NSAttributedString Class Cluster Additions

Class Cluster Description

NSAttributedString objects manage character strings and associated sets of attributes (for example, font and kerning) that apply to individual characters or ranges of characters in the string. An association of characters and their attributes is called an *attributed string*. The cluster's two public classes, NSAttributedString and NSMutableAttributedString, declare the programmatic interface for read-only attributed strings and modifiable attributed strings, respectively. The Foundation Kit defines the basic functionality for attributed strings, while the remainder is defined here in the Application Kit. The Application Kit also uses a subclass of NSMutableAttributedString, called NSTextStorage, to provide the storage for NeXT's extended text-handling system.

Note: NSAttributedString is *not* a subclass of NSString. It contains a string object to which it applies attributes. This protects users of attributed strings from ambiguities caused by the semantic differences between simple and attributed string. For example, equality can't be simply defined between an NSString and an attributed string.

Because of the nature of class clusters, attributed string objects are not actual instances of the NSAttributedString or NSMutableAttributedString classes, but are instances of one of their private concrete subclasses. Although an attributed string object's class is private, its interface is public, as declared by these abstract superclasses, NSAttributedString and NSMutableAttributedString. The attributed string classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert an attributed string from one type to the other.

You create an NSAttributedString object from scratch by using one of **initWithString:**, **initWithString:attributes:**, or **initWithAttributedString:**. These methods initialize an attributed string with data you provide. You can also create an attributed string from RTF data using **initWithRTF:documentAttributes:**, **initWithRTFD:documentAttributes:**, **initWithRTFDFileWrapper:documentAttributes:**, or **initWithPath:documentAttributes:**. See "RTF Document Attributes" for more details on reading RTF.

An attributed string provides basic access to its contents with the **string** and **attributesAtIndex:effectiveRange:** methods, which yield characters and attributes, respectively. These two primitive methods are used by the other access methods to retrieve attributes individually by name, by functional group (font or ruler attributes, for example), and so on.

Feature Overview

NSAttributedString and NSMutableAttributedString add a number of features to the basic content storage of NSString:

- Association of arbitrary, programmer-defined attributes with ranges of characters

- Preservation of attribute-to-character mapping after changes (NSMutableAttributedString)
- Support for RTF, including file attachments and graphics
- Drawing in NSView objects (note that the Application Kit adds drawing methods to NSString as well)
- Linguistic unit (word) and line calculation

An attributed string identifies attributes by name, storing their values as opaque **id**s in an NSDictionary. For example, the text font is stored as an NSFont object under the name given by NSFontAttributeName. You can associate any object value, by any name, with a given range of characters in the attributed string. The basic attributes defined by NSAttributedString are described below under "Accessing Attributes."

A mutable attributed string keeps track of the attribute mapping as characters are added to and deleted from it and as attributes are changed. It allows you to group batches of edits with the **beginEditing** and **endEditing** methods, and to consolidate changes to the attribute-to-character mapping with the **fix...** methods. See "Changing a Mutable Attributed String" below for more information.

An attributed string can be created from rich text (RTF) or rich text with attachments (RTFD), and can write its contents as RTF or RTFD data. Three initialization methods, **initWithRTF:documentAttributes:**, **initWithRTFD:documentAttributes:**, and **initWithRTFDFileWrapper:documentAttributes:**, interpret rich text data. The methods for writing rich text are **RTFFromRange:documentAttributes:** and **RTFDFromRange:documentAttributes:**, which return rich text data for any legal range within the attributed string, and **RTFDFileWrapperFromRange:documentAttributes:**, which returns the attributed string as an NSFileWrapper. NSAttributedString provides limited support for some document-level attributes, as described below under "RTF Document Attributes." Additional support for rich text is provided by other text-handling classes such as NSTextView.

You can draw an attributed string in a focused NSView by invoking either the **drawAtPoint:** or **drawInRect:** method. Note that the Application Kit defines drawing methods for NSString as well, allowing any string object to draw itself. These methods, **drawAtPoint:withAttributes:** and **drawInRect:withAttributes:**, are described in the NSString Additions class specification.

