Text Layout Programming Guide for Cocoa

Cocoa > Text & Fonts

2008-10-15

Contents

Figures and Listings

Introduction to Text Layout Programming Guide

Text Layout Programming Guide describes how the Cocoa text system lays out text. Text layout is the process of converting a string of text characters, font information, and page specifications into lines of glyphs placed at specific locations on a page, suitable for display and printing.

Who Should Read This Document

You should read this document if you need to understand how the text system layout mechanism works and how to work directly with an `NSLayoutManager` object to accomplish the programming goals described in the articles.

To understand the information in this document, you should have read *Text System Overview*. You should also understand basic Cocoa programming conventions, such as delegation.

Organization of This Document

This programming topic contains the following articles:

- "The Layout Manager" (page 9) introduces the `NSLayoutManager` class, describing its features and explaining how it performs text layout.
- "Typesetters" (page 15) describes the responsibilities of the typesetter object, instantiated from a concrete subclass of `NSTypesetter`, which generates the line fragments and glyph positions on behalf of the layout manager.
- "Line Fragment Generation" (page 19) explains how the typesetter and text container work together to create line fragment rectangles.
- "Drawing Strings" (page 23) explains how to use the layout manager, rather than `NSString` convenience methods, to draw strings of text efficiently.
- "Laying Out Text Along an Arbitrary Path" (page 21) shows how to use the layout manager without a text view to lay out glyphs along a calculated path.
- "Calculating Text Height" (page 25) shows how to determine the height of a block of text laid out in a fixed-width area.
- "Counting Lines of Text" (page 27) explains how you can programmatically count the number of lines in a string of text, whether the lines are defined by hard line-break characters or laid out in a text container.
- "Using Text Tables" (page 29) explains how you can add text table support to your application in Mac OS X version 10.4 and later.

See Also

For further reading, refer to the following documents:

- *Text System Storage Layer Overview* discusses the facilities that the Cocoa text system uses to store the text and geometric shape information used for text layout.
- *Text Attributes* describes the text-related attributes maintained by the Cocoa text system, which provide the distinguishing characteristics of rich text and other formatting information for paragraphs and documents.

In addition, please refer to Cocoa Text and Fonts Sample Code and the Application Kit examples installed with Xcode Tools.

The Layout Manager

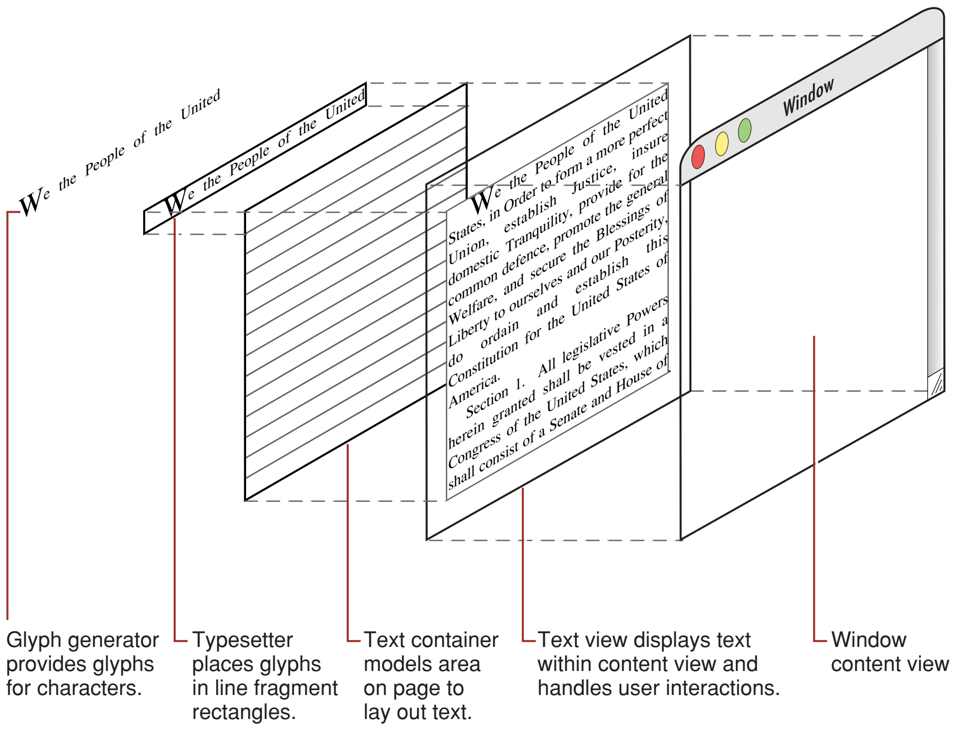
The layout manager class, `NSLayoutManager`, provides the central controlling object for text display in the Cocoa text system.

An `NSLayoutManager` object performs the following actions:

- Controls text storage and text container objects
- Generates glyphs from characters
- Computes glyph locations and stores the information
- Manages ranges of glyphs and characters
- Draws glyphs in text views when requested by the view
- Manages rulers for paragraph style control
- Computes bounding box rectangles for lines of text
- Controls hyphenation
- Manipulates character and glyph attributes

In the model-view-controller paradigm, `NSLayoutManager` is the controller. `NSTextStorage`, a subclass of `NSMutableAttributedString`, provides part of the model, holding a string of text characters with attributes such as typeface, style, color, and size. `NSTextContainer` can also be considered part of the model because it models the geometric layout of the page on which the text is laid out. `NSTextView` (or another `NSView` object) provides the view in which the text is displayed. `NSLayoutManager` serves as the controller for the text system because it directs the glyph generator to translate characters in the text storage object into glyphs, directs the typesetter to lay them out in lines according to the dimensions of one or more text container objects, and coordinates the text display in one or more text view objects. Figure 1 illustrates the composition of the text display, which is coordinated by the layout manager.

Figure 1 Composition of text display


We the People of the United
Window
Glyph generator provides glyphs for characters.
Typesetter places glyphs in line fragment rectangles.
Text container models area on page to lay out text.
Text view displays text within content view and handles user interactions.
Window content view

You can configure the text system to have more than one layout manager if you need the text in the same `NSTextStorage` object to be laid out in more than one way. For example, you might want the text to appear as a continuous galley in one view and to be segmented into pages in another view. For more information about different arrangements of text objects, refer to "Common Configurations."

Thread Safety

Generally speaking, a given layout manager (and associated objects) should not be used on more than one thread at a time. Most layout managers are used on the main thread, since it is the main thread on which their text views are displayed, and since background layout occurs on the main thread. However, you can lay out and render text on secondary threads using `NSLayoutManager` as long as the object graph is contained in a single thread.

If you must use a layout manager on a secondary thread, it's the application's responsibility to ensure that the objects are not accessed simultaneously from other threads. First, make sure that `NSTextView` objects associated with that layout manager (if any) are not displayed while the layout manager is being used on the secondary thread by disabling background layout and auto-display. For example, you could send the text view `lockFocusIfCanDraw` to block the main thread display (and send `unlockFocus` when finished). Second, turn off background layout for that layout manager while it is being used on the secondary thread by sending `setBackgroundLayoutEnabled:` with `NO`.

