NSLayoutManager

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSLayoutManager.h

Class Description

An NSLayoutManager coordinates the layout and display of characters held in an NSTextStorage object. It maps Unicode character codes to glyphs, sets the glyphs in a series of NSTextContainers, and displays them in a series of NSTextViews. In addition to its core function of laying out text, an NSLayoutManager coordinates its NSTextViews, provides services to those NSTextViews to support NSRulerViews for editing paragraph styles, and handles the layout and display of text attributes not inherent in glyphs (such as underline or strikethrough). You can create a subclass of NSLayoutManager to handle additional text attributes, whether inherent or not.

Method Types

Creating an instance	– init
Setting the text storage	– setTextStorage: – textStorage – replaceTextStorage:
Setting text containers	– textContainers – addTextContainer: – insertTextContainer:atIndex: – removeTextContainerAtIndex:
Invalidating glyphs and layout	– invalidateGlyphsForCharacterRange:changeInLength: actualCharacterRange: – invalidateLayoutForCharacterRange:isSoft: actualCharacterRange: – invalidateDisplayForGlyphRange: – textContainerChangedGeometry: – textStorage:edited:range:changeInLength: invalidatedRange:
Turning background layout on/off	– setBackgroundLayoutEnabled: – backgroundLayoutEnabled

Accessing glyphs	– insertGlyph:atGlyphIndex:characterIndex: – glyphAtIndex: – glyphAtIndex:isValidIndex: – replaceGlyphAtIndex:withGlyph: – getGlyphs:range: – deleteGlyphsInRange: – numberOfGlyphs
Mapping characters to glyphs	– setCharacterIndex:forGlyphAtIndex: – characterIndexForGlyphAtIndex: – characterRangeForGlyphRange:actualGlyphRange: – glyphRangeForCharacterRange:actualCharacterRange:
Setting glyph attributes	– setIntAttribute:value:forGlyphAtIndex: – intAttribute:forGlyphAtIndex:
Handling layout for text containers	– setTextContainer:forGlyphRange: – glyphRangeForTextContainer: – textContainerForGlyphAtIndex:effectiveRange: – usedRectForTextContainer:
Handling line fragment rectangles	– setLineFragmentRect:forGlyphRange:usedRect: – lineFragmentRectForGlyphAtIndex:effectiveRange: – lineFragmentUsedRectForGlyphAtIndex:effectiveRange: – setExtraLineFragmentRect:usedRect:textContainer: – extraLineFragmentRect – extraLineFragmentUsedRect – extraLineFragmentTextContainer – setDrawsOutsideLineFragment:forGlyphAtIndex: – drawsOutsideLineFragmentForGlyphAtIndex:
Layout of glyphs	– setLocation:forStartOfGlyphRange: – locationForGlyphAtIndex: – rangeOfNominallySpacedGlyphsContainingIndex: – rectArrayForCharacterRange: withinSelectedCharacterRange: inTextContainer:rectCount: – rectArrayForGlyphRange:withinSelectedGlyphRange: inTextContainer:rectCount: – boundingRectForGlyphRange:inTextContainer: – glyphRangeForBoundingRect:inTextContainer: – glyphRangeForBoundingRectWithoutAdditionalLayout: inTextContainer: – glyphIndexForPoint:inTextContainer: fractionOfDistanceThroughGlyph:

Display of special glyphs	– setNotShownAttribute:forGlyphAtIndex: – notShownAttributeForGlyphAtIndex: – setShowsInvisibleCharacters: – showsInvisibleCharacters – setShowsControlCharacters: – showsControlCharacters
Finding unlaid characters/glyphs	– getFirstUnlaidCharacterIndex:glyphIndex:
Using screen fonts	– setUsesScreenFonts: – usesScreenFonts – substituteFontForFont:
Handling rulers	– rulerAccessoryViewForTextView:paragraphStyle:ruler:enabled: – rulerMarkersForTextView:paragraphStyle:ruler:
Managing the responder chain	– layoutManagerOwnsFirstResponderInWindow: – firstTextView – textViewForBeginningOfSelection
Drawing	– drawBackgroundForGlyphRange:atPoint: – drawGlyphsForGlyphRange:atPoint: – drawUnderlineForGlyphRange:underlineType:baselineOffset: lineFragmentRect:lineFragmentGlyphRange:containerOrigin: – underlineGlyphRange:underlineType:lineFragmentRect: lineFragmentGlyphRange:containerOrigin:
Setting the delegate	– setDelegate: – delegate

Instance Methods

addTextContainer:

– (void)**addTextContainer:**(NSTextContainer *)*aTextContainer*

Appends *aTextContainer* to the series of NSTextContainers where the receiver arranges text. Invalidates glyphs and layout as needed, but doesn't perform glyph generation or layout.

See also: – **insertTextContainer:atIndex:**, – **removeTextContainerAtIndex:**, – **textContainers**, – **invalidateGlyphsForCharacterRange:changeInLength:actualCharacterRange:**, – **invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:**

backgroundLayoutEnabled

– (BOOL)**backgroundLayoutEnabled**

Returns YES if the receiver generates glyphs and lays out text when the application's run loop is idle, NO if it only performs glyph generation and layout when necessary.

See also: – **setBackgroundLayoutEnabled:**

boundingRectForGlyphRange:inTextContainer:

– (NSRect)**boundingRectForGlyphRange:**(NSRange)*glyphRange*
inTextContainer:(NSTextContainer *)*aTextContainer*

Returns a single bounding rectangle enclosing all glyphs and other marks drawn in *aTextContainer* for *glyphRange*, including glyphs that draw outside their line fragment rectangles and text attributes such as underlining. This method is useful for determining the area that needs to be redrawn when a range of glyphs changes.

Performs glyph generation and layout if needed.

