
Font Handling

Data Management: Strings, Text, & Fonts



2008-02-08



Apple Inc.
© 1997, 2008 Apple 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, Aqua, Cocoa, Mac, Mac OS, and Objective-C are trademarks of Apple Inc., registered in the United States and other countries.

Helvetica and Times are registered trademarks 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 Font Handling 7

Who Should Read This Document 7

Organization of This Document 7

See Also 8

Creating a Font Object 9

Getting Font Metrics 11

Querying Aqua Font Variations 13

Characters and Glyphs 15

Calculating Glyph Layout 17

Sequential glyph layout 17

Overstruck glyph layout 17

Stacked glyph layout 18

Special Glyphs 19

Recording the Font in a Selection 21

Initiating Font Changes 23

Creating a Font Manager 25

Responding to Font Changes 27

Converting Fonts Manually 29

Examining Fonts 31

Customizing the Font Conversion System 33

Document Revision History 35

Index 37

Figures and Tables

Getting Font Metrics 11

Figure 1	Font metrics	11
Table 1	Font metrics and related NSFont methods	11

Converting Fonts Manually 29

Table 1	Font conversion methods	29
---------	-------------------------	----

Introduction to Font Handling

Font Handling discusses fonts, the font management system, and the user interface to allow the user to interact with available fonts.

Who Should Read This Document

You should read this document if you need to understand how the text system manages font objects, how the font manager works, and how to modify that behavior. If your application allows users to manipulate fonts, for example, especially if it goes beyond default Cocoa behavior, then you should read this document.

To understand the information in this programming topic, you should understand generally the text system's capabilities and architecture, and you should understand basic Cocoa programming conventions.

Organization of This Document

These articles discuss working with NSFont objects:

- [“Creating a Font Object”](#) (page 9) describes the methods you use to create font objects.
- [“Getting Font Metrics”](#) (page 11) describes font metrics and correlates them with the methods you use to retrieve that information.
- [“Querying Aqua Font Variations”](#) (page 13) lists the methods you use to retrieve the standard fonts used in the Aqua user interface.
- [“Characters and Glyphs”](#) (page 15) defines and differentiates characters and glyphs and explains how the text system converts character strings into glyphs for display.
- [“Calculating Glyph Layout”](#) (page 17) describes the sequential, overstruck, and stacked methods of laying out glyphs.
- [“Special Glyphs”](#) (page 19) describes null glyphs and control glyphs.

These articles discuss working with the font manager:

- [“Recording the Font in a Selection”](#) (page 21) explains how an object that enables the user to select fonts should interact with the font manager.
- [“Initiating Font Changes”](#) (page 23) describes the font-changing action methods of the font manager.
- [“Creating a Font Manager”](#) (page 25) explains how to set up a font manager object programmatically.
- [“Responding to Font Changes”](#) (page 27) explains how a text object should respond to a font-changing message from the font manager.

- [“Converting Fonts Manually”](#) (page 29) describes the NSFontManager methods you use to convert the traits and characteristics of a font.
- [“Examining Fonts”](#) (page 31) describes the methods applications can use to retrieve information about font availability and characteristics.
- [“Customizing the Font Conversion System”](#) (page 33) discusses adding custom controls to the Font panel, subclassing NSFontManager and NSFontPanel, and using your own Font menu.

See Also

For further reading, refer to the following documents:

- *Font Panel* explains how the Font panel interacts with the text system.
- *Attributed String Programming Guide* describes NSAttributedString objects, which manage sets of attributes, such as font and kerning, that are associated with character strings or individual characters.
- *Text Layout Programming Guide for Cocoa* describes how the Cocoa text system converts strings of text characters, font information, and page specifications into lines of glyphs placed at specific locations on a page, suitable for display and printing.

For related reference information, see the following documents:

- NSFont
- NSFontManager
- NSFontPanel

Creating a Font Object

You don't create font objects using the `alloc` and `init` methods (or with constructors in Java), instead, you use either `fontWithName:matrix:` or `fontWithName:size:` to look up an available font and alter its size or matrix to your needs. These methods check for an existing font object with the specified characteristics, returning it if there is one. Otherwise, they look up the font data requested and create the appropriate object.

NSFont also defines a number of methods for specifying standard system fonts, such as `systemFontOfSize:`, `userFontOfSize:`, and `messageFontOfSize:`. To request the default size for these standard fonts, pass 0 or a negative number as the font size. The standard system font methods are listed in ["Querying Aqua Font Variations"](#) (page 13).

Getting Font Metrics

NSFont defines a number of methods for accessing a font's metrics information, when that information is available. Methods such as `boundingRectForGlyph:`, `boundingRectForFont`, `xHeight`, and so on, all correspond to standard font metrics information. Figure 1 shows how the font metrics apply to glyph dimensions, and Table 1 lists the method names that correlate with the metrics. See the various method descriptions for more specific information.

