NSTextView

Inherits From: NSText : NSView : NSResponder : NSObject

Conforms To: NSTextInput
NSChangeSpelling (NSText)
NSIgnoreMisspelledWords (NSText)
NSCoding (NSResponder) — *Note: NSTextView doesn't implement this protocol*
NSObject (NSObject)

Declared In: AppKit/NSTextView.h

Class at a Glance

Purpose

NSTextView is the front-end component of NeXT's extended text system. It displays and manipulates text laid out in an area defined by an NSTextContainer, and adds many features to those defined by its superclass, NSText.

Principal Attributes

- Supports rich text and graphics
- Works with the Font Panel and menu
- Provides delegation and notification
- Works with the pasteboard
- Supports input management and key bindings
- Works with rulers
- Works with the Services facility
- Works with spell-checking services

Creation

Interface Builder

– initWithFrame:	Creates an NSTextView along with all its supporting objects.
– initWithFrame:textContainer:	Designated initializer.

Commonly Used Methods

The methods most commonly used with NSTextView objects are declared in NSText, the superclass. These methods provide access to the other major components of the text system:

– textStorage	Returns the associated NSTextStorage object.
– textContainer	Returns the associated NSTextContainer object.
– layoutManager	Returns the associated NSLayoutManager object.

Class Description

NSTextView is the front-end class to NeXT's extended text-handling system. It draws the text managed by the back-end components and handles user events to select and modify its text. NSTextView is the principal means to obtain a text object that caters to almost all needs for displaying and managing text at the user interface level. While NSTextView is a subclass of NSText—which declares the most general OpenStep interface to the text system—NSTextView adds several major features over and above the capabilities of NSText.

One of the design goals of NSTextView is to provide a comprehensive set of text-handling features so that you should rarely need to create a subclass. In its standard incarnation, NSTextView creates the requisite group of objects that support the text handling system—NSTextContainer, NSLayoutManager, and NSTextStorage objects. Refer to "The OPENSTEP Text System" for a comprehensive overview of the components of the text system. Here are the major features that NSTextView adds to those of NSText:

Rulers. NSTextView works with the NSRulerView class to let users control paragraph formatting, in addition to using commands in the Format Text menu provided by Interface Builder.

Input management and key binding. Certain key combinations are bound to specific NSTextView methods so that the user can move the insertion point, for example, without using the mouse.

Marked text attributes. NSTextView defines a set of text attributes that support special display characteristics during input management. Marked text attributes only affect visual aspects of text—color, underline, and so on—they don't include any attributes that would change the layout of text.

File and graphic attachments. The extended text system provides programmatic access to text attachments as instances of NSTextAttachment, through the NSTextView and NSTextStorage classes.

Delegate messages and notifications. NSTextView adds several delegate messages and notifications to those used by NSText. The delegate and observers of an NSTextView can receive any of the messages or notifications declared by both classes.

Creating NSTextView Objects

The easiest way to add an NSTextView to your application is through Interface Builder. Interface Builder's Data Views palette supplies a specially configured NSScrollView object that contains an NSTextView object as its document view. This NSTextView is configured to work with the NSScrollView and other user-interface controls such as a ruler, the Font menu, the Edit menu, and so on.

Interface Builder also offers other objects—of the NSTextField and NSForm classes—that make use of NSTextView objects for their text-editing facilities. In fact, all NSTextFields and NSForms within the same window share the same NSTextView object (known as the *field editor*), thus reducing the memory demands of an application. If your application requires stand-alone or grouped text fields that support editing (and all the other facilities provided by the NSTextView class), these are the classes to use.

You can also create NSTextView objects programmatically, using either of the methods **initWithFrame:textContainer:** (the designated initializer), or **initWithFrame:**. The **initWithFrame:** method is the simplest way to obtain an NSTextView object—it creates all the other components of the text-handling system for you and releases them when you're done. If you use **initWithFrame:textContainer:**, you must construct (and release) the other components yourself. See the "The OPENSTEP Text System" for more information.

Configuring Editing Behavior

Like NSText, NSTextView allows you to grant or deny the user the ability to select or edit its text, using the **setSelectable:** and **setEditable:** methods. These methods only affect what the user can do; you can still make changes to the NSTextView programmatically. An editable text view can behave as a normal text editor, accepting Tab and Return characters, or as a field editor, interpreting tabs and returns as cues to end editing. The **setFieldEditor:** method controls this behavior. NSTextView also implements the distinction between plain and rich text defined by NSText with its **setRichText:** and **setImportsGraphics:** methods. See the NSText class specification for more information on these various distinctions.

Attachments

While NSText leaves open the nature of imported graphics and other attachments, NSTextView explicitly uses NSTextAttachment objects, which contain NSFileWrappers that represent the attached files. NSTextView declares several delegate methods that let you handle user actions on an attachment's image or icon. **textView:clickedOnCell:inRect:** and **textView:doubleClickedOnCell:inRect:** let the delegate take action on mouse clicks, and **textView:draggedCell:inRect:event:** lets the delegate initiate a dragging session for the attachment. See the NSTextAttachment, NSTextAttachmentCell, and NSFileWrapper class and protocol specifications for more information on working with attachments.

Input Management

NSTextView uses an input manager to turn basic character information into text and commands. It passes uninterpreted keyboard input to the input manager, which examines the characters generated and sends messages to the NSTextView based on those characters. If the typed characters are interpreted as text to input, the input manager sends the text view an **insertText:** message. If they're interpreted as commands to perform, such as moving the insertion point or deleting text, the input manager sends the text view a **doCommandBySelector:** message. Many of the standard commands are described in the NSResponder class specification. NSTextView also gives its delegate a chance to handle a command by sending it a **textView:doCommandBySelector:** message. If the delegate implements this method and returns YES, the text view does nothing further; otherwise it tries to perform the command itself.

See the NSInputManager class and NSTextInput protocol specifications for more information.

Using the Font Panel and Ruler

NSTextView is designed to work with the Application Kit's font conversion system, defined by the NSFontPanel and NSFontManager classes. By default, an NSTextView keeps the Font Panel updated with the first font in its selection, or of its typing attributes (defined below). It also changes the font in response to messages from the Font Panel and Font menu. Such changes apply to the selected text or typing attributes for a rich text view, or to all the text in a plain text view. You can turn this behavior off using the **setUsesFontPanel:** method. Doing so is recommended for a text view that serves as a field editor, for example. Making an NSTextView not use the font conversion system renders some of its other methods unusable, as these methods require access to font information to work. See the description of **setUsesFontPanel:** for these side effects.

NSTextView also defines a comprehensive interface for manipulating paragraph attributes, using the NSRulerView class. If an NSTextView is enclosed in an NSScrollView, it can display a ruler view, which displays margin and tab markers that the user can manipulate to adjust their settings, as well as other controls for setting alignment, paragraph spacing, and so on. **setRulerVisible:** and the inherited **toggleRuler:** control whether the ruler view is displayed. The NSTextView serves as the ruler view's client, as described in the NSRulerView class specification. Similar to the Font Panel, NSTextView can be set not to use a ruler with the **setUsesRuler:** method. This has side effects similar to those of **setUsesFontPanel:**.

Examining and Setting the Selection

Most of the time the selection is determined by the user through mouse or keyboard operations. You can get the range of characters currently selected using the **selectedRange** method. This is the single most commonly used method for examining the selection. You can also set the selection programmatically using **setSelectedRange:**. NSTextView indicates its selection by applying a special set of attributes to it. **selectedTextAttributes** returns these attributes, and **setSelectedTextAttributes:** sets them.

