

# Cocoa Text Architecture Guide



# Contents

## About the Cocoa Text System 8

At a Glance 8

Most Applications Can Use the Cocoa Text System 9

Typographical Concepts Are Essential for Understanding the Text System 9

The Text System Comprises Views, Controllers, and Storage Classes 9

Attributes Characterize Text and Documents 9

Font Objects, the Font Panel, and the Font Manager Provide Typeface Handling 10

Text Objects Are Key to Text Editing 10

Prerequisites 10

See Also 10

## Text Handling Technologies in OS X 12

### Typographical Concepts 13

Characters and Glyphs 13

Typefaces and Fonts 14

Text Layout 15

### Text System Organization 20

Functional Areas of the Cocoa Text System 21

Class Hierarchy of the Cocoa Text System 23

MVC and the Text System 25

Creating Text System Objects 25

Text View Creates the Objects 25

Your App Creates the Objects Explicitly 26

Common Configurations 28

### Text Fields, Text Views, and the Field Editor 33

Text Fields 33

Text Views 34

The Field Editor 35

### Text Attributes 36

Character Attributes 36

Storing Character Attributes	37
Attribute Fixing	37
Temporary Attributes	38
Paragraph Attributes	38
Glyph Attributes	38
Document Attributes	39
<b>Font Handling</b>	40
The Font Panel	40
Creating a Font Panel	40
Using the Font Panel	41
Working with Font Objects	41
Font Descriptors	42
Querying Font Metrics	43
Querying Standard Font Variations	44
Characters, Glyphs, and the Layout Manager	45
Getting the View Coordinates of a Glyph	46
Working with the Font Manager	46
Creating a Font Manager	46
Handling Font Changes	47
Converting Fonts Manually	48
Setting Font Styles and Traits	49
Examining Fonts	50
Customizing the Font Conversion System	50
<b>Text Editing</b>	52
The Editing Environment	52
The Key-Input Message Sequence	53
Intercepting Key Events	55
Text View Delegation	56
Text View Delegate Messages and Notifications	57
Text Field Delegation	58
Synchronizing Editing	59
Batch-Editing Mode	59
Forcing the End of Editing	60
Setting Focus and Selection Programmatically	61
Subclassing NSTextView	62
Updating State	63
Custom Import Types	63
Altering Selection Behavior	64

Preparing to Change Text	64
Text Change Notifications and Delegate Messages	64
Smart Insert and Delete	65
Creating a Custom Text View	65
Implementing Text Input Support	65
Managing Marked Text	66
Communicating with the Text Input Context	67
Working with the Field Editor	68
How the Field Editor Works	68
Using Delegation and Notification with the Field Editor	68
Using a Custom Field Editor	70
Field Editor–Related Methods	71
<b>Document Revision History</b>	<b>75</b>
<b>Objective-C</b>	<b>7</b>

# Figures, Tables, and Listings

## Typographical Concepts 13

- Figure 2-1 Glyphs of the character A 13
- Figure 2-2 Ligatures 14
- Figure 2-3 Fonts in the Times font family 15
- Figure 2-4 Glyph metrics 17
- Figure 2-5 Kerning 17
- Figure 2-6 Alignment of text relative to margins 18
- Figure 2-7 Justified text 19

## Text System Organization 20

- Figure 3-1 Major functional areas of the Cocoa text system 21
- Figure 3-2 Cocoa Text System Class Hierarchy 24
- Figure 3-3 Text object configuration for a single flow of text 28
- Figure 3-4 Text object configuration for paginated text 29
- Figure 3-5 Text object configuration for a multicolumn document 30
- Figure 3-6 Text object configuration for multiple views of the same text 31
- Figure 3-7 Text object configuration with custom text containers 32

## Text Fields, Text Views, and the Field Editor 33

- Figure 4-1 A text field 33
- Figure 4-2 A text view 34
- Figure 4-3 The field editor 35

## Text Attributes 36

- Figure 5-1 The composition of an NSAttributedString object including its attributes dictionary 37

## Font Handling 40

- Figure 6-1 Font metrics 43
- Table 6-1 Font metrics and related NSFont methods 44
- Table 6-2 Standard font methods 44
- Table 6-3 Font conversion methods 48
- Table 6-4 Font menu item actions and tags 51
- Listing 6-1 Font family name matching 42

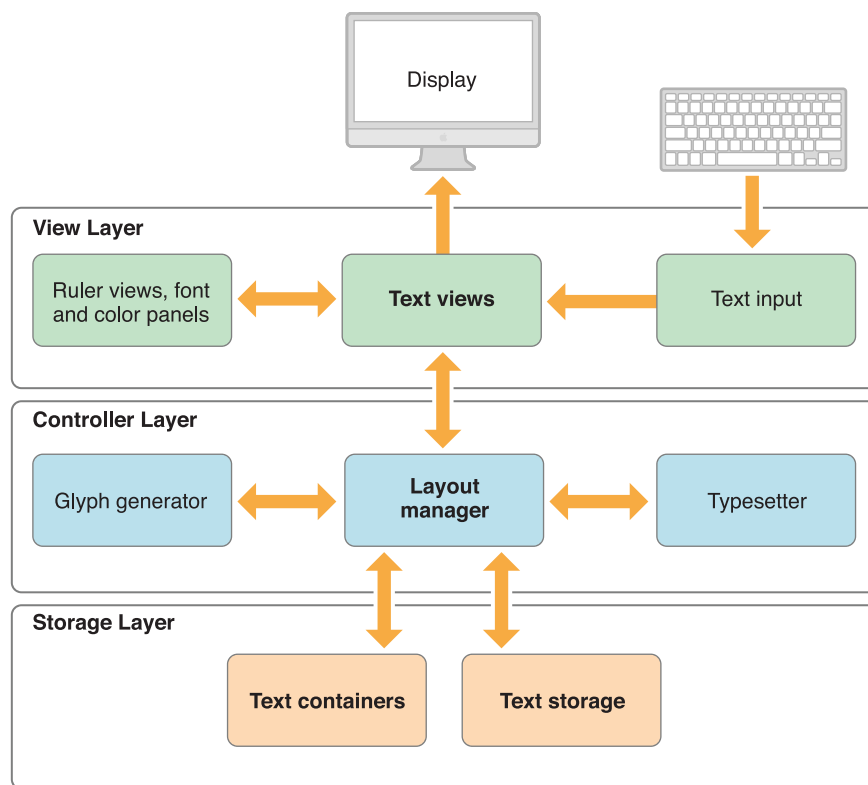
## Text Editing 52

- Figure 7-1     Key-event processing 53
- Figure 7-2     Input context key binding and interpretation 54
- Figure 7-3     Delegate of an NSTextView object 57
- Table 7-1     NSWindow field editor–related methods 71
- Table 7-2     NSTextFieldCell field editor–related method 72
- Table 7-3     NSCell field editor–related methods 72
- Table 7-4     NSControl field editor–related methods 73
- Table 7-5     NSResponder field editor–related methods 73
- Table 7-6     NSText field editor–related methods 74
- Listing 7-1     Forcing layout 60
- Listing 7-2     Forcing the end of editing 61
- Listing 7-3     Forcing the field editor to enter a newline character 69
- Listing 7-4     Substituting a custom field editor 71

Objective-CSwift

# About the Cocoa Text System

The Cocoa text system is the primary text-handling system in OS X, responsible for the processing and display of all visible text in Cocoa. It provides a complete set of high-quality typographical services through the text-related AppKit classes, which enable applications to create, edit, display, and store text with all the characteristics of fine typesetting, such as kerning, ligatures, line-breaking, and justification.



## At a Glance

The Cocoa text system provides text editing and layout for most applications. The object-oriented design of the system provides flexibility and ease of use.



## Most Applications Can Use the Cocoa Text System

If your application needs to display text, and especially if its users need to enter and edit text, then you should use the Cocoa text system. The Cocoa text system is one of two text-handling systems in OS X. The other is Core Text, which provides low-level, basic text layout and font-handling capabilities to higher-level engines such as the AppKit.

---

**Related Chapter:** [Text Handling Technologies in OS X](#) (page 12)

---

## Typographical Concepts Are Essential for Understanding the Text System

The Cocoa text system encodes characters as Unicode values. It translates characters into glyphs, including ligatures and other contextual forms, and handles typefaces, styles, fonts, and families. The system does text layout, placing glyphs horizontally or vertically in either direction, using font metric information, and uses kerning when appropriate. It performs high-quality line breaking and hyphenation to create lines of text with proper alignment or justification.

---

**Related Chapter:** [Typographical Concepts](#) (page 13)

---

## The Text System Comprises Views, Controllers, and Storage Classes

The Cocoa text system is abstracted as a set of classes that represent modular, layered functional areas reflecting the Model-View-Controller design paradigm. The top layer of the system is the user-interface layer of various views, the bottom layer stores the data models, and the middle layer consists of controllers that interpret keyboard input and arrange text for display.

The four primary text system classes—`NSTextView`, `NSLayoutManager`, `NSTextContainer`, and `NSTextStorage`—can be configured in various ways to accomplish different text-handling goals.

---

**Related Chapters:** [Text System Organization](#) (page 20), [Text Fields, Text Views, and the Field Editor](#) (page 33)

---

## Attributes Characterize Text and Documents

The Cocoa text system handles five kinds of attributes: character attributes, such as font and size; temporary attributes used during processing or display, such as underlining of misspelled words; paragraph attributes, such as alignment and tab stops; glyph attributes that may control special handling of particular glyphs; and document attributes, such as margins and paper size.

---

**Related Chapter:** [Text Attributes](#) (page 36)

---

## Font Objects, the Font Panel, and the Font Manager Provide Typeface Handling

The Font panel, also called the *Fonts window*, is a user interface object that displays a list of available font families and styles, letting the user preview them and change the font used to display text. Text views work with `NSFontPanel` and `NSFontManager` objects to implement the font-handling system. You can create font objects using the `NSFont` class and query them for font metrics and detailed glyph layout information.

---

**Related Chapter:** [Font Handling](#) (page 40)

---

## Text Objects Are Key to Text Editing

Usually, text editing is performed by direct user action with a text view, but it can also be accomplished by programmatic interaction with a text storage object. The text input system translates keyboard events into commands and text input. You can customize editing behavior using many methods of text system objects, through the powerful Cocoa mechanisms of notification and delegation, or, in extreme cases, by replacing the text view itself with a custom subclass.

---

**Related Chapter:** [Text Editing](#) (page 52)

---

## Prerequisites

To understand the information in this document, you should understand the material in *Text System User Interface Layer Programming Guide*. In addition, you should have a general knowledge of Cocoa programming paradigms and, to understand the code examples, familiarity with the Objective-C language.

## See Also

The following documents describe other aspects of the Cocoa text system:

*Text System User Interface Layer Programming Guide* describes the high-level interface to the Cocoa text system, which is sufficient for most applications.

*Text System Storage Layer Overview* discusses the lower-level facilities that the Cocoa text system uses to store text.

*Text Layout Programming Guide* describes how the Cocoa text system lays out text on a page, suitable for display and printing.

The following sample code projects illustrate how to use many of the APIs of the Cocoa text system:

*CircleView* is a small application with a demonstration subclass of `NSView` that draws text in a circle.

*NSFontAttributeExplorer* demonstrates how to gather and display various metric information for installed fonts using `NSFont`.

*TextInputView* demonstrates how a view can implement the `NSTextInputClient` protocol.

*TextViewDelegate* demonstrates using a text view's delegate to control selection and user input.

# Text Handling Technologies in OS X

The Macintosh operating system has provided sophisticated text handling and typesetting capabilities from its beginning. In fact, these features sparked the desktop publishing revolution. Over the years, the text handling facilities of the platform have continued to evolve to become more advanced, more efficient, and easier to use. OS X provides modern text handling capabilities that are available to all applications through the classes of the Cocoa text system and the opaque types and functions of Core Text.

The text-handling component of any application presents one of the greatest challenges to software designers. Even the most basic text-handling system must be relatively sophisticated, allowing for text input, layout, display, editing, copying and pasting, and many other features. But developers and users expect even more than these basic features, expecting even simple editors to support multiple fonts, various paragraph styles, embedded images, spell checking, and other features.

The Cocoa text system provides all these basic and advanced text-handling features, and it also satisfies additional requirements from the ever-more-interconnected computing world: support for the character sets of all of the world's living languages, powerful layout capabilities to handle various text directionality and nonrectangular text containers, and sophisticated typesetting capabilities such as control of kerning and ligatures. Cocoa's object-oriented text system is designed to provide all these capabilities without requiring you to learn about or interact with more of the system than is necessary to meet the needs of your application.

Underlying the Cocoa text system is Core Text, which provides low-level, basic text layout and font-handling capabilities to higher-level engines such as AppKit, WebKit, and others. Core Text provides the implementation for many Cocoa text technologies. Application developers typically have no need to use Core Text directly. However, the Core Text API is accessible to developers who must use it directly, such as those writing applications with their own layout engine and those porting ATSUI- or QuickDraw-based codebases to the modern world.

To decide which OS X text technology is right for your application, apply the following guidelines:

- If possible, use Cocoa text. The `NSTextView` class is the most advanced full-featured, flexible text view in OS X. For small amounts of text, use `NSTextField`. For more information about text views, see *Text System User Interface Layer Programming Guide*.
- To display web content in your application, use WebKit. For more information about WebKit, see *WebKit Objective-C Programming Guide*.
- If you have your own page layout engine, you can use Core Text to generate the glyphs and position them relative to each other. For more information about Core Text, see *Core Text Programming Guide*.

# Typographical Concepts

This chapter defines some important typographical concepts relevant to the text system. If you are already familiar with typography, you can skip this chapter.

## Characters and Glyphs

A *character* is the smallest unit of written language that carries meaning. Characters can correspond to a particular sound in the spoken form of the language, as do the letters of the Roman alphabet; they can represent entire words, such as Chinese ideographs; or they can represent independent concepts, such as mathematical symbols. In every case, however, a character is an abstract concept.

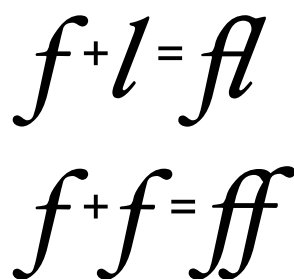
Although characters must be represented in a display area by a recognizable shape, they are not identical to that shape. That is, a character can be drawn in various forms and remain the same character. For example, an “uppercase A” character can be drawn with a different size or a different stroke thickness, it can lean or be vertical, and it can have certain optional variations in form, such as serifs. Any one of these various concrete forms of a character is called a *glyph*. Figure 2-1 shows different glyphs that all represent the character “uppercase A.”