An attributed string supports the typical behavior of editors in selecting a word on a double-click with the **doubleClickAtIndex:** method, and finds word breaks with **nextWordFromIndex:forward:**. It also calculates line breaks with the **lineBreakBeforeIndex:withinRange:** method.

Accessing Attributes

An attributed string identifies attributes by name, storing an **id** value under the attriute name in an NSDictionary, which is in turn associated with an NSRange that indicates the characters to which the dictionary's attributes apply. You can assign any attribute name/value pair you wish to a range of characters, in addition to these standard attributes:

Attribute Identifier	Value Class	Default Value
NSFontAttributeName	NSFont	Helvetica 12-point
NSForegroundColorAttributeName	NSColor	black
NSBackgroundColorAttributeName	NSColor	none (no background drawn)
NSUnderlineStyleAttributeName	NSNumber, as an **int**	none (no underline)
NSSuperscriptAttributeName	NSNumber, as an **int**	0
NSBaselineOffsetAttributeName	NSNumber, as a **float**	0.0
NSKernAttributeName	NSNumber, as a **float**	0.0
NSLigatureAttributeName	NSNumber, as an **int**	1 (standard ligatures)
NSParagraphStyleAttributeName	NSParagraphStyle	(as returned by NSParagraphStyle's **defaultParagraphStyle** method)
NSAttachmentAttributeName	NSTextAttachment	none (no attachment)

The identifiers listed are actually global NSString variables containing the attribute names. The value class is what users of an attributed string should expect the attribute values to be presented as. The default values are what they should assume if no attribute value has been explicitly set for the requested character range.

The natures of several attributes aren't obvious from name alone:

- The underline attribute has only one value defined, NSSingleUnderlineStyle. All characters with this attribute value should be drawn with a single line just below the baseline.
- The superscript attribute indicates an abstract level for both super- and subscripts. The user of the attributed string can interpret this as desired, adjusting the baseline by the same or a different amount for each level, changing the font size, or both.
- The baseline offset attribute is a literal distance by which the characters should be shifted above the baseline (for positive offsets) or below (for negative offsets).
- The kerning attribute indicates how much the following character should be shifted from its default offset as defined by the current character's font; a positive kern indicates a shift farther along and a negative kern indicates a shift closer to the current character.
- The ligature attribute determines what kinds of ligatures should be used when displaying the string. A value of 0 indicates that only ligatures essential for proper rendering of text should be used, 1 indicates that standard ligatures should be used, and 2 indicates that all available ligatures should be used. Which ligatures are standard depends on the script and possibly the font. Arabic text, for example, requires ligatures for many character sequences, but has a rich set of additional ligatures that combine characters. English text has no essential ligatures, and typically has only two standard ligatures, those for "fi" and "fl"—all others being considered more advanced or fancy.

With an immutable attributed string, you assign all attributes on creating the string using methods such as **initWithRTF:documentAttributes:**, which interprets attributes from the RTF data, **initWithString:attributes:**, which explicitly takes an NSDictionary of name/value pairs, or **initWithString:**, which assigns no attributes. See "Changing a Mutable Attributed String" below for information on assigning attributes with a mutable attributed string.

To retrieve attribute values from either type of attributed string, use any of these methods:

– attributesAtIndex:effectiveRange:
– attributesAtIndex:longestEffectiveRange:inRange:
– attribute:atIndex:effectiveRange:
– attribute:atIndex:longestEffectiveRange:inRange:
– fontAttributesInRange:
– rulerAttributesInRange:

The first two methods return all attributes at a given index, the **attribute:...** methods return the value of a single named attribute, and **fontAttributesInRange:** and **rulerAttributesInRange:** return attributes defined to apply only to characters or to whole paragraphs, respectively (see the individual method descriptions for more information).