While changing the selection in response to user input, an NSTextView invokes its **setSelectedRange:affinity:stillSelecting:** method. The first argument is of course the range to select. The second, called the selection affinity, determines which glyph the insertion point displays near when the two glyphs aren't adjacent. It's typically used where 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 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. An additional 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 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, as described below under "Subclass Responsibilities."

An additional aspect of selection, actually 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. **markedRange** 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:**

Setting Text Attributes

NSTextView allows you to change the attributes of its text programmatically through various methods, most inherited from the superclass, NSText. NSTextView adds its own methods for setting the attributes of text that the user types, for setting the baseline offset of text as an absolute value, and for adjusting kerning and use of ligatures. Most of the methods for changing attributes are defined as action methods, and apply to the selected text or typing attributes for a rich text view, or to all of the text in a plain text view.

An NSTextView maintains a set of *typing attributes* (font, size, color, and so on) that it applies to newly entered text, whether typed by the user or pasted as plain text. It automatically sets the typing attributes to the attributes of the first character immediately preceding the insertion point, of the first character of a paragraph if the insertion point is at the beginning of a paragraph, or of the first character of a selection. The user can change the typing attributes by choosing menu commands and using utilities such as the Font Panel. You can also set the typing attributes programmatically using **setTypingAttributes:**, though you should rarely find need to do so unless creating a subclass.

NSText defines the action methods **superscript:**, **subscript:**, and **unscript:**, which raise and lower the baseline of text by predefined increments. NSTextView gives you much finer control over the baseline offset of text by defining the **raiseBaseline:** and **lowerBaseline:** action methods, which raise or lower text by one point each time they're invoked.

Kerning

NSTextView provides convenient action methods for adjusting the spacing between characters. By default, an NSTextView object uses standard kerning (as provided by the data in a font's AFM file). A **turnOffKerning:** message causes this kerning information to be ignored and the selected text to be displayed using nominal widths. The **loosenKerning:** and **tightenKerning:** methods adjust kerning values over the selected text and **useStandardKerning:** reestablishes the default kerning values.

Kerning information is a character attribute that's stored in the text view's NSTextStorage object. If your application needs finer control over kerning than the methods of this class provide, you should operate on the NSTextStorage object directly through methods defined by its superclass, NSMutableAttributedString. See the NSAttributedString Class Cluster Additions specification for information on setting attributes.

Ligatures

NSTextView's support for ligatures provides the minimum required ligatures for a given font and script. The required ligatures for a specific font and script are determined by the mechanisms that generate glyphs for a specific language. Some scripts may well have no ligatures at all—English text, as an example, doesn't require ligatures, although certain ligatures such as "fi" and "fl" are desirable and are used if they're available. Other scripts, such as Arabic, demand that certain ligatures must be available even if a **turnOffLigatures:** message is sent to the NSTextView. Other scripts and fonts have standard ligatures that are used if they're available. The **useAllLigatures:** method extends ligature support to include all possible ligatures available in each font for a given script.

Ligature information is a character attribute that's stored in the text view's NSTextStorage object. If your application needs finer control over ligature use than the methods of this class provide, you should operate on the NSTextStorage object directly through methods defined by its superclass, NSMutableAttributedString. See the NSAttributedString Class Cluster Additions specification for information on setting attributes.

Using Multiple NSTextViews

A single NSLayoutManager can be assigned any number of NSTextContainers, in whose NSTextViews it lays out text sequentially. In such a configuration, many of the attributes accessed through the NSTextView interface are actually shared by all of these text views. Among these attributes are:

- The selection
- The delegate (see "Other Delegate Messages and Notifications" below for details)
- Selectability
- Editability
- Whether they act as a field editor
- Whether they display plain or rich text
- Whether they import graphics
- Whether the ruler is visible
- Whether they use the Font Panel
- Whether they use the ruler

Setting any of these attributes causes all associated NSTextView's to share the new value.

With multiple NSTextViews, only one is the first responder at any time. NSLayoutManager defines these methods for determining and appropriately setting the first responder:

– layoutManagerOwnsFirstResponderInWindow:
– firstTextView
– textViewForBeginningOfSelection

See their descriptions in the NSLayoutManager class specification for more information.

Other 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. Several of the delegate methods have already been mentioned; here are all of the messages that a delegate can receive:

textView:willChangeSelectionFromCharacterRange:toCharacterRange:
textViewDidChangeSelection:

textShouldBeginEditing:
textDidBeginEditing:
textView:shouldChangeTextInRange:replacementString:

textDidChange:
textShouldEndEditing:
textDidEndEditing:

textView:doCommandBySelector:

textView:clickedCell:inRect:
textView:doubleClickedCell:inRect:
textView:draggedCell:inRect:event:

Those whose names begin with “text” rather than “textView” are declared by NSText and described in the NSText class specification. See “Methods Implemented By the Delegate” at the end of this class description for more details. The delegate can be any object you choose, and one delegate can control multiple NSTextView objects (or multiple series of connected NSTextView objects).

All NSTextView objects attached to the same NSLayoutManager share the same delegate: Setting the delegate of one such NSTextView sets the delegate for all the others. Delegate messages pass the **id** of the sender as an argument. For multiple NSTextViews attached to the same NSLayoutManager, the **id** is that of the *notifying text view*, the first NSTextView for the shared NSLayoutManager. As the name implies, this NSTextView is also responsible for posting notifications at the appropriate times.

The notifications posted by NSTextView are:

NSTextViewDidChangeSelectionNotification

NSTextDidBeginEditingNotification
NSTextDidEndEditingNotification
NSTextDidChangeNotification

NSTextViewWillChangeNotifyingTextViewNotification

Of these, the last is crucially import for observers to register for. If a new NSTextView is added at the beginning of a series of connected NSTextViews, 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 NSTextViewWillChangeNotifyingTextViewNotification, which allows all those observers to unregister themselves from the old notifying text view and reregister themselves with the new one. See the description for this notification at the end of this specification for more information.

Subclass Responsibilities

NSTextView expects subclasses to abide by certain rules of behavior, and 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. The following sections describe the major areas where a subclass has obligations or where it can expect help in implementing its new features.

Updating State

NSTextView automatically updates the Font Panel 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. **updateFontPanel** makes the Font Panel display the font of the first character in the selection; you might override it to update the display of an accessory view in the Font Panel. Similarly, **updateRuler** causes the ruler to display the paragraph attributes for the first paragraph in the selection. You can also override this to customize display of items in the ruler. Be sure to invoke **super**'s implementation to have the basic updating performed as well.

Custom Import Types

NSTextView supports the dragging of files and colors into its text. If you customize the ability of your subclass to handle dragging operations for new types of data, you should override the **acceptableDragTypes** method to reflect those types. Your implementation should invoke **super**'s implementation, add to the array returned any types your subclass also supports, and return that array. 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.

Altering Selection Behavior

Your subclass of NSTextView can customize the way selections are made for the various granularities described in "Examining and Setting the Selection." While tracking user changes to the selection, whether by the mouse or keyboard, an NSTextView 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.

Note: These mechanisms aren't meant for changing language word definitions (such as what's selected on a double click). This 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 modify 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. **rangeForUserTextChange** returns the range to which any change to characters themselves—insertions and deletions—should apply.

rangeForUserCharacterAttributeChange 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. **textView:shouldChangeTextInRange:replacementString:**, 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 is 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. **smartDeleteRangeForProposedRange:** expands a proposed deletion range to include any whitespace that should also be deleted. If you need to save the text deleted, though, 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.

