# Text System User Interface Layer Programming Guide for Cocoa

Cocoa > Text & Fonts

2006-06-28

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



Simultaneously published in the United States and Canada.

# Contents

# Figures, Tables, and Listings

# Introduction to Text System User Interface Layer

*Text System User Interface Layer* describes the high-level user interface to the Cocoa text system through the NSTextView class.

## Who Should Read This Document

You should read this document if your application needs to present a user interface to the full capabilities of the text system, that is, if your users need to edit substantial amounts of text.

To understand this material you should have a general understanding of Cocoa programming conventions, and you should have read *Text System Overview*.

## Organization of This Document

This document contains the following articles:

- "The User-Interface Layer: NSTextView Class" (page 9) describes the capabilities and features of the NSTextView class, through which most applications interact with the text system.
- "Creating an NSTextView Object" (page 11) explains how to instantiate an NSTextView object using Interface Builder.
- "Creating an NSTextView Programmatically" (page 15) explains how to create an NSTextView object in code and cause it to create its supporting web of text-handling objects.
- "Putting an NSTextView Object in an NSScrollView" (page 17) shows how to programmatically configure an NSTextView object with scroll bars.
- "Using Multiple NSTextViews" (page 21) describes the attributes held in common by multiple text views configured to share a single layout manager.
- "Plain and Rich Text Objects" (page 23) explains the difference between plain text and rich text and lists the RTF control words that any text object recognizes.
- "Setting Text Attributes" (page 25) discusses text attributes and the action methods you can use to control them programmatically.
- "Setting Text Margins" (page 27) describes the values, maintained by various text system objects, that affect the apparent margins surrounding text on a printed page or display.

# See Also

For more information, refer to the following documents:

- *Text System Overview* provides an overview of the Cocoa text system.
- *Text Editing Programming Guide for Cocoa* explains how the text system supports entering and modifying text and attributes through user interaction with the user interface layer.

# The User-Interface Layer: NSTextView Class

The vast majority of applications interact with the text system through one class: NSTextView. An NSTextView object provides a rich set of text-handling features and can:

- Display text in various fonts, colors, and paragraph styles
- Display images
- Read text and images from (and write them to) disk or the pasteboard
- Let users control text attributes such as font, superscripting and subscripting, kerning, and the use of ligatures
- Cooperate with other views to enable scrolling and display of the ruler
- Cooperate with the Font panel (Fonts window) and Spelling panel
- Support various key bindings, such as those used in Emacs

The interface that this class declares (and inherits from its superclass NSText) lets you programmatically:

- Control the size of the area in which text is displayed
- Control the editability and selectability of the text
- Select and act on portions of the text

NSTextView objects are used throughout the Cocoa user interface to provide standard text input and editing features.

An NSTextView object is a convenient package of the most generally useful text-handling features. If the features of the NSTextView class satisfy your application's requirements and you need more programmatic control over the characters and attributes that make up the text, you'll have to learn something about the object that stores this data, NSTextStorage.

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 system—NSTextContainer, NSLayoutManager, and NSTextStorage objects. Here are the major features that NSTextView adds to those of NSText:

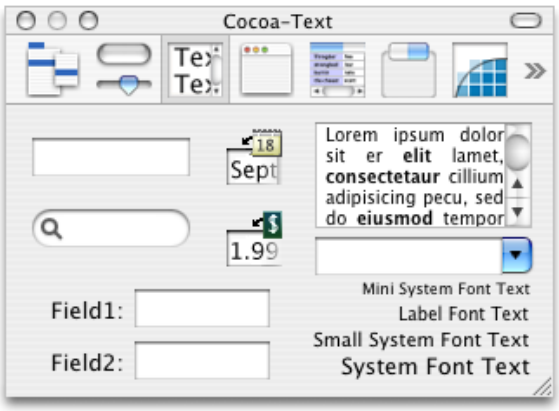
- **Rulers.** NSTextView works with the NSRulerView class to let users control paragraph formatting, in addition to using commands in the Text menu provided by Interface Builder, which is available as a submenu of the Format menu as well as a menu in the menu bar.
- **Input management and key binding.** Certain key combinations are bound to specific NSTextView methods so that the user can, for example, move the insertion point 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 affect only 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 either class.

