

---

# Text System Storage Layer Overview

User Experience: Text Layout



2005-08-11



Apple Inc.  
© 1997, 2005 Apple Computer, Inc.  
All rights reserved.

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

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

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

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Cocoa, Mac, Mac OS, and Objective-C are trademarks of Apple Inc., registered in the United States and other countries.

Helvetica is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Simultaneously published in the United States and Canada.

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

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

**THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.**

**Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.**

# Contents

---

## **Introduction to Text System Storage Layer Overview 7**

Who Should Read This Document 7

Organization of This Document 7

See Also 7

---

## **The Storage Layer: The NSTextStorage Class 9**

---

### **Layout Geometry: The NSTextContainer Class 11**

---

### **Creating Text Storage 13**

---

### **Changing Text Storage 15**

---

### **Displaying a Text Container 17**

---

### **Calculating Region, Bounding Rectangle, and Inset 19**

---

### **Tracking the Size of a Text View 21**

---

### **Creating a Subclass of NSTextStorage 23**

---

### **Document Revision History 25**

---

### **Index 27**

---



# Figures

---

## The Storage Layer: The NSTextStorage Class 9

Figure 1 Capabilities of NSTextStorage 9

---

## Calculating Region, Bounding Rectangle, and Inset 19

Figure 1 Text container region, bounding rectangle, and inset 19



# Introduction to Text System Storage Layer Overview

---

*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.

## Who Should Read This Document

You should read this document if you need to work directly with the text storage layer. For example, you may need to change the text programmatically in a text storage object or extend its capabilities.

To understand this material you should have a general understanding of Cocoa programming conventions. You should also have read *Text System Overview*.

## Organization of This Document

This document contains the following articles:

- ["The Storage Layer: The NSTextStorage Class"](#) (page 9) provides a general introduction to the capabilities of text storage objects.
- ["Layout Geometry: The NSTextContainer Class"](#) (page 11) explains how text containers define the area in which the system lays out text and how they interact with other text system objects.
- ["Creating Text Storage"](#) (page 13) explains how you create and set up text storage objects.
- ["Changing Text Storage"](#) (page 15) describes the process of editing text in a text storage object programmatically.
- ["Displaying a Text Container"](#) (page 17) explains how you can display the text in a text storage object in a text view or other NSView object.
- ["Calculating Region, Bounding Rectangle, and Inset"](#) (page 19) discusses how to define a text container's text layout area.
- ["Tracking the Size of a Text View"](#) (page 21) explains how you can set up a text container so that its geometry interacts with that of its associated text view.
- ["Creating a Subclass of NSTextStorage"](#) (page 23) discusses the requirements of NSTextStorage subclasses.

## See Also

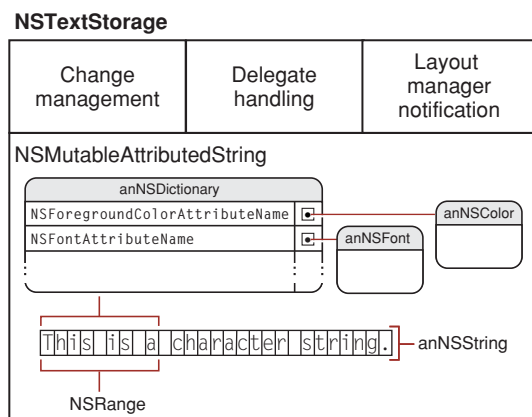
For further reading, refer to the following documents:

- *Attributed String Programming Guide* provides information about the attributed string objects on which NSTextStorage is built. NSTextStorage is a subclass of NSMutableAttributedString.
- *Text Layout Programming Guide for Cocoa* describes the layout process involving text storage, text container, text view, and layout manager objects.

# The Storage Layer: The NSTextStorage Class

An NSTextStorage object serves as the character data repository for the Cocoa text system. The format for this data is an attributed string, which is a sequence of characters (in Unicode encoding) and the attributes (such as font, color, and paragraph style) that apply to them. The classes that represent attributed strings are NSAttributedString and NSMutableAttributedString, of which NSTextStorage is a subclass. Conceptually, each character in a block of text has a dictionary of keys and values associated with it. A key names an attribute (such as NSFontAttributeName), and the associated value specifies the characteristics of that attribute (such as Helvetica 12-point). For more information about attributed strings, see *Attributed String Programming Guide*. Figure 1 illustrates the NSTextStorage class, showing its NSMutableAttributedString component and its additional capabilities.

