NSTextContainer

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSTextContainer.h

Class Description

An NSTextContainer defines a region where text is laid out. An NSLayoutManager uses NSTextContainers to determine where to break lines, lay out portions of text, and so on. NSTextContainer defines rectangular regions, but you can create subclasses that define regions of other shapes, such as circular regions, regions with holes in them, or regions that flow alongside graphics.

You normally use an NSTextView to display the text laid out within an NSTextContainer. An NSTextView can have only one NSTextContainer; however, since the two are separate objects, you can replace an NSTextView's container to change the layout of the text it displays. You can also display an NSTextContainer's text in any NSView by locking the graphics focus on it and using NSLayoutManager's **drawBackgroundForGlyphRange:atPoint:** and **drawGlyphsForGlyphRange:atPoint:** methods. If you have no need of actually displaying the text—if you're only calculating line breaks or number of lines or pages, for example—you can use an NSTextContainer without an NSTextView.

Region, Bounding Rectangle, and Inset

An NSTextContainer's region is defined within a *bounding rectangle* whose coordinate system starts at (0, 0) in the top left corner. The size of this rectangle is returned by the **containerSize** method and set using **setContainerSize:**. You can define a container's region so that it's always the same shape, such as a circle whose diameter is the narrower of the bounding rectangle's dimensions, or you can define the region relative to the bounding rectangle, such as an oval region that fits inside the bounding rectangle (and that's a circle when the bounding rectangle is square). Regardless of a text container's shape, its NSTextView always clips drawing to its bounding rectangle.

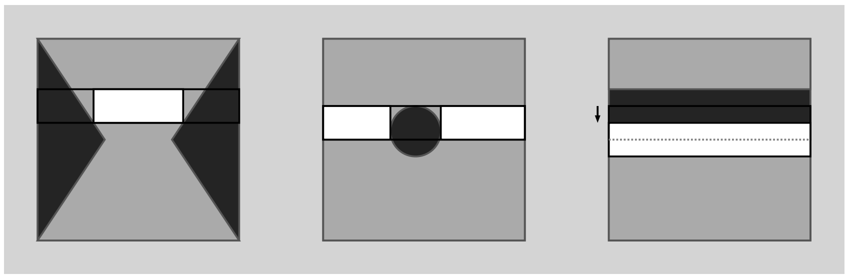
A subclass of NSTextContainer defines its region by overriding three methods. The first, **isSimpleRectangularTextContainer**, indicates whether the region is currently a nonrotated rectangle, thus allowing the NSLayoutManager to optimize layout of text (since custom NSTextContainers typically define more complex regions, your implementation of this method will probably return NO). The second method, **containsPoint:**, is used for testing mouse events and determines whether or not a given point lies in the region. The third method is used for the actual layout of text, defining the region in terms of rectangles available to lay text in; this process is described in "Calculating Text Layout."

An NSTextContainer usually covers its NSTextView exactly, but can be inset within the view frame with NSTextView's **setTextContainerInset:** method. The NSTextContainer's bounding rectangle from the inset position then establishes the limits of the NSTextContainer's region. The inset also helps to determine the size of the bounding rectangle when the NSTextContainer tracks the height or width of its NSTextView, as described in "Tracking the Size of the NSTextView."

Calculating Text Layout

An NSLayoutManager lays text within an NSTextContainer in lines of glyphs, running either horizontally or vertically. The layout of these lines within an NSTextContainer is determined by its shape. For example, if the NSTextContainer 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. This is illustrated in the figure below.

Note: The text system currently supports only horizontal text layout.



The NSLayoutManager proposes a rectangle for a given line, and then asks the NSTextContainer to adjust the rectangle to fit. The proposed rectangle usually spans the NSTextContainer's bounding rectangle, but it can be narrower or wider, and it can also lie partially or completely outside the bounding rectangle. The method that an NSLayoutManager sends the 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 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 NSTextContainer.

Text is laid out 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. The direction in which the lines progress is then called the *line movement direction* and is expressed by the NSLineMovementDirection type. 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.

NSLineSweepDirection values	NSLineMovementDirection values
NSLineSweepLeft	NSLineMovesLeft
NSLineSweepRight	NSLineMovesRight
NSLineSweepDown	NSLineMovesDown
NSLineSweepUp	NSLineMovesUp
	NSLineDoesntMove

For the three examples above, 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 NSLayoutManager will then be this remainder rectangle, and will be returned unchanged by the text container.

In the third example, a gap crosses the entire NSTextContainer. 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 NSTextContainer would have to return NSZeroRect indicating that the line simply doesn't fit. In such a case it's up to the NSLayoutManager 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 NSLayoutManager makes one final adjustment when it actually fits text into the rectangle. This adjustment is a small amount fixed by the NSTextContainer, 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 NSTextContainer's region at the edges and around any holes, and keeps text from abutting any other graphics displayed near the region. You can change the padding from its default value with the **setLineFragmentPadding:** method, or override the default in your subclass. Note that line fragment padding isn't a suitable means for expressing margins; you should set the NSTextView's position and size for document margins or the paragraph margin attributes for text margins.