See also: – **glyphRangeForTextContainer:**, – **drawsOutsideLineFragmentForGlyphAtIndex:**

characterIndexForGlyphAtIndex:

– (unsigned int)**characterIndexForGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns the index in the NSTextStorage for the first character mapped to the glyph at *glyphIndex* within the receiver. In many cases it's better to use the range-mapping methods, **characterRangeForGlyphRange:actualGlyphRange:** and **glyphRangeForCharacterRange:actualCharacterRange:**, which provide more comprehensive information.

Performs glyph generation if needed.

characterRangeForGlyphRange:actualGlyphRange:

– (NSRange)**characterRangeForGlyphRange:**(NSRange)*glyphRange*
actualGlyphRange:(NSRange *)*actualGlyphRange*

Returns the range for the characters in the receiver's text store that are mapped to the glyphs in *glyphRange*. If *actualGlyphRange* is non-NULL, expands the requested range as needed so that it identifies all glyphs mapped to those characters and returns the new range by reference in *actualGlyphRange*.

Suppose the text store begins with the character “Ö” and the glyph cache contains “O” and “¨”. If you get the character range for the glyph range {0, 1} or {1, 1}, *actualGlyphRange* is returned as {0, 2}, indicating that both of the glyphs are mapped to the character “Ö”.

Performs glyph generation if needed.

See also: – **characterIndexForGlyphAtIndex:**,
– **glyphRangeForCharacterRange:actualCharacterRange:**

delegate

– (id)**delegate**

Returns the receiver’s delegate.

See also: – **setDelegate:**

deleteGlyphsInRange:

– (void)**deleteGlyphsInRange:**(NSRange)*glyphRange*

Deletes the glyphs in *glyphRange*.

This method is for use by the glyph generation mechanism, and doesn’t perform any invalidation or generation of the glyphs or layout. You should never directly invoke this method.

See also: – **insertGlyph:atGlyphIndex:characterIndex:**

drawBackgroundForGlyphRange:atPoint:

– (void)**drawBackgroundForGlyphRange:**(NSRange)*glyphRange*
atPoint:(NSPoint)*containerOrigin*

Draws background marks for *glyphRange*, which must lie completely within a single NSTextContainer. *containerOrigin* indicates the position of the NSTextContainer in the coordinate system of the NSView being drawn. This method must be invoked with the graphics focus locked on that NSView.

Background marks are such things as selection highlighting, text background color, and any background for marked text.

Performs glyph generation and layout if needed.

See also: – **drawGlyphsForGlyphRange:atPoint:**, – **glyphRangeForTextContainer:**,
– **textContainerOrigin** (NSTextView)

drawGlyphsForGlyphRange:atPoint:

– (void)**drawGlyphsForGlyphRange:**(NSRange)*glyphRange* **atPoint:**(NSPoint)*containerOrigin*

Draws the glyphs in *glyphRange*, which must lie completely within a single NSTextContainer. *containerOrigin* indicates the position of the NSTextContainer in the coordinate system of the NSView being drawn. This method must be invoked with the graphics focus locked on that NSView.

Performs glyph generation and layout if needed.

See also: – **drawBackgroundForGlyphRange:atPoint:**, – **glyphRangeForTextContainer:**, – **textContainerOrigin** (NSTextView)

drawsOutsideLineFragmentForGlyphAtIndex:

– (BOOL)**drawsOutsideLineFragmentForGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns YES if the glyph at *glyphIndex* exceeds the bounds of the line fragment where it's laid out, NO otherwise. This can happen when text is set at a fixed line height. For example, if the user specifies a fixed line height of 12 points and sets the font size to 24 points, the glyphs will exceed their layout rectangles.

Glyphs that draw outside their line fragment rectangles aren't considered when calculating enclosing rectangles with the **rectArrayForCharacterRange:withinSelectedCharacterRange:inTextContainer:rectCount:** and **rectArrayForGlyphRange:withinSelectedGlyphRange:inTextContainer:rectCount:** methods. They are, however, considered by **boundingRectForGlyphRange:inTextContainer:**

Performs glyph generation and layout if needed.

drawUnderlineForGlyphRange:underlineType:baselineOffset:lineFragmentRect:lineFragmentGlyphRange:containerOrigin:

– (void)**drawUnderlineForGlyphRange:**(NSRange)*glyphRange*
underlineType:(int)*underlineType*
baselineOffset:(float)*baselineOffset*
lineFragmentRect:(NSRect)*lineRect*
lineFragmentGlyphRange:(NSRange)*lineGlyphRange*
containerOrigin:(NSPoint)*containerOrigin*

Draws underlining for the glyphs in *glyphRange*, which must belong to a single line fragment rectangle (as returned by **lineFragmentRectForGlyphAtIndex:effectiveRange:**). *underlineType* indicates the style of underlining to draw; NSLayoutManager accepts only NSSingleUnderlineStyle, but subclasses can define their own underline styles. *baselineOffset* indicates how far below the text baseline the underline should be drawn; it's usually a positive value. *lineRect* is the line fragment rectangle containing the glyphs to draw underlining for, and *lineGlyphRange* is the range of all glyphs within that line fragment rectangle. *containerOrigin* is the origin of the line fragment rectangle's NSTextContainer in its NSTextView.

This method is invoked automatically by **underlineGlyphRange:...**; you should rarely need to invoke it directly.

See also: – **textContainerForGlyphAtIndex:effectiveRange:**, – **textContainerOrigin** (NSTextView)

extraLineFragmentRect

– (NSRect)**extraLineFragmentRect**

Returns the rectangle defining the extra line fragment for the insertion point at the end of a text (either in an empty text or after a final paragraph separator). The rectangle is defined in the coordinate system of its NSTextContainer. Returns NSZeroRect if there is no such rectangle.