Figure 2-1 Glyphs of the character A



Characters and glyphs do not have a one-to-one correspondence. In some cases a character may be represented by multiple glyphs, such as an “é” which may be an “e” glyph combined with an acute accent glyph “’”. In other cases, a single glyph may represent multiple characters, as in the case of a *ligature*, or joined letter. Figure 2-2 shows individual characters and the single-glyph ligature often used when they are adjacent.

Figure 2-2 Ligatures



A ligature is an example of a contextual form in which the glyph used to represent a character changes depending on the characters next to it. Other contextual forms include alternate glyphs for characters beginning or ending a word.

Computers store characters as numbers mapped by encoding tables to their corresponding characters. The encoding scheme native to OS X is called *Unicode*. The Unicode standard provides a unique number for every character in every modern written language in the world, independent of the platform, program, and programming language being used. This universal standard solves a longstanding problem of different computer systems using hundreds of conflicting encoding schemes. It also has features that simplify handling bidirectional text and contextual forms.

Glyphs are also represented by numeric codes called glyph codes. The glyphs used to depict characters are selected by the Cocoa layout manager during composition and layout processing. The layout manager determines which glyphs to use and where to place them in the display, or view. The layout manager caches the glyph codes in use and provides methods to convert between characters and glyphs and between characters and view coordinates. (See [Text Layout](#) (page 15) for more information about the layout process.)

## Typefaces and Fonts

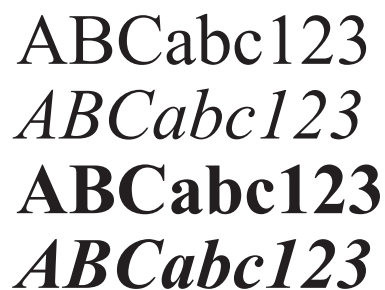
A *typeface* is a set of visually related shapes for some or all of the characters in a written language. For example, Times is a typeface, designed by Stanley Morrison in 1931 for *The Times* newspaper of London. All of the letter forms in Times are related in appearance, having consistent proportions between stems (vertical strokes) and counters (rounded shapes in letter bodies) and other elements. When laid out in blocks of text, the shapes in a typeface work together to enhance readability.

A *typestyle*, or simply *style*, is a distinguishing visual characteristic of a typeface. For example, roman typestyle is characterized by upright letters having serifs and stems thicker than horizontal lines. In italic typestyle, the letters slant to the right and are rounded, similar to cursive or handwritten letter shapes. A typeface usually has several associated typestyles.

A *font* is a series of glyphs depicting the characters in a consistent size, typeface, and typestyle. A font is intended for use in a specific display environment. Fonts contain glyphs for all the contextual forms, such as ligatures, as well as the normal character forms.

A *font family* is a group of fonts that share a typeface but differ in typestyle. So, for example, Times is the name of a font family (as well as the name of its typeface). Times Roman and Times Italic are the names of two individual fonts belonging to the Times family. Figure 2-3 shows several of the fonts in the Times font family.

Figure 2-3    Fonts in the Times font family



ABCAbc123  
*ABCAbc123*  
**ABCAbc123**  
***ABCAbc123***

Styles, also called *traits*, that are available in Cocoa include variations such as bold, italic, condensed, expanded, narrow, small caps, poster fonts, and fixed pitch. Font descriptors in the Cocoa text system provide a font-matching capability, so that you can partially describe a font by creating a font descriptor with, for example, just a family name or weight, and you can then find all the fonts on the system that match the given trait.

## Text Layout

*Text layout* is the process of arranging glyphs on a display device, in an area called a *text view*, which represents an area similar to a page in traditional typesetting. The order in which glyphs are laid out relative to each other is called *text direction*. In English and other languages derived from Latin, glyphs are placed side by side to form words that are separated by spaces. Words are laid out in lines beginning at the top left of the text view proceeding from left to right until the text reaches the right side of the view. Text then begins a new line at the left side of the view under the beginning of the previous line, and layout proceeds in the same manner to the bottom of the text view.

In other languages, glyph layout can be quite different. For example, some languages lay out glyphs from right to left or vertically instead of horizontally. It is common, especially in technical writing, to mix languages with differing text direction, such as English and Hebrew, in the same line. Some writing systems even alternate

layout direction in every other line (an arrangement called boustrophedonic writing). Some languages do not group glyphs into words separated by spaces. Moreover, some applications call for arbitrary arrangements of glyphs; for example, in a graphic design context, a layout may require glyphs to be arranged on a nonlinear path.

To create lines from a string of glyphs, the layout engine must perform *line breaking* by finding a point at which to end one line and begin the next. In the Cocoa text system, you can specify line breaking at either word or glyph boundaries. In Roman text, a word broken between glyphs requires insertion of a hyphen glyph at the breakpoint.

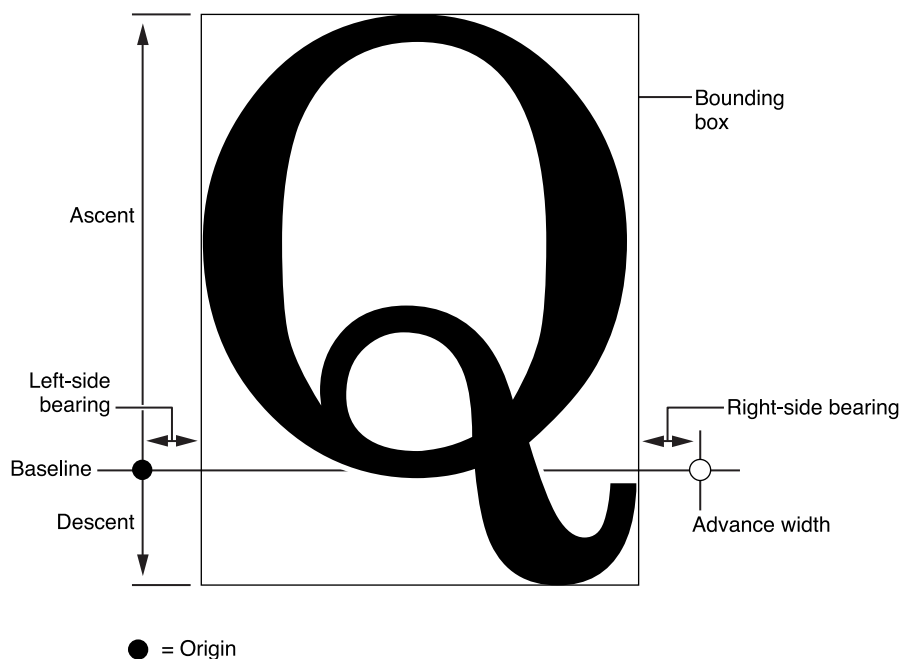
The Cocoa layout manager lays out glyphs along an invisible line called the *baseline*. In Roman text, the baseline is horizontal, and the bottom edge of most of the glyphs rest on it. Some glyphs extend below the baseline, including those for characters like “g” that have *descenders*, or “tails,” and large rounded characters like “O” that must extend slightly below the baseline to compensate for optical effects. Other writing systems place glyphs below or centered on the baseline. Every glyph includes an *origin* point that the layout manager uses to align it properly with the baseline.

Glyph designers provide a set of measurements with a font, called *metrics*, which describe the spacing around each glyph in the font. The layout manager uses these metrics to determine glyph placement. In horizontal text, the glyph has a metric called the *advance width*, which measures the distance along the baseline to the origin point of the next glyph. Typically there is some space between the origin point and the left side of the glyph, which is called the *left-side bearing*. There may also be space between the right side of the glyph and the point described by the advance width, which is called the *right-side bearing*. The vertical dimension of the glyph is provided by two metrics called the *ascent* and the *descent*. The ascent is the distance from the origin



(on the baseline) to the top of the tallest glyphs in the font. The descent, which is the distance below the baseline to the bottom of the font's deepest descender. The rectangle enclosing the visible parts of the glyph is called the *bounding rectangle* or bounding box. Figure 2-4 illustrates these metrics.

Figure 2-4 Glyph metrics



By default, in horizontal text, typesetters place glyphs side-by-side using the advance width, resulting in a standard interglyph space. However, in some combinations, text is made more readable by *kerning*, which is shrinking or stretching the space between two glyphs. A very common example of kerning occurs between an uppercase W and uppercase A, as shown in Figure 2-5. Type designers include kerning information in the metrics for a font. The Cocoa text system provides methods to turn kerning off, use the default settings provided with the font, or tighten or loosen the kerning throughout a selection of text.

Figure 2-5 Kerning

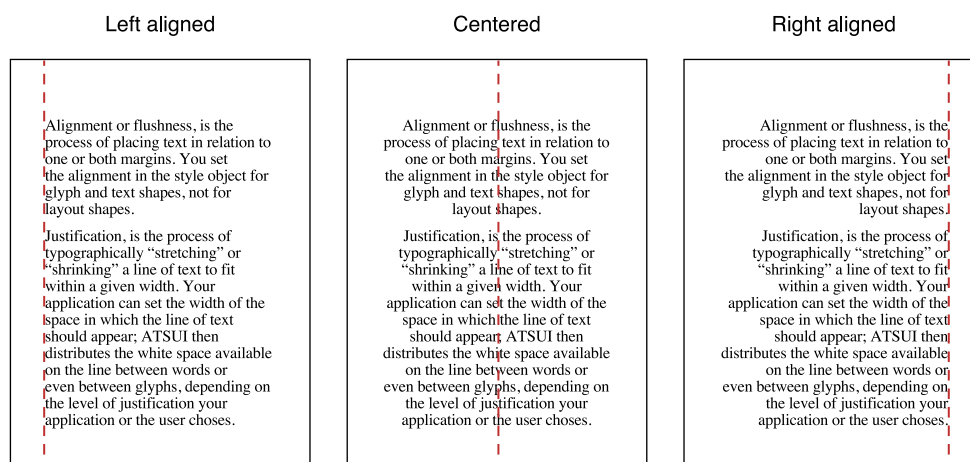


Type systems usually measure font metrics in units called *points*, which in OS X measure exactly 72 per inch. Adding the distance of the ascent and the descent of a font provides the font's *point size*.

Space added during typesetting between lines of type is called *leading*, after the slugs of lead used for that purpose in traditional metal-type page layout. (Leading is sometimes also called *linegap*.) The total amount of ascent plus descent plus leading provides a font's *line height*.

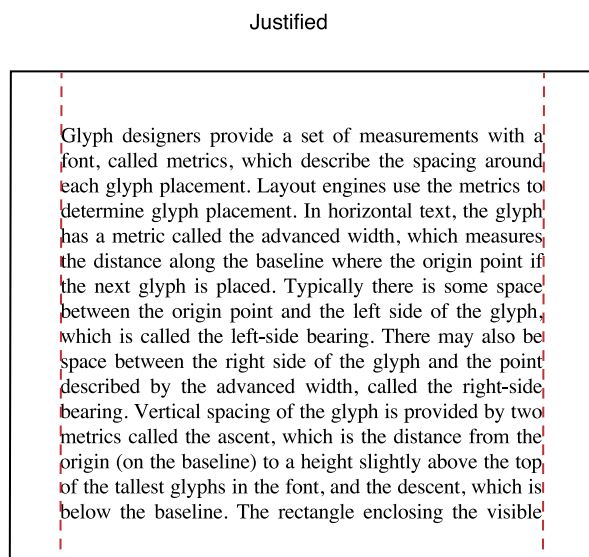
Although the preceding typographic concepts of type design may be somewhat esoteric, most people who have created documents on a computer or typewriter are familiar with the elements of text layout on a page. For example, the *margins* are the areas of white space between the edges of the page and the text area where the layout engine places glyphs. *Alignment* describes the way text lines are placed relative to the margins. For example, horizontal text can be aligned right, left, or centered, as shown in Figure 2-6.

**Figure 2-6** Alignment of text relative to margins



Lines of text can also be *justified*; for horizontal text the lines are aligned on both right and left margin by varying interword and interglyph spacing, as shown in Figure 2-7. The system performs alignment and justification, if requested, after the text stream has been broken into lines and hyphens added and other glyph substitutions made.

Figure 2-7 Justified text



# Text System Organization

The Cocoa text system is abstracted into a set of classes that interact to provide all of the text-handling features of the system. The classes represent specific functional areas with well-defined interfaces that enable application programs to modify the behavior of the system or even to replace parts with custom subclasses. The Cocoa text system is designed so that you don't need to learn about or interact with more of the system than is necessary to meet the needs of your application.

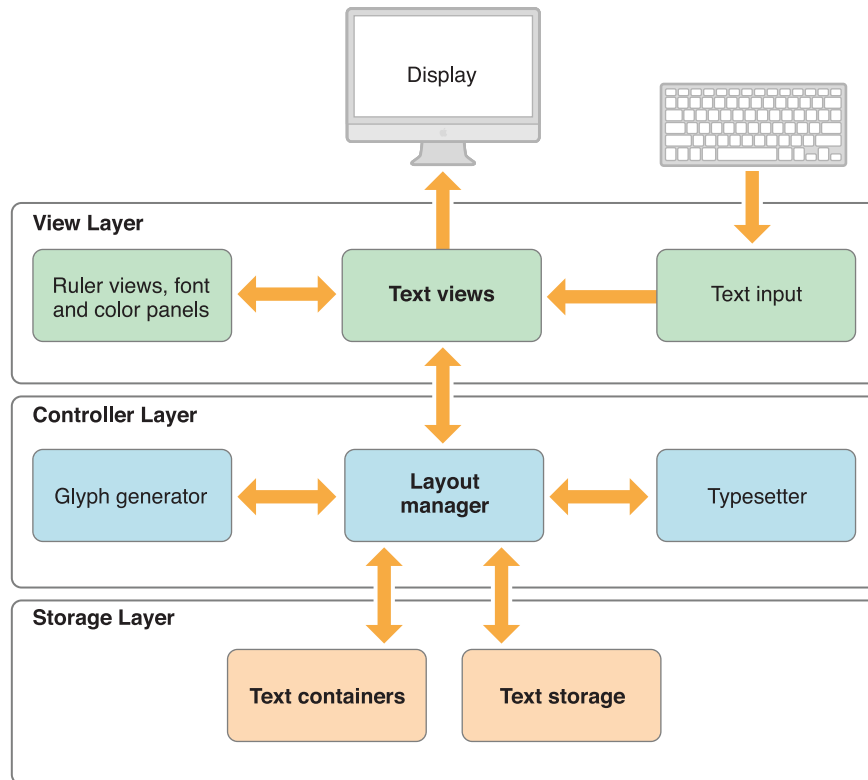
For most developers, the general-purpose programmatic interface of the `NSTextView` class is all you need to learn. `NSTextView` provides the user interface to the text system. See *NSTextView Class Reference* for detailed information about `NSTextView`.