Adopted Protocols

NSTextInput	– conversationIdentifier – doCommandBySelector: – getMarkedText:selectedRange: – hasMarkedText – insertText: – setMarkedText:selectedRange: – unmarkText

Method Types

Creating an instance	– initWithFrame:textContainer: – initWithFrame:
Registering Services information	+ registerForServices
Accessing related text-system objects	– setTextContainer: – replaceTextContainer: – textContainer – setTextContainerInset: – textContainerInset – textContainerOrigin – invalidateTextContainerOrigin – layoutManager – textStorage
Setting graphic attributes	– setBackgroundColor: – backgroundColor – setDrawsBackground: – drawsBackground
Controlling display	– setNeedsDisplayInRect:avoidAdditionalLayout: – shouldDrawInsertionPoint – drawInsertionPointInRect:color:turnedOn: – setConstrainedFrameSize:

Setting behavioral attributes	– setEditable: – isEditable – setSelectable: – isSelectable – setFieldEditor: – isFieldEditor – setRichText: – isRichText – setImportsGraphics: – importsGraphics
Using the Font Panel and menu	– setUsesFontPanel: – usesFontPanel
Using the ruler	– setUsesRuler: – usesRuler – setRulerVisible: – isRulerVisible
Managing the selection	– setSelectedRange: – selectedRange – setSelectedRange:affinity:stillSelecting: – selectionAffinity – setSelectionGranularity: – selectionGranularity – setInsertionPointColor: – insertionPointColor – updateInsertionPointStateAndRestartTimer: – setSelectedTextAttributes: – selectedTextAttributes – markedRange – setMarkedTextAttributes: – markedTextAttributes
Setting text attributes	– setAlignment:range: – setTypingAttributes: – typingAttributes – useStandardKerning: – lowerBaseline: – raiseBaseline: – turnOffKerning: – loosenKerning: – tightenKerning: – useStandardLigatures: – turnOffLigatures: – useAllLigatures:

Other action methods	– pasteAsPlainText: – pasteAsRichText:
Methods that subclasses should use or override	– updateFontPanel – updateRuler – acceptableDragTypes – updateDragTypeRegistration – selectionRangeForProposedRange:granularity: – rangeForUserCharacterAttributeChange – rangeForUserParagraphAttributeChange – rangeForUserTextChange – shouldChangeTextInRange:replacementString: – didChangeText – setSmartInsertDeleteEnabled: – smartInsertDeleteEnabled – smartDeleteRangeForProposedRange: – smartInsertForString:replacingRange:beforeString:afterString:
Changing first responder status	– resignFirstResponder – becomeFirstResponder
Working with the spelling checker	– spellCheckerDocumentTag
NSRulerView client methods	– rulerView:didMoveMarker: – rulerView:didRemoveMarker: – rulerView:didAddMarker: – rulerView:shouldMoveMarker: – rulerView:shouldAddMarker: – rulerView:willMoveMarker:toLocation: – rulerView:shouldRemoveMarker: – rulerView:willAddMarker:atLocation: – rulerView:handleMouseDown:
Assigning a delegate	– setDelegate: – delegate

Class Methods

registerForServices

+ (void)**registerForServices**

Registers send and return types for the Services facility. This method is invoked automatically; you should never need to invoke it directly.

Instance Methods

acceptableDragTypes

– (NSArray *)**acceptableDragTypes**

Returns the data types that the receiver accepts as the destination view of a dragging operation. These types are automatically registered as necessary by the NSTextView. Subclasses should override this method as necessary to add their own types to those returned by NSTextView's implementation. They must then also override the appropriate methods of the NSDraggingDestination protocol to support import of those types. See that protocol's specification for more information.

See also: – **updateTextViewDragTypeRegistration**

backgroundColor

– (NSColor *)**backgroundColor**

Returns the receiver's background color.

See also: – **drawsBackground**, – **setBackgroundColor:**

becomeFirstResponder

– (BOOL)**becomeFirstResponder**

Informs the receiver that it's becoming the first responder. If the previous first responder was not an NSTextView on the same NSLayoutManager as the receiving NSTextView, this method draws the selection and updates the insertion point if necessary. Returns YES.

Use NSWindow's **makeFirstResponder:**, not this method, to make an NSTextView the first responder. Never invoke this method directly.

See also: – **resignFirstResponder**

delegate

– (id)**delegate**

Returns the delegate used by the receiver (and by all other NSTextViews sharing the receiver's NSLayoutManager), or **nil** if there is none.

See also: – **setDelegate:**

didChangeText

– (void)**didChangeText**

Invoked automatically at the end of a series of changes, this method posts an NSTextDidChangeNotification to the default notification center, which also results in the delegate receiving an NSText-delegate **textDidChange:** message. Subclasses implementing methods that change their text should invoke this method at the end of those methods. See the class description for more information.

See also: – **shouldChangeTextInRange:replacementString:**

drawInsertionPointInRect:color:turnedOn:

– (void)**drawInsertionPointInRect:**(NSRect)*aRect*
color:(NSColor *)*aColor*
turnedOn:(BOOL)*flag*

If *flag* is YES, draws the insertion point in *aRect* using *aColor*. If *flag* is NO, this method erases the insertion point. The PostScript focus must be locked on the receiver when this method is invoked.

See also: – **insertionPointColor**, – **shouldDrawInsertionPoint**, – **backgroundColor**, – **lockFocus** (NSView)

drawsBackground

– (BOOL)**drawsBackground**

Returns YES if the receiver draws its background, NO if it doesn't.

See also: – **backgroundColor**, – **setDrawsBackground:**

encodeWithCoder:

@protocol NSCoding
– (void)**encodeWithCoder:**(NSCoder *)*encoder*

Raises an NSInternalInconsistencyException. NSTextView doesn't support coding.

importsGraphics

– (BOOL)**importsGraphics**

Returns YES if the text views sharing the receiver's NSLayoutManager allow the user to import files by dragging, NO if they don't.

A text view that accepts dragged files is also a rich text view.

See also: – **isRichText**, – **textStorage**, + **attributedStringWithAttachment:** (NSAttributedString), – **insertAttributedString:atIndex:** (NSMutableAttributedString), – **setImportsGraphics:**

initWithCoder:

@protocol NSCoding
– (id)**initWithCoder:**(NSCoder *)*decoder*

Raises an NSInternalInconsistencyException. NSTextView doesn't support coding.

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes a newly allocated NSTextView object with *frameRect* as its frame rectangle. This method creates the entire collection of objects associated with an NSTextView—its NSTextContainer, NSLayoutManager, and NSTextStorage—and invokes **initWithFrame:textContainer:**. Returns **self**.

This method creates the text web in such a manner that the NSTextView object is the principal owner of the objects in the web. See "The OPENSTEP Text System" for a detailed description of ownership issues.

initWithFrame:textContainer:

– (id)**initWithFrame:**(NSRect)*frameRect* **textContainer:**(NSTextContainer *)*aTextContainer*

Initializes a newly allocated NSTextView object with *frameRect* as its frame rectangle and *aTextContainer* as its text container. This method is the designated initializer for NSTextView objects. Returns **self**.

Unlike **initWithFrame:**, which builds up an entire group of text-handling objects, you use this method after you've created the other components of the text handling system—an NSTextStorage object, an NSLayoutManager object, and an NSTextContainer object. Assembling the components in this fashion means that the NSTextStorage, not the NSTextView, is the principal owner of the component objects. See "The OPENSTEP Text System" for a detailed description of ownership issues.

See also: – **initWithFrame:**

insertText:

– (void)**insertText:**(NSString *)*aString*

Inserts *aString* into the receiver's text at the insertion point if there is one, otherwise replacing the selection. The inserted text is assigned the current typing attributes, as explained in the class description under "Setting Text Attributes."

This method is the means by which typed text enters an NSTextView. See the NSInputManager class and NSTextInput protocol specifications for more information.