See also: – **extraLineFragmentUsedRect**, – **extraLineFragmentTextContainer**, – **setExtraLineFragmentRect:usedRect:textContainer:**

extraLineFragmentTextContainer

– (NSTextContainer *)**extraLineFragmentTextContainer**

Returns the NSTextContainer that contains the extra line fragment rectangle, or **nil** if there is no extra line fragment rectangle. This rectangle is used to display the insertion point for the insertion point at the end of a text (either in an empty text or after a final paragraph separator).

See also: – **extraLineFragmentRect**, – **extraLineFragmentUsedRect**, – **setExtraLineFragmentRect:usedRect:textContainer:**

extraLineFragmentUsedRect

– (NSRect)**extraLineFragmentUsedRect**

Returns the rectangle enclosing the insertion point drawn in the extra line fragment rectangle. The rectangle is defined in the coordinate system of its NSTextContainer. Returns NSZeroRect if there is no extra line fragment rectangle.

The extra line fragment used rectangle is twice as wide (or tall) as the NSTextContainer's line fragment padding, with the insertion point itself in the middle.

See also: – **extraLineFragmentRect**, – **extraLineFragmentTextConainer**, – **setExtraLineFragmentRect:usedRect:textContainer:**

firstTextView

– (NSTextView *)**firstTextView**

Returns the first NSTextView in the receiver's series of text views. This is the object of various NSText and NSTextView notifications posted.

getFirstUnlaidCharacterIndex:glyphIndex:

– (void)**getFirstUnlaidCharacterIndex:**(unsigned int *)*charIndex*
glyphIndex:(unsigned int *)*glyphIndex*

Returns by reference in *charIndex* and *glyphIndex* the indexes for the first character and glyph that have invalid layout information. Either parameter may be NULL, in which case the receiver simply ignores it.

getGlyphs:range:

– (unsigned int)**getGlyphs:**(NSGlyph *)*glyphArray* **range:**(NSRange)*glyphRange*

Fills *glyphArray* with displayable glyphs from *glyphRange* and returns the actual number of glyphs filled (which may be smaller than *glyphRange*'s length if some glyphs aren't drawn—for example, tab and newline characters). Raises an NSRangeException if the range specified exceeds the bounds of the actual glyph range for the receiver.

Performs glyph generation if needed.

See also: – **glyphAtIndex:**, – **glyphAtIndex:isValidIndex:**, – **notShownAttributeForGlyphAtIndex:**

glyphAtIndex:

– (NSGlyph)**glyphAtIndex:**(unsigned int)*glyphIndex*

Returns the glyph at *glyphIndex*. Raises an NSRangeException if *glyphIndex* is out of bounds.

Performs glyph generation if needed. To avoid an exception with **glyphAtIndex:** you must first check the glyph index against the number of glyphs, which requires generating *all* glyphs. Another method, **glyphAtIndex:isValidIndex:**, generates glyphs only up to the one requested, so using it can be more efficient.

See also: – **getGlyphs:range:**

glyphAtIndex:isValidIndex:

– (NSGlyph)**glyphAtIndex:**(unsigned int)*glyphIndex* **isValidIndex:**(BOOL *)*flag*

If *glyphIndex* is valid, returns the glyph at *glyphIndex* and sets *flag* to YES. Otherwise sets *flag* to NO (in which case the return value is meaningless).

Performs glyph generation if needed.

See also: – **getGlyphs:range:**, – **glyphAtIndex:**

glyphIndexForPoint:inTextContainer:fractionOfDistanceThroughGlyph:

– (unsigned int)**glyphIndexForPoint:**(NSPoint)*aPoint*
inTextContainer:(NSTextContainer *)*aTextContainer*
fractionOfDistanceThroughGlyph:(float *)*partialFraction*

Returns the index for the glyph nearest *aPoint* within *aTextContainer*. *aPoint* is expressed in *aTextContainer*'s coordinate system. If *partialFraction* is non-NULL the ratio of the distance into the glyph relative to the next glyph (in the appropriate sweep direction) is returned by reference in *partialFraction*.

Note: NSLayoutManager currently supports only left-to-right sweep.

For purposes such as dragging out a selection or placing the insertion point, a partial percentage less than or equal to 0.5 indicates that *aPoint* should be considered as falling before the glyph index returned; a partial percentage greater than 0.5 indicates that it should be considered as falling after the glyph index returned. If the nearest glyph doesn't lie under *aPoint* at all (for example, if *aPoint* is beyond the beginning or end of a line) this ratio will be 0 or 1.

Suppose the glyph stream contains the glyphs "A" and "b", with the width of "A" being 13 points. If the user clicks at a location 8 points into "A", *partialFraction* is 8 ÷ 13, or 0.615. In this case, the point given should be considered as falling between "A" and "b" for purposes such as dragging out a selection or placing the insertion point.

Performs glyph generation and layout if needed.

glyphRangeForBoundingRect:inTextContainer:

– (NSRange)**glyphRangeForBoundingRect:**(NSRect)*aRect*
inTextContainer:(NSTextContainer *)*aTextContainer*

Returns the smallest contiguous range for glyphs that are laid out wholly or partially within *aRect* in *aTextContainer*. The range returned can include glyphs that don't fall inside or intersect *aRect*, though the first and last glyphs in the range always do. This method is used to determine which glyphs need to be displayed within a given rectangle.

Performs glyph generation and layout if needed.

See also: – **glyphRangeForBoundingRectWithoutAdditionalLayout:inTextContainer:**

glyphRangeForBoundingRectWithoutAdditionalLayout:inTextContainer:

– (NSRange)**glyphRangeForBoundingRectWithoutAdditionalLayout:**(NSRect)*bounds*
inTextContainer:(NSTextContainer *)*container*

Returns the smallest contiguous range for glyphs that are laid out wholly or partially within *aRect* in *aTextContainer*. The range returned can include glyphs which don't fall inside or intersect *aRect*, though the first and last glyphs in the range always do.

Unlike **glyphRangeForBoundingRect:inTextContainer:**, this method doesn't perform glyph generation or layout. Its results, though faster, can be incorrect. This method is primarily for use by NSTextView; you should rarely need to use it yourself.