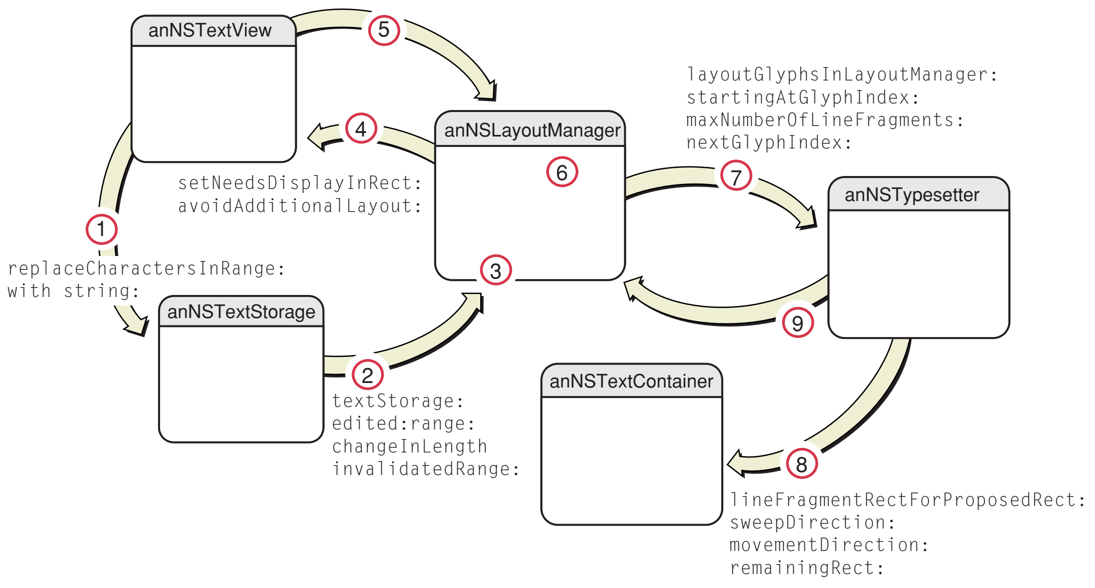
The Layout Process

The layout manager performs text layout in two separate steps: glyph generation and glyph layout. The layout manager performs both layout steps lazily, that is, on an as-needed basis. Accordingly, some `NSLayoutManager` methods cause glyph generation to happen, while others do not, and the same is true with glyph layout. After it generates glyphs and after it calculates their layout locations, the layout manager caches the information to improve performance of subsequent invocations.

The layout manager caches glyphs, attributes, and layout information. It keeps track of ranges of glyphs that have been invalidated by changes to the characters in the text storage. There are two ways in which a character range can be automatically invalidated: if it needs glyphs generated or if it needs glyphs laid out. If you wish, you can manually invalidate either glyph or layout information. When the layout manager receives a message requiring knowledge of glyphs or layout in an invalidated range, it generates the glyphs or recalculates the layout as necessary.

`NSLayoutManager` uses an `NSTypesetter` object to perform the actual glyph layout. See "Typesetters" (page 15) for more information. Figure 2 illustrates the interaction of objects involved in the layout process.

Figure 2 The text layout process



The following steps, numbered to correlate with the numbers in Figure 2, explain how the layout manager controls text layout:

1. Text in the text storage changes, invalidating glyphs or their layout positions or both. Invalidation occurs, for example, because the user edits the text in a text view, and the text view causes the changes to the contents of the text storage. Or another object can change the text programmatically.

2. The text storage notifies its associated layout manager (or managers) of the invalidated character range by sending the message `textStorage:edited:range:changeInLength:invalidatedRange:`. The message specifies whether the change affected characters, attributes, or both; the range of characters that changed; and the range affected after attribute fixing.

3. The layout manager updates its internal data structures to reflect the invalid range. Attribute changes may or may not affect glyph generation and layout. For example, changing the color of text does not affect how it gets laid out.

4. To notify its associated text views that they need to redisplay the invalidated area, the layout manager sends the message `setNeedsDisplayInRect:avoidAdditionalLayout:`. At this point any of several things could happen. If the invalidated portion of the text view is visible, the text view asks the layout manager for any needed glyphs and their positions. If the invalidated area is not currently visible, the view does not immediately call for layout. However, background (idle-time) layout may occur when the application has no events to process. Background layout is on by default, although you can turn it off for any individual layout manager.

5. When the text view asks the layout manager for glyphs and positions, the layout process begins. (Other messages can also invoke layout. The layout manager header file and reference documentation specify which methods cause glyph generation and layout to occur.)

6. The layout manager generates a stream of glyphs from the newly edited character range and caches the glyphs. Glyph generation is a quick first-pass conversion of characters in a particular font to glyphs. (Without font information, glyphs cannot be generated.) Later stages of the layout process can make changes to the glyph stream.

7. After the required glyphs have been generated, the layout manager calls its typesetter to lay out the glyphs into one or more line fragments, sending the `layoutGlyphsInLayoutManager:startingAtGlyphIndex:maxNumberOfLineFragments:nextGlyphIndex:` message to the typesetter. During this process, the typesetter may perform glyph substitution; for example, it can substitute a ligature glyph in place of two or more single-character glyphs.

8. The typesetter generates line fragment rectangles in communication with the text container and determines the placement of each glyph, as described in "Typesetters" (page 15) and "Line Fragment Generation" (page 19).

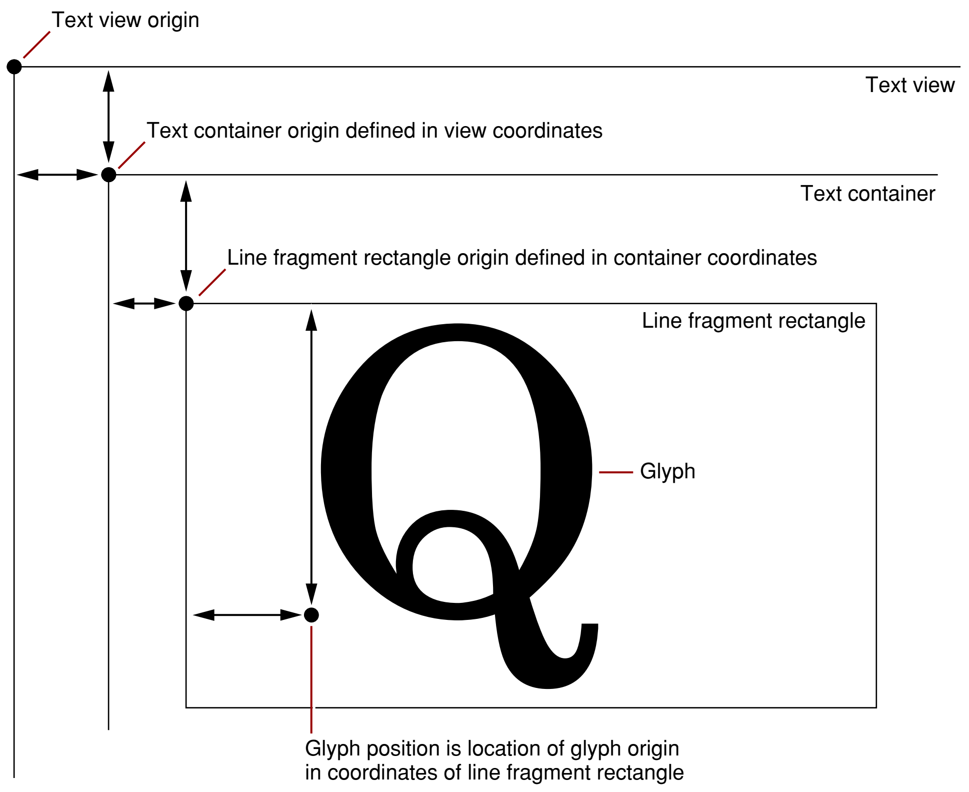
9. The typesetter sends the line fragment rectangles with their glyphs and positions to the layout manager, which commits the information in its internal data structures as valid layout.

Glyph Drawing

In addition to generating glyphs and performing layout, the layout manager does the drawing of the glyphs in the text view. Drawing occurs when the text view asks the layout manager to figure out which glyphs lie within a given view rectangle and to display them. The layout manager has methods for drawing glyphs and their background. These methods do all the necessary drawing by calling into the Quartz graphic layer. They draw the background, set up the font and color, and draw the glyphs, underlines, and any temporary attributes.

Most `NSLayoutManager` methods use container coordinates, rather than view coordinates. The text system expects view coordinates to be flipped, like those of `NSTextView`. If you have a point in view coordinates that you need to convert to container coordinates, subtract the text view's `textContainerOrigin` value to get the container coordinates. Glyph locations are expressed relative to their line fragment bounding rectangle's origin. Figure 3 shows the relationships among these coordinate systems.