If you need more flexible, programmatic access to the text, you'll need to learn about the storage layer and the `NSTextStorage` class. And, of course, to access all the available features, you can interact with any of the classes that support the text system.

## Functional Areas of the Cocoa Text System

Figure 3-1 shows the major functional areas of the text system with the user interface layer on top, the storage layer on the bottom, and, in the middle region, the components that lay out the text for display. These layers represent view, controller, and model concepts, respectively, as described in [MVC and the Text System](#) (page 25).

**Figure 3-1** Major functional areas of the Cocoa text system



The text classes exceed most other classes in the AppKit in the richness and complexity of their interface. One of their design goals is to provide a comprehensive set of text-handling features so that you rarely need to create a subclass. Among other things, a text object such as `NSTextView` can:

- Control whether the user can select or edit text.
- Control the font and layout characteristics of its text by working with the Font menu and Font panel (also called the *Fonts window*).
- Let the user control the format of paragraphs by manipulating a ruler.
- Control the color of its text and background.
- Wrap text on a word or character basis.
- Display graphic images within its text.

- Write text to or read text from files in the form of RTFD—Rich Text Format files that contain TIFF or EPS images, or attached files.
- Let another object, the delegate, dynamically control its properties.
- Let the user copy and paste text within and between applications.
- Let the user copy and paste font and format information between `NSTextView` objects.
- Let the user check the spelling of words in its text.

Graphical user-interface building tools (such as Interface Builder) may give you access to text objects in several different configurations, such as those found in the `NSTextField`, `NSForm`, and `NSScrollView` objects. These classes configure a text object for their own specific purposes. Additionally, all `NSTextField`, `NSForm`, or `NSButton` objects within the same window—in short, all objects that access a text object through associated cells—share the same text object, called the *field editor*. Thus, it's generally best to use one of these classes whenever it meets your needs, rather than create text objects yourself. But if one of these classes doesn't provide enough flexibility for your purposes, you can create text objects programmatically.

Text objects typically work closely with various other objects. Some of these—such as the delegate or an embedded graphic object—require some programming on your part. Others—such as the Font panel, spell checker, or ruler—take no effort other than deciding whether the service should be enabled or disabled.

To control layout of text on the screen or printed page, you work with the objects that link the `NSTextStorage` repository to the `NSTextView` that displays its contents. These objects are of the `NSLayoutManager` and `NSTextContainer` classes.

An `NSTextContainer` object defines a region where text can be laid out. Typically, a text container defines a rectangular area, but by creating a subclass of `NSTextContainer` you can create other shapes: circles, pentagons, or irregular shapes, for example. `NSTextContainer` isn't a user-interface object, so it can't display anything or receive events from the keyboard or mouse. It simply describes an area that can be filled with text, and it's not tied to any particular coordinate system. Nor does an `NSTextContainer` object store text—that's the job of an `NSTextStorage` object.

A layout manager object, of the `NSLayoutManager` class, orchestrates the operation of the other text handling objects. It intercedes in operations that convert the data in an `NSTextStorage` object to rendered text in an `NSTextView` object's display. It also oversees the layout of text within the areas defined by `NSTextContainer` objects.

## Class Hierarchy of the Cocoa Text System

In addition to the four principal classes in the text system—`NSTextStorage`, `NSLayoutManager`, `NSTextContainer`, `NSTextView`—there are a number of auxiliary classes and protocols. Figure 3-2 provides a more complete picture of the text system. Names between angle brackets, such as `<NSCopying>`, are protocols.

**Figure 3-2** Cocoa Text System Class Hierarchy



## MVC and the Text System

The Cocoa text system's architecture is both modular and layered to enhance its ease of use and flexibility. Its modular design reflects the Model-View-Controller paradigm (originating with Smalltalk-80) where the data, its visual representation, and the logic that links the two are represented by separate objects. In the case of the text system, `NSTextStorage` holds the model's text data, `NSTextContainer` models the geometry of the layout area, `NSTextView` presents the view, and `NSLayoutManager` intercedes as the controller to make sure that the data and its representation onscreen stay in agreement.

This factoring of responsibilities makes each component less dependent on the implementation of the others and makes it easier to replace individual components with improved versions without having to redesign the entire system. To illustrate the independence of the text-handling components, consider some of the operations that are possible using different subsets of the text system:

- Using only an `NSTextStorage` object, you can search text for specific characters, strings, paragraph styles, and so on.
- Using only an `NSTextStorage` object, you can programmatically operate on the text without incurring the overhead of laying it out for display.
- Using all the components of the text system except for an `NSTextView` object, you can calculate layout information, determine where line breaks occur, figure the total number of pages, and so forth.

The layering of the text system reduces the amount you have to learn to accomplish common text-handling tasks. In fact, many applications interact with the system solely through the API of the `NSTextView` class.

## Creating Text System Objects

There are two standard ways to create an object web of the four principal classes of the text system to handle text editing, layout, and display: in one case, the text view creates and owns the other objects; in the other case, you create all the objects explicitly and the text storage owns them.

### Text View Creates the Objects

You create and maintain a reference to an `NSTextView` object which automatically creates, interconnects, and owns the other text system objects. The majority of Cocoa apps use this technique and interact with the text system at a high level through `NSTextView`. You can create a text view and have it create the other text objects using Interface Builder, the graphical interface editor of Xcode, or you can do the same thing programmatically.

To create the text view object in Interface Builder, drag a Text View from the Object library onto your app window. When your app launches and its nib file is loaded, it instantiates an `NSTextView` object and embeds it in a scroll view. Behind the scenes, the text view object automatically instantiates and manages `NSTextContainer`, `NSLayoutManager`, and `NSTextStorage` objects.

To create the text view object programmatically and let it create and own the other objects, use the `NSTextView` initialization method `initWithFrame:`.

The text view ownership technique is the easiest and cleanest way to set up the text system object web. However, it creates a single flow of text which does not support pagination or complex layouts, as described in [Common Configurations](#) (page 28). For other configurations you must create the objects explicitly.

## Your App Creates the Objects Explicitly

You create all four text objects explicitly and connect them together, maintaining a reference only to the `NSTextStorage` object. The text storage object then owns and manages the other text objects in the web.

To create the text system objects explicitly and connect them together, use the steps shown in this section. This code could reside in the implementation of the `applicationDidFinishLaunching:` notification method of the app delegate, for example. It assumes that `textStorage` is an instance variable of the delegate object. It also assumes that `window` and `windowView` are properties of the app delegate representing outlets to the app's main window and its content view.

1. Create an `NSTextStorage` object in the normal way using the `alloc` and `init...` messages.

When you create the text system explicitly, you need to keep a reference only to this `NSTextStorage` object. The other objects of the system are owned by the text storage object, and they are released automatically by the system.

```
textStorage = [[NSTextStorage alloc]
               initWithString:@"Here's to the ones who see things
different."];
```

2. Create an `NSLayoutManager` object and connect it to the text storage object.

The layout manager needs a number of supporting objects—such as those that help it generate glyphs or position text within a text container—for its operation. It automatically creates these objects (or connects to existing ones) upon initialization.

```
NSLayoutManager *layoutManager;
layoutManager = [[NSLayoutManager alloc] init];
```

```
[textStorage addLayoutManager:layoutManager];
```

3. Create an `NSTextContainer` object, initialize it with a size, and connect it to the layout manager.

The size of the text container is the size of the view in which it is displayed—in this case `self.windowView` is the content view of the app's main window. Once you've created the text container, you add it to the list of containers that the layout manager owns. If your app has multiple text containers, you can create them and add them in this step, or you can create them lazily as needed.

```
NSTextContainer *textContainer;  
textContainer = [[NSTextContainer alloc]  
                initWithContainerSize:self.windowView.frame.size];  
[layoutManager addTextContainer:textContainer];
```

4. Create an `NSTextView` object, initialize it with a frame, and connect it to the text container.

When you create the text system's object web explicitly, you must use the `initWithFrame:textContainer:` method to initialize the text view. This initialization method does nothing more than initialize the receiver and set its text container (unlike `initWithFrame:`, which not only initializes the receiver, but automatically creates and interconnects its own web of text system objects). Each text view in the system is connected to its own text container.

```
NSTextView *textView;  
textView = [[NSTextView alloc]  
            initWithFrame:self.windowView.frame  
            textContainer:textContainer];
```

Once the `NSTextView` object has been initialized, you make it the content view of the window, which is then displayed. The `makeFirstResponder:` message makes the text view key, so that it accepts keystroke events.

```
[self.window setContentView:textView];  
[self.window makeKeyAndOrderFront:nil];  
[self.window makeFirstResponder:textView];
```

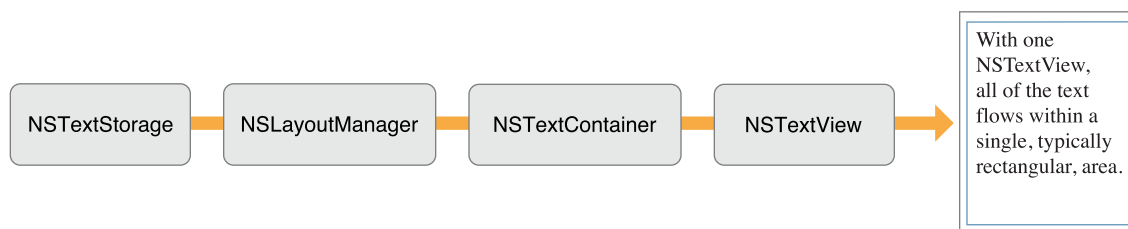
For simplicity, this code puts the text view directly into the window's content view. More commonly, text views are placed inside scroll views, as described in [Putting an NSTextView Object in an NSScrollView](#).

## Common Configurations

The following diagrams give you an idea of how you can configure objects of the four primary text system classes—`NSTextStorage`, `NSLayoutManager`, `NSTextContainer`, and `NSTextView`—to accomplish different text-handling goals.

To display a single flow of text, arrange the objects as shown in Figure 3-3.

**Figure 3-3** Text object configuration for a single flow of text



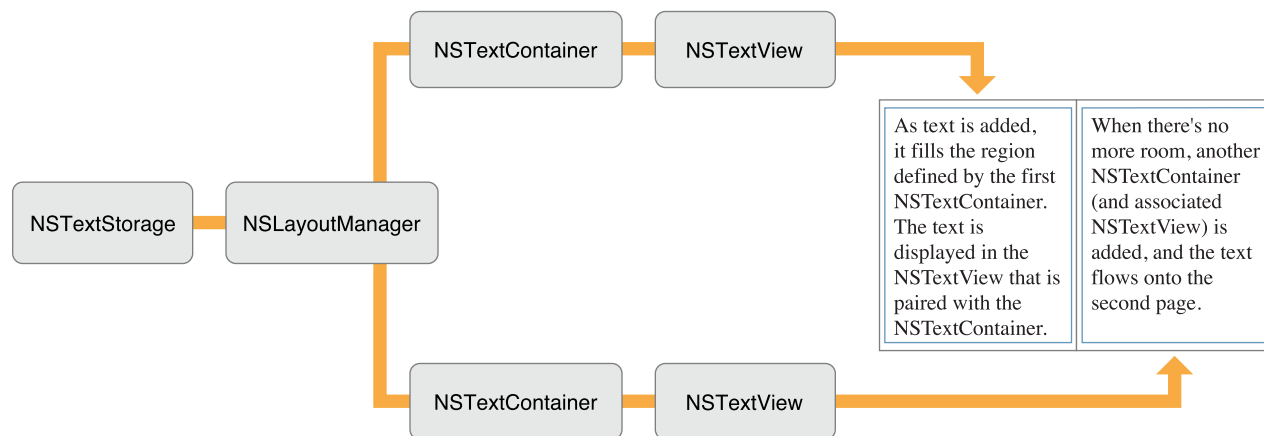
The `NSTextView` object provides the view that displays the glyphs, and the `NSTextContainer` object defines an area within that view where the glyphs are laid out. Typically in this configuration, the `NSTextContainer` object's vertical dimension is declared to be some extremely large value so that the container can accommodate any amount of text, while the `NSTextView` object is set to size itself around the text using the `setVerticallyResizable:` method defined by `NSText`, and given a maximum height equal to the `NSTextContainer` object's height. Then, with the text view embedded in an `NSScrollView` object, the user can scroll to see any portion of this text.

If the text container's area is inset from the text view's bounds, a margin appears around the text. The `NSLayoutManager` object, and other objects not pictured here, work together to generate glyphs from the `NSTextStorage` object's data and lay them out within the area defined by the `NSTextContainer` object.

This configuration is limited by having only one `NSTextContainer`-`NSTextView` pair. In such an arrangement, the text flows uninterrupted within the area defined by the `NSTextContainer`. Page breaks, multicolumn layout, and more complex layouts can't be accommodated by this arrangement.

By using multiple `NSTextContainer`-`NSTextView` pairs, more complex layout arrangements are possible. For example, to support page breaks, an application can configure the text objects as shown in Figure 3-4.