See also: – **glyphRangeForBoundingRect:inTextContainer:**

glyphRangeForCharacterRange:actualCharacterRange:

– (NSRange)**glyphRangeForCharacterRange:**(NSRange)*charRange*
actualCharacterRange:(NSRange *)*actualCharRange*

Returns the range for the glyphs mapped to the characters of the text store in *charRange*. If *actualCharRange* is non-NULL, expands the requested range as needed so that it identifies all characters mapped to those glyphs and returns the new range by reference in *actualCharRange*.

Suppose the text store contains the characters "n˜" and the glyph cache contains "ñ". If you get the glyph range for the character range {0, 1} or {1, 1}, *actualCharRange* is returned as {0, 2}, indicating both of the characters mapped to the glyph "ñ".

Performs glyph generation if needed.

See also: – **characterIndexForGlyphAtIndex:**,
– **glyphRangeForCharacterRange:actualCharacterRange**

glyphRangeForTextContainer:

– (NSRange)**glyphRangeForTextContainer:**(NSTextContainer *)*aTextContainer*

Returns the range for glyphs laid out within *aTextContainer*.

Performs glyph generation and layout if needed.

init

– (id)**init**

Initializes the receiver, a newly created NSLayoutManager object. This is the designated initializer for the NSLayoutManager class. Returns **self**.

See also: – **addLayoutManager:** (NSTextStorage), – **addTextContainer:**

insertGlyph:atGlyphIndex:characterIndex:

– (void)**insertGlyph:**(NSGlyph)*aGlyph*
atGlyphIndex:(unsigned int)*glyphIndex*
characterIndex:(unsigned int)*charIndex*

Inserts *aGlyph* into the glyph cache at *glyphIndex* and maps it to the character at *charIndex*. If the glyph is mapped to several characters, *charIndex* should indicate the first character that it's mapped to.

This method is for use by the glyph generation mechanism, and doesn't perform any invalidation or generation of the glyphs or layout. You should never directly invoke this method.

See also: – **deleteGlyphsInRange:**, – **replaceGlyphAtIndex:withGlyph:**

insertTextContainer:atIndex:

– (void)**insertTextContainer:**(NSTextContainer *)*aTextContainer* **atIndex:**(unsigned int)*index*

Inserts *aTextContainer* into the series of text containers at *index*, and invalidates layout for all subsequent NSTextContainer's. Also invalidates glyph information as needed.

See also: – **addTextContainer:**, – **removeTextContainerAtIndex:**, – **textContainers**

intAttribute:forGlyphAtIndex:

– (int)**intAttribute:**(int)*attributeTag* **forGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns the value of the attribute identified by *attributeTag* for the glyph at *glyphIndex*.

Subclasses that define their own custom attributes must override this method to access their own storage for the attribute values. Non-negative tags are reserved by NeXT; you can define your own attributes with negative tags and set values using **setIntAttribute:value:forGlyphAtIndex:**.

invalidateDisplayForGlyphRange:

– (void)**invalidateDisplayForGlyphRange:**(NSRange)*glyphRange*

Marks the glyphs in *glyphRange* as needing display, as well as the appropriate regions of the NSTextViews that display those glyphs (using NSView's **setNeedsDisplayInRect:**). You should rarely need to invoke this method.

invalidateGlyphsForCharacterRange:changeInLength:actualCharacterRange:

– (void)**invalidateGlyphsForCharacterRange:**(NSRange)*charRange*
changeInLength:(int)*lengthChange*
actualCharacterRange:(NSRange *)*actualCharRange*

Invalidates the cached glyphs for the characters in *charRange* and adjusts the remaining glyph-to-character mapping according to *lengthChange*, which indicates the number of characters added to or removed from the text store. If non-NULL, *actualCharRange* is set to the range of characters mapped to the glyphs just invalidated. This can be larger than the range of characters given due to the effect of context on glyphs and layout.

You should rarely need to invoke this method. It only invalidates glyph information, and performs no glyph generation or layout. Because invalidating glyphs also invalidates layout, after invoking this method you should also invoke **invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:**, passing *charRange* as the first argument and NO as the flag to the **isSoft:** keyword.

invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:

– (void)**invalidateLayoutForCharacterRange:**(NSRange)*charRange*
isSoft:(BOOL)*flag*
actualCharacterRange:(NSRange *)*actualCharRange*

Invalidates the layout information for the glyphs mapped to the characters in *charRange*. If *flag* is YES, attempts to save some layout information to avoid recalculation; if flag is NO, saves no layout information. You should typically pass NO for *flag*. If non-NULL, *actualCharRange* is set to the range of characters mapped to the glyphs whose layout information has been invalidated. This can be larger than the range of characters given due to the effect of context on glyphs and layout.

This method only invalidates information; it performs no glyph generation or layout. You should rarely need to invoke this method.

See also: – **invalidateGlyphsForCharacterRange:changeInLength:actualCharacterRange:**

layoutManagerOwnsFirstResponderInWindow:

– (BOOL)**layoutManagerOwnsFirstResponderInWindow:**(NSWindow *)*aWindow*

Returns YES if the first responder in *aWindow* is an NSTextView associated with the receiver, NO otherwise.

lineFragmentRectForGlyphAtIndex:effectiveRange:

– (NSRect)**lineFragmentRectForGlyphAtIndex:**(unsigned int)*glyphIndex*
effectiveRange:(NSRange *)*lineFragmentRange*

Returns the line fragment rectangle containing the glyph at *glyphIndex*. The rectangle is defined in the coordinate system of its NSTextContainer. If non-NULL, *lineFragmentRange* is set to contain the range for all glyphs in that line fragment.

Performs glyph generation and layout if needed.