Figure 3 View, container, and line fragment coordinates


Text view origin
Text view
Text container origin defined in view coordinates
Text container
Line fragment rectangle origin defined in container coordinates
Line fragment rectangle
Glyph
Glyph position is location of glyph origin
in coordinates of line fragment rectangle

Attributes are qualities the layout manager applies to characters during typesetting and layout, such as font, size, and color. The text storage preserves many attributes in a dictionary stored with the character string, but other attributes are temporary and maintained only by the layout manager during the layout process. Temporary attributes supersede attributes associated with the font or paragraph. The drawing methods also handle additional attributes associated with the text view—for example, a different background color—when the glyphs being drawn are selected in the text view. The drawing methods call some other public `NSLayoutManager` methods, such as `drawUnderlineForGlyphRange:`, which you can override if you want to do things differently. See *Text Attributes* for more information.

The layout manager also handles the representation of attachments during glyph drawing. The text system stores an attachment as an attribute of a special character. A typical attachment is a file, but it can also be in-memory data. A file attachment is usually handled by drawing an icon. However, an attachment can do much more than that if you implement different behavior. During layout the attachment cell (`NSTextAttachmentCell`) tells the layout manager what size it is, so it can be laid out like a glyph. Accordingly, the text's line height and character placement are adjusted to accommodate the attachment cell. During drawing, the layout manager asks the attachment cell to draw itself. See *Text Attachment Programming Topics for Cocoa* for more information.

The layout manager retains and reuses as much layout information as possible, to minimize recalculating glyph positions. For example, if the glyphs have already been generated for an invalidated character range that needs to be laid out, the layout manager tries to optimize the layout process. In the best case, such holes in the layout can be filled just by shifting line fragment locations within the text container.

`NSLayoutManager` provides a public API for getting glyphs from characters. The process is complex, however: You cannot simply convert a single character into a glyph because the relationship between characters and glyphs is many-to-many. That is, one character in the text storage can map to multiple glyphs and vice versa.

Therefore, you use the `NSLayoutManager` methods `glyphRangeForTextContainer:` to get the glyphs for all the characters laid out in a text container or `glyphRangeForCharacterRange:actualCharacterRange:` for a range of characters.

Typesetters

The layout manager uses a helper object called a typesetter to lay out glyphs in line fragments. Typesetter objects are instantiated from a concrete subclass of `NSTypesetter`.

Working with other objects in the Cocoa text system, the typesetter creates line fragment rectangles, places glyphs within the line fragments, determines line breaks by word wrapping and hyphenation, and handles tab positioning. The typesetter also determines interline spacing, paragraph spacing, and the right-to-left positioning of bidirectional glyphs.

Filling Line Fragment Rectangles

The typesetter object generates line fragments by communicating with the text container, as described in "Line Fragment Generation" (page 19). The typesetter determines the suitable line fragment sizes and positions, which it returns in container coordinates.

After creating a line fragment rectangle, the typesetter determines the positions of glyphs within it, in response to the `layoutGlyphsInLayoutManager:startingAtGlyphIndex:maxNumberOfLineFragments:nextGlyphIndex:` message from the layout manager. The typesetter reports the glyph locations relative to the origin of their line fragment's bounding rectangle. The typesetter fills the line fragment until it goes beyond the line fragment's width. Then it creates a line break by wrapping text or hyphenating the last word. In this step, the typesetter performs glyph substitution, if necessary, and may add glyphs to the glyph stream. For example, the typesetter may substitute a ligature glyph for one or more single-character glyphs, or it may add a hyphen to the glyph stream.

Whenever the width of the laid-out line, divided by the width of the line rectangle, exceeds a hyphenation threshold maintained by the layout manager, the typesetter calls an internal hyphenator object which attempts to find hyphenation points in the last word in the line. If the hyphenator finds a good point, the typesetter inserts a hyphen glyph at the end of the line fragment rectangle.

Hyphenation is controlled by a threshold called the hyphenation factor, which is maintained by the layout manager. You can set the threshold using the `NSLayoutManager` method `setHyphenationFactor:`. the hyphenation factor is a float that ranges between 0.0 and 1.0. By default, its value is 0.0, meaning hyphenation is off. Setting the hyphenation factor to 1.0 causes the typesetter to attempt hyphenation always.

Typesetter Behaviors and Versions

The text system uses a shared, reentrant typesetter instance, made available by the `NSLayoutManager` method `typesetter`. The `NSLayoutManager` method `setTypesetterBehavior:` selects among the original default typesetter shipped with Mac OS X prior to version 10.2, a typesetter encapsulating Apple

Type Services (ATS) that shipped with Mac OS X version 10.2, an enhanced version of the ATS-based typesetter that shipped with Mac OS X version 10.3., and the typesetter behavior introduced in Mac OS X version 10.4. The `NSTypesetterBehavior` enumeration defines the relevant constants.

The `NSTypesetter` subclass that implements the original typesetter behavior is `NSSimpleHorizontalTypesetter`, which is defined in the `NSTypesetter.h` header file. `NSSimpleHorizontalTypesetter` supports glyph layout with a left-to-right sweep and downward line movement only. `NSSimpleHorizontalTypesetter` is deprecated in Mac OS X version 10.4 and later.

The typesetter behavior introduced in Mac OS X version 10.2 is implemented by the `NSATSTypesetter` class, which is defined in the `NSATSTypesetter.h` header file. `NSATSTypesetter` provides enhanced line and character spacing accuracy and supports more languages, including bidirectional languages, than the original `NSSimpleHorizontalTypesetter`.

Mac OS X version 10.3 introduced a new version of the `NSATSTypesetter` that declares public APIs for `NSATSTypesetter` and `NSGlyphGenerator`. These APIs open up the typesetter for use with a custom layout engine having a design different from the traditional Cocoa text system, as described in "Design of NSTypesetter" (page 16). In Mac OS X version 10.4, these APIs moved to `NSTypesetter`.

Unless you require a specific behavior of an earlier typesetter version, you should use or subclass the latest version of `NSATSTypesetter`.

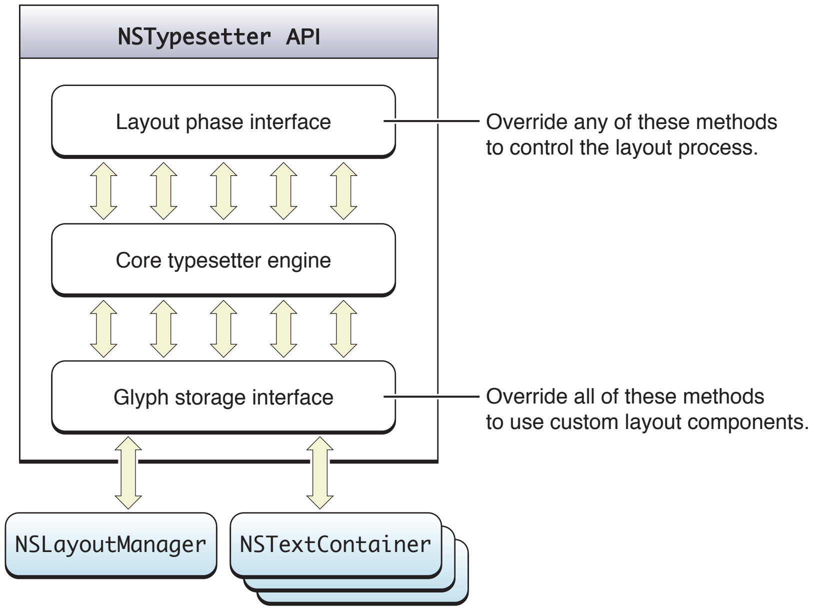
It is important to use the same typesetter behavior when both measuring and rendering text, to avoid differences in paragraph spacing, line spacing, and head indent handling. See "String Drawing and Typesetter Behaviors" (page 24) for more information about typesetter behavior mismatches.

Design of NSTypesetter

In the Cocoa text system, the layout manager owns the typesetter and glyph generator as private objects and maintains an array of text containers, as described in "The Layout Manager" (page 9). The typesetter concept is tightly coupled with the layout manager and text container concepts. The typesetter's responsibility is to fill the text containers in the array with glyphs supplied by the glyph generator. By default, `NSATSTypesetter` works in this way. However, `NSTypesetter` is designed to enable developers to decouple it from the other components of the Cocoa text system.