Tracking the Size of the NSTextView

Normally, if you resize an NSTextView its NSTextContainer doesn't change in size. You can, however, set an NSTextContainer to track the size of its NSTextView and adjust its own size to match whenever the NSTextView's size changes. The **setHeightTracksTextView:** and **setWidthTracksTextView:** methods allow you to control this tracking for either dimension.

When an NSTextContainer adjusts its size to match that of its NSTextView, it takes into account the inset specified by the NSTextView so that the bounding rectangle is inset from every edge possible. In other words, an NSTextContainer that tracks the size of its NSTextView is always smaller than the NSTextView (in the appropriate dimension) by twice the inset. Suppose an NSTextContainer is set to track width and its NSTextView gives it an inset of (10, 10). Now, if the NSTextView's width is changed to 138, the

NSTextContainer’s top left corner is set to lie at (10, 10) and its width is set to 118, so that its right edge is 10 points from the NSTextView’s right edge. Its height remains the same.

Whether it tracks the size of its NSTextView or not, an NSTextContainer doesn’t grow or shrink as text is added or deleted; instead, the NSLayoutManager resizes the NSTextView based on the portion of the NSTextContainer actually filled with text. To allow an NSTextView to be resized in this manner, use NSTextView’s **setVerticallyResizable:** or **setHorizontallyResizable:** methods as needed, set the text container not to track the size of its text view, and set the text container’s size in the appropriate dimension large enough to accommodate a great amount of text—about 1e7 (this incurs no cost whatever in processing or storage). For more information on automatic size adjustment, see “The OPENSTEP Text Handling System.”

Note that an NSTextView can be resized based on its NSTextContainer, and an NSTextContainer can resize itself based on its NSTextView. If you set both objects up to resize automatically in the same dimension, your application can get trapped in an infinite loop. When text is added to the NSTextContainer, the NSTextView is resized to fit the area actually used for text; this causes the NSTextContainer to resize itself and relay its text, which causes the NSTextView to resize itself again, and so on ad infinitum. Each type of size tracking has its proper uses; be sure to use only one for either dimension.

Method Types

Creating an instance	– initWithContainerSize:
Managing text components	– setLayoutManager: – layoutManager – replaceLayoutManager: – setTextView: – textView
Controlling size	– setContainerSize: – containerSize – setWidthTracksTextView: – widthTracksTextView – setHeightTracksTextView: – heightTracksTextView
Setting line fragment padding	– setLineFragmentPadding: – lineFragmentPadding
Calculating text layout	– lineFragmentRectForProposedRect:sweepDirection: movementDirection:remainingRect: – isSimpleRectangularTextContainer
Mouse hit testing	– containsPoint:

Instance Methods

containerSize

– (NSSize)**containerSize**

Returns the size of the receiver’s bounding rectangle, regardless of the size of its region.

See also: – **textContainerInset** (NSTextView), – **setContainerSize:**

containsPoint:

– (BOOL)**containsPoint:**(NSPoint)*aPoint*

Overridden by subclasses to returns YES if *aPoint* lies within the receiver’s region or on the region’s edge—not simply within its bounding rectangle—NO otherwise. For example, if the receiver defines a donut shape and *aPoint* lies in the hole, this method returns NO. This method can be used for hit testing of mouse events.

NSTextContainer’s implementation merely checks that *aPoint* lies within its bounding rectangle.

heightTracksTextView

– (BOOL)**heightTracksTextView**

Returns YES if the receiver adjusts the height of its bounding rectangle when its NSTextView is resized, NO otherwise. The height is adjusted to the height of the NSTextView minus twice the inset height (as given by NSTextView’s **textContainerInset** method).

See the class description for more information on size tracking.

See also: – **widthTracksTextView**, – **setHeightTracksTextView:**

initWithContainerSize:

– (id)**initWithContainerSize:**(NSSize)*aSize*

Initializes the receiver, a newly allocated NSTextContainer, with *aSize* as the size of its bounding rectangle. The new NSTextContainer must be added to an NSLayoutManager before it can be used; it must also have an NSTextView set for text to be displayed. This method is the designated initializer for the NSTextContainer class. Returns **self**.

See also: – **addTextContainer:** (NSLayoutManager), – **setTextView:**

isSimpleRectangularTextContainer

– (BOOL)**isSimpleRectangularTextContainer**

Overridden by subclasses to return YES if the receiver's region is a rectangle with no holes or gaps and whose edges are parallel to the NSTextView's coordinate system axes; returns NO otherwise. An NSTextContainer whose shape changes can return YES if its region is currently a simple rectangle, but when its shape does change it must send **textContainerChangedGeometry:** to its NSLayoutManager so the layout can be recalculated.

NSTextContainer's implementation of this method returns YES.

layoutManager

– (NSLayoutManager *)**layoutManager**

Returns the receiver's NSLayoutManager.

See also: – **setLayoutManager:**, – **replaceLayoutManager:**

lineFragmentPadding

– (float)**lineFragmentPadding**

Returns the amount (in points) by which text is inset within line fragment rectangles.

