The Yellow Box Text System

Note: This document hasn't been fully updated from an earlier version. A number of minor changes have occurred to the text system API that may render some explanations, illustrations, and code samples inaccurate. See the reference documentation and release notes for more information.

The text-handling component of any application framework presents one of the greatest challenges to framework 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 these days developers and users commonly expect even more than these basic features, requiring their simple editors to support multiple fonts, various paragraph styles, embedded images, spell checking, and other features.

A framework that provides these more advanced text-handling features may be adequate for today's programming needs but falls far short when measured against the requirements that are emerging from our ever more interconnected computing world: support for the character sets of the world's living languages, powerful layout capabilities to handle various text directionality and nonrectangular text containers, and sophisticated typesetting capabilities including control of kerning and ligatures.

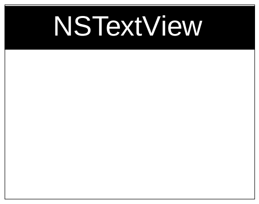
The Yellow Box text-handling system is designed to provide all these capabilities without requiring you to learn about or interact with more of the system than is required to meet the needs of your application. It does this by providing a layering of classes, as described in the next section. The sections that follow the architectural overview give you practical examples of how to work with the text-handling system.

Architectural Overview

You can think of the text-handling system as having three distinct layers of API. For most typical uses, the general-purpose programmatic interface of the NSTextView class is all you need to learn. 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 learn about and interact with any of the classes that support the text-handling system. The following discussion presents these three layers.

The User-Interface Layer: The NSTextView Class

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



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

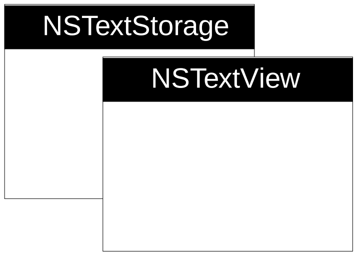
The interface that this class declares (and inherits from its superclass NSText) lets you programmatically:

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

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

An NSTextView object is a convenient package of the most generally useful text-handling features. If the features of the NSTextView class satisfy your application's requirements, you can skip to the section below titled "Working with the Text-Handling System: Basic Operations". However, if you need more programmatic control over the characters and attributes that make up the text, you'll have to learn something about the object that stores this data, NSTextStorage.

The Storage Layer: The NSTextStorage Class



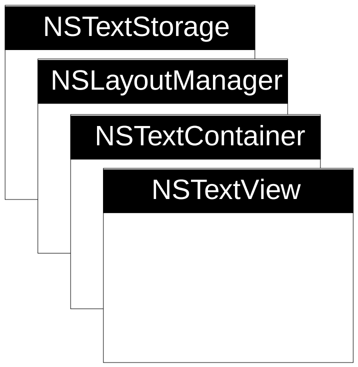
An NSTextStorage object serves as the data repository for a group of text handling objects. The format for this data is called an *attributed string*, which is an association of characters (in Unicode encoding) and the attributes (such as font, color, paragraph style) that apply to them. Conceptually, each character in a text has associated with it a dictionary of keys and values. A key names an attribute (say the font) and the associated value specifies the characteristics of that attribute (such as Helvetica 12 point).

An NSTextView lets users affect character attributes through direct action: The user selects some text and reduces the spacing between characters by choosing the Tighten menu command. NSTextStorage lets you operate on the attributes of the text programmatically: Your code can run through the text loosening the kerning for all characters of a certain font and size. To learn more about the NSTextStorage class, see “Using NSTextStorage” below.

The Complete System

The roster of objects that make up the complete text-handling system is relatively long, so this section concentrates on the major players and only mentions the minor ones in passing.

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, an NSTextContainer 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. Nor does an NSTextContainer store text—that’s the job of NSTextStorage.

An NSLayoutManager 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’s display. It also oversees the layout of text within the areas defined by NSTextContainer objects. To better understand the function of an NSLayoutManager object, you need to understand the difference between characters and glyphs.

Characters and Glyphs

Characters are conceptual entities that correspond to units of written language. Examples of characters include the letters of the Roman alphabet, the Kanji ideographs used in Japanese, and symbols that indicate mathematical operations. Characters are represented as numbers in a computer’s memory or on disk, and a *character encoding* defines the mapping between a numerical value and a specific character. For example, the ASCII and Unicode character encodings both assign the value 97 (decimal) to the character ‘a’. The Yellow Box text-handling system uses the Unicode character encoding internally, although it can read and write other encodings on disk.

You can think of a *glyph* as the rendered image of a character. The words of this sentence are made visible through glyphs. A collection of glyphs that share certain graphic qualities is called a *font*.

The difference between a character and a glyph isn’t immediately apparent in English since there’s typically a one-to-one mapping between the two. But, in some Indic languages, for example, a single character can map to more than one glyph. And, in many languages, two or more characters may be needed to specify a single glyph. To take a simple example, the glyph ‘ö’ can be the result of two characters, one representing the base character ‘o’ and the other representing the diacritical mark ‘¨’. A user of a word processor can strike the 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 stores the derived glyphs. Finding the correspondence between these two streams is another responsibility of the NSLayoutManager.