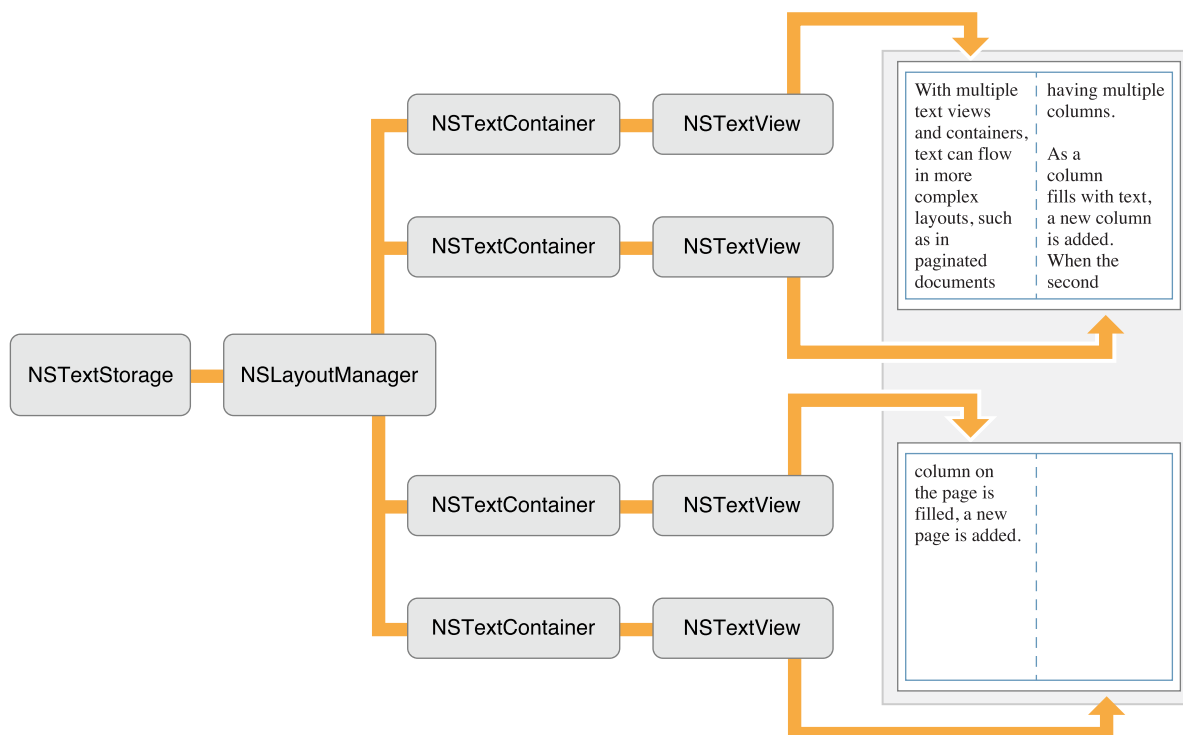
**Figure 3-4** Text object configuration for paginated text



Each `NSTextContainer`-`NSTextView` pair corresponds to a page of the document. The blue rectangle in Figure 3-4 represents a custom view object that your application provides as a background for the `NSTextView` objects. This custom view can be embedded in an `NSScrollView` object to enable the user to scroll through the document's pages.

A multicolumn document uses a similar configuration, as shown in Figure 3-5.

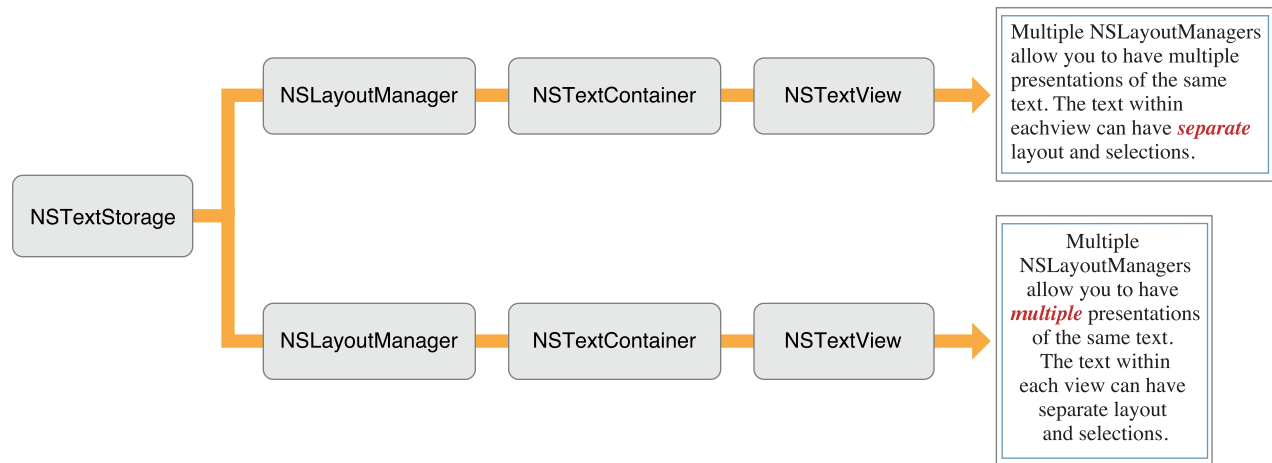
**Figure 3-5** Text object configuration for a multicolumn document



Instead of having one `NSTextView`-`NSTextContainer` pair correspond to a single page, there are now two pairs—one for each column on the page. Each `NSTextContainer`-`NSTextView` pair controls a portion of the document. As the text is displayed, glyphs are first laid out in the top-left view. When there is no more room in that view, the `NSLayoutManager` object informs its delegate that it has finished filling the container. The delegate can check whether there's more text that needs to be laid out and add another `NSTextContainer` and `NSTextView` if necessary. The `NSLayoutManager` object proceeds to lay out text in the next container, notifies the delegate when finished, and so on. Again, a custom view (depicted as a blue rectangle) provides a canvas for these text columns.

Not only can you have multiple `NSTextContainer`-`NSTextView` pairs, you can also have multiple `NSLayoutManager` objects accessing the same text storage. Figure 3-6 illustrates the simplest arrangement with multiple layout managers.

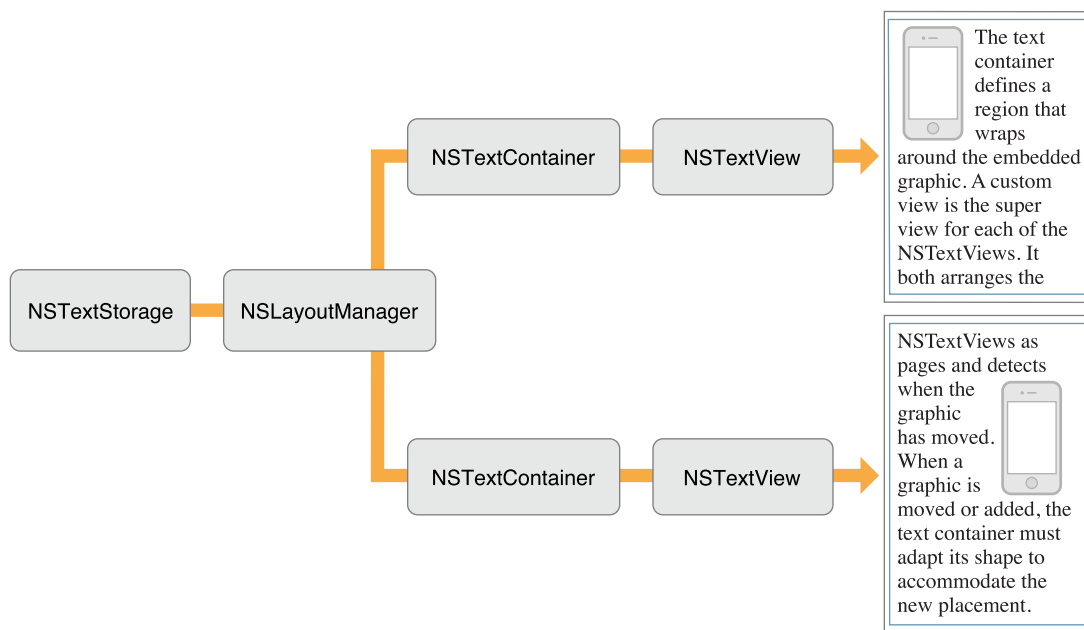
**Figure 3-6** Text object configuration for multiple views of the same text



The effect of this arrangement is to give multiple views on the same text. If the user alters the text in the top view, the change is immediately reflected in the bottom view (assuming the location of the change is within the bottom view's bounds).

Finally, complex page layout requirements, such as permitting text to wrap around embedded graphics, can be achieved by a configuration that uses a custom subclass of `NSTextContainer`. This subclass defines a region that adapts its shape to accommodate the graphic image and uses the object configuration shown in Figure 3-7.

**Figure 3-7** Text object configuration with custom text containers



See *Text Layout Programming Guide* for information about how the text system lays out text.



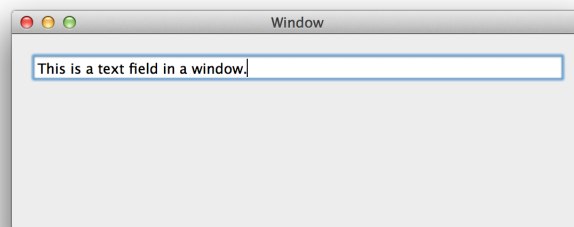
# Text Fields, Text Views, and the Field Editor

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

## Text Fields

A text field is a user interface control object instantiated from the `NSTextField` class. Figure 4-1 shows a text field. Text fields display small amounts of text, typically (although not necessarily) a single line. Text fields provide places for users to enter text responses, such as search parameters. Like all controls, a text field has a target and an action, and it performs validation on its value when edited. If the value isn't valid, it sends a special error action message to its target.

Figure 4-1 A text field



A text field is implemented by two classes: `NSTextFieldCell`, the cell which does most of the work, and `NSTextField`, the control that contains that cell. Every method in `NSTextFieldCell` has a cover in `NSTextField`. (A cover is a method of the same name that calls the original method.) An `NSTextField` object can have a delegate that responds to such delegate methods as `textShouldEndEditing:`.

By default, text fields send their action message when editing ends—that is, when the user presses Return or moves focus to another control. You can control a text field's shape and layout, the font and color of its text, background color, whether the text is editable or read-only, whether it is selectable or not (if read-only), and whether the text scrolls or wraps when the text exceeds the text field's visible area.

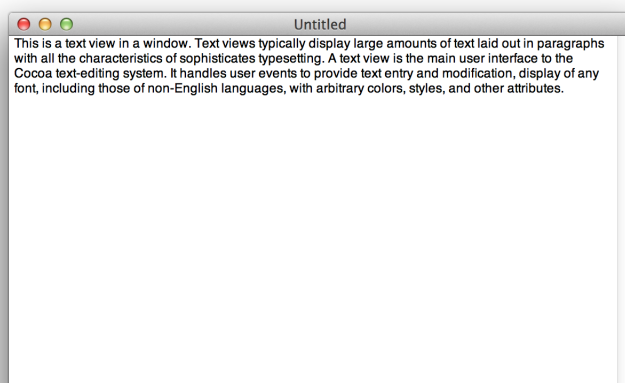
To create a secure text field for password entry, use `NSSecureTextField`, a subclass of `NSTextField`. Secure text fields display bullets in place of characters entered by the user, and they do not allow cutting or copying of their contents. You can get the text field's value using the `stringValue` method, but users have no access to the value. See [Why Use a Custom Field Editor?](#) (page 70) for information about the implementation of secure text fields.

The usual way to instantiate a text field is to drag an `NSTextField` or `NSSecureTextField` object from the Interface Builder objects library and place it in a window of your application's user interface. You can link text fields together in their window's key view loop by setting each field's `nextKeyView` outlet in the Connections pane of the Inspector window, so that users can press Tab to move keyboard focus from one field to the next in the order you specify.

## Text Views

Text views are user interface objects instantiated from the `NSTextView` class. Figure 4-2 shows a text view. Text views typically display multiple lines of text laid out in paragraphs with all the characteristics of sophisticated typesetting. A text view is the main user interface to the Cocoa text-editing system. It handles user events to provide text entry and modification, and to display any font, including those of non-English languages, with arbitrary colors, styles, and other attributes.

**Figure 4-2** A text view



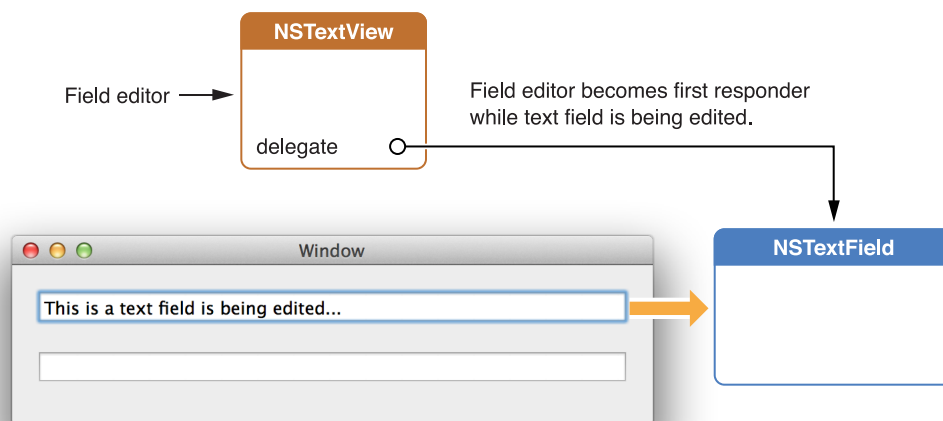
The Cocoa text system supports text views with many other underlying objects providing text storage, layout, font and attribute manipulation, spell checking, undo and redo, copy and paste, drag and drop, saving of text to files, and other features. `NSTextView` is a subclass of `NSText`, which is a separate class for historical reasons.

You don't instantiate `NSText`, although it declares many of the methods you use with `NSTextView`. When you put an `NSTextView` object in an `NSWindow` object, you have a full-featured text editor whose capabilities are provided “for free” by the Cocoa text system.

## The Field Editor

The field editor is a single `NSTextView` object that is shared among all the controls, including text fields, in a window. This text view object inserts itself into the view hierarchy to provide text entry and editing services for the currently active text field. When the user shifts focus to a text field, the field editor begins handling keystroke events and display for that field. Figure 4-3 illustrates the field editor in relation to the text field it is editing.

**Figure 4-3** The field editor



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

For more information about the field editor, see [Working with the Field Editor](#) (page 68).

# Text Attributes

The Cocoa text system handles five kinds of text attributes: character attributes, temporary attributes, paragraph attributes, glyph attributes, and document attributes. Character attributes include traits such as font, color, and subscript, which can be associated with an individual character or a range of characters. Temporary attributes are character attributes that apply only to a particular layout and are not persistent. Paragraph attributes are traits such as indentation, tabs, and line spacing. Glyph attributes affect the way the layout manager renders glyphs and include traits such as overstriking the previous glyph. Document attributes include document-wide traits such as paper size, margins, and view zoom percentage.

This chapter provides a brief introduction to the various types of text attributes with cross-references to more detailed documentation.

## Character Attributes

The text system stores character attributes persistently in attributed strings along with the characters to which they apply. The text system's predefined character attributes control the appearance of characters (font, foreground color, background color, and ligature handling) and their placement (superscript, baseline offset, and kerning).

Two special character attributes pertain to links and attachments. A link attribute points to a URL (encapsulated in an `NSURL` object) or any other object of your choice. An attachment attribute is associated with a special attachment character and points to an `NSFileWrapper` object containing the attached file or in-memory data.