The design of `NSTypesetter` isolates the primitive, core typesetter from the rest of the Cocoa text system, as shown in Figure 1. `NSTypesetter` has a core typesetting engine, a layout phase interface, and a glyph storage interface layer that communicates with the text system and drives the layout engine. The core typesetting engine encapsulates the complexity of the Apple Type Services for Unicode Imaging (ATSUI) to provide advanced typographic capabilities through a simplified API. The core typesetting engine lays out glyphs in an infinite horizontal line and knows nothing about text containers or text direction. The glyph storage interface layer calls out to the text system to generate line fragment rectangles and make sure they fit onto the page properly.

Figure 1 Design of NSTypesetter


NSTypesetter API
Layout phase interface
Override any of these methods to control the layout process.
Core typesetter engine
Glyph storage interface
Override all of these methods to use custom layout components.
NSLayoutManager
NSTextContainer

The API design for `NSTypesetter` has two primary goals. The first is to break the tie between the two classes and `NSLayoutManager`, allowing developers to tap deeply into Cocoa's typographic capabilities without using `NSLayoutManager`. The second goal is to provide override points to allow developers to extend various aspects of the typesetting process. In addition, direct access to these classes makes it easier to port Carbon, Windows, or UNIX applications with their own layout engines to Cocoa.

`NSTypesetter` categorizes its methods as follows:

- The glyph storage interface (`NSGlyphStorageInterface`) declares all the primitive methods interfacing to the glyph storage facility, which is `NSLayoutManager` in Cocoa. By overriding all these methods, an application can implement an `NSTypesetter` subclass that interacts with a custom glyph storage facility and layout manager. The default implementations of these methods call `NSLayoutManager`.
- The layout phase interface (`NSLayoutPhaseInterface`) declares control points called during text layout, if implemented. These method calls act as notifications of events occurring in the layout process. An `NSTypesetter` subclass can override any of these methods, if desired, to modify various aspects of the layout process. For example, the typesetter calls `willSetLineFragmentRect:forGlyphRange:usedRect:baselineOffset:` immediately before it calls `setLineFragmentRect:forGlyphRange:usedRect:baselineOffset:` to store the actual line fragment rectangle location in the layout manager.
- The remainder of the `NSTypesetter` methods are primitive typesetter methods that a custom layout manager can call to control the typesetter directly.

With its layered design, `NSTypesetter` can be instantiated and used in its standard configuration with the Cocoa text system or subclassed and adapted to work with another text system, even one that has entirely different concepts of how to perform page layout.

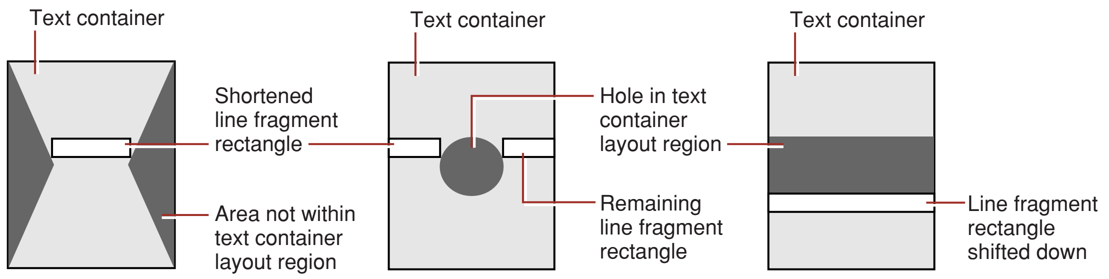
Line Fragment Generation

An `NSTypesetter` object lays text within an `NSTextContainer` object in lines of glyphs. The layout of these lines within an `NSTextContainer` object is determined by its shape. For example, if the text container is narrower in some parts than in others, the lines in those parts must be shortened; if there are holes in the region, some lines must be fragmented; if there's a gap across the entire region, the lines that would overlap it have to be shifted to compensate.

The built-in typesetters currently provided with text system support only horizontal text layout. However, the text system can support typesetters that lay out text along lines that run either horizontally or vertically, and in either direction. This type of movement is called the **sweep direction** and is expressed by the `NSLineSweepDirection` type in Objective-C and the sweep direction constants in Java. The direction in which the lines then progress is called the **line movement direction** and is expressed by the `NSLineMovementDirection` type in Objective-C and the line movement constants in Java. Each affects the adjustment of a line fragment rectangle in a different way: The rectangle can be moved or shortened along the sweep direction and shifted (but not resized) in the line movement direction.

The typesetter object proposes a rectangle for a given line, and then asks the `NSTextContainer` object to adjust the rectangle to fit. The proposed rectangle usually spans the text container's bounding rectangle, but it can be narrower or wider, and it can also lie partially or completely outside the bounding rectangle. The message that the typesetter sends the text container to adjust the proposed rectangle is `lineFragmentRectForProposedRect:sweepDirection:movementDirection:remainingRect:`, which returns the largest rectangle available for the proposed rectangle, based on the direction in which text is laid out. It also returns a rectangle containing any remaining space, such as that left on the other side of a hole or gap in the text container. This process is illustrated in the Figure 1.

Figure 1 Line fragment fitting in irregular text containers


Text container
Shortened line fragment rectangle
Area not within text container layout region
Text container
Hole in text container layout region
Remaining line fragment rectangle
Text container
Line fragment rectangle shifted down

For the three examples in Figure 1, the sweep direction is `NSLineSweepRight` and the line movement direction is `NSLineMovesDown`. In the first example, the proposed rectangle spans the region's bounding rectangle and is shortened by the text container to fit inside the hourglass shape with no remainder.

In the second example, the proposed rectangle crosses a hole, so the text container must return a shorter rectangle (the white rectangle on the left) along with a remainder (the white rectangle on the right). The next rectangle proposed by the typesetter will then be this remainder rectangle which will be returned unchanged by the text container.

In the third example, a gap crosses the entire text container. Here the text container shifts the proposed rectangle down until it lies completely within the container's region. If the line movement direction here were `NSLineDoesntMove`, the text container would have to return `NSRect.ZeroRect` indicating that the line simply doesn't fit. In such a case it's up to the typesetter to propose a different rectangle or to move on to a different container. When a text container shifts a line fragment rectangle, the layout manager takes this into account for subsequent lines.

The typesetter makes one final adjustment when it actually fits text into the rectangle. This adjustment is a small amount fixed by the `NSTextContainer` object, called the line fragment padding, which defines the portion on each end of the line fragment rectangle left blank. Text is inset within the line fragment rectangle by this amount (the rectangle itself is unaffected). Padding allows for small-scale adjustment of the text container's region at the edges and around any holes and keeps text from directly abutting any other graphics displayed near the region. You can change the padding from its default value with the `setLineFragmentPadding:` method. Note that line fragment padding isn't a suitable means for expressing margins; you should set the `NSTextView` object's position and size for document margins or the paragraph margin attributes for text margins.

In addition to the line fragment rectangle itself, the typesetter returns a rectangle called the used rectangle. This is the portion of the line fragment rectangle that actually contains glyphs or other marks to be drawn. By convention, both rectangles include the line fragment padding and the interline space calculated from the font's line height metrics and the paragraph's line spacing parameters. However, the paragraph spacing (before and after) and any space added around the text, such as that caused by center-spaced text, are included only in the line fragment rectangle and not in the used rectangle.

See "Layout Geometry: the NSTextContainer Class" for more information about text containers. See "The Layout Manager" (page 9) for more information about the layout process.

Laying Out Text Along an Arbitrary Path

The Cocoa text system typically lays out text in horizontal lines in a text view. However, it is also possible to use just the text objects needed to store characters and generate glyphs while manually calculating final glyph positions and drawing the glyphs yourself.

To lay out text along an arbitrary path, you need to use the three basic, nonview text objects: `NSTextStorage` to hold the text, `NSTextContainer` to model the region in which the text is laid out, and `NSLayoutManager` to generate the glyphs and layout information. Finally you draw the glyphs in a custom `NSView` object.