See also: – **lineFragmentUsedRectForGlyphAtIndex:effectiveRange:**,
– **setLineFragmentRect:forGlyphRange:usedRect:**

lineFragmentUsedRectForGlyphAtIndex:effectiveRange:

– (NSRect)**lineFragmentUsedRectForGlyphAtIndex:**(unsigned int)*glyphIndex*
effectiveRange:(NSRange *)*lineFragmentRange*

Returns the portion of the line fragment rectangle containing *glyphAtIndex* that actually contains glyphs (such as for a partial or wrapped line), plus the line fragment padding defined by the NSTextContainer where the glyphs reside. This rectangle is defined in the coordinate system of its NSTextContainer, and is based on line calculation only—that is, it isn't a bounding box for the glyphs in the line fragment.

If non-NULL, *lineFragmentRange* is set to contain the range for all glyphs in the line fragment.

Performs glyph generation and layout if needed.

See also: – **lineFragmentRectForGlyphAtIndex:effectiveRange:**,
– **setLineFragmentRect:forGlyphRange:usedRect:**

locationForGlyphAtIndex:

– (NSPoint)**locationForGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns the location, in terms of its line fragment rectangle, for the glyph at *glyphIndex*. The line fragment rectangle in turn is defined in the coordinate system of the text container where it resides.

Performs glyph generation and layout if needed.

See also: – **lineFragmentRectForGlyphAtIndex:effectiveRange:**,
– **lineFragmentUsedRectForGlyphAtIndex:effectiveRange:**

notShownAttributeForGlyphAtIndex:

– (BOOL)**notShownAttributeForGlyphAtIndex:**(unsigned int)*glyphIndex*

Returns YES if the glyph at *glyphIndex* isn't shown (in the sense of the PostScript **show** operator), NO if it is. For example, a tab, newline, or attachment glyph doesn't get shown; it just affects the layout of following glyphs or locates the attachment graphic. Space characters, however, typically are shown as glyphs with a displacement, though they leave no visible marks. Raises an NSRangeException if *glyphIndex* is out of bounds.

Performs glyph generation and layout if needed.

See also: – **setNotShownAttribute:forGlyphAtIndex:**

numberOfGlyphs

– (unsigned int)**numberOfGlyphs**

Returns the number of glyphs in the receiver, performing glyph generation if needed to determine this number.

rangeOfNominallySpacedGlyphsContainingIndex:

– (NSRange)**rangeOfNominallySpacedGlyphsContainingIndex:**(unsigned int)*glyphIndex*

Returns the range for the glyphs around *glyphIndex* that can be displayed with a single PostScript **show** operation; in other words, glyphs with no pairwise kerning or other adjustments to spacing.

Performs glyph generation and layout if needed.

rectArrayForCharacterRange:withinSelectedCharacterRange:inTextContainer: rectCount:

– (NSRect *)**rectArrayForCharacterRange:**(NSRange)*charRange*
withinSelectedCharacterRange:(NSRange)*selCharRange*
inTextContainer:(NSTextContainer *)*aTextContainer*
rectCount:(unsigned int *)*rectCount*

Returns a C array of rectangles for the glyphs in *aTextContainer* that correspond to *charRange*, and by reference in *rectCount* the number of such rectangles. These rectangles can be used to draw the background or highlight for the given range of characters. *selCharRange* indicates selected characters, which can affect the size of the rectangles; it must be equal to or contain *charRange*. To calculate the rectangles for drawing the background, use a selected character range whose location is NSNotFound. To calculate the rectangles for drawing highlighting for *charRange*, use a selected character range that contains *charRange*.

The number of rectangles returned isn't necessarily the number of lines enclosing the specified range. Contiguous lines can share an enclosing rectangle, and lines broken into several fragments have a separate enclosing rectangle for each fragment.

The array of rectangles returned is owned by the receiver, and is overwritten by various NSLayoutManager methods. You should never free it, and should copy it if you need to keep the values or use them after sending other messages to the layout manager.

The purpose of this method is to calculate line rectangles for drawing the text background and highlighting. These rectangles don't necessarily enclose glyphs that draw outside their line fragment rectangles; use **boundingRectForGlyphRange:inTextContainer:** to determine the area that contains all drawing performed for a range of glyphs.

Performs glyph generation and layout if needed.

See also: – **glyphRangeForTextContainer:**, – **characterRangeForGlyphRange:actualGlyphRange:**, – **drawsOutsideLineFragmentForGlyphAtIndex:**

rectArrayForGlyphRange:withinSelectedGlyphRange:inTextContainer: rectCount:

– (NSRect *)**rectArrayForGlyphRange:**(NSRange)*glyphRange*
withinSelectedGlyphRange:(NSRange)*selGlyphRange*
inTextContainer:(NSTextContainer *)*aTextContainer*
rectCount:(unsigned *)*rectCount*

Returns a C array of rectangles for the glyphs in *aTextContainer* in *glyphRange*, and by reference in *rectCount* the number of such rectangles. These rectangles can be used to draw the background or highlight for the given range of glyphs. *selGlyphRange* indicates selected glyphs. To calculate the rectangles for drawing the background, use a selected glyph range whose location is NSNotFound. To calculate the rectangles for highlighting, use a selected glyph range that contains *glyphRange*.

The number of rectangles returned isn't necessarily the number of lines enclosing the specified range. Contiguous lines can share an enclosing rectangle, and lines broken into several fragments have a separate enclosing rectangle for each fragment.

The array of rectangles returned is owned by the receiver, and is overwritten by various NSLayoutManager methods. You should never free it, and should copy it if you need to keep the values or use them after sending other messages to the layout manager.

The purpose of this method is to calculate line rectangles for drawing the text background and highlighting. These rectangles don't necessarily enclose glyphs that draw outside their line fragment rectangles; use **boundingRectForGlyphRange:inTextContainer:** to determine the area that contains all drawing performed for a range of glyphs.

Performs glyph generation and layout if needed.