Figure 1 Capabilities of NSTextStorage



The NSTextStorage methods let you operate programmatically on the attributes of the text displayed by the NSTextView object; for example, your code can iterate through the text, tightening or loosening the kerning for all characters of a certain font and size. An NSTextView object enables users to affect character attributes through direct action; for example, the user selects some text and reduces the spacing between characters by choosing the Tighten menu command.



# Layout Geometry: The NSTextContainer Class

---

An NSTextContainer object defines the area on a page in which the Cocoa text system lays out text. By default, a text container defines a simple, rectangular area, but you can create subclasses that define areas with any geometrical shape, including regions with holes around which text flows.

NSTextContainer provides one of the four primary text objects in the Cocoa text system. Text containers work with text storage objects, layout managers, and text views to store, lay out, and display attributed text strings. In particular, a text container works directly with a layout manager, which uses an NSTypesetter object to generate line-fragment rectangles in which to place glyphs (character shapes), as described in “Line Fragment Generation.”

When the typesetter generates line fragments, the text container is particularly concerned with the direction in which text layout proceeds. There are two aspects to layout direction: line sweep and line movement. Line sweep is the direction in which the system lays out glyphs within lines. Line movement is the direction in which the system lays out lines upon the page. The typesetter object determines these parameters and passes them as constant values to the text container. Both line sweep and line movement can proceed from left to right, right to left, top to bottom, and bottom to top. In addition, the typesetter can specify no line movement.

**Note:** The built-in typesetters currently support only top-to-bottom line movement and left-to-right sweep. These typesetters do handle bidirectional text (Hebrew and Arabic) by laying it out in the proper display order within the line fragments, but they do not use the line sweep mechanism.

The layout manager maintains an array of text containers. It sends a message to its delegate whenever it fills a text container, and the delegate can then add a new text container to be filled. If a text container changes size, or if changes to laid-out text in a container invalidate layout at that point, then the system invalidates the layout in all the subsequent containers in the layout manager’s array.

You can specify that a text container track the size of its text view; that is, if the user resizes the view, the text container resizes itself to match. For more information, see [“Tracking the Size of a Text View”](#) (page 21).

NSTextContainer instances have methods for initialization, managing connection to layout managers and text views, getting and setting the container size, generating line fragments, and hit-testing. In addition, NSTextContainer has methods for getting and setting the amount of padding to apply to line fragments. Line fragment padding is extra space included at the ends of line fragments so laid-out glyphs don’t directly touch other elements on the page, such as graphics.



# Creating Text Storage

---

As an abstract class of a class cluster, allocating and initializing an `NSTextStorage` object in Objective-C actually produces an instance of a private subclass. You can use any `NSAttributedString` or `NSMutableAttributedString` initialization method to create an `NSTextStorage` object. To create an `NSTextStorage` in Java, use one of its constructors.

Having created the text storage object, you add `NSLayoutManager` objects to the text storage using `addLayoutManager:`. A single text storage object can have multiple layout managers (which is why this method name begins with “add” rather than “set”).

Creating a text storage object in this way and adding a layout manager is part of the process of assembling the text system programmatically, which is described in more detail in “Assembling the Text System by Hand.” You can also create an `NSTextView` object and let it assemble the text system automatically, in which case the text view creates (and later deallocates) the text storage object. For more information, see “Creating a Text View Programmatically.”



# Changing Text Storage

---

The behavior of an `NSTextStorage` object is best illustrated by following the messages you send to change its text. There are three stages to editing a text storage object programmatically:

1. The first stage is to send it a `beginEditing` message to announce a group of changes.
2. In the second stage, you send it some editing messages, such as `deleteCharactersInRange:` and `addAttributes:range:`, to effect the changes in characters or attributes. Each time you send such a message, the text storage object invokes `edited:range:changeInLength:` to track the range of its characters affected since it received the `beginEditing` message.
3. For the third stage, when you're done changing the text storage object, you send it the `endEditing` message. This causes it to invoke its own `processEditing` method, fixing attributes within the recorded range of changed characters. (See *Text Attributes* for information about attribute fixing.)

After fixing its attributes, the text storage object sends a message to each associated layout manager indicating the range in the text storage object that has changed, along with the nature of those changes. The layout managers in turn use this information to recalculate their glyph locations and redisplay if necessary. `NSTextStorage` also keeps a delegate and sends it messages before and after processing edits.