First you create and initialize instances of the text storage, text container, and layout manager. You initialize the text container with the string of text to be laid out. Then you hook the objects together: the text storage object retains a reference to the layout manager, and the layout manager retains a reference to the text container. Listing 1, which could reside in the initialization method for the custom `NSView` object that displays the text, illustrates this process.

Listing 1 Creating and configuring the nonview text objects

```
NSTextStorage *textStorage;
NSLayoutManager *layoutManager;
NSTextContainer *textContainer;

textStorage = [[NSTextStorage alloc] initWithString:@"This is the string of text
 in the text storage."];
layoutManager = [[NSLayoutManager alloc] init];
textContainer = [[NSTextContainer alloc] init];
[layoutManager addTextContainer:textContainer];
[textContainer release];
[textStorage addLayoutManager:layoutManager];
[layoutManager release];
```

The reason you “add” these references, rather than “set” them, is because a layout manager can have multiple text containers, and a text storage object can have multiple layout managers. Also note the memory management implications of this procedure: Because the layout manager retains the text container and the text storage retains the layout manager, you can release them as soon as you connect the objects. However, you should explicitly release the text storage object in your `dealloc` method.

Tell the layout manager not to use screen fonts, because they do not scale or rotate properly (by default, screen fonts are allowed):

```
[layoutManager setUsesScreenFonts:NO];
```

Next, force the layout manager to generate glyphs for the characters in the text storage object and have it calculate glyph positions laid out in a simple, rectangular container. You then transform the positions and call the layout manager to draw the glyphs. This can be done in the view’s `drawRect:` method.

The following message forces layout and returns glyphs for the character string in the text storage object:

```
NSRange glyphRange = [layoutManager
                       glyphRangeForTextContainer:textContainer];
```

The code in Listing 2 does the actual drawing.

Listing 2 Drawing the string

```
NSGraphicsContext *context = [NSGraphicsContext currentContext];
NSAffineTransform *transform = [NSAffineTransform transform];
[transform rotateByDegrees:30.0];
[context saveGraphicsState];
[transform concat];
[self lockFocus];
[layoutManager drawGlyphsForGlyphRange:glyphRange
                               atPoint:NSMakePoint(50.0, 50.0)];
[self unlockFocus];
[context restoreGraphicsState];
```

This fragment merely rotates the graphic context 30 degrees counterclockwise before it asks the layout manager to draw the glyphs, but another algorithm could be used to calculate a more complex layout path. This discussion simplifies the technique in order to concentrate on interactions with the layout manager.

The CircleView example distributed with the Xcode Tools provides complete source code for an application that illustrates the technique but does it in a more robust manner. CircleView calculates a position and draws each glyph individually. CircleView is located at the following path when Xcode Tools are installed:

```
/Developer/Examples/AppKit/CircleView
```

Drawing Strings

There are three ways to draw text programmatically in Cocoa: using methods of `NSString` or `NSAttributedString`, using those of `NSCell`, and using `NSLayoutManager` directly. `NSLayoutManager` is the most efficient.

Using the String-Drawing Convenience Methods

The `NSString` class has two convenience methods for drawing string objects directly in an `NSView` object: `drawAtPoint:withAttributes:` and `drawInRect:withAttributes:`. For strings that have multiple attributes associated with ranges and individual characters, you must use `NSAttributedString`. You can draw a string (in a focused `NSView`) with either the `drawAtPoint:` or `drawInRect:` method. These methods are designed for drawing small amounts of text or text that must be drawn rarely. They create and dispose of various supporting text objects, including `NSLayoutManager`, every time you call them.

For repeated drawing of text, however, the string drawing convenience methods are not efficient because they do a lot of work behind the scenes. For example, to draw Unicode text, you must first convert the characters into glyphs, the elements of a font. Glyph generation is complicated because several characters may produce a single glyph and vice versa, depending on the context and other factors. In addition, the system does a lot of work setting up for glyph conversion, and the string drawing convenience methods do this work each time they draw a string. Using the layout manager directly provides significant performance improvements because it caches glyph layout and size information.

Drawing Text With NSCell

The `NSCell` class also provides primitives for displaying and editing text. `NSCell` text drawing methods are used by `NSBrowser` and `NSTableView`. Text drawing by `NSCell` is more efficient than using the string convenience methods because it caches some information, such as the size of the text rectangle. So, for displaying the same text repeatedly, `NSCell` works well, but for the most efficient display of an arbitrary text string, use `NSLayoutManager` directly.

Drawing Text With NSLayoutManager

If you use the `NSTextView` class to display text, either by dragging a text view object from the Interface Builder Data palette or creating a text view programmatically using the `NSTextView` method `initWithFrame:` method, Cocoa automatically creates an `NSLayoutManager` instance to draw your text. If you create a text view using the `NSTextView` `initWithFrame:textContainer:` method, however, or if you need to draw text directly into a different type of `NSView` object, you must create the `NSLayoutManager` explicitly.

To use `NSLayoutManager` to draw a text string directly into a view, you must create and initialize the three basic nonview components of the text system. First create an `NSTextStorage` object to hold the string. Then create an `NSTextContainer` object to describe the geometric area for the text. Then create the `NSLayoutManager` object and hook the three objects together by adding the layout manager to the text storage object and adding the text container to the layout manager. The code in Listing 1, which could reside in the view's `initWithFrame:` method, illustrates this procedure.

Listing 1 Creating and configuring the nonview text objects

```
NSTextStorage *textStorage = [[NSTextStorage alloc]
    initWithString:@"This is the text string."];
NSLayoutManager *layoutManager = [[NSLayoutManager alloc] init];
NSTextContainer *textContainer = [[NSTextContainer alloc] init];
[layoutManager addTextContainer:textContainer];
[textContainer release];
[textStorage addLayoutManager:layoutManager];
[layoutManager release];
```

You can release the text container because the layout manager retains it, and you can release the layout manager because the text storage object retains it.

To draw glyphs directly in a view, you can use the `NSLayoutManager` method `drawGlyphsForGlyphRange:` in the view's `drawRect:` method. However, you must first convert the character range you want to draw into a glyph range. If you need to select a subrange of the text in the text storage object, you can use the `glyphRangeForCharacterRange:actualCharacterRange:` method. If you want to draw the entire string in the text storage object, use the `glyphRangeForTextContainer:` method as in Listing 2 (which uses the `layoutManager` variable name from Listing 1).

Listing 2 Drawing glyphs directly in a view

```
NSRange glyphRange = [layoutManager
    glyphRangeForTextContainer:textContainer];
[self lockFocus];
[layoutManager drawGlyphsForGlyphRange: glyphRange atPoint: rect.origin];
[self unlockFocus];
```

String Drawing and Typesetter Behaviors

There are differences among Cocoa's three ways to draw text with regard to typesetter behavior, which is described in "Typesetter Behaviors and Versions" (page 15). By default, the string-drawing convenience methods and `NSCell` objects supplied by the Application Kit use `NSTypesetterBehavior_10_2_WithCompatibility`, whereas `NSLayoutManager` objects use `NSTypesetterLatestBehavior`. It is important to use the same typesetter behavior when both measuring and rendering text, to avoid differences in paragraph spacing, line spacing, and head indent handling.

In cases where you must measure text one way and render it another, set the typesetter behavior to match using the `setTypesetterBehavior:` method defined by `NSLayoutManager` and `NSTypesetter`. For example, if you need to use an `NSLayoutManager` object to measure text and convenience string drawing methods to draw it, change the layout manager's typesetter behavior to `NSTypesetterBehavior_10_2_WithCompatibility`.

Calculating Text Height

There may be times when you need to know the height of a block of text formed by a text string after it is laid out in a fixed-width area. The `NSLayoutManager` class can do this very simply. This article illustrates the technique implemented in a single function.

Note: You don't need to use this technique to find the height of a single line of text. The `NSLayoutManager` method `defaultLineHeightForFont:` returns that value. The default line height is the sum of a font's tallest ascender, plus the absolute value of its deepest descender, plus its leading.

The basic technique for calculating text height uses the three basic nonview components of the text system: `NSTextStorage`, `NSTextContainer`, and `NSLayoutManager`. The text storage object holds the string to be measured; the text container specifies the width of the layout area; and the layout manager does the layout and returns the width.