For a given glyph the NSLayoutManager can find the corresponding character or characters in the character stream. Similarly, for a given character, the NSLayoutManager can locate the associated glyph or glyphs. For example, when a user selects a range of text, the NSLayoutManager 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”, the ‘f’ and ‘l’ can be represented by the ‘fl’ ligature rather than the two glyphs ‘f’ and ‘l’. The NSLayoutManager has new glyphs generated as needed. Once the glyphs are regenerated, the text must be laid out and displayed. Again, the NSLayoutManager is instrumental in this step. Working with the NSTextContainer and other objects of the text system, the NSLayoutManager determines where each glyph appears in the NSTextView. Finally, the NSTextView renders the text.

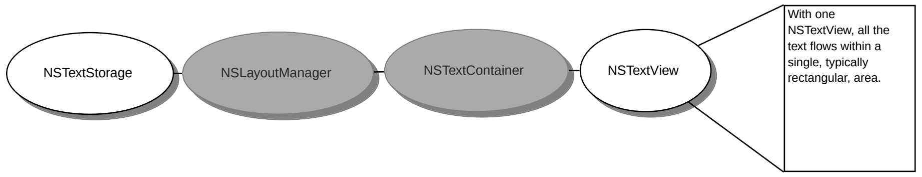
Since an NSLayoutManager is central to the operation of the text-handling system, it also serves as the repository of information shared by various components of the system.

These are just some of the functions of an NSLayoutManager; others are discussed in later sections.

Common Configurations

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

To display a single flow of text, the objects are arranged like this:

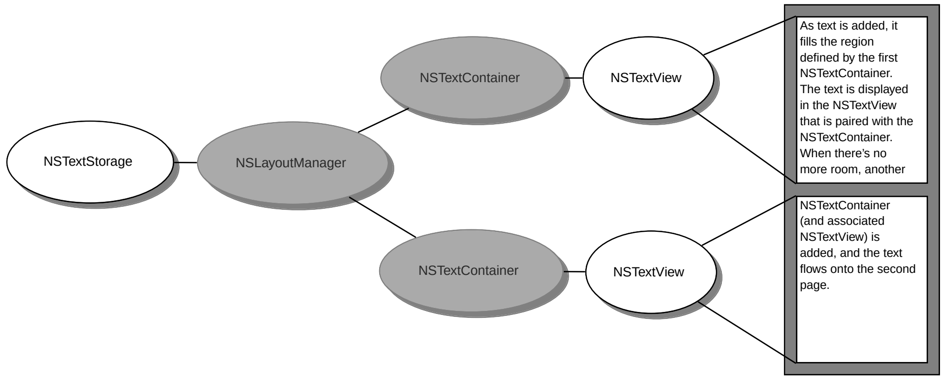

NSTextStorage
NSLayoutManager
NSTextContainer
NSTextView
With one NSTextView, all the text flows within a single, typically rectangular, area.

The NSTextView 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's vertical dimension is declared to be some extremely large value so that the container can accommodate any amount of text, while the NSTextView is set to size itself around the text using the **setVerticallyResizable:** method defined by NSText, and given a maximum height equal to the NSTextContainer's height. Then, with the NSTextView embedded in an NSScrollView, the user can scroll to see any portion of this text.

If the NSTextContainer's area is inset from the NSTextView'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's data and lay them out within the area defined by the NSTextContainer.

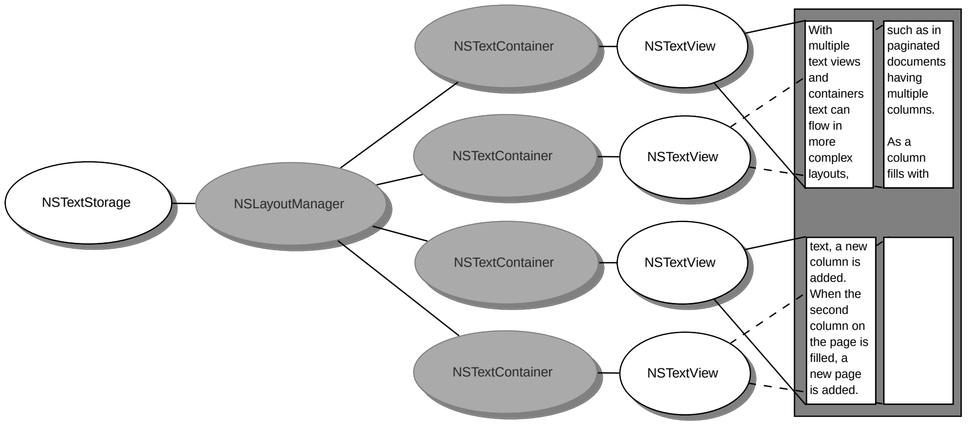
This configuration is limited by having only one NSTextContainer-NSTextView pair. In such an arrangement, the text flows uninterrupted within the area defined by the NSTextContainer. Page breaks, multi-column layout, and more complex layouts can't be accommodated by this arrangement.

By using multiple NSTextContainer-NSTextView pairs, more complex layout arrangements are possible. For example, to support page breaks, an application can configure the text-handling objects like this:


NSTextContainer
NSTextView
NSTextStorage
NSLayoutManager
NSTextContainer
NSTextView
As text is added, it fills the region defined by the first NSTextContainer. The text is displayed in the NSTextView that is paired with the NSTextContainer. When there's no more room, another
NSTextContainer (and associated NSTextView) is added, and the text flows onto the second page.