# Creating an NSTextView Object

The easiest way to use the text system is through Interface Builder. Interface Builder's Cocoa-Text palette, shown in Figure 1, 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.

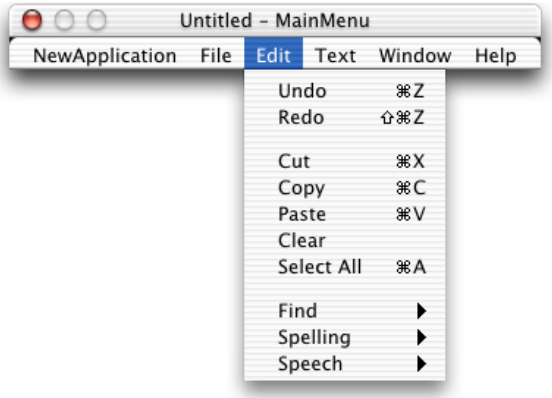
**Figure 1** Cocoa-Text palette




Using Interface Builder's Info window (also called the inspector) you can specify, among other things, whether the contained NSTextView allows multiple fonts and embedded graphics.

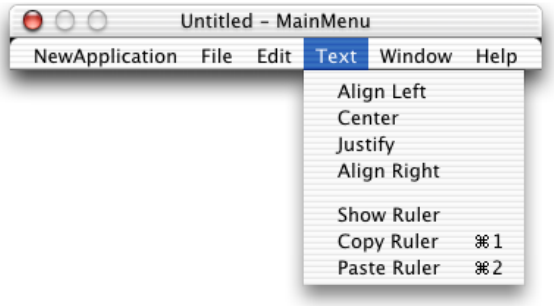
Much more of NSTextView's functionality is accessible through menu commands. Interface Builder's Cocoa-Menus palette offers the ready-made Edit menu that contains text-editing commands shown in Figure 2.

**Figure 2** Edit menu



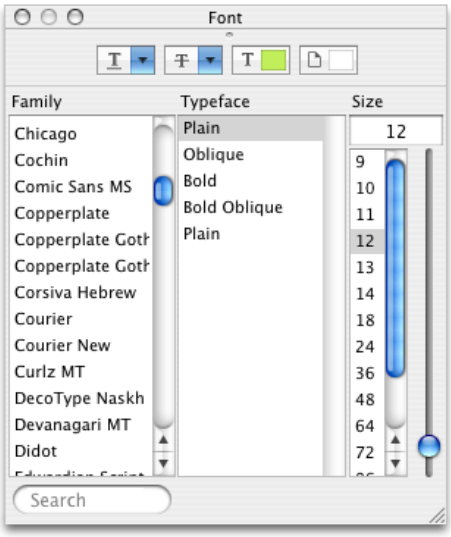
The Cocoa-Menus palette also has the Text menu, shown in Figure 3, which contains paragraph style controls and provides user access to the document's ruler.

**Figure 3** Text menu




The Cocoa-Menus palette also has the system Font panel (or Fonts window) shown in Figure 4.

**Figure 4** Font panel




By default, most of the commands in these menus operate on the first responder, that is, the view within the key window that the user has selected for input. (See the reference documentation for NSResponder, NSView, and NSWindow for more information on the first responder.) In practice, the first responder is the object that's displaying the selection, a drawing object in the case of a graphical selection or an NSTextView in the case of a textual selection. By adding these menus to your application, you can offer the user access to many powerful text-editing features.

NSTextViews cooperate with the Services facility through the Services menu, also available from the Cocoa-Menus palette. By simply adding the Services menu item to your application's main menu, the NSTextViews in your application can access services provided by other applications. For example, if the user selects a word within an NSTextView and chooses the Mail > Send Selection service, the NSTextView passes its selected text to the Mail application which places the text in a new message.