See also: – **typingAttributes**

insertionPointColor

– (NSColor *)**insertionPointColor**

Returns the color used to draw the insertion point.

See also: – **drawInsertionPointInRect:color:turnedOn:**, – **shouldDrawInsertionPoint**, – **setInsertionPointColor:**

invalidateTextContainerOrigin

– (NSColor *)**invalidateTextContainerOrigin**

Informs the receiver that it needs to recalculate the origin of its text container, usually because it's been resized or the contents of the text container have changed. This method is invoked automatically; you should never need to invoke it directly.

See also: – **textContainer**, – **textContainerOrigin**

isEditable

– (BOOL)**isEditable**

Returns YES if the text views sharing the receiver's NSLayoutManager allow the user to edit text, NO if they don't. If a text view is editable, it's also selectable.

See also: – **isSelectable**, – **setEditable:**

isFieldEditor

– (BOOL)**isFieldEditor**

Returns YES if the text views sharing the receiver's NSLayoutManager interpret Tab, Shift-Tab, and Return (Enter) as cues to end editing, and possibly to change the first responder; no if they accept them as text input. See the NSWindow class specification for more information on field editors. By default, NSTextViews don't behave as field editors.

See also: – **setFieldEditor:**

isRichText

– (BOOL)**isRichText**

Returns YES if the text views sharing the receiver's NSLayoutManager allow the user to apply attributes to specific ranges of the text, NO if they don't.

See also: – **importsGraphics**, – **textStorage**, – **setRichText:**

isRulerVisible

– (BOOL)**isRulerVisible**

Returns YES if the scroll view enclosing the text views sharing the receiver's NSLayoutManager shows its ruler, NO otherwise.

See also: – **usesRuler**, – **setRulerVisible:**, – **toggleRuler:** (NSText)

isSelectable

– (BOOL)**isSelectable**

Returns YES if the text views sharing the receiver's NSLayoutManager allow the user to select text, NO if they don't.

See also: – **isEditable**, – **setSelectable:**

layoutManager

– (NSLayoutManager *)**layoutManager**

Returns the NSLayoutManager that lays out text for the receiver's text container, or **nil** if there's no such object (which is the case when a text view isn't linked into a group of text objects).

See also: – **textContainer**, – **setLayoutManager:** (NSTextContainer), – **replaceLayoutManager:** (NSTextContainer)

loosenKerning:

– (void)**loosenKerning:**(id)*sender*

This action method increases the space between glyphs in the receiver's selection, or in all text if the receiver is a plain text view. Kerning values are determined by the point size of the fonts in the selection.

See also: – **tightenKerning**, – **turnOffKerning:**, – **useStandardKerning:**

lowerBaseline:

– (void)**lowerBaseline:**(id)*sender*

This action method lowers the baseline offset of selected text by one point, or of all text if the receiver is a plain text view. As such, this method defines a more primitive operation than subscripting.

See also: – **raiseBaseline:**, – **subscript:** (NSText), – **unscript:** (NSText)

markedRange

– (NSRange)**markedRange**

Returns the range of marked text. If there's no marked text, returns a range whose location is NSNotFound.

See also: – **setMarkedTextAttributes:**

markedTextAttributes

– (NSDictionary *)**markedTextAttributes**

Returns the attributes used to draw marked text.

See also: – **setMarkedTextAttributes:**

pasteAsPlainText:

– (void)**pasteAsPlainText:**(id)*sender*

This action method inserts the contents of the pasteboard into the receiver's text as plain text, in the manner of **insertText:**.

See also: – **pasteAsRichText:**, – **insertText:**

pasteAsRichText:

– (void)**pasteAsRichText:**(id)*sender*

This action method inserts the contents of the pasteboard into the receiver's text as rich text, maintaining its attributes. The text is inserted at the insertion point if there is one, otherwise replacing the selection.

See also: – **pasteAsRichText:**, – **insertText:**

raiseBaseline:

– (void)**raiseBaseline:**(id)*sender*

This action method raises the baseline offset of selected text by one point, or of all text if the receiver is a plain text view. As such, this method defines a more primitive operation than superscripting.

See also: – **lowerBaseline:**, – **superscript:** (NSText), – **unscript:** (NSText)

rangeForUserCharacterAttributeChange

– (NSRange)rangeForUserCharacterAttributeChange

Returns the range of characters affected by an action method that changes character (not paragraph) attributes, such as the NSText action method **changeFont:**. For rich text this is typically the range of the selection. For plain text this is the entire contents of the receiver.

If the receiver isn't editable or doesn't use the Font Panel, the range returned has a location of NSNotFound.

See also: – **rangeForUserParagraphAttributeChange**, – **rangeForUserTextChange**, – **isEditable**, – **usesFontPanel**

rangeForUserParagraphAttributeChange

– (NSRange)rangeForUserParagraphAttributeChange

Returns the range of characters affected by a method that changes paragraph (not character) attributes, such as the NSText action method **alignLeft:**. For rich text this is typically calculated by extending the range of the selection to paragraph boundaries. For plain text this is the entire contents of the receiver.

If the receiver isn't editable the range returned has a location of NSNotFound.

See also: – **rangeForUserParagraphAttributeChange**, – **rangeForUserTextChange**, – **isEditable**, – **usesRuler**

rangeForUserTextChange

– (NSRange)rangeForUserTextChange

Returns the range of characters affected by a method that changes characters (as opposed to attributes), such as **insertText:**. This is typically the range of the selection.

If the receiver isn't editable or doesn't use a ruler, the range returned has a location of NSNotFound.

See also: – **rangeForUserParagraphAttributeChange**, – **rangeForUserTextChange**, – **isEditable**, – **usesRuler**

replaceTextContainer:

– (void)**replaceTextContainer:**(NSTextContainer *)*aTextContainer*

Replaces the NSTextContainer for the group of text-system objects containing the receiver with *aTextContainer*, keeping the association between the receiver and its layout manager intact, unlike **setTextContainer:**. Raises NSInvalidArgumentException if *aTextContainer* is **nil**.

See also: – **initWithFrame:textContainer:**, – **setTextContainer:**

resignFirstResponder

– (BOOL)**resignFirstResponder**

Notifies the receiver that it's been asked to relinquish its status as first responder in its NSWindow. If the object that will become the new first responder is an NSTextView attached to the same NSLayoutManager as the receiver, this method returns YES with no further action. Otherwise, this method sends a **textShouldEndEditing:** message to its delegate (if any). If the delegate returns NO, this method returns NO. If the delegate returns YES this method hides the selection highlighting and posts an NSTextDidEndEditingNotification to the default notification center.

Use NSWindow's **makeFirstResponder:**, not this method, to make an NSTextView the first responder. Never invoke this method directly.

See also: – **becomeFirstResponder**

rulerView:didAddMarker:

– (void)**rulerView:**(NSRulerView *)*aRulerView* **didAddMarker:**(NSRulerMarker *)*aMarker*

This NSRulerView client method modifies the paragraph style of the paragraphs containing the selection to accommodate a new NSTextTab represented by *aMarker*. It then records the change by invoking **didChangeText**.

NSTextView checks for permission to make the change in its **rulerView:shouldAddMarker:** method, which invokes **shouldChangeTextInRange:replacementString:** to send out the proper request and notifications, and only invokes this method if permission is granted.

See also: – **representedObject** (NSRulerMarker), – **rulerView:didMoveMarker:**, – **rulerView:didRemoveMarker:**

rulerView:didMoveMarker:

– (void)**rulerView:**(NSRulerView *)*aRulerView* **didMoveMarker:**(NSRulerMarker *)*aMarker*

This NSRulerView client method modifies the paragraph style of the paragraphs containing the selection to record the new location of the NSTextTab represented by *aMarker*. It then records the change by invoking **didChangeText**.