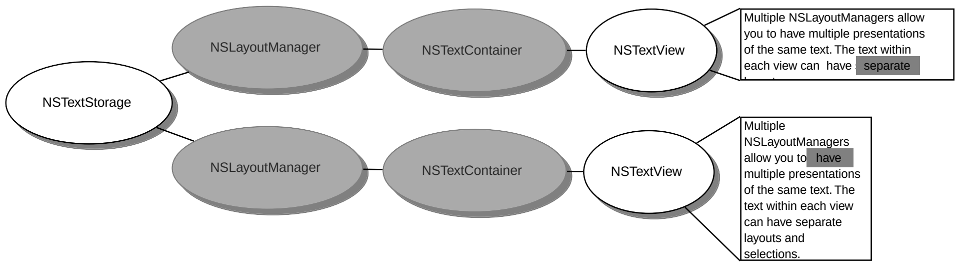
Each NSTextContainer-NSTextView pair corresponds to a page of the document. The gray rectangle in the diagram above represents a custom view object that your application provides as a background for the NSTextViews. This custom view can be embedded in an NSScrollView to allow the user to scroll through the document's pages.

A multi-column document uses a similar configuration:


NSTextStorage
NSLayoutManager
NSTextContainer
NSTextView
NSTextContainer
NSTextView
NSTextContainer
NSTextView
NSTextContainer
NSTextView
With multiple text views and containers text can flow in more complex layouts,
such as in paginated documents having multiple columns.
As a column fills with
text, a new column is added. When the second column on the page is filled, a new page is added.

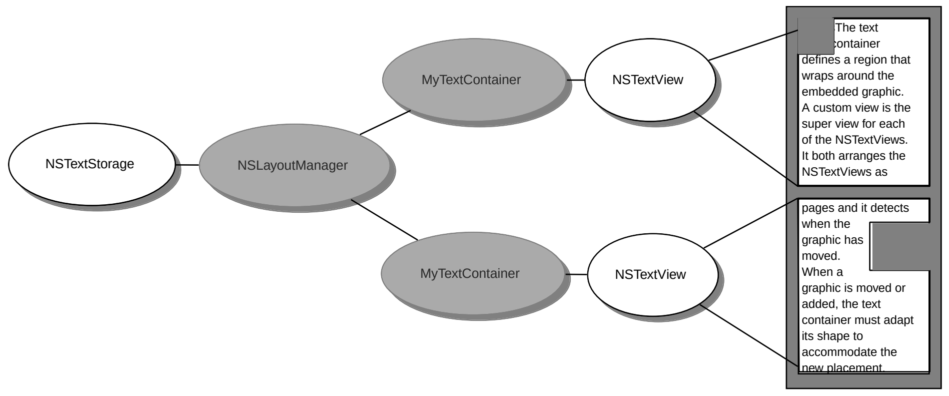
Instead of having one NSTextView-NSTextContainer pair correspond to a single page, there are now two pairs—one for each column on the page. Each NSTextContainer-NSTextView controls a portion of the document. As the text is displayed, glyphs are first laid out in the top-left view. When there is no more room in that view, the NSLayoutManager 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. The NSLayoutManager proceeds to lay out text in the next container, notifies the delegate when finished, and so on. Again, a custom view (depicted as a gray rectangle) provides a canvas for these text columns.

Not only can you have multiple NSTextContainer-NSTextView pairs, you can also have multiple NSLayoutManagers accessing the same NSTextStorage. The simplest arrangement looks like this:


NSTextStorage
NSLayoutManager
NSTextContainer
NSTextView
NSLayoutManager
NSTextContainer
NSTextView
Multiple NSLayoutManagers allow you to have multiple presentations of the same text. The text within each view can have separate
Multiple NSLayoutManagers allow you to have multiple presentations of the same text. The text within each view can have separate layouts and selections.