To set up the text system for the calculation, you need the text string to be measured, a font for the string, and a width for the area modeled by the text container. You can pass these values into a function with a declaration such as the following:

```
float heightForStringDrawing(NSString *myString, NSFont *myFont,
    float myWidth);
```

The argument names in the function declaration appear as variables in the following code fragments that define the method body.

First, you instantiate the needed text objects and hook them together. You use the designated initializer for the text storage object, which takes the string pointer as an argument. Likewise, the text container's designated initializer takes the container size as its argument. You set the container width to your desired width and set the height to an arbitrarily large value, as shown in the following code fragment:

```
NSTextStorage *textStorage = [[[NSTextStorage alloc]
        initWithString:myString] autorelease];
NSTextContainer *textContainer = [[[NSTextContainer alloc]
        initWithContainerSize: NSMakeSize(myWidth, FLT_MAX)] autorelease];
NSLayoutManager *layoutManager = [[[NSLayoutManager alloc] init]
        autorelease];
```

Once the text objects are created, you can hook them together:

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

You don't need to release the text container and layout manager because you added them to the autorelease pool at initialization time. Next, set the font by adding the font attribute to the range of the entire string in the text storage object. Set the line fragment padding to `0` to get an accurate width measurement. (Padding is used in page layout to prevent the text in a text container from abutting too closely other elements on a page, such as graphics.)

```
[textStorage addAttribute:NSFontAttributeName value:myFont
```

```
        range:NSMakeRange(0, [textStorage length])];
[textContainer setLineFragmentPadding:0.0];
```

Finally, because the layout manager performs layout lazily, on demand, you must force it to lay out the text, even though you don't need the glyph range returned by this function. Then you can simply ask the layout manager for the height of the rectangle occupied by the laid-out text and, assuming this code is in a function implementation, return the value:

```
(void) [layoutManager glyphRangeForTextContainer:textContainer];
return [layoutManager
        usedRectForTextContainer:textContainer].size.height;
```

Counting Lines of Text

This task shows how to count the number of lines in a block of text programmatically. Lines can be defined by hard linebreak characters in the text string, or they can be lines generated by the text layout mechanism when it wraps the text to fit into a text container.

Lines of text defined by hard linebreak characters, such as carriage return and newline, are considered to be paragraphs by the text system. That is, the text layout engine generates line fragments sized to fit into a text container until it reaches a hard linebreak character, so the last fragment is typically shorter than the container width. However, if lines are laid out into a text container wider than the longest glyph run between hard linebreak characters, then each paragraph is a single line.

To count the number of hard linebreak characters in a text string, you can use the `NSString` methods `getLineStart:end:contentsEnd:forRange:` and `lineRangeForRange:`. These methods take a character range as input and return the lines that contain that range. They define lines as character ranges ending in carriage return, newline, carriage return and newline together (in that order, often called CRLF), and the Unicode characters for line separator and paragraph separator. For example, the code in Listing 1 puts into `numberOfLines` the number of lines in an `NSString` variable `string`.

Listing 1 Counting hard line breaks

```
NSString *string;
unsigned numberOfLines, index, stringLength = [string length];
for (index = 0, numberOfLines = 0; index < stringLength; numberOfLines++)
    index = NSMaxRange([string lineRangeForRange:NSMakeRange(index, 0)]);
```

This compact code fragment begins by making a range containing only the first character in string. The `lineRangeForRange:` method returns the line containing that character as a range that includes the hard linebreak character (or characters). The `NSMaxRange` function returns the index of the first character in the next line, which is the value assigned to `index`. The `numberOfLines` variable increments, and the `for` loop repeats, until `index` is greater than the length of `string`, at which point `numberOfLines` contains the number of lines in `string`, as defined by hard linebreak characters.

To count the lines generated by the text layout mechanism when it wraps the text to fit into a text container, you can get the information from the layout manager, as shown in Listing 2.

Listing 2 Counting lines of wrapped text

```
NSLayoutManager *layoutManager = [textView layoutManager];
unsigned numberOfLines, index, numberOfGlyphs =
        [layoutManager numberOfGlyphs];
NSRange lineRange;
for (numberOfLines = 0, index = 0; index < numberOfGlyphs; numberOfLines++){
    (void) [layoutManager lineFragmentRectForGlyphAtIndex:index
            effectiveRange:&lineRange];
    index = NSMaxRange(lineRange);
}
```

This code assumes you have a reference to a text view configured with a layout manager, text storage, and text container. The text view returns a reference to the layout manager, which then returns the number of glyphs for all the characters in its associated text storage, performing glyph generation if necessary. The `for` loop then begins laying out the text and counting the resulting line fragments. The `NSLayoutManager` method `lineFragmentRectForGlyphAtIndex:effectiveRange:` forces layout of the line containing the glyph at the index passed to it. The method returns the rectangle occupied by the line fragment (here ignored) and, by reference, the range of the glyphs in the line after layout. After the method calculates a line, the `NSMaxRange`function returns the index one greater than the maximum value in the range, that is, the index of the first glyph in the next line. The `numberOfLines` variable increments, and the `for` loop repeats, until `index` is greater than the number of glyphs in the text, at which point `numberOfLines` contains the number of lines resulting from the layout process, as defined by word wrapping.

This strategy causes layout of the entire text contained in the text storage object and calculates the number of lines required to lay it out, regardless of the number of text containers filled or text views required to display it. To obtain the number of lines in a single page (which is modeled by a text container), you would use the `NSLayoutManager` method `glyphRangeForTextContainer:` and restrict the line-counting `for` loop to that range, rather than the `{0, numberOfGlyphs}` range, which includes all of the text.

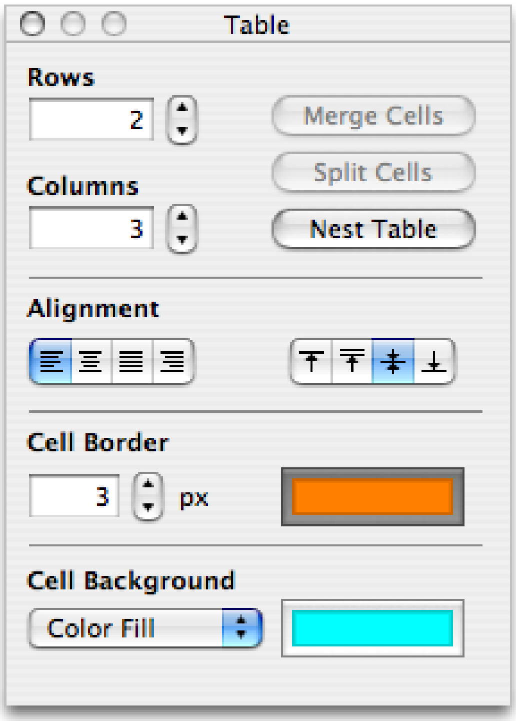
Using Text Tables

The Cocoa text system supports text tables in Mac OS X version 10.4 and later. The main classes involved are `NSTextTable`, which represents a table, `NSTextTableBlock`, which represents a block of text appearing as a cell in the table, and its superclass, `NSTextBlock`. This article explains how to add table support to your application.

Adding the Text Table Panel

`NSTextView` has built-in support for text tables, which provides the easiest way to add table support to your text view. This table support is in the form of the action method `orderFrontTablePanel:`. This method inserts a table into the text view and opens a modeless utility window that floats over the application windows. This table panel enables the user to manipulate attributes of a table while the cursor or selection is in the table. The table panel is shown in "Text table panel" (page 29).