NSTextView checks for permission to make the change in its **rulerView:shouldMoveMarker:** method, which invokes **shouldChangeTextInRange:replacementString:** to send out the proper request and notifications, and only invokes this method if permission is granted.

See also: – **representedObject** (NSRulerMarker), – **rulerView:didAddMarker:**, – **rulerView:didRemoveMarker:**

rulerView:didRemoveMarker:

– (void)**rulerView:**(NSRulerView *)*aRulerView* **didRemoveMarker:**(NSRulerMarker *)*aMarker*

This NSRulerView client method modifies the paragraph style of the paragraphs containing the selection—if possible—by removing the NSTextTab represented by *aMarker*. It then records the change by invoking **didChangeText**.

NSTextView checks for permission to move or remove a tab stop in its **rulerView:shouldMoveMarker:** method, which invokes **shouldChangeTextInRange:replacementString:** to send out the proper request and notifications, and only invokes this method if permission is granted.

See also: – **representedObject** (NSRulerMarker), – **shouldChangeTextInRange:replacementString:**, – **rulerView:didAddMarker:**, – **rulerView:didMoveMarker:**

rulerView:handleMouseDown:

– (void)**rulerView:**(NSRulerView *)*aRulerView* **handleMouseDown:**(NSEvent *)*theEvent*

This NSRulerView client method adds a left tab marker to the ruler, but a subclass can override this method to provide other behavior, such as creating guidelines. This method is invoked once with *theEvent* when the user first clicks in the *aRulerView*'s ruler area, as described in the NSRulerView class specification.

rulerView:shouldAddMarker:

– (BOOL)**rulerView:**(NSRulerView *)*aRulerView* **shouldAddMarker:**(NSRulerMarker *)*aMarker*

This NSRulerView client method controls whether a new tab stop can be added. The receiver checks for permission to make the change by invoking **shouldChangeTextInRange:replacementString:** and returning the return value of that message. If the change is allowed, the receiver is then sent a **rulerView:didAddMarker:** message.

See also: – **rulerView:shouldMoveMarker:**, – **rulerView:shouldRemoveMarker:**

rulerView:shouldMoveMarker:

– (BOOL)**rulerView:**(NSRulerView *)*aRulerView* **shouldMoveMarker:**(NSRulerMarker *)*aMarker*

This NSRulerView client method controls whether an existing tab stop can be moved. The receiver checks for permission to make the change by invoking **shouldChangeTextInRange:replacementString:** and returning the return value of that message. If the change is allowed, the receiver is then sent a **rulerView:didMoveMarker:** message.

See also: – **rulerView:shouldAddMarker:**, – **rulerView:shouldRemoveMarker:**

rulerView:shouldRemoveMarker:

– (BOOL)**rulerView:**(NSRulerView *)*aRulerView*
shouldRemoveMarker:(NSRulerMarker *)*aMarker*

This NSRulerView client method controls whether an existing tab stop can be removed. Returns YES if *aMarker* represents an NSTextTab, NO otherwise. Because this method can be invoked repeatedly as the

user drags a ruler marker, it returns that value immediately. If the change is allows and the user actually removes the marker, the receiver is also sent a **rulerView:didRemoveMarker:** message.

See also: – **rulerView:shouldAddMarker:**, – **rulerView:shouldMoveMarker:**

rulerView:willAddMarker:atLocation:

– (float)**rulerView:**(NSRulerView *)*aRulerView*
willAddMarker:(NSRulerMarker *)*aMarker*
atLocation:(float)*location*

This NSRulerView client method ensures that the proposed *location* of *aMarker* lies within the appropriate bounds for the receiver's text container, returning the modified location.

See also: – **rulerView:didAddMarker:**

rulerView:willMoveMarker:toLocation:

– (float)**rulerView:**(NSRulerView *)*aRulerView*
willMoveMarker:(NSRulerMarker *)*aMarker*
toLocation:(float)*location*

This NSRulerView client method ensures that the proposed *location* of *aMarker* lies within the appropriate bounds for the receiver's text container, returning the modified location.

See also: – **rulerView:didMoveMarker:**

selectedRange

– (NSRange)**selectedRange**

Returns the range of characters selected in the receiver's layout manager.

See also: – **selectedTextAttributes**, – **setSelectedRange:affinity:stillSelecting:**, – **selectionRangeForProposedRange:granularity:**, – **setSelectedRange:**

selectedTextAttributes

– (NSDictionary *)**selectedTextAttributes**

Returns the attributes used to indicate the selection. This is typically just the text background color.

See also: – **selectedRange**, – **setSelectedTextAttributes:**

selectionAffinity

– (NSSelectionAffinity)**selectionAffinity**

Returns the preferred direction of selection, either NSSelectionAffinityUpstream or NSSelectionAffinityDownstream. Selection affinity determines whether, for example, the insertion point appears after the last character on a line or before the first character on the following line in cases where text wraps across line boundaries.

See also: – **setSelectedRange:affinity:stillSelecting:**

selectionGranularity

– (NSSelectionGranularity)**selectionGranularity**

Returns the current selection granularity, used during mouse tracking to modify the range of the selection. This is one of:

NSSelectByCharacter
NSSelectByWord
NSSelectByParagraph

See also: – **selectionRangeForProposedRange:granularity:**, – **setSelectionGranularity:**

selectionRangeForProposedRange:granularity:

– (NSRange)**selectionRangeForProposedRange:**(NSRange)*proposedSelRange*
granularity:(NSSelectionGranularity)*granularity*

Adjusts the *proposedSelRange* if necessary, based on *granularity*, which is one of:

NSSelectByCharacter
NSSelectByWord
NSSelectByParagraph

Returns the adjusted range. This method is invoked repeatedly during mouse tracking to modify the range of the selection. Override this method to specialize selection behavior.

See also: – **setSelectionGranularity:**

setAlignment:range:

– (void)**setAlignment:**(NSTextAlignment)*alignment* **range:**(NSRange)*aRange*

Sets the alignment of the paragraphs containing characters in *aRange* to *alignment*, which is one of:

NSLeftTextAlignment
NSRightTextAlignment
NSCenterTextAlignment
NSJustifiedTextAlignment
NSNaturalTextAlignment

See also: – **rangeForUserParagraphAttributeChange**

setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the receiver's background color to *aColor*.

See also: – **setDrawsBackground:**, – **backgroundColor**

setConstrainedFrameSize:

– (void)**setConstrainedFrameSize:**(NSSize)*desiredSize*

Attempts to set the frame size for the NSTextView to *desiredSize*, constrained by the receiver's existing minimum and maximum sizes and by whether resizing is permitted.

See also: – **minSize** (NSText), – **maxSize** (NSText), – **isHorizontallyResizable** (NSText), – **isVerticallyResizable** (NSText)

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the delegate for all NSTextViews sharing the receiver's NSLayoutManager to *anObject*, without retaining it.

See also: – **delegate**

setDrawsBackground:

– (void)**setDrawsBackground:**(BOOL)*flag*

Controls whether the receiver draws its background. If *flag* is YES, the receiver fills its background with the background color; if *flag* is NO, it doesn't.

See also: – **setBackgroundColor:**, – **drawsBackground**

setEditable:

– (void)**setEditable:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager allow the user to edit text. If *flag* is YES, they allow the user to edit text and attributes; if *flag* is NO, they don't. If n NSTextView is made editable, it's also made selectable. NSTextViews are by default editable.