Interface Builder offers these direct ways of accessing the features of the text system. You can also configure your own menu items or other controls within Interface Builder to send messages to an NSTextView object. For example, you can make an NSTextView output its text for printing or faxing by sending it a `print:` or `fax:` message. One way to do this is to drag a menu item from the Cocoa-Menus palette into your application's File menu and hook it up to an NSTextView (either through the first responder or by direct connection). By specifying that the item send a `print:` message to its target, the NSTextView's contents can be printed or faxed when the application is running.

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 NSTextField and NSForm objects 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 standalone or grouped text fields that support editing (and all the other facilities provided by the NSTextView class), these are the classes to use.

Using the Info window (inspector), you can set many text-related attributes of these controls. For example, you can specify whether the text in a text field is selectable, editable, scrollable, and so on. The Info window also lets you set the text alignment and background and foreground colors.

# Creating an NSTextView Programmatically

At times, you may need to assemble the text system programmatically. You can do this in either of two ways: by creating an NSTextView object and letting it create its network of supporting objects or by building the network of objects yourself. In most cases, you'll find it sufficient to create an NSTextView object and let it create the underlying network of text-handling objects, as discussed in this article. If your application has complex text-layout requirements, you'll have to create the network yourself; see "Assembling the Text System by Hand" for information.

You create an NSTextView object programmatically in the usual way: by sending the `alloc` and `init...` messages.

You can also create NSTextView objects using one of its constructors in Java or either of these methods in Objective-C:

- `initWithFrame:textContainer:` (the designated initializer)
- `initWithFrame:`

The method that takes one argument, `initWithFrame:`, is the simplest way to obtain an NSTextView object—it creates all the other components of the text system for you and releases them when you're done. If you use the method that takes two arguments, `initWithFrame:textContainer:`, you must create (and release) the other components yourself.

Listing 1 shows how you can create an NSTextView object, given an NSWindow object represented by `aWindow`.

**Listing 1** Creating an NSTextView programmatically

```
/* determine the size for the NSTextView */
NSRect cFrame =[[aWindow contentView] frame];

/* create the NSTextView and add it to the window */
NSTextView *theTextView = [[NSTextView alloc] initWithFrame:cFrame];
[aWindow setContentView:theTextView];
[aWindow makeKeyAndOrderFront:nil];
[aWindow makeFirstResponder:theTextView];
```

This code determines the size for the NSTextView's frame rectangle by asking `aWindow` for the size of its content view. The NSTextView is then created and made the content view of `aWindow` using `setContentView:`. Finally, the `makeKeyAndOrderFront:` and `makeFirstResponder:` messages display the window and cause `theTextView` to prepare to accept keyboard input.

NSTextView's `initWithFrame:` method not only initializes the receiving NSTextView object, it causes the object to create and interconnect the other components of the text system. This is a convenience that frees you from having to create and interconnect them yourself. Since the NSTextView created these supporting objects, it's responsible for releasing them when they are no longer needed. When you're done with the NSTextView, release it and it takes care of releasing the other objects of the text system. Note that this

ownership policy is in effect only if you let NSTextView create the components of the text system. See "Assembling the Text System by Hand" for more information on object ownership when you create the components yourself.

# Putting an NSTextView Object in an NSScrollView

A scrolling text view is commonly required in applications, and Interface Builder provides an NSTextView configured just for this purpose. However, at times you may need to create a scrolling text view programmatically.

The process consists of three steps: setting up the NSScrollView, setting up the NSTextView, and assembling the pieces. This article describes these steps in terms of a typical text view configured with a vertical scroll bar only, then shows alternate statements used to configure a horizontal scroll bar.

## Setting Up the Scroll View

Assuming an object has the variable `theWindow` that represents the window where the scrolling view is displayed, you can set up the NSScrollView using the code in Listing 1.