The first four methods also return by reference the *effective range* and the *longest effective range* of the attributes. These ranges allow you to determine the extent of attributes. Conceptually, each character in an attributed string has its own collection of attributes; however, it's often useful to know when the attributes and values are the same over a series of characters. This allows a routine to progress through an attributed string in chunks larger than a single character. In retrieving the effective range, an attributed string simply looks up information in its attribute mapping, essentially the dictionary of attributes that apply at the index requested. In retrieving the longest effective range, the attributed string continues checking characters past this basic range as long as the attribute values are the same. This extra comparison increases the execution time for these methods but guarantees a precise maximal range for the attributes requested.The code fragment below progresses through an attributed string in chunks based on the effective range. The fictitious **analyzer** object here counts the number of characters in each font. The **while** loop progresses as long as the effective range retrieved doesn't include the end of the attributed string, retrieving the font in effect just past the latest retrieved range. For each font attribute retrieved, **analyzer** is asked to tally the number of characters in the effective range. In this example, it's possible that consecutive invocations of **attribute:atIndex:effectiveRange:** will return the same value.

```
NSAttributedString *attrStr;
unsigned int length;
NSRange effectiveRange;
id attributeValue;

length = [attrStr length];
effectiveRange = NSMakeRange(0, 0);

while (NSMaxRange(effectiveRange) < length) {
    attributeValue = [attrStr attribute:NSFontAttributeName
        atIndex:NSMaxRange(effectiveRange) effectiveRange:&effectiveRange];
    [analyzer tallyCharacterRange:effectiveRange font:attributeValue];
}
```

In contrast, the next code fragment progresses through the attributed string according to the maximum effective range for each font. In this case, **analyzer** counts font *changes*, which may not be represented by merely retrieving effective ranges. In this case the **while** loop is predicated on the length of the limiting

range, which begins as the entire length of the attributed string and is whittled down as the loop progresses. After **analyzer** records the font change, the limit range is adjusted to account for the longest effective range retrieved.

```
NSAttributedString *attrStr;
NSRange limitRange;
NSRange effectiveRange;
id attributeValue;

limitRange = NSMakeRange(0, [attrStr length]);

while (limitRange.length > 0) {
    attributeValue = [attrStr attribute:NSFontAttributeName
        atIndex:limitRange.location longestEffectiveRange:&effectiveRange
        inRange:limitRange];
    [analyzer recordFontChange:attributeValue];
    limitRange = NSMakeRange(NSMaxRange(effectiveRange),
        NSMaxRange(limitRange) - NSMaxRange(effectiveRange));
}
```

Note that the second code fragment is more complex. Because of this, and because **attribute:atIndex:longestEffectiveRange:inRange:** is somewhat slower than **attribute:atIndex:effectiveRange:**, you should typically use it only when absolutely necessary for the work you're performing. In most cases working by effective range is enough.

Changing a Mutable Attributed String

NSMutableAttributedString declares a number of methods for changing both characters and attributes, such as the primitive **replaceCharactersInRange:withString:** and **setAttributes:range:**, or the more convenient methods **addAttribute:value:range:**, **applyFontTraits:range:**, **setAlignment:range:**, and so on. All of the methods for changing a mutable attributed string properly update the mapping between characters and attributes, but after a change some inconsistencies can develop. Here are some examples of attribute consistency requirements:

- Paragraph styles must apply to entire paragraphs.
- Scripts may only be assigned fonts that support them. For example, Kanji and Arabic characters can't be assigned the Times-Roman font, and must be reassigned fonts that support these scripts.
- Deleting attachment characters from the string requires the corresponding attachment objects to be released. Similarly, removing attachment objects requires the corresponding attachment characters to be removed from the string.
- A code editing application that displays all language keywords in boldface can automatically assign this attriute as the user changes the font or edits the text.

NSMutableAttributedString defines methods to fix these inconsistencies as changes are made. This allows the attributes to be cleaned up at a low level, hiding potential problems from higher levels and providing for very clean update of display as attributes change. There are six methods for fixing attributes:

– fixAttributesInRange:
– fixAttachmentAttributeInRange:
– fixFontAttributeInRange:
– fixParagraphStyleAttributeInRange:
– beginEditing
– endEditing

The first method, **fixAttributesInRange:**, invokes the other three **fix...** methods to clean up deleted attachment references, font attributes, and paragraph attributes, respectively. The individual method descriptions explain what cleanup entails for each case.