See also: – **glyphRangeForTextContainer:**, – **drawsOutsideLineFragmentForGlyphAtIndex:**

removeTextContainerAtIndex:

– (void)**removeTextContainerAtIndex:**(unsigned int)*index*

Removes the NSTextContainer at *index* and invalidates the layout as needed.Also invalidates glyph information as needed.

See also: – **addTextContainer:**, – **insertTextContainer:atIndex:**, – **textContainers**,
– **invalidateGlyphsForCharacterRange:changeInLength:actualCharacterRange:**,
– **invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:**

replaceGlyphAtIndex:withGlyph:

– (void)**replaceGlyphAtIndex:**(unsigned int)*glyphIndex* **withGlyph:**(NSGlyph)*newGlyph*

Replaces the glyph at *glyphIndex* with *newGlyph*. Doesn't alter the glyph-to-character mapping or invalidate layout information.

This method is for use by the glyph generation mechanism, and doesn't perform any invalidation or generation of the glyphs or layout. You should never directly invoke this method.

See also: – **setCharacterIndex:forGlyphAtIndex:**,
– **invalidateGlyphsForCharacterRange:changeInLength:actualCharacterRange:**,
– **invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:**

replaceTextStorage:

– (void)**replaceTextStorage:**(NSTextStorage *)*newTextStorage*

Replaces the NSTextStorage for the group of text-system objects containing the receiver with *newTextStorage*. All NSLayoutManagers sharing the original NSTextStorage then share the new one. This method makes all the adjustments necessary to keep these relationships intact, unlike **setTextStorage:**.

rulerAccessoryViewForTextView:paragraphStyle:ruler:enabled:

– (NSView *)**rulerAccessoryViewForTextView:**(NSTextView *)*aTextView*
paragraphStyle:(NSParagraphStyle *)*paraStyle*
ruler:(NSRulerView *)*aRulerView*
enabled:(BOOL)*flag*

Returns the accessory NSView for *aRulerView*. This accessory contains tab wells, text alignment buttons, and so on. *paraStyle* is used to set the state of the controls in the accessory NSView; it must not be **nil**. If *flag* is YES the accessory view is enabled and accepts mouse and keyboard events; if NO it's disabled.

This method is invoked automatically by the NSTextView object using the layout manager. You should rarely need to invoke it, but you can override it to customize ruler support. If you do this method directly, not that it neither installs the ruler accessory view nor sets the markers for the NSRulerView. You must install the accessory view into the ruler using NSRulerView's **setAccessoryView:** method. To set the markers, use **rulerMarkersForTextView:paragraphSyle:ruler:** to get the markers needed and then send **setMarkers:** to the ruler.

See also: – **horizontalRulerView** (NSScrollView)

rulerMarkersForTextView:paragraphStyle:ruler:

– (NSArray *)**rulerMarkersForTextView:**(NSTextView *)*aTextView*
paragraphStyle:(NSParagraphStyle *)*paraStyle*
ruler:(NSRulerView *)*aRulerView*

Returns the NSRulerMarkers for *aRulerView* in *aTextView*, based on *paraStyle*. These markers represent such things as left and right margins, first-line indent, and tab stops. You can set these markers immediately with NSRulerView's **setMarkers:** method.

This method is invoked automatically by the NSTextView object using the layout manager. You should rarely need to invoke it; but you can override it to add new kinds of markers or otherwise customize ruler support.

See also: – **rulerAccessoryViewForTextView:paragraphStyle:ruler:enabled:**

setBackgroundLayoutEnabled:

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

Sets according to *flag* whether the receiver generates glyphs and lays them out when the application's run loop is idle.

See also: – **backgroundLayoutEnabled**

setCharacterIndex:forGlyphAtIndex:

– (void)**setCharacterIndex:**(unsigned int)*charIndex* **forGlyphAtIndex:**(unsigned int)*glyphIndex*

Maps the character at *charIndex* to the glyph at *glyphIndex*.

This method is for use by the glyph generation mechanism, and doesn't perform any invalidation or generation of the glyphs or layout. You should never directly invoke this method.

See also: – **characterIndexForGlyphAtIndex:**,
– **characterRangeForGlyphRange:actualGlyphRange:**,
– **glyphRangeForCharacterRange:actualCharacterRange:**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the receiver's delegate to *anObject*, without retaining it.

See also: – **delegate**

setDrawsOutsideLineFragment:forGlyphAtIndex:

– (void)**setDrawsOutsideLineFragment:**(BOOL)*flag* **forGlyphAtIndex:**(unsigned int)*glyphIndex*

Sets according to *flag* whether the glyph at *glyphIndex* exceeds the bounds of the line fragment where it's laid out. This can happen when text is set at a fixed line height. For example, if the user specifies a fixed line height of 12 points and sets the font size to 24 points, the glyphs will exceed their layout rectangles.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **drawsOutsideLineFragmentForGlyphAtIndex**

setExtraLineFragmentRect:usedRect:textContainer:

– (void)**setExtraLineFragmentRect:**(NSRect)*aRect*
usedRect:(NSRect)*usedRect*
textContainer:(NSTextContainer *)*aTextContainer*

Sets a line fragment rectangle for displaying an empty last line in a body of text. *aRect* is the rectangle to set, and *aTextContainer* is the NSTextContainer where the rectangle should be laid out. *usedRect* indicates where the insertion point is drawn.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **extraLineFragmentRect**, – **extraLineFragmentUsedRect**, – **textContainer**

setIntAttribute:value:forGlyphAtIndex:

– (void)**setIntAttribute:**(int)*attributeTag*
value:(int)*anInt*
forGlyphAtIndex:(unsigned int)*glyphIndex*

Sets a custom attribute value for the glyph at *glyphIndex*. *attributeTag* identifies the custom attribute, and *anInt* is its new value.