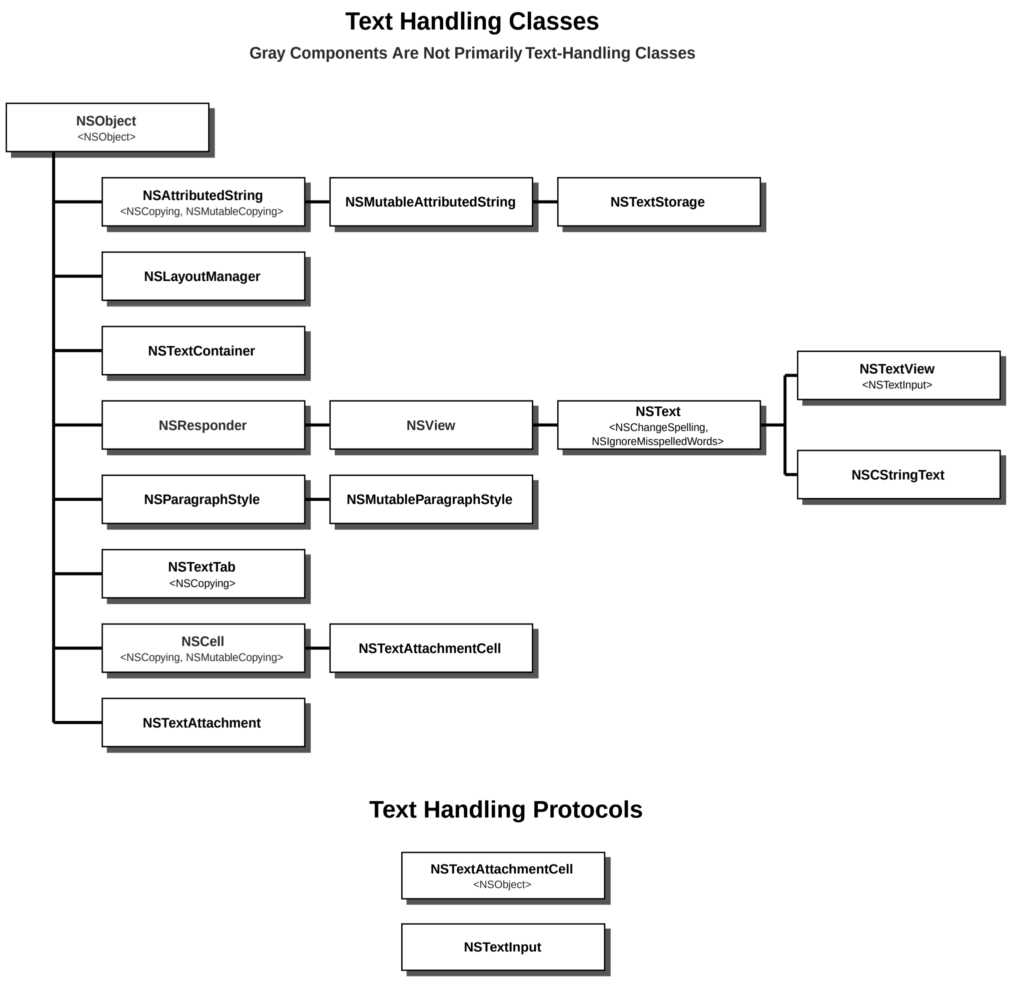
The effect of this arrangement is to give multiple views on the same text. If the user alters the text in the top view, the change is immediately reflected in the bottom view (assuming the location of the change is within the bottom view's bounds).

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:


NSTextStorage
NSLayoutManager
MyTextContainer
MyTextContainer
NSTextView
NSTextView
The text container defines a region that wraps around the embedded graphic. A custom view is the super view for each of the NSTextViews. It both arranges the NSTextViews as pages and it detects when the graphic has moved. When a graphic is moved or added, the text container must adapt its shape to accommodate the new placement.

Class Hierarchy of the Text-Handling System

You’ve seen the four principal classes in the text-handling system, but there are a number of auxiliary classes and protocols that make up the system. The diagrams below give you a picture of the complete system. Following the diagrams is a list of the elements that haven’t been introduced so far (see the individual class specifications for more information).



- NSFileWrapper, NSTextAttachment, and NSTextAttachmentCell
- NSTextInput protocol, NSInputManager, and NSInputServer
- NSParagraphStyle, NSMutableParagraphStyle, and NSTextTab

Summary

The text-handling 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-handling system, NSTextStorage holds the model’s data, NSTextContainer and

NSTextView work together to present the view, and NSLayoutManager intercedes as the controller to make sure that the data and its representation on screen 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-handling 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, determining where line breaks occur, the total number of pages, etc.

The layering of the text-handling system reduces the amount you have to learn to accomplish common text-handling tasks. Many applications interact with this system solely through the API of the NSTextView class.

The following sections examine the text-handling system from a practical point of view, showing you how to work with the system to achieve particular goals, starting with the most basic.

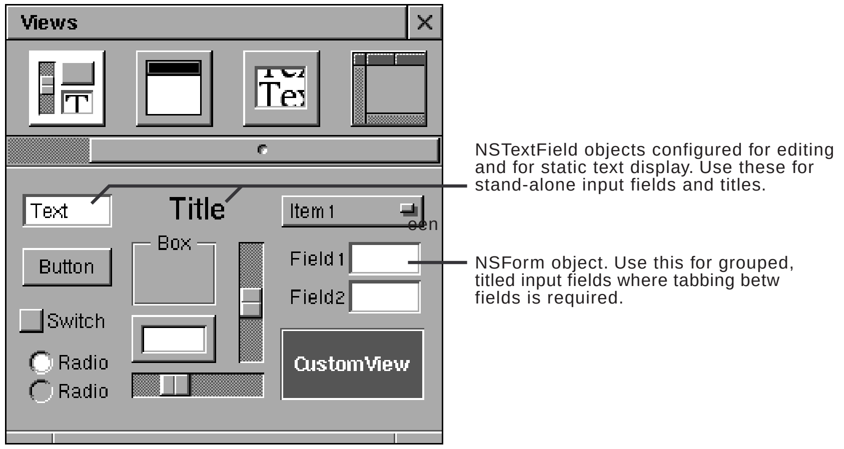
Working with the Text-Handling System: Basic Operations

Creating an NSTextView Object

All applications use the text-handling system, if only to display the titles of buttons and other labels. Most applications have far greater need of the system than that. This section describes the most direct ways of assembling the network of objects that make up that system.