The **beginEditing** and **endEditing** methods are provided for subclasses of NSMutableAttributedString to override. Their default implementations do nothing. These methods allow instances of a subclass to record or buffer groups of changes and clean themselves up on receiving an **endEditing** message. **endEditing** also allows the receiver to notify any observers that it has been changed. NSTextStorage's implementation of **endEditing**, for example, fixes changed attributes and then notifies its NSLayoutManagers that they need to re-lay and redisplay their text.

RTF Document Attributes

Attributed strings keep attribute information for their text only, while RTF allows for more general attributes of a document, especially regarding paper size and layout. To support higher-level objects that use attributed strings, the methods that work with RTF also read and write some RTF directives for document attributes, stored in an NSDictionary under these keys:

Attribute Key	Value Class
PaperSize	NSValue, as an NSSize
LeftMargin	NSNumber, as a **float**
RightMargin	NSNumber, as a **float**
TopMargin	NSNumber, as a **float**
BottomMargin	NSNumber, as a **float**

The **init** methods, such as **initWithRTF:documentAttributes:**, return by reference a dictionary containing the attributes read from the RTF data, which your application can then use to set up its page layout. Similarly, RTF extraction methods such as **RTFFromRange:documentAttributes:**, accept a dictionary containing those attributes and writes them into the RTF data, thus preserving the page layout information.

Attachments

Attachments, such as embedded images or files, are represented in an attributed string by both a special character and an attribute. The character is identified by the global name NSAttachmentCharacter, and

indicates the presence of an attachment at its location in the string. The attribute, identified in the string by the attribute name NSAttachmentAttributeName, is an NSTextAttachment object. An NSTextAttachment contains the data for the attachment itself, as well as an image to display when the string is drawn. You can use NSAttributedString's **attributedStringWithAttachment:** class method to construct an attachment string, which you can then add to a mutable attributed string using **appendAttributedString:** or **insertAttributedString:atIndex:**.

NSAttributedString Additions

Inherits From: NSObject

Declared In: AppKit/NSAttributedString.h
AppKit/NSStringDrawing.h
AppKit/NSTextAttachment.h

Class Description

The Application Kit extends the Foundation Kit's NSAttributedString class by adding:

- Support for RTF, with or without attachments
- Graphic attributes, including font and ruler attributes
- Methods for drawing attributed strings
- Methods for calculating significant linguistic units

Method Types

Creating an NSAttributedString	– initWithRTF:documentAttributes: – initWithRTFD:documentAttributes: – initWithRTFDFileWrapper:documentAttributes: – initWithPath:documentAttributes: + attributedStringWithAttachment:
Retrieving attribute information	– fontAttributesInRange: – rulerAttributesInRange: – containsAttachments
Calculating linguistic units	– doubleClickAtIndex: – lineBreakBeforeIndex:withinRange: – nextWordFromIndex:forward:
Drawing the string	– drawAtPoint: – drawInRect: – size
Generating RTF data	– RTFFromRange:documentAttributes: – RTFDFileWrapperFromRange:documentAttributes: – RTFDFromRange:documentAttributes:

Class Methods

attributedStringWithAttachment:

+ (NSAttributedString *)**attributedStringWithAttachment:**(NSTextAttachment *)*attachment*

Returns an NSAttributedString object containing only the attachment marker character (NSAttachmentCharacter), which is assigned an attribute whose name is NSTextAttachmentName and whose value is *attachment*. Use this method, along with **appendAttributedString:** or **insertAttributedString:atIndex:**, to add an attachment to an attributed string.

Instance Methods

containsAttachments

– (BOOL)**containsAttachments**

Returns YES if the receiver contains any attachment attributes, NO otherwise. This method checks only for attachment attributes, not for NSAttachmentCharacter.

doubleClickAtIndex:

– (NSRange)**doubleClickAtIndex:**(unsigned int)*index*

Returns the range of characters that form a word (or other linguistic unit) surrounding *index*, taking language characteristics into account. Raises an NSRangeException if *index* lies beyond the end of the receiver's characters.