# Displaying a Text Container

---

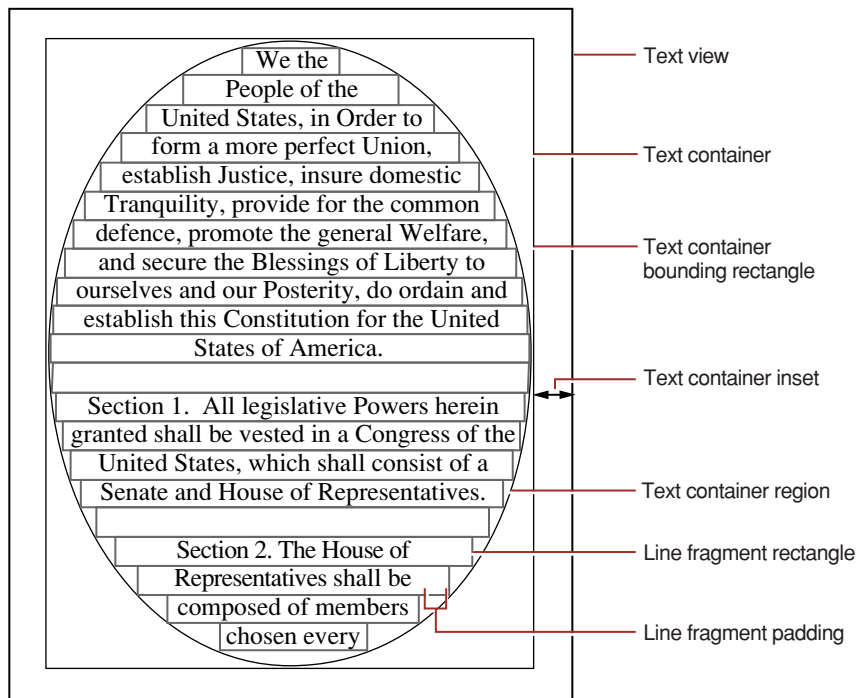
You normally use an `NSTextView` object to display the text laid out in an `NSTextContainer`. An `NSTextView` can have only one `NSTextContainer`; however, because the two are separate objects, you can replace an `NSTextView`'s container to change the layout of the text it displays. You can also display an `NSTextContainer`'s text in any `NSView` by locking the graphic focus on it with `lockFocus:` and using the `NSLayoutManager` methods `drawBackgroundForGlyphRange:atPoint:` and `drawGlyphsForGlyphRange:atPoint:`. If you have no need to actually display 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`.



# Calculating Region, Bounding Rectangle, and Inset

An `NSTextContainer` object's region is defined by 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. Figure 1 illustrates these aspects of a text container.

**Figure 1** Text container region, bounding rectangle, and inset



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, `lineFragmentRectForProposedRect:sweepDirection:movementDirection:remainingRect:`, 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 "Line Fragment Generation."

An `NSTextContainer` usually covers its `NSTextView` exactly, but it can be inset within the view frame with the `NSTextView` `setTextContainerInset:` method. The `NSTextContainer` object's bounding rectangle from the inset position then establishes the limits of the `NSTextContainer` object'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 a Text View"](#) (page 21).

Note that the text container inset does not fully determine the position of the container in the text view. The text view calculates the position of the text container within it, and it tries to maintain the amount of space given by the text container inset, but depending on the relative sizes of the text view and text container, that may not be possible. It's also possible that there's more space to be distributed than that specified by the text container inset. If you want to determine the true location of the text container, for example, to convert between container and view coordinates, you should use the `NSTextView` `textContainerOrigin` method, which is the actual value calculated by the text view.

# Tracking the Size of a Text View

---

You can set an `NSTextContainer` to track the size of its `NSTextView` and adjust its own size to match whenever the `NSTextView` 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 (which are inherited from `NSText`) 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—for example, 10,000,000 points (this incurs no cost whatever in processing or storage).

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.