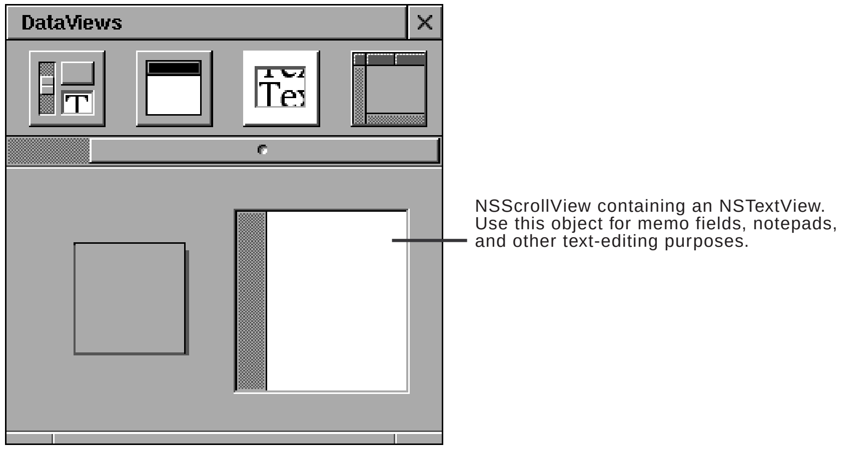
Interface Builder and the Text-Handling System

The easiest way to use the text-handling system is through the objects on Interface Builder's palettes. The control objects (NSForm and NSTextField) provide objects that are preconfigured for specific uses:


Views
Text
Title
Item 1
Box
Button
Field 1
Field 2
Switch
Radio
Radio
CustomView
NSTextField objects configured for editing and for static text display. Use these for stand-alone input fields and titles.
NSForm object. Use this for grouped, titled input fields where tabbing betw fields is required.

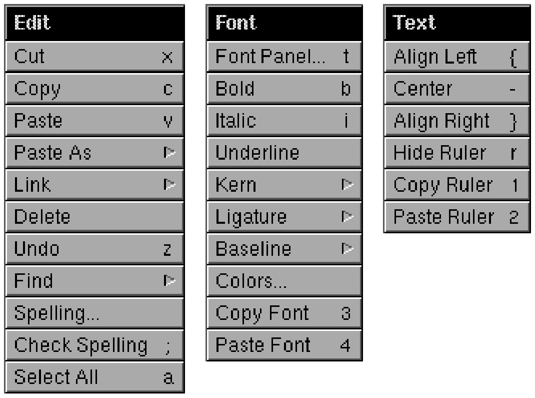
Using Interface Builder's Inspector panel, you can set many text-related attributes of these controls. For example, you can specify whether the text in a text field is selectable, editable, scrollable, and so on. The Inspector panel also lets you set the text alignment and background and foreground colors.

Interface Builder also provides a scrolling text view that supports the features of a basic text editor:


DataViews
NSScrollView containing an NSTextView. Use this object for memo fields, notepads, and other text-editing purposes.

The NSScrollView inspector in Interface Builder lets you specify, among other things, whether the contained NSTextView allows multiple fonts and embedded graphics.

Much more of NSTextView's functionality is accessible through menu commands. Interface Builder's Palettes window offers these ready-made menus that contain text-related commands:



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

NSTextViews cooperate with the Services facility through the Services menu, also available from Interface Builder's Menus palette. By simply adding the Services menu item to your application's main menu, the NSTextViews in your application can access services provided by other applications. For example, if the user selects a word within an NSTextView and chooses the Define in Webster service, the NSTextView passes its selected text to the Webster application for look up.

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

Creating an NSTextView Programmatically

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

You create an NSTextView object in the usual way: by sending the **alloc** and **init...** messages. Given an NSWindow object represented by **aWindow**, you can create an NSTextView object in this way:

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

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

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

NSTextView's **initWithFrame:** method not only initializes the receiving NSTextView object, it causes the object to create and interconnect the other components of the text-handling system. This is a convenience that frees you from having to create and interconnect them yourself. Since the NSTextView created these supporting objects, it's responsible for releasing them when they are no longer needed. When you're done with the NSTextView, release it and it takes care of releasing the other objects of the text-handling system. Note that this ownership policy is only in effect if you let NSTextView create the components of the text-handling system. See "Assembling the Text System by Hand" for more information on object ownership when you create the components yourself.

Text Input and Output

The text-handling system provides a convenient interface to the file system allowing you to read, display, and write files in these formats:

Format	Description
Plain Text	Characters unaccompanied by attribute information.
Rich Text Format (RTF)	Character and attribute information expressed in the Rich Text Format® (RTF). See the *Rich Text Format Specification* by Microsoft Corporation for more information.
Rich Text Format Directory (RTFD)	Character and attribute information expressed in the Rich Text Format but stored in a directory along with the images and other attachments that are embedded in the text.