See also: – **lineFragmentRectForProposedRect:sweepDirection:movementDirection: remainingRect:**, – **setLineFragmentPadding:**

lineFragmentRectForProposedRect:sweepDirection:movementDirection: remainingRect:

– (NSRect)**lineFragmentRectForProposedRect:**(NSRect)*proposedRect*
sweepDirection:(NSLineSweepDirection)*sweepDirection*
movementDirection:(NSLineMovementDirection)*movementDirection*
remainingRect:(NSRect *)*remainingRect*

Overridden by subclasses to calculate and return the longest rectangle available for *proposedRect* for displaying text, or NSZeroRect if there is none according to the receiver's region definition.The receiver should examine *proposedRect* to see that it intersects its bounding rectangle, and should return a modified rectangle based on *sweepDirection* and *movementDirection*, whose possible values are listed in the class description. If *sweepDirection* is NSLineSweepRight, for example, the receiver uses this information to trim the right end of *proposedRect* as needed rather than the left end.

If *proposedRect* doesn't completely overlap the region along the axis of *movementDirection* and *movementDirection* isn't NSLineDoesntMove, this method can either shift the rectangle in that direction as much as needed so that it does completely overlap, or return NSZeroRect to indicate that the proposed rectangle simply doesn't fit.

Upon returning, *remainingRect* contains the unused, possibly shifted, portion of *proposedRect* that's available for further text, or NSZeroRect if there is no remainder.

See the class description for more information on overriding this method.

replaceLayoutManager:

– (void)**replaceLayoutManager:**(NSLayoutManager *)*aLayoutManager*

Replaces the NSLayoutManager for the group of text-system objects containing the receiver with *aLayoutManager*. All NSTextContainers and NSTextViews sharing the original NSLayoutManager then share the new one. This method makes all the adjustments necessary to keep these relationships intact, unlike **setLayoutManager:**.

See also: – **layoutManager**

setContainerSize:

– (void)**setContainerSize:**(NSSize)*aSize*

Sets the size of the receiver's bounding rectangle to *aSize* and sends **textContainerChangedGeometry:** to the NSLayoutManager.

See also: – **setTextContainerInset:** (NSTextView), – **containerSize**

setHeightTracksTextView:

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

Controls whether the receiver adjusts the height of its bounding rectangle when its NSTextView is resized. If *flag* is YES, the receiver follows changes to the height of its text view; if *flag* is NO, it doesn't.

See the class description for more information on size tracking.

See also: – **setContainerSize:**, – **setWidthTracksTextView:**, – **heightTracksTextView**

setLayoutManager:

– (void)**setLayoutManager:**(NSLayoutManager *)*aLayoutManager*

Sets the receiver's NSLayoutManager to *aLayoutManager*. This method is invoked automatically when you add an NSTextContainer to an NSLayoutManager; you should never need to invoke it directly, but might want to override it. If you want to replace the NSLayoutManager for an established group of text-system objects, use **replaceLayoutManager:**.

See also: – **addTextContainer:** (NSLayoutManager), – **layoutManager**

setLineFragmentPadding:

– (void)**setLineFragmentPadding:**(float)*aFloat*

Sets the amount (in points) by which text is inset within line fragment rectangles to *aFloat*. Also sends **textContainerChangedGeometry:** to the receiver's NSLayoutManager to inform it of the change.

See also: – **lineFragmentRectForProposedRect:sweepDirection:movementDirection: remainingRect:**, – **lineFragmentPadding**

setTextView:

– (void)**setTextView:**(NSTextView *)*aTextView*

Sets the receiver's NSTextView to *aTextView* and sends **setTextContainer:** to *aTextView* to complete the association of the text container and text view. Since you usually specify an NSTextContainer when you create an NSTextView, you should rarely need to invoke this method. An NSTextContainer doesn't need an NSTextView to calculate line fragment rectangles, but must have one to display text.

You can use this method to disconnect an NSTextView from a group of text-system objects by sending this message to its text container and passing **nil** as *aTextView*.

See also: – **initFrame:textContainer:** (NSTextView), – **replaceTextContainer:** (NSTextView)

setWidthTracksTextView:

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

Controls whether the receiver adjusts the width of its bounding rectangle when its NSTextView is resized. If *flag* is YES, the receiver follows changes to the width of its text view; if *flag* is NO, it doesn't.

See the class description for more information on size tracking.

See also: – **setContainerSize:**, – **setHeightTracksTextView:**, – **widthTracksTextView**

textView

– (NSTextView *)**textView**

Returns the receiver's NSTextView, or **nil** if it has none.

See also: – **setTextView:**

widthTracksTextView

– (BOOL)**widthTracksTextView**

Returns YES if the receiver adjusts the width of its bounding rectangle when its NSTextView is resized, NO otherwise. The width is adjusted to the width of the NSTextView minus twice the inset width (as given by NSTextView's **textContainerInset** method).

See the class description for more information on size tracking.

See also: – **heightTracksTextView**, – **setWidthTracksTextView:**