Figure 1 Text table panel

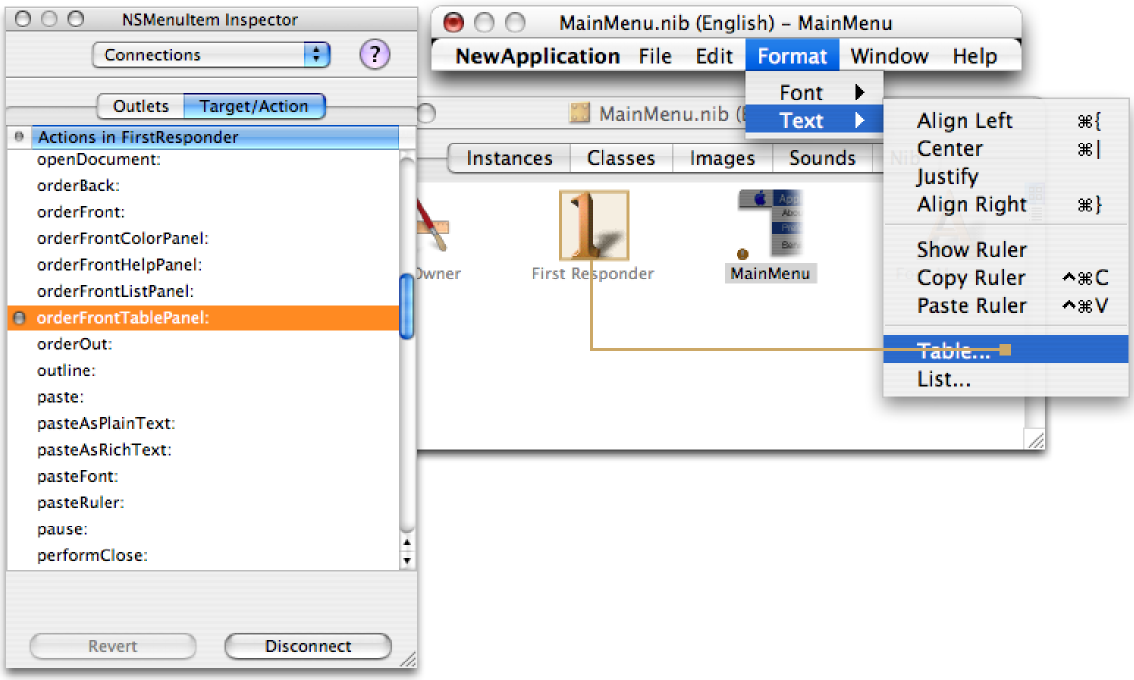


Rows: 2
Columns: 3
Cell Border: 3 px

The user can change other aspects of the table, such as cell size and contents, by direct manipulation with the cursor.

To make the text table panel available in your text view, use Interface Builder to add the `orderFrontTablePanel:` action method to your first responder and connect it to a menu item, as shown in "Connecting the action method" (page 30).

Figure 2 Connecting the action method



`NSTextView` defines similar action methods for opening list, link, and paragraph spacing panels.

Supporting Text Tables Programmatically

If you don't want to use the text table panel, you can support tables programmatically by using `NSTextTable` and related classes directly. The basic class in this group is `NSTextBlock`, which represents a block of text laid out in a subregion of a text container. When working with tables, you use its subclass, `NSTextTableBlock`, which represents a text block that appears as a cell in a table. The table itself is represented by a separate class, `NSTextTable`. All of the `NSTextTableBlock` objects, representing the cells in the table, refer to the `NSTextTable` object, which controls their size and positioning.

Text blocks appear as attributes on paragraphs, as part of the paragraph style. An `NSParagraphStyle` object can have an array of text blocks representing the table cells that contain the paragraph. The paragraph style uses an array because table cells can be nested, and the text blocks are ordered in the array from outermost to innermost. For example, if block1 contains four paragraphs, and the middle two are also in inner block2, then the text blocks array for first and fourth paragraphs is (block1) which the array for the second and third paragraphs is (block1, block2).

You add text blocks to a paragraph style object using the `NSMutableParagraphStyle` method `setTextBlocks:`, and the `NSParagraphStyle` method `textBlocks` returns the array.

To implement a text table programmatically, use the following sequence of steps:

1. Create an attributed string for the table.
2. Create the table object, setting the number of columns.
3. Create the text table block for the first cell of the row, referring to the table object.
4. Set the attributes for the text block.
5. Create a paragraph style object for the cell, setting the text block as an attribute (along with any other paragraph attributes, such as alignment).
6. Create an attributed string for the cell, adding the paragraph style as an attribute. The cell string must end with a paragraph marker, such as a newline character.
7. Append the cell string to the table string.
8. Repeat steps 3–7 for each cell in the table.

The methods shown in "Table creation method" (page 31) and "Table cell creation method" (page 32) perform the steps from the preceding list. (All of the example methods in this article are defined in the `NSDocument` subclass of a document-based application, but they could as easily belong to another object, such as a text view.) "Table cell creation method" (page 32) performs steps 3–6 for each cell in the table, using fat borders and contrasting colors for illustrative purposes.

Listing 1 Table creation method

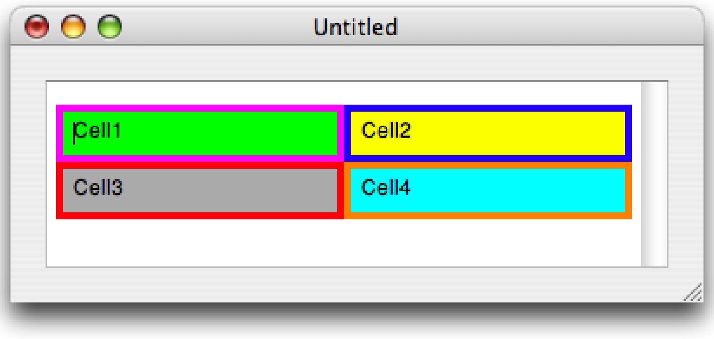
```
- (NSMutableAttributedString *) tableAttributedString
{
    // tableString is an ivar declared in the header file as NSMutableAttributedString
 *tableString;
    tableString = [[NSMutableAttributedString alloc] initWithString:@"\n\n"];
    NSTextTable *table = [[NSTextTable alloc] init];
    [table setNumberOfColumns:2];

    [tableString appendAttributedString:[self
tableCellAttributedStringWithString:@"Cell1\n"
        table:table
        backgroundColor:[NSColor greenColor]
        borderColor:[NSColor magentaColor]
        row:0
        column:0]];

    [tableString appendAttributedString:[self
tableCellAttributedStringWithString:@"Cell2\n"
        table:table
        backgroundColor:[NSColor yellowColor]
        borderColor:[NSColor blueColor]
        row:0
        column:1]];

    [tableString appendAttributedString:[self
tableCellAttributedStringWithString:@"Cell3\n"
        table:table
        backgroundColor:[NSColor lightGrayColor]
        borderColor:[NSColor redColor]
        row:1
        column:0]];

    [tableString appendAttributedString:[self
tableCellAttributedStringWithString:@"Cell4\n"
        table:table
        backgroundColor:[NSColor cyanColor]
        borderColor:[NSColor orangeColor]
        row:1
        column:1]];

    [table release];
    return [tableString autorelease];
}
```

Listing 2 Table cell creation method

```
- (NSMutableAttributedString *) tableCellAttributedStringWithString:(NSString *)string
        table:(NSTextTable *)table
        backgroundColor:(NSColor *)backgroundColor
        borderColor:(NSColor *)borderColor
        row:(int)row
        column:(int)column
{
    NSTextTableBlock *block = [[NSTextTableBlock alloc]
        initWithTable:table
        startingRow:row
        rowSpan:1
        startingColumn:column
        columnSpan:1];
    [block setBackgroundColor:backgroundColor];
    [block setBorderColor:borderColor];
    [block setWidth:4.0 type:NSTextBlockAbsoluteValueType forLayer:NSTextBlockBorder];
    [block setWidth:6.0 type:NSTextBlockAbsoluteValueType forLayer:NSTextBlockPadding];

    NSMutableParagraphStyle *paragraphStyle =
        [[NSParagraphStyle defaultParagraphStyle] mutableCopy];
    [paragraphStyle setTextBlocks:[NSArray arrayWithObjects:block, nil]];
    [block release];

    NSMutableAttributedString *cellString =
        [[NSMutableAttributedString alloc] initWithString:string];
    [cellString addAttribute:NSParagraphStyleAttributeName
        value:paragraphStyle
        range:NSMakeRange(0, [cellString length])];
    [paragraphStyle release];

    return [cellString autorelease];
}
```

The code in "Table creation method" (page 31) and "Table cell creation method" (page 32) produces the table shown in "Table output" (page 33).