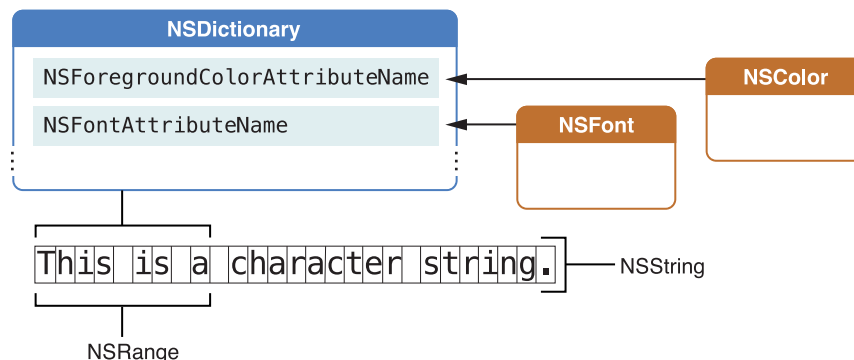
The predefined character attribute `NSCharacterShapeAttributeName` enables you to set a value for the character shape feature used in font rendering by Apple Type Services. This feature is currently used to specify traditional shapes in Chinese and Japanese scripts, but font developers could use it for other scripts as well.

The predefined character attribute `NSGlyphInfoAttributeName` points to an `NSGlyphInfo` object that provides a means to override the standard glyph generation process and substitute a specified glyph over the attribute's range.

## Storing Character Attributes

An attributed string stores character attributes as key-value pairs in `NSDictionary` objects. The key is an attribute name, represented by an identifier (an `NSString` constant) such as `NSForegroundColorAttributeName`. Figure 5-1 shows an attributed string with an attribute dictionary applied to a range within the string.

Figure 5-1 The composition of an `NSAttributedString` object including its attributes dictionary



Conceptually, each character in an attributed string has an associated dictionary of attributes. Typically, however, an attribute dictionary applies to a longer range of characters. The `NSAttributedString` class provides methods that take a character index and return the associated attribute dictionary and the range to which its attribute values apply. See [Accessing Attributes](#) for more information about using these methods.

In addition to the predefined attributes, you can assign any attribute key-value pair you wish to a range of characters. You add the attributes to the appropriate character range in the `NSTextStorage` object using the `NSMutableAttributedString` method `addAttribute:value:range:`. You can also create an `NSDictionary` object containing the names and values of a set of custom attributes and add them to the character range in a single step using the `addAttributes:range:` method. To make use of your custom attributes, you need a custom subclass of `NSLayoutManager` that understands what to do with them. Your subclass should override the `drawGlyphsForGlyphRange:atPoint:` method first to call the superclass to draw the glyph range, then draw your own attributes on top, or else draw the glyphs entirely your own way.

## Attribute Fixing

Editing attributed strings can cause inconsistencies that must be cleaned up by *attribute fixing*. The AppKit extensions to `NSMutableAttributedString` define `fix...` methods to fix inconsistencies among attachment, font, and paragraph attributes. These methods ensure that attachments don't remain after their attachment characters are deleted, that font attributes apply only to characters available in that font, and that paragraph attributes are consistent throughout paragraphs.

See *Attributed String Programming Guide* for more details about character attributes and attribute fixing.

## Temporary Attributes

Temporary attributes are character attributes that are not stored with an attributed string. Rather, the layout manager assigns temporary attributes during the layout process and uses them only when drawing the text. For example, you can use temporary attributes to underline misspelled words or color key words in a programming language.

Temporary attributes affect only the appearance of text, not the way in which it is laid out. You store temporary attributes in an `NSDictionary` object using the same keys as regular character attributes, or using custom attribute names (if you have an `NSLayoutManager` subclass that can handle them). Then you add the attributes using an `NSLayoutManager` method such as `addTemporaryAttributes:forCharacterRange:`. By default, the only temporary attributes recognized are those affecting color and underlines. During layout, temporary attributes supersede regular character attributes. So, for example, if a character has a stored `NSForegroundColorAttributeName` value specifying blue and a temporary attribute of the same identifier specifying red, then the character is rendered in red.

For more information on temporary attributes, see *NSLayoutManager Class Reference*.

## Paragraph Attributes

Paragraph attributes affect the way the layout manager arranges lines of text into paragraphs on a page. The text system encapsulates paragraph attributes in objects of the `NSParagraphStyle` class. The value of one of the predefined character attributes, `NSParagraphStyleAttributeName`, points to an `NSParagraphStyle` object containing the paragraph attributes for that character range. Attribute fixing ensures that only one `NSParagraphStyle` object pertains to the characters throughout each paragraph.

Paragraph attributes include traits such as alignment, tab stops, line-breaking mode, and line spacing (also known as leading). Users of text applications control paragraph attributes through ruler views, defined by the `NSRulerView` class.

See *Ruler and Paragraph Style Programming Topics* for more details about paragraph attributes.

## Glyph Attributes

Glyphs are the concrete representations of characters that the text system actually draws on a display. Glyph attributes are not complex data structures like character attributes but are simply integer values that the layout manager uses to denote special handling for particular glyphs during rendering.

The text system uses glyph attributes rarely, and applications should have little reason to be concerned with them. Nonetheless, `NSLayoutManager` provides public methods that handle glyph attributes, so you can use subclasses to extend the mechanism to handle custom glyph attributes if necessary.

The glyph generator sets built-in glyph attributes as required on glyphs during typesetting. They are maintained in the layout manager's glyph cache during that process, but they are not stored persistently. Two examples of glyph attributes are the elastic attribute for spaces, used to lay out fully justified text, and the attribute `NSGlyphAttributeInscribe`, which is used for situations such as drawing an umlaut over a character when the font does not include a built-in character-with-umlaut.

For more information about glyph attributes, see the description of the `setIntAttribute:value:forGlyphAtIndex:` method in *NSLayoutManager Class Reference*.

## Document Attributes

Document attributes pertain to a document as a whole. Document attributes include traits such as paper size, margins, and view zoom percentage. Although the text system has no built-in mechanism to store document attributes, initialization methods such as `initWithRTF:documentAttributes:` can populate an `NSDictionary` object that you provide with document attributes derived from a stream of RTF or HTML data. Conversely, methods that write RTF data, such as `RTFFromRange:documentAttributes:`, write document attributes if you pass a reference to an `NSDictionary` object containing them with the message.

See *RTF Files and Attributed Strings* and *NSAttributedString AppKit Additions Reference* for more information.

# Font Handling

This chapter explains how the Cocoa text system deals with fonts. It explains how to use the Font panel in your application, how to work directly with font objects, and how to work with the font manager.

## The Font Panel

The Font panel, also called the *Fonts window*, is a user interface object that displays a list of available font families and styles, letting the user preview them and change the font used to display text. Text objects, such as `NSTextView`, work with `NSFontPanel` and `NSFontManager` objects to implement the AppKit's font conversion system. By default, a text object keeps the Font panel updated with the first font in its selection, or with its typing attributes. It also changes the font in which it displays text 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 object or to all the text in a plain text object.

`NSFontManager` is the hub for font conversion—that is, changing the traits of a font object, such as its size or typeface. The font manager receives the messages from the Font panel and sends messages up the responder chain for action on the text objects.

Normally, an application's Font panel displays all the standard fonts available on the system. If this isn't appropriate for your application—for example, if only fixed-pitch fonts should be used—you can assign a delegate to the `NSFontPanel` object to filter the available fonts. Before the `NSFontPanel` object adds a particular font family or face to its list, the `NSFontPanel` object asks its delegate to confirm the addition by sending the delegate a `fontManager:willIncludeFont:` message. If the delegate returns `TRUE` (or doesn't implement this method), the font is added. If the delegate returns `FALSE`, the font isn't added. This method must be invoked before the loading of the main nib file.

## Creating a Font Panel

In general, you add the facilities of the Font panel to your application, along with the `NSFontManager` object and the Font menu, through which the user opens the Font panel, using Interface Builder. You do this by dragging a Font or Format menu (which contains a Font submenu) into one of your application's menus. At runtime, the Font panel object is created and hooked into the font conversion system. You can also create (or access) the Font panel using the `sharedFontPanel` class method.



You can add a custom view object to an `NSFontPanel` object using `setAccessoryView:`, allowing you to add custom controls to the Font panel. You can also limit the fonts displayed (by default, all fonts are displayed) by assigning a delegate to the application's font manager object.

If you want the `NSFontManager` object to instantiate the Font panel from some class other than `NSFontPanel`, use the `NSFontManager` class method `setFontPanelFactory:`. See [Converting Fonts Manually](#) for more information on using the font conversion system.

## Using the Font Panel

You can enable the interaction between a text object and the Font panel using the `NSTextView` (or `NSText`) method `setUsesFontPanel:` method. Doing so is recommended for a text view that serves as a field editor, for example.

You can use the Font panel on objects other than standard text fields. The `NSFontManager` method `setAction:` sets the action (specified by a selector) that is sent up the first responder chain when a new font is selected. The default selector is `changeFont:`. Any object that receives this message from the responder chain should send a `convertFont:` message back to the `NSFontManager` to convert the font in a manner the user has specified.

This example assumes there is only one font selected:

```
- (void)changeFont:(id)sender
{
    NSFont *oldFont = [self font];
    NSFont *newFont = [sender convertFont:oldFont];
    [self setFont:newFont];
    return;
}
```

If multiple fonts are selected, `changeFont:` must send conversion messages for each selected font. This is useful for objects such as table views, which do not inherently respond to messages from the Font panel.

## Working with Font Objects

A computer font is a data file in a format such as OpenType or TrueType, containing information describing a set of glyphs, as described in [Characters and Glyphs](#) (page 13), and various supplementary information used in glyph rendering. The `NSFont` class provides the interface for getting and setting font information. An `NSFont` instance provides access to the font's characteristics and glyphs. The text system combines character information

with font information to choose the glyphs used during text layout. You use font objects by passing them to methods that accept them as a parameter. Font objects are immutable, so it is safe to use them from multiple threads in your app.

You don't create font objects using the `alloc` and `init` methods; instead, you use methods such as `fontWithName:matrix:` or `fontWithName:size:` to look up an available font and alter its size or matrix to your needs. These methods check for an existing font object with the specified characteristics, returning it if there is one. Otherwise, they look up the font data requested and create the appropriate object. You can also use font descriptors to create fonts. See [Font Descriptors](#) (page 42).

`NSFont` also defines a number of methods for specifying standard system fonts, such as `systemFontOfSize:`, `userFontOfSize:`, and `messageFontOfSize:`. To request the default size for these standard fonts, pass `0` or a negative number as the font size. The standard system font methods are listed in [Querying Standard Font Variations](#) (page 44).

## Font Descriptors

Font descriptors, instantiated from the `NSFontDescriptor` class, provide a way to describe a font with a dictionary of attributes. The font descriptor can then be used to create or modify an `NSFont` object. In particular, you can make an `NSFont` object from a font descriptor, you can get a descriptor from an `NSFont` object, and you can change a descriptor and use it to make a new font object. You can also use a font descriptor to specify custom fonts provided by an app.

Font descriptors provide a font matching capability by which you can partially describe a font by creating a font descriptor with, for example, just a family name. You can then find all the available fonts on the system with a matching family name using `matchingFontDescriptorsWithMandatoryKeys:`.

Font descriptors can be archived, which is preferable to archiving a font object because a font object is immutable and provides access to a particular font installed on a particular system. A font descriptor, on the other hand, describes a font in terms of its characteristics, its attributes, and provides access at runtime to the fonts currently available that match those attributes.

That is, you can use font descriptors to query the system for available fonts that match particular attributes, and then create instances of `NSFont` matching those attributes, such as names, traits, languages, and other features. For example, you can use a font descriptor to retrieve all the fonts matching a given font family name, using the PostScript family names (as defined by the CSS standard) as shown in Listing 6-1.

**Listing 6-1** Font family name matching

```
NSFontDescriptor *helveticaNeueFamily =  
    [NSFontDescriptor fontDescriptorWithFontAttributes:
```

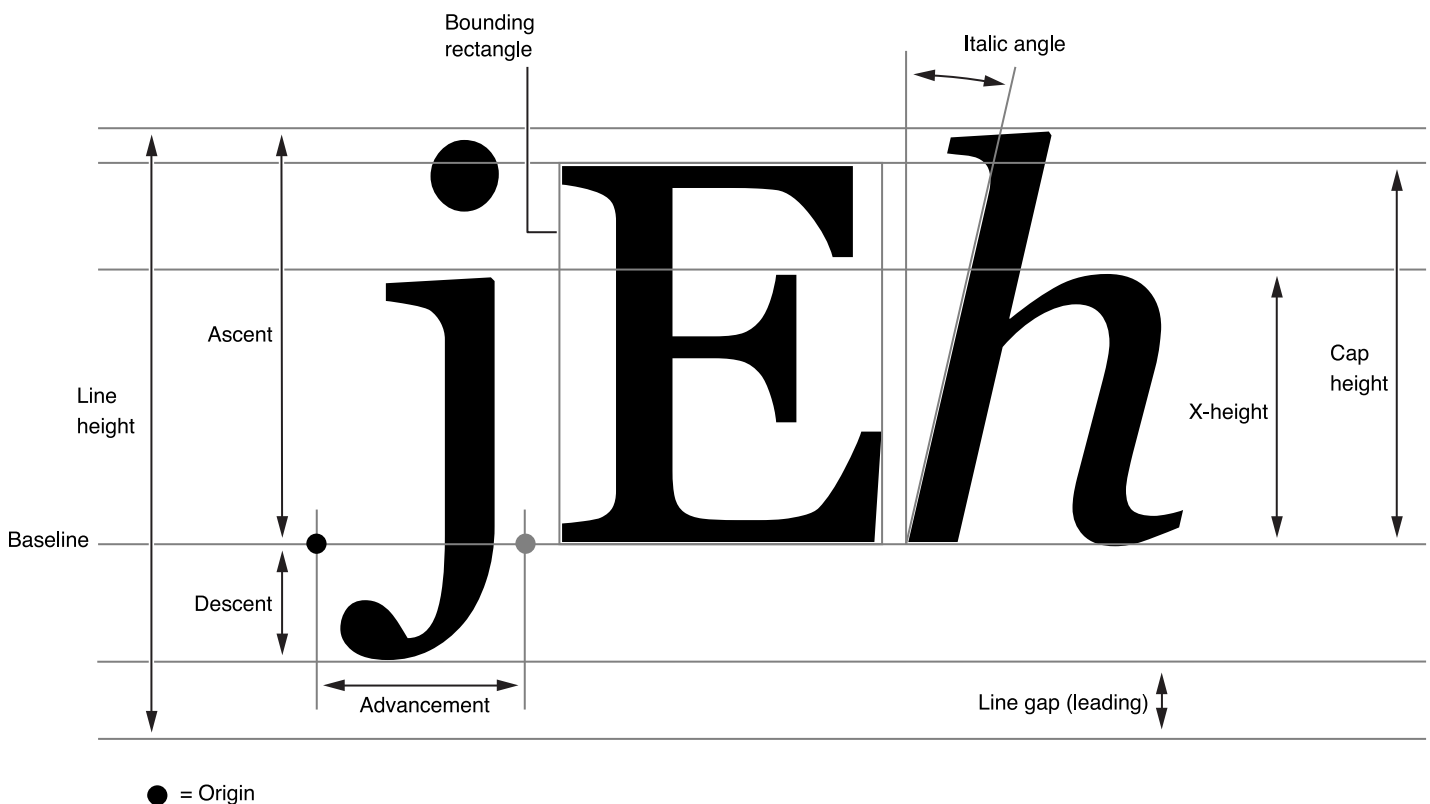
```
@{ NSFontFamilyAttribute: @"Helvetica Neue" }];  
NSArray *matches =  
[helveticaNeueFamily matchingFontDescriptorsWithMandatoryKeys: nil];
```

The `matchingFontDescriptorsWithMandatoryKeys:` method as shown returns an array of font descriptors for all the Helvetica Neue fonts on the system, such as `HelveticaNeue`, `HelveticaNeue-Medium`, `HelveticaNeue-Light`, `HelveticaNeue-Thin`, and so on. (PostScript font and font family names for the fonts available on a given OS X installation are displayed by the Font Book app in the Font Info window.)