Subclasses that define their own custom attributes must override this method and provide their own storage for the attribute values. Non-negative tags are reserved by NeXT; you can define your own attributes with negative tags and set values using this method.

This method doesn't perform glyph generation or layout. The glyph at *glyphIndex* must already have been generated.

See also: – **intAttribute:forGlyphAtIndex:**

setLineFragmentRect:forGlyphRange:usedRect:

– (void)**setLineFragmentRect:**(NSRect)*fragmentRect*
forGlyphRange:(NSRange)*glyphRange*
usedRect:(NSRect)*usedRect*

Sets to *fragmentRect* the line fragment rectangle where the glyphs in *glyphRange* are laid out. The text container must be specified first with **setTextContainer:forGlyphRange:**, and the exact positions of the glyphs must be set after the line fragment rectangle with **setLocation:forStartOfGlyphRange:**. *usedRect* indicates the portion of *fragmentRect*, in the NSTextContainer's coordinate system, that actually contains glyphs or other marks that are drawn (including the text container's line fragment padding). *usedRect* must be equal to or contained within *fragmentRect*.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **lineFragmentRectForGlyphRange:effectiveRange:**,
– **lineFragmentUsedRectForGlyphRange:effectiveRange:**

setLocation:forStartOfGlyphRange:

– (void)**setLocation:**(NSPoint)*aPoint* **forStartOfGlyphRange:**(NSRange)*glyphRange*

Sets the location where the glyphs in *glyphRange* are laid out to *aPoint*, which is expressed relative to the origin of the line fragment rectangle for *glyphRange*. *glyphRange* defines a series of glyphs that can be displayed with a single PostScript **show** operation (a nominal range). Setting the location for a series of glyphs implies that the glyphs preceding it can't be included in a single **show** operation.

Before setting the location for a glyph range, you must specify the text container with **setTextContainer:forGlyphRange:** and the line fragment rectangle with **setLineFragmentRect:forGlyphRange:usedRect:**.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **rangeOfNominallySpacedGlyphsContainingIndex:**

setNotShownAttribute:forGlyphAtIndex:

– (void)**setNotShownAttribute:**(BOOL)*flag* **forGlyphAtIndex:**(unsigned int)*glyphIndex*

Sets according to *flag* whether the glyph at *glyphIndex* is one that isn't shown. For example, a tab or newline character doesn't leave any marks; it just indicates where following glyphs are laid out. Raises an NSRangeException if *glyphIndex* is out of bounds.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **notShownAttributeForGlyphAtIndex**

setShowsControlCharacters:

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

Controls whether the receiver makes control characters visible in layout where possible. If *flag* is YES, it substitutes visible glyphs for control characters if the font and script support it; if *flag* is NO it doesn't.

See also: – **setShowsInvisibleCharacters:**, – **showsControlCharacters**

setShowsInvisibleCharacters:

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

Controls whether the receiver makes whitespace and other typically nonvisible characters visible in layout where possible. If *flag* is YES, it substitutes visible glyphs for invisible characters if the font and script support it; if *flag* is NO it doesn't.

See also: – **setShowsControlCharacters:**, – **showsInvisibleCharacters**

setTextContainer:forGlyphRange:

– (void)**setTextContainer:**(NSTextContainer *)*aTextContainer*
forGlyphRange:(NSRange)*glyphRange*

Sets to *aTextContainer* the NSTextContainer where the glyphs in *glyphRange* are laid out. You specify the layout within the container with the **setLineFragmentRect:forGlyphRange:usedRect:** and **setLocation:forStartOfGlyphRange:** methods.

This method is used by the layout mechanism; you should never invoke it directly.

See also: – **textContainerForGlyphRange:effectiveRange:**

setTextStorage:

– (void)**setTextStorage:**(NSTextStorage *)*textStorage*

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

See also: – **addLayoutManager:** (NSTextStorage)

setUsesScreenFonts:

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

Sets according to *flag* whether the receiver calculates layout and displays text using screen fonts when possible.

See also: – **usesScreenFonts**, – **substituteFontForFont:**

showsControlCharacters

– (BOOL)**showsControlCharacters**

Returns YES if the receiver substitutes visible glyphs for control characters if the font and script support it, NO if it doesn't.

See also: – **showsInvisibleCharacters**, – **setShowsControlCharacters:**

showsInvisibleCharacters

– (BOOL)**showsInvisibleCharacters**

Returns YES if the receiver substitutes visible glyphs for invisible characters if the font and script support it, NO if it doesn't.

See also: – **showsControlCharacters**, – **setShowsInvisibleCharacters:**

substituteFontForFont:

– (NSFont *)**substituteFontForFont:**(NSFont *)*originalFont*

Returns a screen font suitable for use in place of *originalFont*, or simply returns *originalFont* if a screen font can't be used or isn't available. A screen font can be substituted if the receiver is set to use screen fonts and if no NSTextView associated with the receiver are scaled or rotated.

See also: – **usesScreenFonts**

textContainerChangedGeometry:

– (void)**textContainerChangedGeometry:**(NSTextContainer *)*aTextContainer*

Invalidates the layout information, and possibly glyphs, for *aTextContainer* and all subsequent NSTextContainers. This method is invoked automatically by other components of the text system; you should rarely need to invoke it directly. Subclasses of NSTextContainer, however, must invoke this method any time their size of shape changes (a text container that dynamically adjusts its shape to wrap text around placed graphics, for example, must do so when a graphic is added, moved, or removed).

textContainerChangedTextView:

– (void)**textContainerChangedTextView:**(NSTextContainer *)*aTextContainer*

Updates information needed to manage NSTextView objects. This method is invoked automatically by other components of the text system; you should rarely need to invoke it directly.

textContainerForGlyphAtIndex:effectiveRange:

– (NSTextContainer *)**textContainerForGlyphAtIndex:**(unsigned int)*glyphIndex*
effectiveRange:(NSRange *)*effectiveGlyphRange*

Returns the NSTextContainer where the glyph at *glyphIndex* is laid out. If non-NULL, *effectiveGlyphRange* is set to the range for all glyphs laid out in that text container.