Reading Text from a File

To read text from a file, you have to first determine format of the text. To illustrate how this is done, consider an object of the custom class Controller. A Controller object is responsible for opening and closing files. It stores an NSTextView and declares a variable that records the format of the text that it reads in. Here's the interface declaration:

```
#import <AppKit/AppKit.h>

typedef enum _dataFormat {
    Unknown = 0,
    PlainText = 1,
    RichText = 2,
    RTFD = 3,
} DataFormat;

@interface Controller : NSObject
{
    DataFormat theFormat;
    NSTextView *theTextView;
}

- (void)openFile:(id)sender;
- (void)saveFile:(id)sender;
@end
```

Now, the Controller object's **openFile:** method can be implemented like this:

```
- (void)openFile:(id)sender
{
    NSOpenPanel *panel = [NSOpenPanel openPanel];

    if ([panel runModal] == NSOKButton) {
        NSString *fileName = [panel filename];
        if ([[fileName pathExtension] isEqualToString:@"rtfd"]) {
            [theTextView readRTFDFromFile:fileName];
            theFormat = RTFD;
        } else if([[fileName pathExtension] isEqualToString:@"rtf"]) {
           NSData *rtfData = [NSData dataWithContentsOfFile:fileName];
              [theTextView replaceRange:NSMakeRange(0, [[theTextView
string]
                  length]) withRTF:rtfData];
              theFormat = RichText;
          } else {
              NSString *fileContents = [NSString
                  stringWithContentsOfFile:fileName];
             [theTextView setString:fileContents range:NSMakeRange(0,
                  [[theTextView string] length])];
              theFormat = PlainText;
          }
      }
      return;
  }
```

The **openFile:** method checks the file name returned by the Open panel for the extensions “rtfd” or “rtf” and uses the appropriate means of loading data for each type. Files having any other extension are loaded as plain text. Note that the Controller object records the format of the loaded data in its **theFormat** variable. This information is used to determine how the file should be saved, as discussed in the next section.

Writing Text to a File

Depending on the format of an NSTextView’s text, you use slightly different approaches to write the text to a file. For plain text, you extract the contents of the NSTextView as an NSString object and use NSString’s **writeToFile:atomically:** method to write the data to disk. RTF text is treated similarly, except that the contents is extracted as an NSData object. Easiest of all is RTFD data, which the NSTextView itself knows how to write to a file:

```
- (void)saveFile:(id)sender
{
    NSSavePanel *panel = [NSSavePanel savePanel];

    switch (theFormat) {
      case PlainText:
        [panel setRequiredFileType:@""];
        if ([panel runModal] == NSOKButton) {
            [[theTextView string] writeToFile:[panel filename]
                atomically:YES];
        }
        break;
      case RichText:
        [panel setRequiredFileType:@"rtf"];
        if ([panel runModal] == NSOKButton) {
           [[theTextView RTFFromRange:NSMakeRange(0, [[theTextView
string]
             length])] writeToFile:[panel filename] atomically:YES];
        }
        break;
      case RTFD:
        [panel setRequiredFileType:@"rtfd"];
        if ([panel runModal] == NSOKButton) {
            [theTextView writeRTFDToFile:[panel filename]
atomically:YES];
        }
        break;
      default:
        NSRunAlertPanel(@"Save Error",
           @"Couldn’t save file (unknown data format).\n", nil, nil,
nil);
        break;
    }
    return;
}
```

Putting an NSTextView Object in an NSScrollView

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

The process consists of three steps: setting up the NSScrollView, setting up the NSTextView, and assembling the pieces.

Assuming an object has the variable **theWindow** that represents the window where the scrolling view is displayed, you can set up the NSScrollView like this:

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

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

Note that we create an NSScrollView that completely covers the content area of the window it’s displayed in. We also specify a vertical scroll bar but no horizontal scroll bar, since this scrolling text view wraps text within the horizontal extent of the NSTextView, but lets text flow beyond the vertical extent of the NSTextView.

Finally, we set how the NSScrollView reacts when the window it’s displayed in changes size. By turning on the NSViewWidthSizable and NSViewHeightSizable bits of its resizing mask, we ensure that the NSScrollView grows and shrinks to match the window’s dimensions.

The next step is to create and configure an NSTextView to fit in the NSScrollView:

```
theTextView = [[NSTextView alloc] initWithFrame:NSMakeRect(0, 0,
    contentSize.width, contentSize.height)];
[theTextView setMinSize:NSMakeSize(0.0, contentSize.height)];
[theTextView setMaxSize:NSMakeSize(1e7, 1e7)];
[theTextView setVerticallyResizable:YES];
[theTextView setHorizontallyResizable:NO];
[theTextView setAutoresizingMask:NSViewWidthSizable];

[[theTextView textContainer] setContainerSize:contentSize];
[[theTextView textContainer] setWidthTracksTextView:YES];
```

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

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

The last message in this step is to the NSTextContainer, not the NSTextView. It tells the NSTextContainer to resize its width according to the width of the NSTextView. Recall that the text-handling system lays out text according to the dimensions stored in NSTextContainer objects. An NSTextView provides a place for the text to be displayed, but its dimensions and those of its NSTextContainer can be quite different. The **setWidthTracksTextView:**YES message ensures that as the NSTextView is resized, the dimensions stored in its NSTextContainer are likewise resized, causing the text to be laid out within the new boundaries.

The last step is to assemble and display the pieces:

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

Working with the Text-Handling System: Intermediate Operations

The previous section discussed basic operations that can be implemented using the NSTextView and NSTextContainer classes. This section explores those classes in greater depth and brings in the other major classes of the text-handling system, showing you how to use them to accomplish various goals.

Changing Character Attributes

Interface Builder's Font and Text menus offer many standard commands for altering text attributes: Bold, Superscript, Center, and so on. These work by invoking standard action methods, such as **changeFont:**, **superscript:**, and **alignCenter:**, that effect a specific change in one step. But how do you define and implement new commands?

