's implementation of this method returns an object other than `nil`, `NSWindow` substitutes it for the field editor.

The `NSTextFieldCell` method related to the field editor is listed in Table 2.

Table 2 NSTextFieldCell field editor–related method

Method	Description
`setUpFieldEditorAttributes:`	You never invoke this method directly; by overriding it, however, you can customize or replace the field editor.

The `NSCell` methods related to the field editor are listed in Table 3.

Table 3 NSCell field editor–related methods

Method	Description
`selectWithFrame:inView: editor:delegate:start:length:`	Uses the field editor passed with the message to select text in a range.
`editWithFrame:inView:editor: delegate:event:`	Begins editing of the receiver's text using the field editor passed with the message.
`endEditing:`	Ends any editing of text, using the field editor passed with the message, begun with either of the other two `NSCell` field editor–related methods.

The `NSControl` methods related to the field editor are listed in Table 4. The `NSControl` delegate methods listed in Table 4 are control-specific versions of the delegate methods and notifications defined by `NSText`. The field editor, derived from `NSText`, initiates sending the delegate messages and notifications through its editing actions.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Table 4 NSControl field editor–related methods

Method	Description
`abortEditing`	Terminates and discards any editing of text displayed by the receiver and removes the field editor's delegate.
`currentEditor`	If the receiver is being edited, this method returns the field editor; otherwise, it returns `nil`.
`validateEditing`	Sets the object value of the text in a cell of the receiving control to the current contents of the cell's field editor.
`control:textShouldBeginEditing:`	Sent directly to the delegate when the user tries to enter a character in a cell of the control passed with the message.
`control:textShouldEndEditing:`	Sent directly to the delegate when the insertion point tries to leave a cell of the control that has been edited.
`controlTextDidBeginEditing:`	Sent by the default notification center to the delegate (and all observers of the notification) when a control begins editing text, passing `NSControlTextDidBeginEditingNotification`.
`controlTextDidChange:`	Sent by the default notification center to the delegate and observers when the text in the receiving control changes, passing `NSControlTextDidChangeNotification`.
`controlTextDidEndEditing:`	Sent by the default notification center to the delegate and observers when a control ends editing text, passing `NSControlTextDidEndEditingNotification`.

The `NSResponder` methods related to the field editor are listed in Table 5.

Table 5 NSResponder field editor–related methods

Method	Description
`insertBacktab:`	Implemented by subclasses to handle a "backward tab."
`insertNewlineIgnoringFieldEditor:`	Implemented by subclasses to insert a line-break character at the insertion point or selection.
`insertTabIgnoringFieldEditor:`	Implemented by subclasses to insert a tab character at the insertion point or selection.

The `NSText` methods related to the field editor are listed in Table 6.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Table 6 NSText field editor–related methods

Method	Description
`isFieldEditor`	Returns YES if the receiver interprets Tab, Shift-Tab, and Return (Enter) as cues to end editing and possibly to change the first responder; NO if it accepts them as text input.
`setFieldEditor:`	Controls whether the receiver interprets Tab, Shift-Tab, and Return (Enter) as cues to end editing and possibly to change the first responder.
`textDidBeginEditing:`	Informs the delegate that the user has begun changing text, passing `NSTextDidBeginEditingNotification`.
`textDidChange:`	Informs the delegate that the text object has changed its characters or formatting attributes, passing `NSTextDidChangeNotification`.
`textDidEndEditing:`	Informs the delegate that the text object has finished editing (that it has resigned first responder status), passing `NSTextDidEndEditingNotification`.
`textShouldBeginEditing:`	Invoked from a text object's implementation of `becomeFirstResponder`, this method requests permission to begin editing.
`textShouldEndEditing:`	Invoked from a text object's implementation of `resignFirstResponder`, this method requests permission to end editing.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Handling Drops in a Text Field

To handle drag and drop in a text field, you need to subclass `NSTextField` and add support for the operation. This works well as long as the text field is not currently being edited. To handle drag and drop while the text field is being edited, you must implement support in the field editor.

To provide a custom field editor for your text field (or any other control) you need to implement a method to respond to the `NSWindow` delegate message `windowWillReturnFieldEditor:toObject:` in the delegate of the window containing the text field you want to respond to drags. The client specified in the `toObject:` argument is the text field that is about to be edited, for which it uses the `NSTextView` object you return instead of the standard field editor.

`NSTextView` has support for drag and drop through the `NSDragging` category. However, an `NSTextView` object registers for draggable pasteboard types only if it is set up to handle rich text (see the `setRichText:` method) and allows attached files (see the `setImportsGraphics:` method). By default, `NSTextView` does not accept dragged files.

To support new data types 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 the superclass, and return the modified array. You must also override the appropriate methods of the `NSDraggingDestination` protocol to support importing those types. See that protocol reference for more information. Also see the *Drag and Drop Programming Topics for Cocoa* programming topic.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Document Revision History

This table describes the changes to *Text Editing Programming Guide for Cocoa*.