See also: – **setSelectable:**, – **isEditable**

setFieldEditor:

– (void)**setFieldEditor:**(BOOL)flag

Controls whether the text views sharing the receiver's NSLayoutManager interpret Tab, Shift-Tab, and Return (Enter) as cues to end editing, and possibly to change the first responder. If *flag* is YES, they do; if *flag* is NO, they don't, instead accepting these characters as text input. See the NSWindow class specification for more information on field editors. By default, NSTextViews don't behave as field editors.

See also: – **isFieldEditor**

setImportsGraphics:

– (void)**setImportsGraphics:**(BOOL)flag

Controls whether the text views sharing the receiver's NSLayoutManager allow the user to import files by dragging. If *flag* is YES, they do; if *flag* is NO, they don't. If an NSTextView is set to accept dragged files, it's also set for rich text. By default, NSTextViews don't accept dragged files.

See also: – **textStorage**, – **setRichText:**, – **importsGraphics**

setInsertionPointColor:

– (void)**setInsertionPointColor:**(NSColor *)*aColor*

Sets the color of the insertion point to *aColor*.

See also: – **drawInsertionPointInRect:color:turnedOn:**, – **shouldDrawInsertionPoint**, – **insertionPointColor**

setMarkedTextAttributes:

– (void)**setMarkedTextAttributes:**(NSDictionary *)*attributes*

Sets the attributes used to draw marked text to *attributes*. Text color, background color, and underline are the only supported attributes for marked text.

See also: – **markedTextAttributes**, – **markedRange**

setNeedsDisplayInRect:avoidAdditionalLayout:

– (void)**setNeedsDisplayInRect:**(NSRect)*aRect* **avoidAdditionalLayout:**(BOOL)*flag*

Marks the receiver as requiring display within *aRect*. If *flag* is YES, the receiver won't perform any layout that might be required to complete the display, even if this means that portions of the NSTextView remain empty. If *flag* is NO, the receiver performs at least as much layout as needed to display *aRect*.

NSTextView overrides the NSView **setNeedsDisplayInRect:** method such that it invokes this method with NO as *flag*.

setRichText:

– (void)**setRichText:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager allow the user to apply attributes to specific ranges of the text. If *flag* is YES they do; if *flag* is NO they don't. If *flag* is NO, they're also set not to accept dragged files. By default, NSTextViews let the user apply multiple attributes to text, but don't accept dragged files.

See also: – **textStorage**, – **isRichText**, – **setImportsGraphics:**

setRulerVisible:

– (void)**setRulerVisible:**(BOOL)*flag*

Controls whether the scroll view enclosing text views sharing the receiver's NSLayoutManager displays the ruler. If *flag* is YES it shows the ruler; if *flag* is NO it hides the ruler. By default, the ruler is not visible.

See also: – **setUsesRuler:**, – **isRulerVisible**, – **toggleRuler:** (NSText)

setSelectable:

– (void)**setSelectable:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager allow the user to select text. If *flag* is YES, they do; if *flag* is NO, they don't. If an NSTextView is made not selectable, it's also made not editable. NSTextViews are by default both editable and selectable.

See also: – **setEditable:**, – **isSelectable**

setSelectedRange:

– (void)**setSelectedRange:**(NSRange)*charRange*

Sets the selection to the characters in *charRange*, resets the selection granularity to NSSelectByCharacter, posts an NSTextViewDidChangeSelectionNotification to the default notification center. Also removes the marking from marked text if the new selection is greater than the marked region.

charRange must begin and end on glyph boundaries and not split base glyphs and their non-spacing marks.

See also: – **setSelectedRange:affinity:stillSelecting:**, – **selectionAffinity**, – **selectionGranularity**, – **selectedRange**

setSelectedRange:affinity:stillSelecting:

– (void)**setSelectedRange:**(NSRange)*charRange*
affinity:(NSSelectionAffinity)*affinity*
stillSelecting:(BOOL)*flag*

Sets the selection to the characters in *charRange*, using *affinity* if needed to determine how to display the selection or insertion point (see the description for **selectionAffinity** for more information). *flag* indicates whether this method is being invoked during mouse-dragging or after the user releases the mouse. If *flag* is YES the receiver doesn't send notifications or remove the marking from its marked text; if *flag* is NO it does as appropriate. This method also resets the selection granularity to NSSelectByCharacter.

charRange must begin and end on glyph boundaries and not split base glyphs and their non-spacing marks.

See also: – **setSelectedRange:**, – **selectionAffinity**, – **selectionGranularity**, – **setSelectedRange:**

setSelectedTextAttributes:

– (void)setSelectedTextAttributes:(NSDictionary *)*attributes*

Sets the attributes used to indicate the selection to *attributes*. Text color, background color, and underline are the only supported attributes for selected text.

See also: – **selectedRange**, – **selectedTextAttributes**

setSelectionGranularity:

– (void)**setSelectionGranularity:**(NSSelectionGranularity)*granularity*

Sets the selection granularity for subsequent extension of a selection to *granularity*, which may be one of:

NSSelectByCharacter
NSSelectByWord
NSSelectByParagraph

Selection granularity is used to determine how the selection is modified when the user Shift-clicks or drags the mouse after a double- or triple-click. For example, if the user selects a word by double-clicking, the selection granularity is set to NSSelectByWord. Subsequent shift-clicks then extend the selection by words.

Selection granularity is reset to NSSelectByCharacter whenever the selection is set. You should always set the selection granularity after setting the selection.

See also: – **selectionGranularity**, – **setSelectedRange:**

setSmartInsertDeleteEnabled:

– (void)setSmartInsertDeleteEnabled:(BOOL)*flag*

Controls whether the receiver inserts or deletes space around selected words so as to preserve proper spacing and punctuation. If *flag* is YES it does; if *flag* is NO it inserts and deletes exactly what's selected.

See also: – **smartInsertForString:replacingRange:beforeString:afterString:**,
– **smartDeleteRangeForProposedRange:**, – **smartInsertDeleteEnabled**

setTextContainer:

– (void)**setTextContainer:**(NSTextContainer *)*aTextContainer*

Sets the receiver's text container to *aTextContainer*. The receiver then uses the layout manager and text storage of *aTextContainer*. This method is invoked automatically when you create an NSTextView; you should never invoke it directly, but might want to override it. To change the text view for an established group of text-system objects, send **setTextView:** to the text container. To replace the text container for a text view and maintain the view's association with the existing layout manager and text storage, use **replaceTextContainer:**.

See also: – **textContainer**

setTextContainerInset:

– (void)**setTextContainerInset:**(NSSize)*inset*

Sets the empty space the NSTextView leaves around its associated text container to *inset*.

See also: – **textContainerOrigin**, – **invalidateTextContainerOrigin**, – **textContainerInset**

setTypingAttributes:

– (void)**setTypingAttributes:**(NSDictionary *)*attributes*

Sets the receiver's typing attributes to *attributes*. Typing attributes are reset automatically whenever the selection changes. If you add any user actions that change text attributes, you should use this method to apply those attributes to a zero-length selection.

See also: – **typingAttributes**

setUsesFontPanel:

– (void)**setUsesFontPanel:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager use the Font Panel and Font menu. If *flag* is YES, they respond to messages from the Font Panel and from the Font menu, and update the Font Panel with the selection font whenever it changes. If *flag* is NO they disallow character attribute changes. By default, NSTextView objects use the Font Panel and menu.

See also: – **rangeForUserCharacterAttributeChange**, – **usesFontPanel**

setUsesRuler:

– (void)**setUsesRuler:**(BOOL)*flag*

Controls whether the text views sharing the receiver's NSLayoutManager use an NSRulerView and respond to Format menu commands. If *flag* is YES, they respond to NSRulerView client messages and to paragraph-related menu actions, and update the ruler (when visible) as the selection changes with its paragraph and tab attributes. If *flag* is NO, the ruler is hidden and the text views disallow paragraph attribute changes. By default, NSTextView objects use the ruler.