# Creating a Subclass of NSTextStorage

---

NSTextStorage isn't a fully concrete class; rather, it is the abstract superclass of a class cluster, as described in "Class Clusters". It defines the storage for its NSLayoutManager objects and implements some methods, but doesn't provide the primitive attributed string methods to subclasses. A subclass must define the storage for its attributed string, typically as an instance variable of type NSMutableAttributedString, override `init` and define its own initialization methods (or define constructors in Java), and implement the primitive methods of both NSAttributedString and NSMutableAttributedString. The Objective-C primitive methods are:

```
string
attributesAtIndex:effectiveRange:
replaceCharactersInRange:withString:
setAttributes:range:
```

The Java methods are:

```
stringReference
attributesAtIndex
replaceCharactersInRange
setAttributesInRange
```

Beyond these requirements, if a subclass overrides or adds any methods that change its characters or attributes directly, those methods must invoke `edited:range:changeInLength:` (in Java, `editedInRange`) after performing the change in order to keep the change-tracking information up to date. See the description of this method for more information.



# Document Revision History

---

This table describes the changes to *Text System Storage Layer Overview*.

Date	Notes
2005-08-11	Revised "Creating a Subclass of NSTextStorage" to remove ambiguity about invoking <code>edited:range:changeInLength:</code> (in Java, <code>editedInRange</code> ). Changed title from "Text System Storage Layer."
2004-02-13	Rewrote introduction and added an index.
2003-05-02	Added NSTextContainer article, clarifications, links, and minor revisions throughout.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.



# Index

---

## A

---

`addAttributes:range:` [method 15](#)  
`addLayoutManager:` [method 13](#)  
[attribute fixing 15](#)  
[attributed string 9](#)  
[attributes of text. See text attributes](#)  
`attributesAtIndex` [method \(Java\) 23](#)  
`attributesAtIndex:effectiveRange:` [method 23](#)

## B

---

`beginEditing` [method 15](#)  
[bounding rectangle](#)  
  of `NSTextContainer` [region 19](#)

## C

---

`containerSize` [method 19](#)  
`containsPoint:` [method 19](#)

## D

---

[delegation 11](#)  
`deleteCharactersInRange:` [method 15](#)  
`drawBackgroundForGlyphRange:atPoint:` [method 17](#)  
`drawGlyphsForGlyphRange:atPoint:` [method 17](#)

## E

---

`edited:range:changeInLength:` [method 15, 23](#)  
`editedInRange` [method \(Java\) 23](#)  
[editing](#)  
  [programmatic 15](#)

`endEditing` [method 15](#)

## I

---

`isSimpleRectangularTextContainer` [method 19](#)

## L

---

[layout manager](#)  
  [adding to text storage 13](#)  
  [programmatic editing and 15](#)  
  [text containers and 11](#)  
[layout of text. See text layout](#)  
[line movement 11](#)  
[line sweep 11](#)  
[line-fragment rectangles 11](#)  
`lineFragmentRectForProposedRect:sweepDirection:`  
  `movementDirection:remainingRect:` [method 19](#)  
`lockFocus:` [method 17](#)

## N

---

`NSAttributedString` [class 9, 13, 23](#)  
`NSLayoutManager` [class 13, 23](#)  
`NSMutableAttributedString` [class 9, 13, 23](#)  
`NSTextContainer` [class](#)  
  [described 11](#)  
  [displaying 17](#)  
  [methods 11](#)  
  [region 19](#)  
  [tracking text view size 21](#)  
`NSTextStorage` [class 9, 23](#)  
`NSTextView` [class 9, 17, 20](#)  
`NSTypesetter` [class 11](#)

## P

---

primitive methods for NSTextStorage subclasses [23](#)  
processEditing [method 15](#)  
programmatic editing [15](#)

## R

---

replaceCharactersInRange [method \(Java\) 23](#)  
replaceCharactersInRange:withString: [method 23](#)

## S

---

setAttributes:range: [method 23](#)  
setAttributesInRange [method \(Java\) 23](#)  
setContainerSize: [method 19](#)  
setHeightTracksTextView: [method 21](#)  
setHorizontallyResizable: [method 21](#)  
setTextContainerInset: [method 20](#)  
setVerticallyResizable: [method 21](#)  
setWidthTracksTextView: [method 21](#)  
string [method 23](#)  
stringReference [method \(Java\) 23](#)

## T

---

text attributes  
  fixing [15](#)  
  NSTextStorage and [9](#)  
text containers  
  described [11](#)  
  inset [21](#)  
  region [19](#)  
  resizing [21](#)  
text layout [11, 19](#)  
text objects [11](#)  
text views [11](#)  
typesetter [11](#)

## U

---

Unicode  
  NSTextStorage and [9](#)