**Listing 1** Setting up the scroll view

```
NSScrollView *scrollview = [[NSScrollView alloc]
            initWithFrame:[[theWindow contentView] frame]];
NSSize contentSize = [scrollview contentSize];

[scrollview setBorderType:NSNoBorder];
[scrollview setHasVerticalScroller:YES];
[scrollview setHasHorizontalScroller:NO];
[scrollview setAutoresizingMask:NSViewWidthSizable |
            NSViewHeightSizable];
```

Note that the code creates an NSScrollView that completely covers the content area of the window it's displayed in. It also specifies a vertical scroll bar but no horizontal scroll bar, since this scrolling text view wraps text within the horizontal extent of the NSTextView, but lets text flow beyond the vertical extent of the NSTextView. To use a horizontal scroll bar, you must configure the scroll view and text view slightly differently, as described in "Setting Up a Horizontal Scroll Bar" (page 18).

Finally, the code sets how the NSScrollView reacts when the window it's displayed in changes size. Turning on the NSViewWidthSizable and NSViewHeightSizable bits of its resizing mask ensures that the NSScrollView grows and shrinks to match the window's dimensions.

## Setting Up the Text View

The next step is to create and configure an NSTextView to fit in the NSScrollView. Listing 2 shows the statements that accomplish this step.

**Listing 2** Setting up the text view

```
theTextView = [[NSTextView alloc] initWithFrame:NSMakeRect(0, 0,
            contentSize.width, contentSize.height)];
[theTextView setMinSize:NSMakeSize(0.0, contentSize.height)];
[theTextView setMaxSize:NSMakeSize(FLT_MAX, FLT_MAX)];
[theTextView setVerticallyResizable:YES];
[theTextView setHorizontallyResizable:NO];
[theTextView setAutoresizingMask:NSViewWidthSizable];

[[theTextView textContainer]
            setContainerSize:NSMakeSize(contentSize.width, FLT_MAX)];
[[theTextView textContainer] setWidthTracksTextView:YES];
```

Listing 2 specifies that the NSTextView's width and height initially match those of the content area of the NSScrollView. The `setMinSize:` message tells the NSTextView that it can get arbitrarily small in width, but no smaller than its initial height. The `setMaxSize:` message allows the receiver to grow arbitrarily in either dimension. These limits are used by the NSLayoutManager when it resizes the NSTextView to fit the text laid out.

The next three messages determine how the NSTextView's dimensions change in response to additions or deletions of text and to changes in the scroll view's size. The NSTextView is set to grow vertically as text is added but not horizontally. Its resizing mask is set to allow it to change width in response to changes in the width of its `superview`. Since, except for the minimum and maximum values, the NSTextView's height is determined by the amount of text it has in it, its height should not change with that of its superview.

The last two messages in this step are to the NSTextContainer, not the NSTextView. One message sets the text container's initial width to that of the scroll view and its height to the maximum size of the text view. The last message tells the NSTextContainer to resize its width according to the width of the NSTextView. Recall that the text system lays out text according to the dimensions stored in NSTextContainer objects. An NSTextView provides a place for the text to be displayed, but its dimensions and those of its NSTextContainer can be quite different. The `setWidthTracksTextView:YES` message ensures that as the NSTextView is resized, the width dimension stored in its NSTextContainer is likewise resized, causing the text to be laid out within the new boundaries.

## Assembling the Pieces

The last step is to assemble and display the pieces. Listing 3 shows the statements that accomplish this step.

**Listing 3** Assembling the pieces

```
[scrollview setDocumentView:theTextView];
[theWindow setContentView:scrollview];
[theWindow makeKeyAndOrderFront:nil];
[theWindow makeFirstResponder:theTextView];
```

## Setting Up a Horizontal Scroll Bar

To set up both horizontal and vertical scroll bars, use the statements in Listing 4 in place of the corresponding statements in the previous listings.