Let's say that you want to define a command that emphasizes the selected text in some way. For example, using Interface Builder, you wish to add a menu command that sends an **emphasizeText:** message, perhaps to a custom object that owns an NSTextView. The custom object then sets the font in the NSTextView.

In doing so, the custom object can invoke any NSTextView method that changes attributes, but it must first ask for permission to do so and inform the NSTextView that changes are occurring so that the NSTextView can batch them together and send out the appropriate notifications to observers. Given this, and assuming the custom object has an instance variable (named **theTextView**) that identifies the NSTextView containing the selection, you can implement the **emphasizeText:** method like this:

```
- (void)emphasizeText:(id)sender
{
    NSRange changeCharRange = [theTextView
        rangeForUserCharacterAttributeChange];

    if ([theTextView shouldChangeTextInRange:changeCharRange
                     replacementString:nil]) {
        [theTextStore beginEditing];

        [myTextView setFont:[NSFont fontWithName:@"Helvetica-Oblique"
            size:12.0] range:changeCharRange];

        [theTextStore endEditing];
        [theTextView didChangeText];
    }

    return;
}
```

The custom object gets the range of the selected text and then applies a new font to that range. It then determines the range of characters that should be changed, and proceeds to attempt the change. To do so it invokes **shouldChangeTextInRange:replacementString:**, which gives the NSTextView's delegate a chance to reject the change. If the change is approved, this method sets the font of the characters being changed, bracketing the change with **beginEditing** and **endEditing** messages that allow the NSTextView to optimize multiple changes (though only one change is made here). Finally, this method invokes **didChangeText** to send out the appropriate delegate message and notifications.

This implementation sent a **setFont:range:** message to the NSTextView to effect its change. NSTextView defines other, similar, methods to set some common attributes (such as font, text color, and alignment). These are “cover” methods that hide the work of invoking the NSTextStorage methods that actually modify the attributed string. If you want to set attributes other than those accessible through the NSTextView API, you have to interact more intimately with the NSTextStorage object.

Fortunately, working with NSTextStorage is quite straightforward. For example, a reimplemented **emphasizeText:** method that acts on the underlying NSTextStorage object looks like this:

```
- (void)emphasizeText:(id)sender
{
    NSTextStorage *theTextStore = [theTextView textStorage];
    NSRange changeCharRange = [theTextView
        rangeForUserCharacterAttributeChange];

    if (changeCharRange.location == NSNotFound) return;

    if ([theTextView shouldChangeTextInRange:changeCharRange
                     replacementString:nil]) {
        [theTextStore beginEditing];

        [theTextStore addAttribute:NSFontAttributeName
         value:[NSFont fontWithName:@"Helvetica-Oblique" size:12.0]
           range:changeCharRange];

        [theTextStore endEditing];
        [theTextView didChangeText];
    }

    return;
}
```

Except for interacting with the NSTextStorage instead of the NSTextView, this implementation is identical to the first one, asking for permission to make the change, and informing the NSTextView as things proceed.

Regarding the change itself: An NSTextStorage object stores text attributes in dictionaries (see the NSDictionary class specification for more information). Each range of characters that share the same attributes conceptually share a dictionary. Within the dictionary, attributes are identified by a key which has an associated value. In the preceding implementation of **emphasizeText:**, the attribute we add to the selected text is identified by the globally scoped key NSFontAttributeName whose value is set to the NSFont object representing the Helvetica-Oblique type face.

Perhaps setting the font to an oblique angle doesn't provide enough emphasis, so you decide to additionally have the text drawn in blue on a red background. You can accomplish this by sending two more **addAttributeValue:range:** messages, in which case the **beginEditing** and **endEditing** messages are required, and not merely good coding practice. However, since you plan to use this set of attributes repeatedly, a better idea is to create a dictionary containing this set. This dictionary defines a style that you can use repeatedly:

```
NSDictionary *emphasisAttributes = [NSDictionary
    dictionaryWithObjectsAndKeys:
        [NSColor blueColor],NSForegroundColorAttributeName,
        [NSColor redColor], NSBackgroundColorAttributeName,
        [NSFont fontWithName:@"Helvetica-Oblique" size:12.0],
        NSFontAttributeName, nil];

- (void)emphasizeText:(id)sender
{
    NSTextStorage *theTextStore = [theTextView textStorage];
    NSRange changeCharRange = [theTextView
        rangeForUserCharacterAttributeChange];

    if (changeCharRange.location == NSNotFound) return;

    if ([theTextView shouldChangeTextInRange:changeCharRange
                     replacementString:nil]) {
        [theTextStore beginEditing];

        [theTextStore addAttributes:emphasisAttributes
            range:changeCharRange];

        [theTextStore endEditing];
        [theTextView didChangeText];
    }
    return;
```

Note the use of the **addAttributes:range:** method. This method is similar to the **addAttribute:range:** method, but applies a dictionary of attributes rather than a single attribute. With either method, an added attribute replaces an existing one. For example, if the foreground color is set to green and you then invoke the **emphasizeText:** method above, the new value of the foreground color is blue. Of course, this is the correct behavior and is a result of storing attributes in a dictionary, where a given key can have only one value.