## Querying Font Metrics

`NSFont` defines a number of methods for accessing a font's metrics information, when that information is available. Methods such as `boundingRectForGlyph:`, `boundingRectForFont`, `xHeight`, and so on, all correspond to standard font metrics information. Figure 6-1 shows how the font metrics apply to glyph dimensions, and Table 6-1 lists the method names that correlate with the metrics. See the various method descriptions for more specific information.

Figure 6-1 Font metrics



**Table 6-1** Font metrics and related NSFont methods

Font metric	Methods
Advancement	advancementForGlyph:, maximumAdvancement
X-height	xHeight
Ascent	ascender
Bounding rectangle	boundingRectForFont, boundingRectForGlyph:
Cap height	capHeight
Line height	defaultLineHeightForFont, pointSize, labelFontSize, smallSystemFontSize, systemFontSize, systemFontSizeForControlSize:
Descent	descender
Italic angle	italicAngle

## Querying Standard Font Variations

Using the methods of NSFont listed in Table 6-2, you can query all of the standard font variations. To request the default font size for the standard fonts, you can either explicitly pass in default sizes (obtained from class methods such as `systemFontSize` and `labelFontSize`), or pass in 0 or a negative value.

**Table 6-2** Standard font methods

Font	Methods
System font	<code>[NSFont systemFontOfSize:[NSFont systemFontSize]]</code>
Emphasized system font	<code>[NSFont boldSystemFontOfSize:[NSFont systemFontSize]]</code>
Small system font	<code>[NSFont systemFontOfSize:[NSFont smallSystemFontSize]]</code>
Emphasized small system font	<code>[NSFont boldSystemFontOfSize:[NSFont smallSystemFontSize]]</code>
Mini system font	<code>[NSFont systemFontSizeForControlSize: NSMiniControlSize]</code>
Emphasized mini system font	<code>[NSFont boldSystemFontOfSize:[NSFont systemFontSizeForControlSize: NSMiniControlSize]]</code>

Font	Methods
Application font	<code>[UIFont systemFontOfSize:-1.0]</code>
Application fixed-pitch font	<code>[UIFont systemFontOfSize:-1.0]</code>
Label Font	<code>[UIFont systemFontOfSize:[UIFont systemFontOfSize]]</code>

## Characters, Glyphs, and the Layout Manager

Characters are conceptual entities that correspond to units of written language. Generally, a glyph is the concrete, rendered image of a character. ([Typographical Concepts](#) (page 13) presents a more detailed discussion of characters and glyphs.)

In English, there's often a one-to-one mapping between characters and glyphs, but that is not always the case. For example, the glyph “ö” can be the result of two characters, one representing the base character “o” and the other representing the *umlaut* diacritical mark “”. A user of a word processor can press an arrow key one time to move the insertion point from one side of the “ö” glyph to the other; however, the current position in the character stream must be incremented by two to account for the two characters that make up the single glyph.

Thus, the text system must manage two related but different streams of data: the stream of characters (and their attributes) and the stream of glyphs that are derived from these characters. The `NSTextStorage` object stores the attributed characters, and the `NSLayoutManager` object stores the derived glyphs. Finding the correspondence between these two streams is the responsibility of the layout manager. For example, when a user selects a range of text, working with glyphs displayed on screen, the layout manager must determine which range of characters corresponds to the selection.

When characters are deleted, some glyphs may have to be redrawn. For example, if the user deletes the characters “ee” from the word “feel”, then the “f” and “l” are now adjacent and can be represented by the “fl” ligature rather than the two glyphs “f” and “l”. The `NSLayoutManager` object directs a glyph generator object to generate new glyphs as needed. Once the glyphs are regenerated, the text must be laid out and displayed. Working with the `NSTextContainer` object and other objects of the text system, the layout manager determines where each glyph appears in the text view. Finally, the text view renders the text.

Because an `NSLayoutManager` object is central to the operation of the text system, it also serves as the repository of information shared by various components of the system. For more information about `NSLayoutManager`, refer to *NSLayoutManager Class Reference* and to *Text Layout Programming Guide*.

## Getting the View Coordinates of a Glyph

Glyph locations are figured relative to the origin of the bounding rectangle of the line fragment in which they are laid out. To get the rectangle of the glyph's line fragment in its container coordinates, use the `NSLayoutManager` method `lineFragmentRectForGlyphAtIndex:effectiveRange:`. Then add the origin of that rectangle to the location of the glyph returned by `locationForGlyphAtIndex:` to get the glyph location in container coordinates.

The following code fragment from the *CircleView* sample code project illustrates this technique.

```
usedRect = [layoutManager usedRectForTextContainer:textContainer];
NSRect lineFragmentRect = [layoutManager lineFragmentRectForGlyphAtIndex:glyphIndex
                                     effectiveRange:NULL];
NSPoint viewLocation, layoutLocation = [layoutManager
                                     locationForGlyphAtIndex:glyphIndex];
// Here layoutLocation is the location (in container coordinates) where the glyph
// was laid out.
layoutLocation.x += lineFragmentRect.origin.x;
layoutLocation.y += lineFragmentRect.origin.y;
```

## Working with the Font Manager

Any object that records fonts that the user can change should tell the font manager what the font of its selection is whenever it becomes the first responder and whenever its selection changes while it's the first responder. The object does so by sending the shared font manager a `setSelectedFont:isMultiple:` message. It should pass in the first font of the selection, along with a flag indicating whether there's more than one font.

The font manager uses this information to update the Font panel and Font menu to reflect the font in the selection. For example, suppose the font is Helvetica Oblique 12.0 point. In this case, the Font panel selects that font and displays its name; the Font menu adds a check mark before its Italic command; if there's no Bold variant of Helvetica available, the Bold menu item is disabled; and so on.

## Creating a Font Manager

You normally set up a font manager and the Font menu using Interface Builder. However, you can also do so programmatically by getting the shared font manager instance and having it create the standard Font menu at runtime, as in this example:

```
NSFontManager *fontManager = [NSFontManager sharedFontManager];
```

```
NSMenu *fontMenu = [fontManager fontMenu:YES];
```

You can then add the Font menu to your application's menus. Once the Font menu is installed, your application automatically gains the functionality of both the Font menu and the Font panel.

## Handling Font Changes

The user normally changes the font of the selection by manipulating the Font panel (also called the Fonts window) and the Font menu. These objects initiate the intended change by sending an action message to the font manager. There are four font-changing action methods:

```
addFontTrait:  
removeFontTrait:  
modifyFont:  
modifyFontViaPanel:
```

The first three cause the font manager to query the sender of the message in order to determine which trait to add or remove, or how to modify the font. The last causes the font manager to use the settings in the Font panel to modify the font. The font manager records this information and uses it in later requests to convert fonts.

When the font manager receives an `addFontTrait:` or `removeFontTrait:` message, it queries the sender with a `tag` message, interpreting the return value as a trait mask for use with `convertFont:toHaveTrait:` or `convertFont:toNotHaveTrait:`, as described in [Converting Fonts Manually](#) (page 48). The Font menu commands Italic and Bold, for example, have trait mask values of `NSItalicFontMask` and `NSBoldFontMask`, respectively. See the Constants section in *NSFontManager Class Reference* for a list of trait mask values.

When the font manager receives a `modifyFont:` message, it queries the sender with a `tag` message and interprets the return value as a particular kind of conversion to perform, via the various conversion methods described in [Converting Fonts Manually](#) (page 48). For example, a button whose tag value is `SizeUpFontAction` causes the font manager's `convertFont:` method to increase the size of the `NSFont` object passed as the parameter. See the `NSFontManager` method `modifyFont:` for a list of conversion tag values.

For `modifyFontViaPanel:`, the font manager sends the application's Font panel a `panelConvertFont:` message. The Font panel in turn uses the font manager to convert the font provided according to the user's choices. For example, if the user selects only the font family in the Font panel (perhaps Helvetica), then for whatever fonts are provided to `panelConvertFont:`, only the family is changed: Courier Medium 10.0 point becomes Helvetica Medium 10.0 point, and Times Italic 12.0 point becomes Helvetica Oblique 12.0 point.

The font manager responds to a font-changing action method by sending a `changeFont:` action message up the responder chain. A text-bearing object that receives this message should have the font manager convert the fonts in its selection by invoking `convertFont:` for each font and using the `NSFont` object returned. The `convertFont:` method uses the information recorded by the font-changing action method, such as `addFontTrait:`, modifying the font provided appropriately. (There's no way to explicitly set the font-changing action or trait; instead, you use the methods described in [Converting Fonts Manually](#) (page 48).)

This simple example assumes there's only one font in the selection:

```
- (void)changeFont:(id)sender
{
    NSFont *oldFont = [self selectionFont];
    NSFont *newFont = [sender convertFont:oldFont];
    [self setSelectionFont:newFont];
    return;
}
```

Most text-bearing objects have to scan the selection for ranges with different fonts and invoke `convertFont:` for each one.

## Converting Fonts Manually

`NSFontManager` defines a number of methods for explicitly converting particular traits and characteristics of a font. Table 6-3 lists these conversion methods.

**Table 6-3** Font conversion methods

Methods	Behavior
<code>convertFont:toFace:</code>	Alters the basic design of the font provided. Requires a fully specified typeface name, such as "Times-Roman" or "Helvetica-BoldOblique".
<code>convertFont: toFamily:</code>	Alters the basic design of the font provided. Requires only a family name, such as "Times" or "Helvetica".
<code>convertFont:toHaveTrait:</code>	Uses a trait mask to add a single trait such as Italic, Bold, Condensed, or Extended.
<code>convertFont:toNotHaveTrait:</code>	Uses a trait mask to remove a single trait such as Italic, Bold, Condensed, or Extended.



Methods	Behavior
<code>convertFont:toSize:</code>	Returns a font of the requested size, with all other characteristics the same as those of the original font.
<code>convertWeight: ofFont:</code>	Either increases or decreases the weight of the font provided, according to a Boolean flag. Font weights are typically indicated by a series of names, which can vary from font to font. Some go from Light to Medium to Bold, while others have Book, SemiBold, Bold, and Black. This method offers a uniform way of incrementing and decrementing any font's weight.

Each method returns a transformed version of the font provided, or the original font if it can't be converted.

The default implementation of font conversion is very conservative, making a change only if no other trait or aspect is affected. For example, if you try to convert Helvetica Oblique 12.0 point by adding the Bold trait, and only Helvetica Bold is available, the font isn't converted. You can create a subclass of `NSFontManager` and override the conversion methods to perform less conservative conversion, perhaps using Helvetica Bold in this case and losing the Oblique trait.

In addition to the font-conversion methods, `NSFontManager` defines `fontWithFamily:traits:weight:size:` to construct a font with a given set of characteristics. If you don't care to make a subclass of `NSFontManager`, you can use this method to perform approximate font conversions yourself.

## Setting Font Styles and Traits

This section shows how to programmatically set font styles, such as bold or italic, and font attributes, such as underlining, in an attributed string.

Underlining is an attribute that can be easily set on an attributed string, using the `NSUnderlineStyleAttributeName` constant, as explained in *NSMutableAttributedString Class Reference*. Use the following method:

```
- (void)addAttribute:(NSString *)name value:(id)value range:(NSRange)aRange
```

Pass `NSUnderlineStyleAttributeName` for the name parameter with a value of `[NSNumber numberWithInt:1]`.

Unlike underlining, bold and italic are traits of the font, so you need to use a font manager instance to convert the font to have the desired trait, then add the font attribute to the mutable attributed string. For a mutable attributed string named `attributedString`, use the following technique:

```
NSFontManager *fontManager = [NSFontManager sharedFontManager];
unsigned idx = range.location;
NSRange fontRange;
NSFont *font;

while (NSLocationInRange(idx, range)){
    font = [attributedString attribute:NSFontAttributeName atIndex:idx
                                   longestEffectiveRange:&fontRange inRange:range];
    fontRange = NSIntersectionRange(fontRange, range);
    [attributedString applyFontTraits:NSBoldFontMask range:fontRange];
    idx = NSMaxRange(fontRange);
}
```

If your mutable attributed string is actually an `NSTextStorage` object, place this code between `beginEditing` and `endEditing` calls.

## Examining Fonts

In addition to converting fonts, `NSFontManager` provides information on which fonts are available to the application and on the characteristics of any given font. The `availableFonts` method returns an array of the names of all fonts available. The `availableFontNamesWithTraits:` method filters the available fonts based on a font trait mask.

There are three methods for examining individual fonts. The `fontName:hasTraits:` method returns `true` if the font matches the trait mask provided. The `traitsOfFont:` method returns a trait mask for a given font. The `weightOfFont:` method returns an approximate ranking of a font's weight on a scale of 0–15, where 0 is the lightest possible weight, 5 is Normal or Book weight, 9 is the equivalent of Bold, and 15 is the heaviest possible (often called Black or Ultra Black).