See also: – **nextWordFromIndex:forward:**

drawAtPoint:

– (void)**drawAtPoint:**(NSPoint)*point*

Draws the receiver with its font and other display attributes at *point* in the currently focused NSView. Text is drawn in such a way that the upper left corner of its bounding box lies at *point*, regardless of the line sweep direction or whether the NSView is flipped.

Don't invoke this method while no NSView is focused.

See also: – **lockFocus** (NSView), – **size**, – **drawInRect:**

drawInRect:

– (void)**drawInRect:**(NSRect)*rect*

Draws the receiver with its font and other display attributes within *rect* in the currently focused NSView, clipping the drawing to this rectangle. Text is drawn within *rect* according to its line sweep direction; for example, Arabic text will begin at the right edge and potentially be clipped on the left.

Don't invoke this method while no NSView is focused.

See also: – **lockFocus** (NSView), – **drawAtPoint:**

fontAttributesInRange:

– (NSDictionary *)**fontAttributesInRange:**(NSRange)*aRange*

Returns the font attributes in effect for the character at *aRange.location*. Font attributes are all those listed under "Accessing Attributes" in the class cluster description except NSParagraphStyleAttributeName and NSAttachmentAttributeName. Use this method to obtain font attributes that are to be copied or pasted with "copy font" operations. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **rulerAttributesInRange:**

initWithPath:documentAttributes:

– (id)**initWithPath:**(NSString *)*path* **documentAttributes:**(NSDictionary **)*docAttributes*

Initializes a new NSAttributedString from RTF or RTFD data contained in the file at *path*. Also returns by reference in *docAttributes* a dictionary containing document-level attributes, as listed in the class cluster description under "RTF Document Attributes." *docAttributes* may be NULL, in which case no document attributes are returned. Returns **self**, or **nil** if *rtfData* can't be decoded.

initWithRTF:documentAttributes:

– (id)**initWithRTF:**(NSData *)*rtfData* **documentAttributes:**(NSDictionary **)*docAttributes*

Initializes a new NSAttributedString by decoding the stream of RTF commands and data contained in *rtfData*. Also returns by reference in *docAttributes* a dictionary containing document-level attributes, as listed in the class cluster description under "RTF Document Attributes." *docAttributes* may be NULL, in which case no document attributes are returned. Returns **self**, or **nil** if *rtfData* can't be decoded.

initWithRTFD:documentAttributes:

– (id)**initWithRTFD:**(NSData *)*rtfdData* **documentAttributes:**(NSDictionary **)*docAttributes*

Initializes a new NSAttributedString by decoding the stream of RTFD commands and data contained in *rtfdData*. Also returns by reference in *docAttributes* a dictionary containing document-level attributes, as listed in the class cluster description under "RTF Document Attributes." *docAttributes* may be NULL, in which case no document attributes are returned. Returns **self**, or **nil** if *rtfData* can't be decoded.

initWithRTFDFileWrapper:documentAttributes:

– (id)**initWithRTFDFileWrapper:**(NSFileWrapper *)*wrapper*
documentAttributes:(NSDictionary **)*docAttributes*

Initializes a new NSAttributedString from *wrapper* an NSFileWrapper object containing an RTFD document. Also returns by reference in *docAttributes* a dictionary containing document-level attributes, as listed in the class cluster description under "RTF Document Attributes." *docAttributes* may be NULL, in which case no document attributes are returned. Returns **self**, or **nil** if *wrapper* can't be interpreted as an RTFD document.

lineBreakBeforeIndex:withinRange:

– (unsigned int)**lineBreakBeforeIndex:**(unsigned int)*index* **withinRange:**(NSRange)*aRange*

Returns the index of the closest character before *index* and within *aRange* that can be placed on a new line when laying out text. In other words, finds the appropriate line break when the character at *index* won't fit on the same line as the character at the beginning of *aRange*. Returns NSNotFound if no line break is possible before *index*.