See also: – **setRulerVisible:**, – **rangeForUserParagraphAttributeChange**, – **usesRuler**

shouldChangeTextInRange:replacementString:

– (BOOL)**shouldChangeTextInRange:**(NSRange)*affectedCharRange*
replacementString:(NSString *)*replacementString*

Initiates a series of delegate messages (and general notifications) to determine whether modifications can be made to the receiver's text. If characters in the text string are being changed, *replacementString* contains the characters that will replace the characters in *affectedCharRange*. If only text attributes are being changed, *replacementString* is **nil**. This method checks with the delegate as needed using **textShouldBeginEditing:** and **textView:shouldChangeTextInRange:replacementString:**, returning YES to allow the change, and NO to prohibit it.

This method must be invoked at the start of any sequence of user-initiated editing changes. If your subclass of NSTextView implements new methods that modify the text, make sure to invoke this method to determine whether the change should be made. If the change is allowed, complete the change by invoking the **didChangeText** method. See "Notifying About Changes to the Text" in the class description for more information. If you can't determine the affected range or replacement string before beginning changes, pass (NSNotFound, 0) and **nil** for these values.

See also: – **isEditable**

shouldDrawInsertionPoint

– (BOOL)**shouldDrawInsertionPoint**

Returns YES if the receiver should draw its insertion point, NO if the insertion point can't or shouldn't be drawn (for example, if the receiver's window isn't key).

See also: – **drawInsertionPointInRect:color:turnedOn:**

smartDeleteRangeForProposedRange:

– (NSRange)smartDeleteRangeForProposedRange:(NSRange)*proposedCharRange*

Given *proposedCharRange*, returns an extended range that includes adjacent whitespace that should be deleted along with the proposed range in order to preserve proper spacing and punctuation of the text surrounding the deletion.

NSTextView uses this method as necessary; you can also use it in implementing your own methods that delete text, typically when the selection granularity is NSSelectByWord. To do so, invoke this method with the proposed range to delete, then actually delete the range returned. If placing text on the pasteboard, however, you should put only the characters from the proposed range onto the pasteboard.

See also: – **smartInsertForString:replacingRange:beforeString:afterString:**, – **selectionGranularity**, – **smartInsertDeleteEnabled**

smartInsertDeleteEnabled

– (BOOL)smartInsertDeleteEnabled

Returns YES if the receiver inserts or deletes space around selected words so as to preserve proper spacing and punctuation, NO if it inserts and deletes exactly what's selected.

See also: – **smartInsertForString:replacingRange:beforeString:afterString:**, – **smartDeleteRangeForProposedRange:**, – **setSmartInsertDeleteEnabled:**

smartInsertForString:replacingRange:beforeString:afterString:

– (void)smartInsertForString:(NSString *)*aString*
replacingRange:(NSRange)*charRange*
beforeString:(NSString **)*beforeString*
afterString:(NSString **)*afterString*

Determines whether whitespace needs to be added around *aString* to preserve proper spacing and punctuation when it's inserted into the receiver's text over *charRange*. Returns by reference in *beforeString* and *afterString* any whitespace that should be added, unless either or both is NULL. Both are returned as **nil** if *aString* is **nil** or if smart insertion and deletion is disabled.

NSTextView uses this method as necessary; you can also use it in implementing your own methods that insert text. To do so, invoke this method with the proper arguments, then insert *beforeString*, *aString*, and *afterString* in order over *charRange*.

See also: – **smartDeleteRangeForProposedRange:**, – **smartInsertDeleteEnabled**

spellCheckerDocumentTag

– (int)**spellCheckerDocumentTag**

Returns a tag identifying the NSTextView text as a document for the spell checker server. The document tag is obtained by sending a **uniqueSpellDocumentTag** message to the spell server the first time this method is invoked for a particular group of NSTextViews. See the NSSpellChecking and NSSpellServer class specifications for more information on how this tag is used.

textContainer

– (NSTextContainer *)**textContainer**

Returns the receiver's text container.

See also: – **setTextContainer:**

textContainerInset

– (NSSize)**textContainerInset**

Returns the empty space the NSTextView leaves around its text container.

See also: – **textContainerOrigin**, – **invalidateTextContainerOrigin**, – **setTextContainerInset:**

textContainerOrigin

– (NSPoint)**textContainerOrigin**

Returns the origin of the receiver's text container, which is calculated from the receiver's bounds rectangle, container inset, and the container's used rect.

See also: – **invalidateTextContainerOrigin**, – **textContainerInset**,
– **usedRectForTextContainer:** (NSLayoutManager)

textStorage

– (NSTextStorage *)**textStorage**

Returns the receiver's text storage object.

tightenKerning:

– (void)**tightenKerning:**(id)*sender*

This action method decreases the space between glyphs in the receiver's selection, or for all glyphs if the receiver is a plain text view. Kerning values are determined by the point size of the fonts in the selection.

See also: – **loosenKerning:**, – **useStandardKerning:**, – **turnOffKerning:**

turnOffKerning:

– (void)**turnOffKerning:**(id)*sender*

This action method causes the receiver to use nominal glyph spacing for the glyphs in its selection, or for all glyphs if the receiver is a plain text view.

See also: – **useStandardKerning:**, – **loosenKerning:**, – **tightenKerning:**, – **isRichText**

turnOffLigatures:

– (void)**turnOffLigatures:**(id)*sender*

This action method causes the receiver to use only required ligatures when setting text, for the glyphs in the selection if the receiver is a rich text view, or for all glyphs if it's a plain text view.

See also: – **useAllLigatures:**, – **isRichText**, – **useStandardLigatures:**

typingAttributes

– (NSDictionary *)**typingAttributes**

Returns the current typing attributes.

See also: – **setTypingAttributes:**

updateDragTypeRegistration

– (void)**updateDragTypeRegistration**

If the receiver is editable and is a rich text view, causes all NSTextViews associated with the receiver's NSLayoutManager to register their acceptable drag types. If the NSTextView isn't editable or isn't rich text, causes those NSTextViews to unregister their dragged types.

Subclasses can override this method to change the conditions for registering and unregistering drag types, whether as a group or individually based on the current state of the NSTextView. They can then invoke this method when that state changes to perform that reregistration.

See also: – **acceptableDragTypes**, – **registerForDraggedTypes:** (NSView),
– **unregisterDraggedTypes** (NSView), – **isEditable**, – **importsGraphics**, – **isRichText**

updateFontPanel

– (void)**updateFontPanel**

Updates the Font Panel to contain the font attributes of the selection. Does nothing if the receiver doesn't use the Font Panel. You should never need to invoke this method directly, but you can override it if needed to handle additional font attributes.

See also: – **usesFontPanel**

updateInsertionPointStateAndRestartTimer:

– (void)**updateInsertionPointStateAndRestartTimer:**(BOOL)*flag*

Updates the insertion point's location and, if *flag* is YES, restarts the blinking cursor timer. This method is invoked automatically whenever the insertion point needs to be moved; you should never need to invoke it directly, but you can override it to add different insertion point behavior.

See also: – **shouldDrawInsertionPoint**, – **drawInsertionPointInRect:color:turnedOn:**

updateRuler

– (void)**updateRuler**

Updates the NSRulerView in the receiver's enclosing scroll view to reflect the selection's paragraph and marker attributes. Does nothing if the ruler isn't visible or if the receiver doesn't use the ruler. You should never need to invoke this method directly, but you can override this method if needed to handle additional ruler attributes.