**Listing 4** Setting up a horizontal scroll bar

```
[[theTextView enclosingScrollView] setHasHorizontalScroller:YES];
[theTextView setHorizontallyResizable:YES];
[theTextView setAutoresizingMask:(NSViewWidthSizable | NSViewHeightSizable)];
[[theTextView textContainer] setContainerSize:NSMakeSize(FLT_MAX, FLT_MAX)];
[[theTextView textContainer] setWidthTracksTextView:NO];
```

This code fragment adds the horizontal scroll bar to the scroll view and makes the text view horizontally resizable so it can display text of any width. The code sets the text view's resizing mask so that it changes in both width and height in response to corresponding changes in its superview. The next-to-last message sets both dimensions of the text container to an arbitrarily large value, which essentially means the text is laid out in one long line, and the last message ensures that the text container does not resize horizontally with the text view.

# 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
- Selectability
- Editability
- Whether they act as a field editor
- Whether they display plain or rich text
- Whether they import graphics
- Whether they use the ruler
- Whether the ruler is visible
- Whether they use the Font panel (Fonts window)

Setting any of these attributes causes all associated NSTextView objects 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.

# Plain and Rich Text Objects

Text objects such as NSText and NSTextView can contain either plain text or rich text. Plain text objects allow only one set of text attributes for all of their text; rich text objects allow multiple fonts, sizes, indents, and other attributes for different sets of characters and paragraphs. You can control whether a text object is plain or rich using the `setRichText:` method. Rich text objects are also capable of allowing the user to drag images and files into them. This behavior is controlled by the `setImportsGraphics:` method.

A rich NSText object can use RTF (Rich Text Format) as an interchange format. Not all RTF control words are supported, however. On input, an NSText object ignores any control word it doesn't recognize; some of those it can read and interpret it doesn't write out. Table 1 lists the RTF control words that any text object recognizes. Subclasses may recognize more.

**Table 1** RTF control words recognized by all text objects

| Control word | Can be written out |
|---|---|
| `\ansi` | yes |
| `\b` | yes |
| `\cb` | yes |
| `\cf` | yes |
| `\colortbl` | yes |
| `\dnn` | yes |
| `\fin` | yes |
| `\fn` | yes |
| `\fonttbl` | yes |
| `\fsn` | yes |
| `\i` | yes |
| `\lin` | yes |
| `\margrn` | yes |
| `\paperwn` | yes |
| `\mac` | no |
| `\margln` | yes |
| `\par` | yes |

| Control word | Can be written out |
|---|---|
| \pard | no |
| \pca | no |
| \qc | yes |
| \ql | yes |
| \qr | yes |
| \sn | no |
| \tab | yes |
| \upn | yes |

# 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 (Fonts window). 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 reference documentation for NSAttributedString Additions 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:`

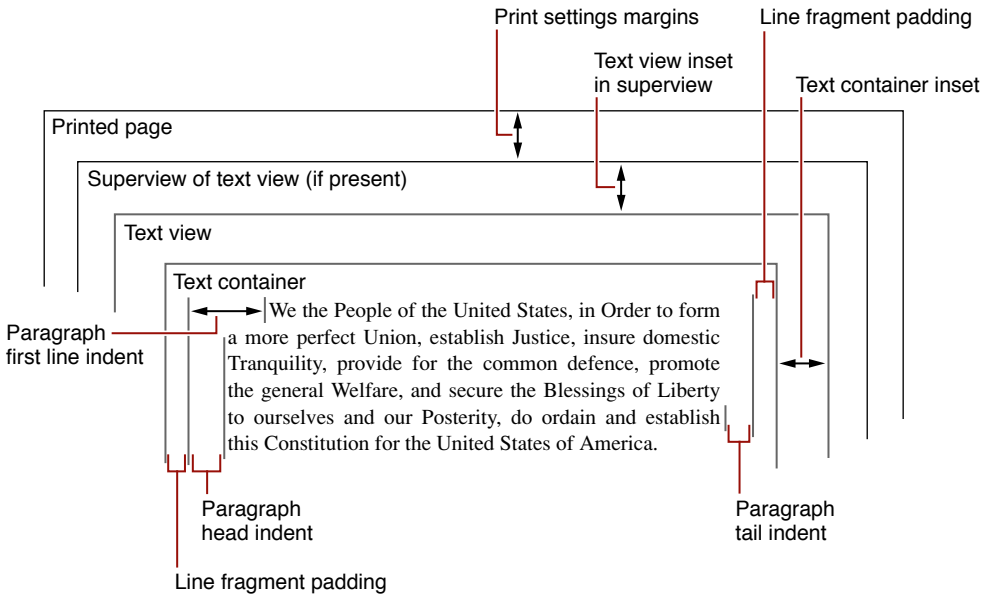
message is sent to the NSTextView. Other scripts and fonts have standard ligatures that are used if they're available. The `useAllLigatures:` method extends ligature support to include all possible ligatures available in each font for a given script.