Assembling the Text System by Hand

You build the network of objects that make up the text-handling system from the bottom up, starting with the NSTextStorage object. Here's the process:

1. Set up an NSTextStorage object.

You create an NSTextStorage object in the normal way, using the **alloc** and **init...** messages. In the simplest case, where there's no initial contents for the NSTextStorage, the initialization looks like this

```
textStorage = [[NSTextStorage alloc] init];
```

If, on the other hand, you want to initialize an NSTextStorage object with rich text data from a file, the initialization looks like this (assume **fileName** is defined):

```
NSAttributedString *attrString = [NSAttributedString
    attributedStringFromRTF:[NSData
dataWithContentsOfFile:fileName]];

textStorage = [[NSTextStorage alloc]
    initWithAttributedString:attrString];
```

We've assumed that **textStorage** is an instance variable of the object that contains this method. When you create the text-handling system by hand, you need to keep a reference only to the NSTextStorage object as we've done here. The other objects of the system are owned either directly or indirectly by this NSTextStorage object, as you'll see in the next steps.

2. Set up an NSLayoutManager object:

   Next, create an NSLayoutManager object:

```
NSLayoutManager *layoutManager;

layoutManager = [[NSLayoutManager alloc] init];
[textStorage addLayoutManager:layoutManager];
[layoutManager release];
```

   Note that **layoutManager** is released after being added to **textStorage**. This is because the NSTextStorage object retains each NSLayoutManager that's added to it—that is, the NSTextStorage object *owns* its NSLayoutManagers.

   The NSLayoutManager 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. You only need to connect the NSLayoutManager to the NSTextStorage object and to the NSTextContainer object, as seen in the next step.

3. Set up an NSTextContainer object.

Next, create an NSTextContainer and initialize it with a size. Assume that **theWindow** is defined and represents the window that displays the text view.

```
NSRect cFrame = [[theWindow contentView] frame];
NSTextContainer *container;

container = [[NSTextContainer alloc]
    initWithContainerSize:cFrame.size];
[layoutManager addTextContainer:container];
[container release];
```

Once you've created the NSTextContainer, you add it to the list of containers that the NSLayoutManager owns, and then you release it. The NSLayoutManager now owns the NSTextContainer and is responsible for releasing it when it's no longer needed. If your application has multiple NSTextContainers, you can create them and add them at this time.

4. Set up an NSTextView object.

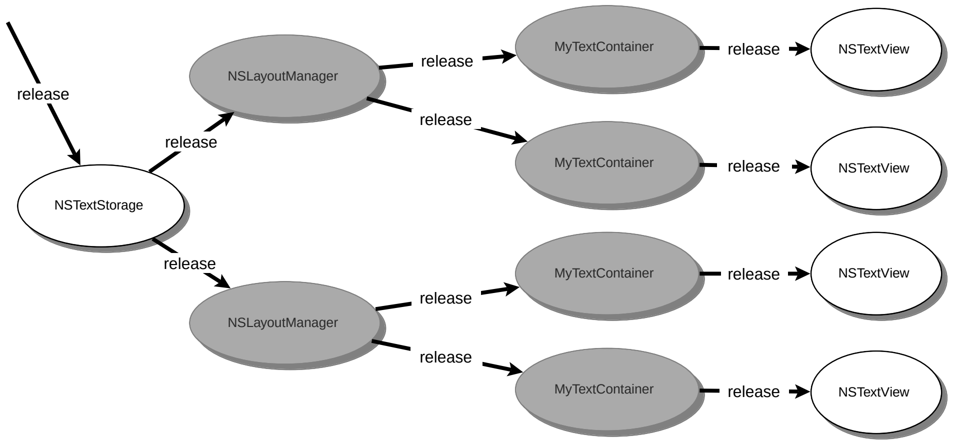
   Finally, create the NSTextView (or NSTextViews) that displays the text:

```
NSTextView *textView = [[NSTextView alloc]
    initWithFrame:cFrame textContainer:container];

[theWindow setContentView:textView];
[theWindow makeKeyAndOrderFront:nil];

[textView release];
```

Note that we use **initWithFrame:textContainer:** to initialize the NSTextView. This initialization method does nothing more than what it says: initialize the receiver and set its text container. This is in contrast to **initWithFrame:**, as discussed in "Creating an NSTextView Programmatically" above, which not only initializes the receiver, but creates and interconnects the network of objects that make up the text-handling system. Once the NSTextView has been initialized, it's added to the window, which is then displayed. Finally, you release the NSTextView.

Note that in creating the text-handling network by hand, we created four objects but then released three as they were added to the network. We are left with a reference only to the NSTextStorage object. The NSTextView is retained by both its NSTextContainer and its superview, though; to fully destroy this group of text objects you must send **removeFromSuperview** to the NSTextView object and then release the NSTextStorage object.

An NSTextStorage object is conceptually the owner of any network of text-handling objects, no matter how complex. When you release the NSTextStorage object, it releases its NSLayoutManagers, which release their NSTextContainers, which in turn release their NSTextViews.



However, recall that the text system implements a simplified ownership policy for those whose only interaction with the system is through the NSTextView class. See “Creating an NSTextView Programmatically” above for more information.

The code in the four steps above overlooks an important issue: resizing. As the window is resized, does the text rewrap within the new boundaries? What happens when there’s more text than fits within the content view of the NSWindow? For information on these subjects, see the “Controlling the Size of the NSTextView” and “Putting an NSTextView Object in an NSScrollView” sections.