## Customizing the Font Conversion System

If you need to customize the font conversion system by creating subclasses of `NSFontManager` or `NSFontPanel`, you must inform the `NSFontManager` class of this change with a `setFontManagerFactory:` or `setFontPanelFactory:` message, before either the shared font manager or shared Font panel is created. These methods record your class as the one to instantiate the first time the font manager or Font panel is requested. You may be able to avoid using subclasses if all you need is to add some custom controls to the Font panel. In this case, you can invoke the `NSFontPanel` method `setAccessoryView:` to add an `NSView` object below its font browser.

If you provide your own Font menu, you should register it with the font manager using the `setFontMenu:` method. The font manager is responsible for validating Font menu items and changing their titles and tags according to the selected font. For example, when the selected font is Italic, the font manager adds a check mark to the Italic Font menu item and changes its tag to `UnitalicMask`. Your Font menu's items should use the appropriate action methods and tags, as shown in Table 6-4.

**Table 6-4** Font menu item actions and tags

Font menu item	Action	Tag
Italic	<code>addFontTrait:</code>	<code>ItalicMask</code>
Bold	<code>addFontTrait:</code>	<code>BoldMask</code>
Heavier	<code>modifyFont:</code>	<code>HeavierFontAction</code>
Larger	<code>modifyFont:</code>	<code>SizeUpFontAction</code>

See also the following documents: *Attributed String Programming Guide* describes `NSAttributedString` objects, which manage sets of attributes, such as font and kerning, that are associated with character strings or individual characters. *Text Layout Programming Guide* describes how the Cocoa text system converts strings of text characters, font information, and page specifications into lines of glyphs placed at specific locations on a page, suitable for display and printing.

# Text Editing

## Objective-C/Swift

This chapter describes ways in which you can control the behavior of the Cocoa text system as it performs text editing. Text editing is the modification of text characters or attributes by interacting with text objects. Usually, editing is performed by direct user action with a text view, but it can also be accomplished by programmatic interaction with a text storage object. This document also discusses the text input system that translates keyboard events into commands and text input.

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

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

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

## The Editing Environment

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

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

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

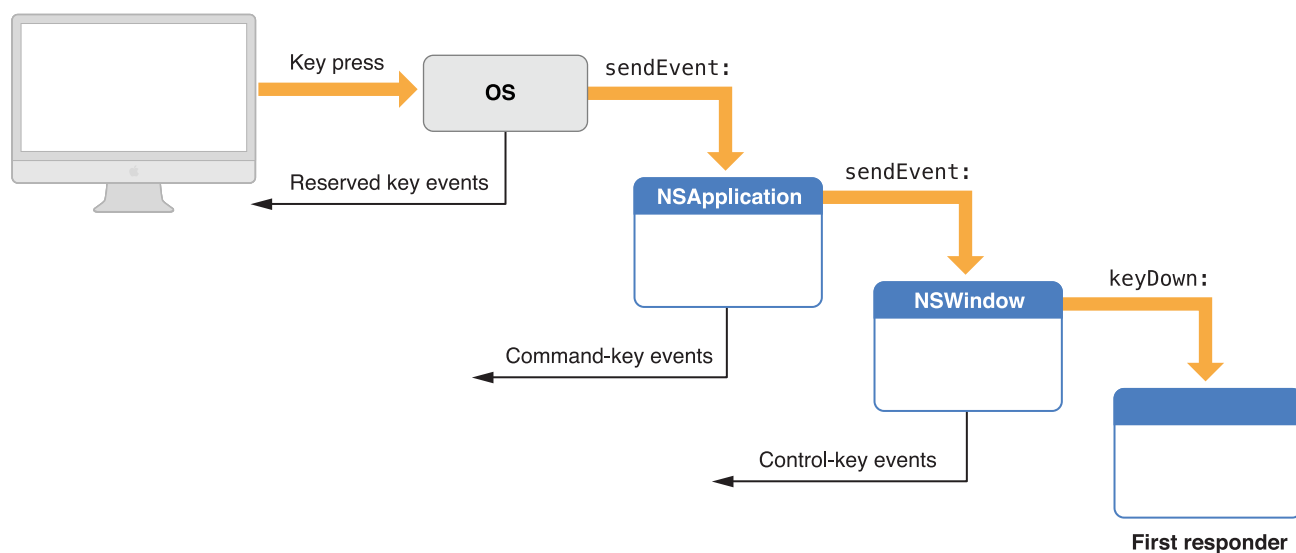
See [Working with the Field Editor](#) (page 68) for more information about the field editor.

## The Key-Input Message Sequence

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

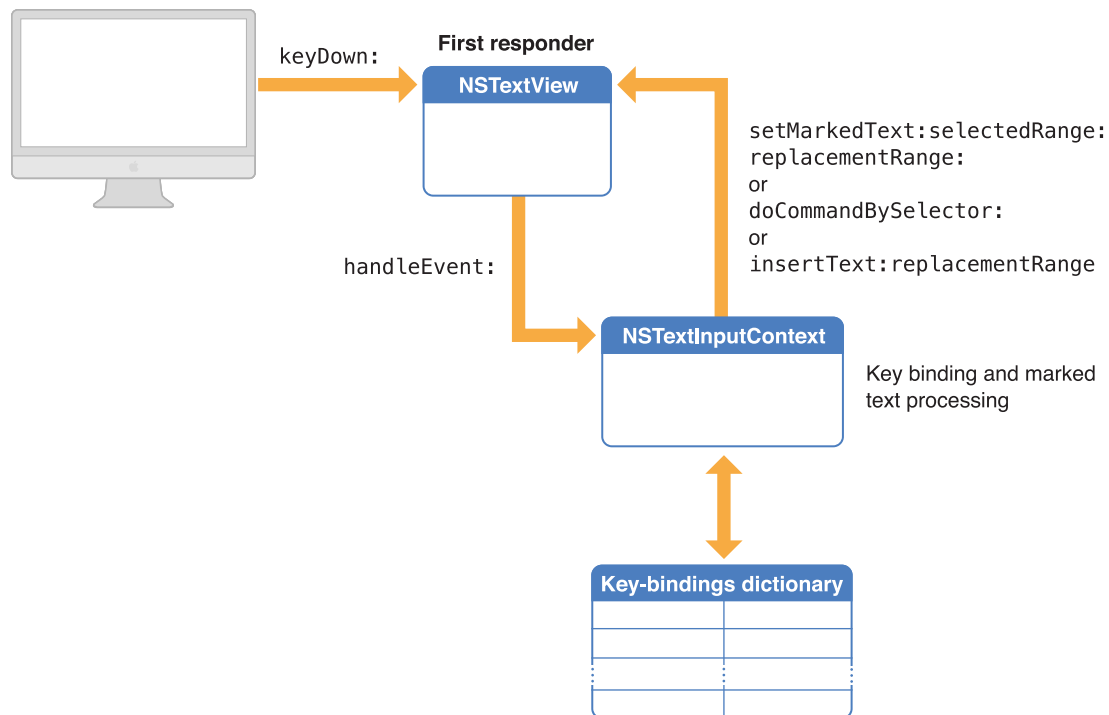
The message sequence invoked when a text view receives key events involves four methods declared by `NSResponder`. When the user presses a key, the operating system handles certain reserved key events and sends others to the `NSApplication` object, which handles Command-key events as key equivalents. The key events not handled are sent by the application object to the key window, which processes key events mapped to keyboard navigation actions (such as Tab moving focus on the next view) and sends other key events to the first responder. Figure 7-1 illustrates this sequence.

Figure 7-1 Key-event processing



If the first responder is a text view, the key event enters the text system. The key window sends the text view a `keyDown:` message with the event as its argument. The `keyDown:` method passes the event to `handleEvent:`, which sends the character input to the input context for key binding and interpretation. In response, the input context sends either `insertText:replacementRange:`, `setMarkedText:selectedRange:replacementRange:`, or `doCommandBySelector:` to the text view. Figure 7-2 illustrates the sequence of text-input event processing.

**Figure 7-2** Input context key binding and interpretation



The text input system uses a dictionary property list, called a *key-bindings dictionary*, to interpret keyboard events before passing them to the Input Method Kit framework for mapping to characters.

During the processing of a keyboard event, the event passes through the `NSMenu` object, then to the first responder via the `keyDown:` method. The default implementation of the method provided by the `NSResponder` class propagates the message up the responder chain until an overridden `keyDown:` implementation stops the propagation. Typically, an `NSResponder` subclass can choose to process certain keys and ignore others (for example, in a game) or to send the `handleEvent:` message to its input context.

The input context checks the event to see if it matches any of the keystrokes in the user's key-bindings dictionary. A key-bindings dictionary maps a keystroke (including its modifier keys) to a method name. For example, the default key-bindings dictionary maps `^d` (Control-D) to the method name `deleteForward:`. If the keyboard event is in the dictionary, then the input context calls the text view's `doCommandBySelector:` method with the selector associated with the dictionary entry.

If the input context cannot match the keyboard event to an entry in the key-bindings dictionary, it passes the event to the Input Method Kit for mapping to characters.

The standard key-bindings dictionary is in the file `/System/Library/Frameworks/AppKit.framework/Resources/StandardKeyBinding.dict`. You can override the standard dictionary entirely by providing a dictionary file at the path `~/Library/KeyBindings/DefaultKeyBinding.dict`. However, defining custom key bindings dynamically (that is, while the application is running) is not supported.

For more information about text-input key event processing, see “Text System Defaults and Key Bindings” in *Cocoa Event Handling Guide*.

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

## Intercepting Key Events

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

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

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

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

When a text view receives a key event, it sends the character input to the input context for key binding and interpretation. In response, the input context sends either `insertText:replacementRange:` or `doCommandBySelector:` to the text view, depending on whether the key event represents text to be inserted or a command to perform. The input context can also send the `setMarkedText:selectedRange:replacementRange:` message to set the marked text in the text view's associated text storage object. The message sequence invoked when a text view receives key events is described in more detail in [The Key-Input Message Sequence](#) (page 53).

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

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

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

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

## Text View Delegation

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

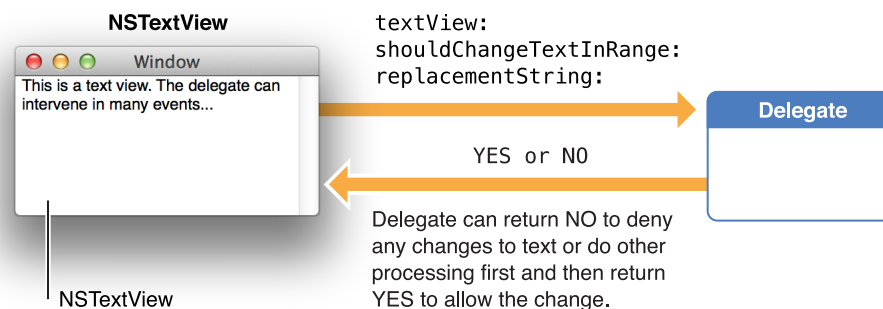


Before a text view makes any change to its text, it sends its delegate a `textView:shouldChangeTextInRange:replacementString:` message, which returns a Boolean value. (As with all delegate messages, it sends the message only if the delegate implements the method.) This mechanism provides the delegate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

## Text View Delegate Messages and Notifications

An `NSTextView` object can have a delegate that it informs of certain actions or pending changes to the state of the text. The delegate can be any object you choose, and one delegate can control multiple `NSTextView` objects (or multiple series of connected `NSTextView` objects). Figure 7-3 illustrates the activity of the delegate of an `NSTextView` object receiving the delegate message `textView:shouldChangeTextInRange:replacementString:`.

Figure 7-3 Delegate of an `NSTextView` object



*`NSTextDelegate Protocol Reference`* and *`NSTextViewDelegate Protocol Reference`* describe the delegate messages the delegate can receive. The delegating object sends a message only if the delegate implements the method.

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

---

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

---

The notifications posted by `NSTextView` are:

`NSTextDidBeginEditingNotification`

```
NSNotification  
NSNotification  
NSNotification  
NSTextViewWillChangeNotifyingTextViewNotification
```

It is particularly important for observers to register for the last of these notifications. If a new `NSTextView` object is added at the beginning of a series of connected `NSTextView` objects, it becomes the new notifying text view. It doesn't have access to which objects are observing its group of text objects, so it posts an `NSTextViewWillChangeNotifyingTextViewNotification`, which allows all those observers to unregister themselves from the old notifying text view and reregister themselves with the new one. For more information, see the description for this notification in *NSTextView Class Reference*.

## Text Field Delegation

Text fields (that is, instances of `NSTextField`, as opposed to instances of `NSTextView`) can also use delegation to control their editing behavior. One way in which this is done is for the text field itself to designate a delegate. Typically, you do this in Interface Builder by Control-dragging from the text field object to the delegate object, but you can also do it at run time by sending the text field a `setDelegate:` message, for example, in an `awakeFromNib` method. The delegate must respond to the messages defined by the `NSTextFieldDelegate` protocol (which adopts the `NSControlTextEditingDelegate` protocol). In addition to the methods defined by the `NSControlTextEditingDelegate` protocol, a text field delegate can respond to the delegate methods of `NSControl`.

As an example of how a text field's delegate can control its editing behavior, you can disable text completion in a text field by having its delegate implement the delegate method `control:textView:completions:forPartialWordRange:indexOfSelectedItem:` simply to return `nil`.

Another way in which you can customize editing behavior in a text field by delegation involves the field editor, an `NSTextView` object that handles the actual editing, in turn, for all the text fields in a window. The field editor automatically designates any text field it is editing as its delegate, so you can encapsulate special editing behavior for a text field with the text field itself by implementing the delegate methods defined by `NSTextDelegate` and `NSTextViewDelegate` protocols. For information about controlling the editing behavior of text fields through delegate messages and notifications sent by the field editor, see [Using Delegation and Notification with the Field Editor](#) (page 68).

## Synchronizing Editing

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

The system provides a rich set of notifications to delegates and observers to enable your code to interact with this logic, as described in [Text View Delegate Messages and Notifications](#) (page 57).

## Batch-Editing Mode

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

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

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

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

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

Listing 7-1 illustrates a situation in which the `NSText` method `scrollRangeToVisible:` forces layout to occur and raises an exception.

#### Listing 7-1 Forcing layout

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

Scrolling a character range into visibility requires layout to be complete through that range so the text view can know where the range is located. But in Listing 7-1, the text storage is in batch-editing mode. It is in an inconsistent state, so the layout manager has no way to do layout at this time. Moving the `scrollRangeToVisible:` call after `endEditing` would solve the problem.