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 reference documentation for NSAttributedString Additions for information on setting attributes.

# Setting Text Margins

Many text system objects cooperate in the display of text, and several of them maintain inset values that affect the apparent margins of text on a printed page or display. This article describes those settings and their proper use. Figure 1 illustrates the various margins and insets you can place around text.

**Figure 1** Text margins and insets




Paragraph style objects maintain head indent values for the first and subsequent lines and a tail indent value. These values describe space between the beginning and end of text lines and the edge of the text container. For left-to-right text, as shown in Figure 1, the head indents appear on the left side of the paragraph and the tail indent on the right side. You can find the indent values using the NSParagraphStyle methods `firstLineHeadIndent`, `headIndent`, and `tailIndent`. You set the values using the corresponding NSMutableParagraphStyle methods `setFirstLineHeadIndent:`, `setHeadIndent:`, and `setTailIndent:`.

By default, a text container covers its text view exactly. However, you can specify blank space between the edges of the text container and the edges of the text view with the NSTextView method `setTextContainerInset:`. This method specifies a width and height by which the text container's top-left origin point is offset from the origin of the text view. The text container's right and bottom edges are then inset by an equal amount. The container inset is respected even when the container is set to track the height and width of the text view. It's possible to set the text container and text view sizes and resizing behavior so that the inset cannot be maintained exactly, but the text system maintains it whenever possible.

The text container inset refers to the bounding rectangle of the text container's region. However, you can define the region to be a nonrectangular shape, in which case some lines of text can have additional space between the ends of the lines and the bounding rectangle. See "Calculating Region, Bounding Rectangle, and Inset" for more information.

Another parameter that you can set to leave space at the ends of lines of type is called line fragment padding. You can set the padding value with the NSTextContainer method `setLineFragmentPadding:`. This adjustment is meant to specify a small amount of blank space on each end of the line fragment rectangles in which the typesetter sets lines of text. Line fragment padding keeps text from directly abutting any graphics or other elements positioned next to the text container.

Finally, the text view itself can optionally be inset in a superview, as in TextEdit's multiple-page view, and views can be inset on a printed page using print settings.

# Document Revision History

This table describes the changes to *Text System User Interface Layer Programming Guide for Cocoa*.

| Date | Notes |
| --- | --- |
| 2006-06-28 | Corrected line fragment padding representation in Figure 1 of "Setting Text Margins." |
| 2004-07-27 | Made editorial revisions to previously unedited articles. |
| 2004-02-06 | Added section to “Putting an NSTextView Object in an NSScrollView.” Added a new article titled “Setting Text Margins.” Rewrote introduction and added an index. |
| 2003-05-02 | Moved four articles to new *Text Editing* programming topic. |
| 2003-01-16 | Corrected error in example code in the article “Putting an NSTextView Object in an NSScrollView.” |
| 2002-11-12 | Revision history added to existing topic. |

# Index

## P



## R



## S



## T



## U