Figure 3 Table output



To insert the table in text displayed in a text view, implement an action method such as the one shown in Listing 3 (page 33). This method inserts the table string constructed in "Table creation method" (page 31) in the text view, replacing the current selection (or at the insertion point, if there's no selection), and ensures that the proper notifications and delegate messages are sent.

Listing 3 Table insertion action method

```
- (void) insertMyTable:(id)sender
{
    NSRange charRange = [myTextView rangeForUserTextChange];
    NSTextStorage *myTextStorage = [myTextView textStorage];

    if ([myTextView isEditable] && charRange.location != NSNotFound)
        {
          NSMutableAttributedString *attrStringToInsert = [self tableAttributedString];
            if ([myTextView shouldChangeTextInRange:charRange replacementString:nil])
                {
                    [myTextStorage replaceCharactersInRange:charRange
                        withAttributedString:attrStringToInsert];
                    [myTextView setSelectedRange:NSMakeRange(charRange.location, 0)
                        affinity:NSSelectionAffinityUpstream stillSelecting:NO];
                    [myTextView didChangeText];
                }
        }
}
```

`NSAttributedString` has the following convenience methods you can use to determine the range in the string that is covered by a text block or table:

`rangeOfTextBlock:atIndex:`

`rangeOfTextTable:atIndex:`

If the given location is not in the specified block or table, these methods return a range of `(NSNotFound, 0)`.

The Text Table Model

The Cocoa text table model is derived primarily from the table model defined by HTML and CSS, in which tables are built up from rows of cells. For a description of the CSS table model, refer to the following URL:

http://www.w3.org/TR/CSS21/tables.html

This affinity provides another way to create tables in text. You can define a table in HTML and use that data to initialize an attributed string. The attributes of that string then define the table for rendering by the Cocoa text system. You can use the following `NSAttributedString` initialization methods for this purpose:

`initWithHTML:documentAttributes:`

`initWithHTML:options:documentAttributes:`

`initWithHTML:baseURL:documentAttributes:`

`initWithData:options:documentAttributes:error:`

Controlling Text Block Appearance

The position of a text block is determined by its text container or containing block. In the case of a text table block, which represents a cell in a table, size and position are controlled by the text table and the block's relation to other blocks in the table. When you initialize an `NSTextTableBlock` object, you specify its row and column position as a cell within its table, and you also specify whether it spans multiple rows or columns. The `NSTextTable` initialization method is:

`initWithTable:startingRow:rowSpan:startingColumn:columnSpan:`

"Table cell creation method" (page 32) shows this method in use.

In addition, you can specify the value of a number of dimensions for each block, either as an absolute value or as a percentage of the containing block. These dimensions include the following:

- Width
- Height
- Minimum width
- Minimum height
- Maximum width
- Maximum height
- Padding width for each of the four sides. Padding is space surrounding the content area of the block, extending to the border.
- Border width for each of the four sides. A border is space between the padding and the margin, typically colored to present a visible boundary.
- Margin width for each of the four sides. The margin is space surrounding the border.

The default value for each of these dimensions is `0` , indicating no padding, border, or margin, and the natural width and height. Natural width and height of a single text block extend to the width and height of its containing block (or text container); natural width and height of multiple blocks divide the space of their containing block evenly.

The following methods specify or return values associated with these dimensions:

`setValue:type:forDimension:`

`valueForDimension:`

`valueTypeForDimension:`

`setWidth:type:forLayer:`

`setWidth:type:forLayer:edge:`

`widthForLayer:edge:`

`widthValueTypeForLayer:edge:`

In these methods, the value type refers to absolute or percentage values. The dimension refers to minimum, maximum, and full width and height of the block. The layer refers to padding, border, and margin. These parameters are specified by constants described in `NSTextBlock`.

`NSTextBlock` provides the following methods to specify and return the background and border color of the block:

`backgroundColor`

`setBackgroundColor:`

`borderColorForEdge:`

`setBorderColor:forEdge:`

`setBorderColor:`

By default the color values are `nil` , meaning no color. Note that a border with no color is invisible.

Table Layout Process

During text layout, the typesetter works with `NSTextBlock` to determine the layout rectangles for the text block. If the text block is an instance of `NSTextTableBlock`, it calls its containing `NSTextTable` instance to perform the calculations. The typesetter stores the results of these calculations in its layout manager. There are methods in `NSTextBlock`, `NSTextTable`, and `NSLayoutManager` specific to this layout process that you can use if you need to intervene in the process.

To begin the text block layout process, the typesetter proposes a large rectangle within which the text block should fit. For the outermost block, this is determined by the text container; for inner blocks, it is determined by the containing block. The block object then decides what area within the proposed rectangle it should actually occupy.

The text block actually determines two rectangles: first, the layout rectangle, within which the text in the block is to be laid out; second, the bounds rectangle, which contains additional space for padding, borders, border decoration, and margins. The text block calculates the layout rectangle immediately before the typesetter lays out the first glyph because it is needed for all subsequent layout of text in the block. The layout rectangle is often quite tall because, at this point, the height of the text to be laid out has not yet been determined. The text block calculates the bounds rectangle immediately after the last glyph in the block has been laid out, and it is based on the actual rectangle used for the text within the block. Under some circumstances, the bounds rectangle may be adjusted subsequently as additional blocks in the same table are laid out.

To find the layout and bounds rectangles, the typesetter calls the following `NSTextBlock` methods:

`rectForLayoutAtPoint:inRect:textContainer:characterRange:`

`boundsRectForContentRect:inRect:textContainer:characterRange:`

An `NSTextTableBlock` object, in turn, calls its `NSTextTable` object to perform these calculations, using the following methods:

`rectForBlock:layoutAtPoint:inRect:textContainer:characterRange:`

`boundsRectForBlock:contentRect:inRect:textContainer:characterRange:`

The typesetter stores the results of these methods in the layout manager using the following methods:

`setLayoutRect:forTextBlock:glyphRange:`

`setBoundsRect:forTextBlock:glyphRange:`

The typesetter uses the following `NSLayoutManager` methods when it needs to determine the space used by previously laid text blocks:

`layoutRectForTextBlock:glyphRange:`

`layoutRectForTextBlock:atIndex:effectiveRange:`

`boundsRectForTextBlock:glyphRange:`

`boundsRectForTextBlock:atIndex:effectiveRange:`

The preceding methods cause glyph generation but do not force layout. This avoids an infinite recursion when the methods are called during layout. For the same reason, the following variants of existing `NSLayoutManager` methods have an option to prevent them from causing layout:

`lineFragmentRectForGlyphAtIndex:effectiveRange:withoutAdditionalLayout:`

`lineFragmentUsedRectForGlyphAtIndex:effectiveRange:withoutAdditionalLayout:`

`textContainerForGlyphAtIndex:effectiveRange:withoutAdditionalLayout:`

If no rectangle has been set, the preceding methods return `NSZeroRect`.

At display time the text is drawn as usual, as described in "“The Layout Manager” (page 9)," except that the text block draws background and border decoration while glyph backgrounds are being drawn, using the following method:

```
drawBackgroundWithFrame:inView:characterRange:layoutManager:
```

If the text block is an `NSTextTableBlock` object, it calls its text table for this purpose, using the following `NSTextTable` method:

```
drawBackgroundForBlock:withFrame:inView:characterRange:layoutManager:
```

Document Revision History

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

Date	Notes
2008-10-15	Added information on thread safety to "The Layout Manager" article.
2007-12-11	Corrected indexing error in sample code in "Using Text Tables."
2006-11-07	Fixed minor error in function declaration in "Calculating Text Height" and label in diagram in "Typesetters" article.
2006-10-03	Fixed a broken link, corrected a minor error in sample code, and added information about typesetter compatibility
2005-06-04	Added article "Using Text Tables" describing use of NSTextTable in Mac OS X v10.4. Changed title from "Text Layout."
2004-02-13	Rewrote introduction and added an index.
2003-08-08	Revised "Typesetters" (page 15) to include information about NSATSTypesetter class and new API design.
2003-05-09	First version.

Index