See also: – **usesRuler**

useAllLigatures:

– (void)**useAllLigatures:**(id)*sender*

This action method causes the receiver to use all ligatures available for the fonts and languages used when setting text, for the glyphs in the selection if the receiver is a rich text view, or for all glyphs if it's a plain text view.

See also: – **turnOffLigatures:**, – **useStandardLigatures:**

usesFontPanel

– (BOOL)**usesFontPanel**

Returns YES if the text views sharing the receiver's NSLayoutManager use the Font Panel, NO otherwise. See **setUsesFontPanel:** and **rangeForUserCharacterAttributeChange** for the effect this has on an NSTextView's behavior.

usesRuler

– (BOOL)**usesRuler**

Returns YES if the text views sharing the receiver's NSLayoutManager use a ruler view, NO otherwise. See **setUsesRuler:** and **rangeForUserParagraphAttributeChange** for the effect this has on an NSTextView's behavior

See also: – **setUsesRuler:**

useStandardKerning:

– (void)**useStandardKerning:**(id)*sender*

This action method causes the receiver to use pair kerning data for the glyphs in its selection, or for all glyphs if the receiver is a plain text view. This data is taken from a font's AFM file

See also: – **isRichText**, – **loosenKerning:**, – **tightenKerning:**, – **turnOffKerning:**

useStandardLigatures:

– (void)**useStandardLigatures:**(id)*sender*

This action method causes the receiver to use the standard ligatures available for the fonts and languages used when setting text, for the glyphs in the selection if the receiver is a rich text view, or for all glyphs if it's a plain text view.

See also: – **turnOffLigatures:**, – **useAllLigatures:**

Methods Implemented By the Delegate

NSTextView communicates with its delegate through methods declared both by NSTextView and by its superclass, NSText. See the NSText class specification for those other delegate methods.

textView:clickedOnCell:inRect:

– (void)**textView:**(NSTextView *)*aTextView*
clickedOnCell:(id <NSTextAttachmentCell>)*attachmentCell*
inRect:(NSRect)*cellFrame*

Invoked after the user clicks on *attachmentCell* within *cellFrame* in an NSTextView and the cell wants to track the mouse. The delegate can use this message as its cue to perform an action or select the attachment cell's character. *aTextView* is the first NSTextView in a series shared by an NSLayoutManager, not necessarily the one that draws *attachmentCell*.

The delegate may subsequently receive a **textView:doubleClickedOnCell:** message if the user continues to perform a double click.

See also: – **wantsToTrackMouse** (NSTextAttachmentCell)

textView:doCommandBySelector:

– (BOOL)**textView:**(NSTextView *)*aTextView*
doCommandBySelector:(SEL)*aSelector*

Sent from NSTextView's **doCommandBySelector:**, this method allows the delegate to perform the command for the text view. If the delegate returns YES, the text view doesn't perform *aSelector*; if the delegate returns NO, the text view attempts to perform it. *aTextView* is the first NSTextView in a series shared by an NSLayoutManager.

textView:doubleClickedOnCell:inRect:

– (void)**textView:**(NSTextView *)*aTextView*
doubleClickedOnCell:(id <NSTextAttachmentCell>)*attachmentCell*
inRect:(NSRect)*cellFrame*

Invoked when the user double-clicks on *attachmentCell* within *cellFrame* in an NSTextView and the cell want to track the mouse. The delegate can use this message as its cue to perform an action, such as opening the file represented by the attachment. *aTextView* is the first NSTextView in a series shared by an NSLayoutManager, not necessarily the one that draws *attachmentCell*.

See also: – **wantsToTrackMouse** (NSTextAttachmentCell)

textView:draggedCell:inRect:event:

– (void)**textView:**(NSTextView *)*aTextView*
draggedCell:(id <NSTextAttachmentCell>)*attachmentCell*
inRect:(NSRect)*aRect*
event:(NSEvent *)*theEvent*

Invoked when the user attempts to drag *attachmentCell* from *aRect* within an NSTextView and the cell wants to track the mouse. *theEvent* is the mouse-down event that preceded the mouse-dragged event. The delegate can use this message as its cue to initiate a dragging operation.

See also: – **wantsToTrackMouse** (NSTextAttachmentCell),
– **dragImage:at:offset:event:pasteboard:source:slideBack:** (NSView),
– **dragFile:fromRect:slideBack:event:** (NSView)

textView:shouldChangeTextInRange:replacementString:

– (BOOL)**textView:**(NSTextView *)*aTextView*
shouldChangeTextInRange:(NSRange)*affectedCharRange*
replacementString:(NSString *)*replacementString*

Invoked when an NSTextView needs to determine if text in the range *affectedCharRange* should be changed. If characters in the text string are being changed, *replacementString* contains the characters that will replace the characters in *affectedCharRange*. If only text attributes are being changed, *replacementString* is **nil**. The delegate can return YES to allow the replacement, or NO to reject the change.

aTextView is the first NSTextView in a series shared by an NSLayoutManager.

textView:willChangeSelectionFromCharacterRange:toCharacterRange:

– (NSRange)**textView:**(NSTextView *)*aTextView*
willChangeSelectionFromCharacterRange:(NSRange)*oldSelectedCharRange*
toCharacterRange:(NSRange)*newSelectedCharRange*

Invoked before an NSTextView finishes changing the selection—that is, when the last argument to a **setSelectedRange:affinity:stillSelecting:** message is NO. *oldSelectedCharRange* is the original range of the selection. *newSelectedCharRange* is the proposed character range for the new selection. The delegate can return and adjusted range or return *newSelectedCharRange* unmodified.

aTextView is the first NSTextView in a series shared by an NSLayoutManager.

textViewDidChangeSelection:

– (void)**textViewDidChangeSelection:**(NSNotification *)*aNotification*

Invoked when the selection changes in the NSTextView. The name of *aNotification* is NSTextViewDidChangeSelectionNotification.

See also: NSTextViewDidChangeSelectionNotification

Notifications

NSTextView posts the following notifications as well as those declared by its superclasses, particularly NSText. See the NSText class specification for those other notifications.

NSTextViewDidChangeSelectionNotification

Posted when the selected range of characters changes. NSTextView posts this notification whenever **setSelectedRange:affinity:stillSelecting:** is invoked either directly, or through the many methods (**mouseDown:**, **selectAll:**, and so on) that invoke it indirectly. When the user is selecting text, this

notification is posted only once, at the end of the selection operation. The NSTextView's delegate receives a **textViewDidChangeSelection:** message when this notification is posted.

The notification contains:

Notification Object	The notifying NSTextView.
Userinfo	
Key	**Value**
NSOldSelectedCharacterRange	An NSValue object containing an NSRange

NSTextViewWillChangeNotifyingTextViewNotification

Posted when a new NSTextView is established as the NSTextView that sends notifications. This allows observers to reregister themselves for the new NSTextView. Methods such as **removeTextContainerAtIndex:**, **textContainerChangedTextView**:, and **insertTextContainer:atIndex:** cause this notification to be posted.

The notification contains:

Notification Object	The old notifying NSTextView, or **nil**.
Userinfo	
Key	**Value**
NSOldNotifyingTextView	The old NSTextView, if one exists
NSNewNotifyingTextView	The new NSTextView, if one exists

There's no delegate method associated with this notification. The text-handling system ensures that when a new NSTextView replaces an old one as the notifying NSTextView, the existing delegate becomes the delegate of the new NSTextView and the delegate is registered to receive NSTextView notifications from the new notifying NSTextView. All other observers are responsible for registering themselves on receiving this notification.

See also: – **removeObserver:** (NSNotificationCenter),
– **addObserver:selector:name:object:** (NSNotificationCenter)