Performs glyph generation and layout if needed.

See also: – **setTextContainer:forGlyphAtIndex:**

textContainers

– (NSArray *)**textContainers**

Returns the receiver's NSTextContainers.

See also: – **addTextContainer:**, – **insertTextContainer:atIndex:**, – **removeTextContainerAtIndex:**

textStorage

– (NSTextStorage *)**textStorage**

Returns the receiver's NSTextStorage.

See also: – **setTextStorage:**, – **replaceTextStorage:**

textStorage:edited:range:changeInLength:invalidatedRange:

– (void)**textStorage:**(NSTextStorage *)*aTextStorage*
edited:(unsigned int)*mask*
range:(NSRange)*range*
changeInLength:(int)*lengthChange*
invalidatedRange:(NSRange)*invalidatedCharRange*

Invalidates glyph and layout information for a portion of text in *aTextStorage*. This message is sent from NSTextStorage's **processEditing** method to indicate that its characters or attributes have been changed. This method invalidates glyphs and layout for the affected characters, and performs a soft invalidation of the layout information for all subsequent characters. *mask* specifies the nature of the changes. Its value is made by combining these options with the C bitwise OR operator:

Option	Meaning
NSTextStorageEditedAttributes	Attributes were added, removed, or changed.
NSTextStorageEditedCharacters	Characters were added, removed, or replaced.

range indicates the extent of characters resulting from the edits. If the NSTextStorageEditedCharacters bit of *mask* is set, *lengthChange* gives the number of characters added to or removed from the original range (otherwise its value is irrelevant). For example, after replacing "The" with "Several" to produce the string "Several files couldn't be saved", *range* is {0, 7} and *lengthChange* is 4. The receiver uses this information to update its character-to-glyph mapping and to update the selection range based on the change.

invalidatedRange represents the range of characters affected after attributes have been fixed. For example, deleting a paragraph separator character invalidates the layout information for all characters in the paragraphs that precede and follow the separator.

textStorage:edited:range:changeInLength:invalidatedRange: messages are sent in a series to each NSLayoutManager associated with the text storage object, so the NSLayoutManagers receiving them shouldn't edit *aTextStorage*. If one of them does, the *range*, *lengthChange*, and *invalidatedRange* arguments will be incorrect for all following NSLayoutManagers that receive the message.

See also: – **invalidateLayoutForCharacterRange:isSoft:actualCharacterRange:**

textViewForBeginningOfSelection

– (NSTextView *)**textViewForBeginningOfSelection**

Returns the NSTextView containing the first glyph in the selection, or **nil** if there's no selection or if there isn't enough layout information to determine the text view.

underlineGlyphRange:underlineType:lineFragmentRect:lineFragmentGlyphRange:containerOrigin:

– (void)**underlineGlyphRange:**(NSRange)*glyphRange*
underlineType:(int)*underlineType*
lineFragmentRect:(NSRect)*lineRect*
lineFragmentGlyphRange:(NSRange)*lineGlyphRange*
containerOrigin:(NSPoint)*containerOrigin*

Calculates and draws underlining for the glyphs in *glyphRange*, which must belong to a single line fragment rectangle (as returned by **lineFragmentRectForGlyphAtIndex:effectiveRange:**). *underlineType* indicates the style of underlining to draw; NSLayoutManager accepts only NSSingleUnderlineStyle, but subclasses can define their own underline styles. *lineRect* is the line fragment rectangle containing the glyphs to draw underlining for, and *lineGlyphRange* is the range of all glyphs within that line fragment rectangle. *containerOrigin* is the origin of the line fragment rectangle's NSTextContainer in its NSTextView.

This method determines which glyphs actually need to be underlined based on *underlineType*. With NSSingleUnderlineStyle, for example, leading and trailing whitespace isn't underlined, but whitespace between visible glyphs is. A potential word-underline style would omit underlining on any whitespace.

After determining which glyphs to draw underlining on, this method invokes **drawUnderlineForGlyphRange:...** for each contiguous range of glyphs that requires it.

See also: – **textContainerForGlyphAtIndex:effectiveRange:**, – **textContainerOrigin** (NSTextView)

usedRectForTextContainer:

– (NSSize)**usedRectForTextContainer:**(NSTextContainer *)*aTextContainer*

Returns the bounding rectangle for the glyphs laid out in *aTextContainer*, which tells "how full" it is. This rectangle is given in the *aTextContainer*'s coordinate system.

See also: – **containerSize** (NSTextContainer)

usesScreenFonts

– (BOOL)**usesScreenFonts**

Returns YES if the receiver calculates layout and displays text using screen fonts when possible, NO otherwise.

See also: – **setUsesScreenFonts:**, – **substituteFontForFont:**

Methods Implemented By the Delegate

layoutManager:didCompleteLayoutForTextContainer:atEnd:

– (void)**layoutManager:**(NSLayoutManager *)*aLayoutManager*
didCompleteLayoutForTextContainer:(NSTextContainer *)*aTextContainer*
atEnd:(BOOL)*flag*

Informs the delegate that *aLayoutManager* has finished laying out text in *aTextContainer*. *aTextContainer* is **nil** if there aren't enough containers to hold all the text; the delegate can use this information as a cue to add another container. If *flag* is YES, *aLayoutManager* is finished laying out its text—this also means that *aTextContainer* is the final text container used by the layout manager. Delegates can use this information to show an indicator or background or to enable or disable a button that forces immediate layout of text.

layoutManagerDidInvalidateLayout:

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

Informs the delegate that *aLayoutManager* has invalidated layout information (not glyph information). This method is invoked only when layout was complete and then became invalidated for some reason. Delegates

can use this information to show an indicator or background layout or to enable a button that forces immediate layout of text.