Date	Notes
2008-02-08	Corrected typographical errors.
2006-06-28	Made minor corrections to code snippets, fixed a grammatical error, and added a cross-reference.
2004-04-19	Added a new article, "Working With the Field Editor" (page 27).
2004-02-13	Rewrote introduction and added an index.
2003-05-07	First version.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

Index

A

abortEditing method 32
acceptableDragTypes method 21, 35

B

beginEditing method 13
beginning of editing 13

C

control points of editing mechanism 9
control:textShouldBeginEditing: method 32
control:textShouldEndEditing: method 32
controlTextDidBeginEditing: method 32
controlTextDidChange: method 32
controlTextDidEndEditing: method 32
currentEditor method 32

D

delegation 11, 13, 19, 28
deletion, smart 23
didChangeText method 14, 22, 23
doCommandBySelector: method 10, 11, 17
drag and drop 35
  field editors and 30
  text fields and 30

E

editing
  batch mode 13
  customizing behavior 9, 11, 21
  environment 9
  message sequence 9
  modes 9
  synchronizing 13
editWithFrame:inView:editor: delegate:event: method 31
end of editing
  caused by user 17
  defined 13
  endEditing: method and 31
  field editor and 13, 28
  forcing programatically 14
endEditing method 13
endEditing: method 31
endEditingFor: method 31

F

field editors
  custom 29
  defined 27
  drag and drop handling 35
  end of editing and 13
  intercepting key events 17
  methods related to 31
  notifications and 28
  operation 27
  text fields and 9
  using delegation and notification with 28
fieldEditor:forObject: method 31
first responder
  delegate methods and 23
  end of editing and 14
  field editor and 13, 30
  keyboard focus and 25
firstResponder method 27
focus, setting programmatically 25
Font window 21

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

I

import types 21
input management 25
insertBacktab: method 32
insertion point 25
insertNewline: method 17
insertNewlineIgnoringFieldEditor: method 32
insertTabIgnoringFieldEditor: method 32
insertText: method 10, 17
interpretKeyEvents: method 10
isFieldEditor method 33

K

key events
  intercepting 17
  processing by a text view 10
key window 10
key-input message sequence 9
keyDown: method 10, 12, 21

L

layout manager 13, 19

M

makeFirstResponder: method 14
marked text 25
markedRange method 25
markedTextAttributes method 25
message sequence of editing mechanism 9

N

newline characters
  textfields and 29
notifications
  field editors and 28
  of text changes 22
  posted by NSTextView 19
NSApp object 17
NSApplication class 10
NSCell class 31
NSControl class 27, 29, 31
NSDragging category 35
NSDraggingDestination protocol 35
NSFormatter class 29
NSLayoutManager class 19
NSNotFound constant 22
NSResponder class 10, 32
NSSecureTextField class 30
NSTableView class 14
NSText class 31, 32
NSTextDidBeginEditingNotification 19, 22
NSTextDidChangeNotification 19, 23
NSTextDidEndEditingNotification 13, 19
NSTextField class 29
NSTextField class, subclassing 35
NSTextFieldCell class 31
NSTextInput protocol 12
NSTextStorage class 11
NSTextView class
  as field editor 27
  delegate of 11, 19
  features 9
  subclassing 12, 21
NSTextViewDidChangeSelectionNotification 19
NSTextViewWillChangeNotifyingTextViewNotification 19
NSWindow class 27, 30, 31, 35
NSWindowDidResignKey notification 14

P

paragraph attributes 14
pasteboard 21

R

rangeForUserCharacterAttributeChange method 14, 22
rangeForUserParagraphAttributeChange method 14, 22
rangeForUserTextChange method 14, 22
readablePasteboardTypes method 21
readSelectionFromPasteboard:type: method 22
replaceCharactersInRange:withString: method 11
responder chain 27
ruler 21

S

scrollRangeToVisible: method 13

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.

selectedRange method 25
selectedTextAttributes method 25
selection
  affinity 25
  altering behavior 22
  granularity 22, 25
  setting programmatically 25
selectionAffinity method 25
selectionRangeForProposedRange:granularity: method 22, 25
selectWithFrame:inView: editor:delegate:start:length: method 31
setAttributes:range: method 11
setEditable: method 9
setFieldEditor: method 9, 17, 33
setImportsGraphics: method 9, 35
setMarkedTextAttributes: method 25
setRichText: method 9, 35
setSelectable: method 9
setSelectedRange method 14
setSelectedRange:affinity:stillSelecting: method 25
setSelectedTextAttributes: method 25
setSmartInsertDeleteEnabled: method 23
setUpFieldEditorAttributes: method 31
shouldChangeTextInRange:replacementString: method 22, 23
smart insertion and deletion 23
smartDeleteRangeForProposedRange: method 23
smartInsertForString:replacingRange:beforeString: afterString: method 23

T

table view, editing a cell 14
text attributes 21
text delegates 11, 22
text fields
  newline characters and 29
  secure 30
text ranges, modifying for changes 22
text storage 11
text views
  creating your own 12
  defined 9
textDidBeginEditing: method 23, 33
textDidChange: method 23, 33
textDidEndEditing: method 14, 33
textShouldBeginEditing: method 22, 23, 33
textShouldEndEditing: method 33
textView:doCommandBySelector: method 11, 17
textView:shouldChangeTextInRange: replacementString: method 12, 14, 17, 19
textView:willChangeSelectionFromCharacterRange: toCharacterRange: method 22

U

updateDragTypeRegistration method 21
updateFontPanel method 21
updateRuler method 21

V

validateEditing method 32

W

windowWillReturnFieldEditor: toObject: method 31
windowWillReturnFieldEditor:toObject: method 35
writablePasteboardTypes method 21
writeSelectionToPasteboard:type: method 22

2008-02-08 | © 2003, 2008 Apple Inc. All Rights Reserved.