Raises an NSRangeException if *index* or any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **nextWordFromIndex:forward:**

nextWordFromIndex:forward:

– (unsigned int)**nextWordFromIndex:**(unsigned int)*index* **forward:**(BOOL)*flag*

Returns the index of the first character of the word after or before *index*. If *flag* is YES, this is the first character after *index* that begins a word; if *flag* is NO, it's the first character before *index* that begins a word, whether *index* is located within a word or not. If *index* lies at either end of the string and the search direction would progress past that end, it's returned unchanged. This method is intended for moving the insertion point during editing, not for linguistic analysis or parsing of text.

Raises an NSRangeException if *index* lies beyond the end of the receiver's characters.

See also: – **lineBreakBeforeIndex:withinRange:**

RTFDFileWrapperFromRange:documentAttributes:

– (NSFileWrapper *)**RTFDFileWrapperFromRagne:**(NSRange)*aRange*
documentAttributes:(NSDictionary *)*docAttributes*

Returns an NSFileWrapper object that contains an RTFD document corresponding to the characters and attributes within *aRange*. The file wrapper also includes the document-level attributes in *docAttributes*, as explained in the class cluster description under "RTF Document Attributes." If there are no document-level attributes, *docAttributes* can be **nil**. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

You can save the file wrapper using NSFileWrapper's **writeToFile:atomically:updateFilenames:** method.

See also: – **RTFDFromRange:documentAttributes:**, – **RTFFromRange:documentAttributes:**

RTFDFromRange:documentAttributes:

– (NSData *)**RTFDFromRange:**(NSRange)*aRange*
documentAttributes:(NSDictionary *)*docAttributes*

Returns an NSData object that contains an RTFD stream corresponding to the characters and attributes within *aRange*. Also writes the document-level attributes in *docAttributes*, as explained in the class cluster description under "RTF Document Attributes." If there are no document-level attributes, *docAttributes* can be **nil**. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

When writing data to the pasteboard, you can use the NSData object as the first argument to NSPasteboard's **setData:forType:** method, with a second argument of NSRTFDPboardType.

See also: – **RTFFromRange:documentAttributes:**,
– **RTFDFileWrapperFromRange:documentAttributes:**

RTFFromRange:documentAttributes:

– (NSData *)**RTFFromRange:**(NSRange)*aRange*
documentAttributes:(NSDictionary *)*docAttributes*

Returns an NSData object that contains an RTF stream corresponding to the characters and attributes within *aRange*, omitting all attachment attributes. Also writes the document-level attributes in *docAttributes*, as explained in the class cluster description under "RTF Document Attributes." If there are no document-level attributes, *docAttributes* can be **nil**. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

When writing data to the pasteboard, you can use the NSData object as the first argument to NSPasteboard's **setData:forType:** method, with a second argument of NSRTFPboardType.

Although this method strips attachments, it leaves the attachment characters in the text itself. NSText's **RTFFromRange:** method, on the other hand, does strip attachment characters when extracting RTF.

See also: – **RTFDFromRange:documentAttributes:**,
– **RTFDFileWrapperFromRange:documentAttributes:**

rulerAttributesInRange:

– (NSDictionary *)**rulerAttributesInRange:**(NSRange)*aRange*

Returns the ruler (paragraph) attributes in effect for the characters within *aRange*. The only ruler attribute currently defined is that named by NSParagraphStyleAttributeName. Use this method to obtain attributes that are to be copied or pasted with "copy ruler" operations. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **fontAttributesInRange:**

size

– (NSSize)**size**

Returns the bounding box of the marks that the receiver draws.

See also: – **drawAtPoint:**, – **drawInRect:**

NSMutableAttributedString

Inherits From: NSAttributedString : NSObject

Declared In: AppKit/NSAttributedString.h
AppKit/NSStringDrawing.h
AppKit/NSTextAttachment.h

Class Description

Additions to the NSMutableAttributedString class primarily involve setting graphical attributes, such as font, super- or subscripting, and alignment, and making these attributes consistent after changes. See the class cluster description for more information.