There are additional actions that you should take if you implement new user actions in a text view, such as a menu action or key binding method that changes the text. For example, you can modify the selected range of characters using the `NSTextView` method `setSelectedRange:`, depending on the type of change performed by the command, using the results of the `NSTextView` methods `rangeForUserTextChange`, `rangeForUserCharacterAttributeChange`, or `rangeForUserParagraphAttributeChange`. For example, `rangeForUserParagraphAttributeChange` returns the entire paragraph containing the original selection—that is the range affected if your action modifies paragraph attributes. Also, you should call `textView:shouldChangeTextInRange:replacementString:` before you make the change and `didChangeText` afterwards. These actions ensure that the correct text gets changed and the system sends the correct notifications and delegate messages to the text view's delegate. See [Subclassing NSTextView](#) (page 62) for more information.

## Forcing the End of Editing

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

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

Listing 7-2 presents an implementation of the `textDidEndEditing:` delegate method that ends editing in an `NSTableView` subclass. By default, when the user is editing a cell in a table view and presses Tab or Return, the field editor ends editing in the current cell and begins editing the next cell. In this case, you want to end

editing altogether if the user presses Return. This method distinguishes which key the user pressed; for Tab it does the normal behavior, and for Return it forces the end of editing completely by making the window first responder.

**Listing 7-2** Forcing the end of editing

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

## Setting Focus and Selection Programmatically

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

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

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

When an object conforming to the `NSTextInputClient` protocol becomes the first responder in the key window, its `NSTextInputContext` object becomes active and bound to the active text input sources, such as character palette, keyboards, and input methods.

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

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

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

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

## Subclassing `NSTextView`

Using `NSTextView` directly is the easiest way to interact with the text system, and its delegate mechanism provides an extremely flexible way to modify its behavior. In cases where delegation does not provide required behavior, you can subclass `NSTextView`.

---

**Note:** To modify editing behavior, your first resort should be to notification or delegation, rather than subclassing. It may be tempting to start by subclassing `NSTextView` and overriding `keyDown:`, but that's usually not appropriate, unless you really need to deal with raw key events before input management or key binding. In most cases it's more appropriate to work with one of the text view delegate methods or with text view notifications, as described in [Text View Delegate Messages and Notifications](#) (page 57).

---

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

## Updating State

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

## Custom Import Types

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

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

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

## Altering Selection Behavior

Your subclass of `NSTextView` can customize the way selections are made for the various granularities (such as character, word, and paragraph) described in [Setting Focus and Selection Programmatically](#) (page 61). While tracking user changes to the selection, an `NSTextView` object repeatedly invokes `selectionRangeForProposedRange:granularity:` to determine what range to actually select. When finished tracking changes, it sends the delegate a `textView:willChangeSelectionFromCharacterRange:toCharacterRange:` message. By overriding the `NSTextView` method or implementing the delegate method, you can alter the way the selection is extended or reduced. For example, in a code editor you can provide a delegate that extends a double click on a brace or parenthesis character to its matching delimiter.

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

## Preparing to Change Text

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

## Text Change Notifications and Delegate Messages

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 `NSNotification`, 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 `NSNotification`—are skipped in the sequence. The `textView:shouldChangeTextInRange:replacementString:` method, however, must be invoked for each individual change.

## Smart Insert and Delete

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

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

## Creating a Custom Text View

A strategy even more complicated than subclassing `NSTextView` is to create your own custom text view object. If you need more sophisticated text handling than `NSTextView` provides, for example in a word processing application, it is possible to create a text view by subclassing `NSView`, implementing the `NSTextInputClient` protocol, and interacting directly with the input management system.

## Implementing Text Input Support

Custom Cocoa views can provide varying levels of support for the text input system. There are essentially three levels of support to choose from:

1. Override the `keyDown:` method.
2. Override `keyDown:` and use `handleEvent:` to support key bindings.
3. Also implement the full `NSTextInputClient` protocol.

In the first level of support, the `keyDown:` method recognizes a limited set of events and ignores others. This level of support is typical of games. (When overriding `keyDown:`, you must also override `acceptsFirstResponder` to make your custom view respond to key events, as described in “Event Handling Basics” in *Cocoa Event Handling Guide*.)

In the second level of support, you can override `keyDown:` and use the `handleEvent:` method to receive key-binding support without implementing the `NSTextInputClient` protocol. Because the `NSView` method `inputContext` does not instantiate `NSTextInputContext` automatically if the view does not conform to `NSTextInputClient`, the custom view must instantiate it manually. You then implement the standard key-binding methods that your view wants to support, such as `moveForward:` or `deleteForward:`. (The full list of key-binding methods can be found in `NSResponder.h`.)

If you are writing your own text view from scratch, you should use the third level of support and implement the `NSTextInputClient` protocol in addition to overriding `keyDown:` and using `handleEvent:`. `NSTextView` and its subclasses are the only classes provided in Cocoa that implement `NSTextInputClient`, and if your application needs more complex behavior than `NSTextView` can provide, as a word processor might, you may need to implement a text view from the ground up. To do this, you must subclass `NSView` and implement the `NSTextInputClient` protocol. (A class implementing this protocol—by inheriting from `NSTextView` or by implementing the protocol directly—is called a *text view*.)

If you are implementing the `NSTextInputClient` protocol, your view needs to manage marked text and communicate with the text input context to support the text input system. These tasks are described in the next two sections.

## Managing Marked Text

One of the primary things that a text view must do to cooperate with an input context is to maintain a (possibly empty) range of *marked text* within its text storage. The text view should highlight text in this range in a distinctive way, and it should allow selection within the marked text. A text view must also maintain an *insertion point*, which is usually at the end of the marked text, but the user can place it within the marked text. The text view also maintains a (possibly empty) *selection* range within its text storage, and if there is any marked text, the selection must be entirely within the marked text.

A common example of marked text appears when a user enters a character with multiple keystrokes, such as “é” in an `NSTextView` object. To enter this character, the user needs to type Option-E followed by the E key. After pressing Option-E, the accent mark appears in a highlighted box, indicating that the text is marked (not final). After the final E is pressed, the “é” character appears and the highlight disappears.

## Communicating with the Text Input Context

A text view and a text input context must cooperate so that the input context can implement its user interface. The `NSTextInputContext` class represents the interface to the text input system, that is, a state or context unique to its client object such as the key binding state, input method communication session, and so on. Most of the `NSTextInputClient` protocol methods are called by an input context to manipulate text within the text view for the input context’s user-interface purposes.

Each `NSTextInputClient`-compliant object (typically an `NSView` subclass) has its own `NSTextInputContext` instance. The default implementation of the `NSView` method `inputContext` manages an `NSTextInputContext` instance automatically if the view subclass conforms to the `NSTextInputClient` protocol.

A text view must inform the current input manager when a mouse or keyboard event happens by sending the `handleEvent:` message to the current input context. When its marked text range is no longer needed, the text view sends a `discardMarkedText` message to the current input context.

In addition, a text view must tell the input context when position information for a character range changes, such as when the text view scrolls, by sending the `invalidateCharacterCoordinates` message to the input context. The input context can then update information previously queried via methods like `firstRectForCharacterRange:actualRange:` when, for example, it wants to show a selection pop-up menu for marked text (as with a Japanese input method). There is an optional method, `drawsVerticallyForCharacterAtIndex:`, that can inform the text input system whether the protocol-conforming client renders the character at the given index vertically.

The input context generally uses all of the methods in the `NSTextInputClient` protocol. You can also register to receive a notification from the input context when the keyboard layout changes.

For more information, refer to *NSText Class Reference*, *NSTextView Class Reference*, *NSView Class Reference*, *NSTextInputContext Class Reference*, and *NSTextInputClient Protocol Reference*.

## Working with the Field Editor

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

### How the Field Editor Works

The text system automatically instantiates the field editor from the `NSTextView` class when the user begins editing text of an `NSControl` object such as a text field. While it is editing, the system inserts the field editor into the responder chain as first responder, so it receives keystroke events in place of the text field or other control object. When the focus shifts to another text field, the field editor attaches itself to that field instead. The field editor designates the current text field as its delegate, which enables the text field to control changes to its contents. This mechanism can be confusing if you're not familiar with the workings of the field editor, because the `NSWindow` method `firstResponder` returns the field editor, which is not visible, rather than the onscreen object that currently has keyboard focus.

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

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

The end of editing triggers synchronization of the contents of the field editor and the `NSTextFieldCell` object that controls editing in the text field. At that point Cocoa detaches the field editor from the text field and reveals the text field at the top of the view hierarchy.

### Using Delegation and Notification with the Field Editor

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

### Changing Default Behavior

It's straightforward to change the default behavior of the field editor by implementing delegate methods. For example, the delegate can change the behavior that occurs when the user presses Return while editing a text view. By default, that action ends editing and selects the next control in the key view loop. If, for example, you

want pressing Return to end editing but not select the next control, you can implement the `textDidEndEditing:` delegate method in the text field. The field editor automatically calls this method if the delegate implements it, and passes `NSTextDidEndEditingNotification`. The implementation can examine this notification to discover the event that ended editing and respond appropriately.

## Getting Newlines into an NSTextField Object

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

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

A better method is to implement another `NSControl` delegate method, `control:textView:doCommandBySelector:`, which enables the text field's delegate to check whether the user is attempting to insert a newline character and, if so, force to field editor to insert it. The implementation could appear as shown in Listing 7-3.

**Listing 7-3** Forcing the field editor to enter a newline character

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

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

## Using a Custom Field Editor

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

### Why Use a Custom Field Editor?

It's not necessary to use a custom field editor if you simply need to validate, interpret, format, or even edit the contents of text fields as the user types. You can attach an `NSFormatter`, such as `NSNumberFormatter`, `NSDateFormatter`, or a custom formatter, for that purpose. See *Data Formatting Guide* for more information about using formatters. Delegation and notification also provide many opportunities for you to intervene, as described in [Using Delegation and Notification with the Field Editor](#) (page 68).

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

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

### How to Substitute a Custom Field Editor

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

For example, the implementation shown in Listing 7-4 tests whether or not the requesting object is instantiated from a custom text field class named `CustomTextField`, and, if it is, returns a custom field editor.

**Listing 7-4** Substituting a custom field editor

```
- (id)windowWillReturnFieldEditor:(NSWindow *)sender toObject:(id)anObject
{
    if ([anObject isKindOfClass:[CustomTextField class]])
    {
        if (!myCustomFieldEditor) {
            myCustomFieldEditor = [[CustomFieldEditor alloc] init];
            [myCustomFieldEditor setFieldEditor:YES];
        }
        return myCustomFieldEditor;
    }
    return nil;
}
```

If the requesting object is not a custom text field or subclass, the delegate method returns `nil` and the window uses its default field editor. This arrangement has the advantage that it does not instantiate the custom field editor unless it is needed.

In OS X v10.6 and later, another way of providing a custom field editor is to override the `NSCell` method `fieldEditorForView:`. This method, rather than the window delegate method, is more suitable for custom cell subclasses.

You can find more information about subclassing `NSTextView` in [Subclassing NSTextView](#) (page 62).

## Field Editor–Related Methods

This section lists the AppKit methods most directly related to the field editor. You can peruse these tables to understand where Cocoa provides opportunities for you to interact with the field editor. Refer to *AppKit Framework Reference* for details. The `NSWindow` methods related to the field editor are listed in Table 7-1.

**Table 7-1** `NSWindow` field editor–related methods

Method	Description
<code>fieldEditor: forObject:</code>	Returns the receiver's field editor, creating it if needed.

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

The `NSTextFieldCell` method related to the field editor is listed in Table 7-2.

**Table 7-2** `NSTextFieldCell` field editor–related method

Method	Description
<code>setUpFieldEditorAttributes:</code>	You never invoke this method directly; by overriding it, however, you can customize the field editor.

The `NSCell` methods related to the field editor are listed in Table 7-3.

**Table 7-3** `NSCell` field editor–related methods

Method	Description
<code>fieldEditorForView:</code>	The primary way to substitute a custom field editor in OS X v10.6 and later.
<code>selectWithFrame: inView:editor: delegate:start: length:</code>	Uses the field editor passed with the message to select text in a range.
<code>editWithFrame: inView:editor: delegate:event:</code>	Begins editing of the receiver's text using the field editor passed with the message.
<code>endEditing:</code>	Ends any editing of text, using the field editor passed with the message, begun with either of the other two <code>NSCell</code> field editor–related methods.



The `NSControl` methods related to the field editor are listed in Table 7-4. The `NSControl` delegate methods listed in Table 7-4 are control-specific versions of the delegate methods and notifications defined by `NSText`. The field editor, derived from `NSText`, initiates sending the delegate messages and notifications through its editing actions.

**Table 7-4** `NSControl` field editor–related methods

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

The `NSResponder` methods related to the field editor are listed in Table 7-5.

**Table 7-5** `NSResponder` field editor–related methods

Method	Description
<code>insertBacktab:</code>	Implemented by subclasses to handle a “backward tab.”
<code>insertNewline- IgnoringFieldEditor:</code>	Implemented by subclasses to insert a line-break character at the insertion point or selection.

Method	Description
<code>insertTabIgnoringFieldEditor:</code>	Implemented by subclasses to insert a tab character at the insertion point or selection.

The `NSText` and `NSTextDelegate` methods related to the field editor are listed in Table 7-6.

**Table 7-6** `NSText` field editor–related methods

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

# Document Revision History

This table describes the changes to *Cocoa Text Architecture Guide*.

Date	Notes
2014-02-11	Added "Font Descriptors" section to Font Handling chapter. Fixed typos and dead link in introduction.
2013-04-23	Removed references to retired sample code.
2012-09-19	Added section on creating text objects programmatically. Fixed typo.
2012-07-23	Added section titled "Text Field Delegation" to "Text Editing" chapter.
2012-04-27	Removed out-of-date text editor tutorial and section describing deprecated glyph-handling methods of NSFont. Removed references to manual reference counting. Fixed typos.
2010-05-04	Updated text system class hierarchy diagram and configuration diagrams. Additional legacy documents providing content to this document are Font Handling and Font Panel.
2010-03-23	New document that explains how the objects of the Cocoa text system interact. This document contains content previously published in the following documents, which remain in the legacy area of the ADC library: Text System Overview, Text Attributes, Text Editing Programming Guide for Cocoa, and Text Input and Output.



Apple Inc.  
Copyright © 2014 Apple Inc.  
All rights reserved.

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

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

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

Apple, the Apple logo, Cocoa, Mac, Macintosh, Objective-C, OS X, QuickDraw, TrueType, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

Helvetica and Times are registered trademarks of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Smalltalk-80 is a trademark of ParcPlace Systems.

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

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

**Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.**