Method Types

Changing attributes	– applyFontTraits:range: – setAlignment:range: – subscriptRange: – superscriptRange: – unscriptRange:
Updating attachment contents	– updateAttachmentsFromPath:
Fixing attributes after changes	– fixAttributesInRange: – fixAttachmentAttributeInRange: – fixFontAttributeInRange: – fixParagraphAttributeInRange:

Instance Methods

applyFontTraits:range:

– (void)**applyFontTraits:**(NSFontTraitMask)*mask* **range:**(NSRange)*aRange*

Apply the font attributes specified by *mask* to the characters in *aRange*. See the NSFontManager class specification for a description of the font traits available. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **setAlignment:range:**, – **setAttributes:range:**

fixAttachmentAttributeInRange:

– (void)**fixAttachmentAttributeInRange:**(NSRange)*aRange*

Cleans up attachment attributes in *aRange*, removing all attachment attributes assigned to characters other than NSAttachmentCharacter. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **fixFontAttributeInRange:**, – **fixParagraphStyleAttributeInRange:**, – **fixAttributesInRange:**

fixAttributesInRange:

– (void)**fixAttributesInRange:**(NSRange)*aRange*

Invokes the other **fix...** methods, allowing you to clean up an attributed string with a single message. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **fixAttachmentAttributeInRange:**, – **fixFontAttributeInRange:**, – **fixParagraphStyleAttributeInRange:**

fixFontAttributeInRange:

– (void)**fixFontAttributeInRange:**(NSRange)*aRange*

Fixes the font attribute in *aRange*, assigning default fonts to characters with illegal fonts for their scripts and otherwise correcting font attribute assignments. For example, Kanji characters in assigned a Latin font are reassigned an appropriate Kanji font. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **fixParagraphStyleAttributeInRange:**, – **fixAttachmentAttributeInRange:**, – **fixAttributesInRange:**

fixParagraphStyleAttributeInRange:

– (void)**fixParagraphStyleAttributeInRange:**(NSRange)*aRange*

Fixes the paragraph style attributes in *aRange*, assigning the first paragraph style attribute value in each paragraph to all characters of the paragraph. This method extends the range as needed to cover the last paragraph partially contained. A paragraph is delimited by any of these characters, the longest possible sequence being preferred to any shorter:

U+000D (**\r** or CR)	U+2028 (Unicode line separator)
U+000A (**\n** or LF)	U+2029 (Unicode paragraph separator)

\r\n, in that order (also known as CRLF)

Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **fixFontAttributeInRange:**, – **fixAttachmentAttributeInRange:**, – **fixAttributesInRange:**

setAlignment:range:

– (void)**setAlignment:**(NSTextAlignment)*alignment* **range:**(NSRange)*aRange*

Sets the alignment characteristic of the paragraph style attribute for the characters in *aRange* to *alignment*. When attribute fixing takes place, this change will only affect paragraphs whose first character was included in *aRange*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **addAttributes:range:**, – **applyFontTraits:range:**, – **fixParagraphStyleAttributeInRange:**

subscriptRange:

– (void)**subscriptRange:**(NSRange)*aRange*

Decrements the value of the superscript attribute for characters in *aRange* by 1. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **superscriptRange:**, – **unscriptRange:**

superscriptRange:

– (void)**superscriptRange:**(NSRange)*aRange*

Increments the value of the superscript attribute for characters in *aRange* by 1. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **subscriptRange:**, – **unscriptRange:**

unscriptRange:

– (void)**unscriptRange:**(NSRange)*aRange*

Removes the superscript attribute from the characters in *aRange*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **subscriptRange:**, – **superscriptRange:**

updateAttachmentsFromPath:

– (void)**updateAttachmentsFromPath:**(NSString *)*path*

Updates all attachments based on files contained in the RTFD file package at *path*.

See also: – **updateFromPath:** (NSFileWrapper)