Figure 1 Font metrics

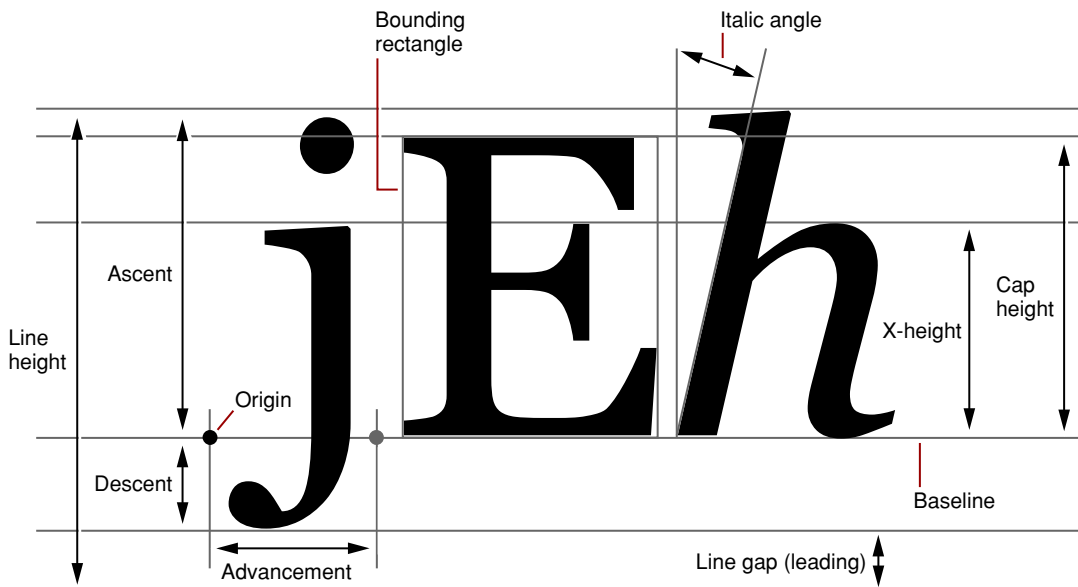


Table 1 Font metrics and related NSFont methods

Font metric	Methods
Advancement	<code>advancementForGlyph:</code> , <code>maximumAdvancement</code>
X-height	<code>xHeight</code>
Ascent	<code>ascender</code>
Bounding rectangle	<code>boundingRectForFont</code> , <code>boundingRectForGlyph:</code>
Cap height	<code>capHeight</code>
Line height	<code>defaultLineHeightForFont</code> , <code>pointSize</code> , <code>labelFontSize</code> , <code>smallSystemFontSize</code> , <code>systemFontSize</code> , <code>systemFontSizeForControlSize:</code>
Descent	<code>descender</code>

Font metric	Methods
Italic angle	<code>italicAngle</code>

Querying Aqua Font Variations

Using the methods of `NSFont`, you can query all of the Aqua font variations. To request the default font size for the standard fonts, you can either explicitly pass in default sizes (obtained from class methods such as `systemFontSize` and `labelFontSize`), or pass in 0 or a negative value.

For Objective-C, use the following method invocations:

Font	Objective-C methods
System font	<code>[NSFont systemFontOfSize:[NSFont systemFontOfSize]]</code>
Emphasized system font	<code>[NSFont boldSystemFontOfSize:[NSFont systemFontOfSize]]</code>
Small system font	<code>[NSFont systemFontOfSize:[NSFont smallSystemFontSize]]</code>
Emphasized small system font	<code>[NSFont boldSystemFontOfSize:[NSFont smallSystemFontSize]]</code>
Mini system font	<code>[NSFont systemFontOfSizeForControlSize: NSMiniControlSize]</code>
Emphasized mini system font	<code>[NSFont boldSystemFontOfSize:[NSFont systemFontOfSizeForControlSize: NSMiniControlSize]]</code>
Application font	<code>[NSFont userFontOfSize:-1.0]</code>
Application fixed-pitch font	<code>[NSFont userFixedPitchFontOfSize:-1.0]</code>
Label Font	<code>[NSFont labelFontOfSize:[NSFont labelFontSize]]</code>

The equivalent Java invocations are as follows:

Font	Java methods
System font	<code>NSFont.systemFontOfSize(-1.0)</code>
Emphasized system font	<code>NSFont.boldSystemFontOfSize(-1.0)</code>
Small system font	<code>NSFont.systemFontOfSize(NSFont.smallSystemFontSize())</code>
Emphasized small system font	<code>NSFont.boldSystemFontOfSize(NSFont.smallSystemFontSize())</code>
Mini system font	<code>NSFont.systemFontSizeForControlSize(-1.0)</code>
Emphasized mini system font	<code>NSFont.boldSystemFontOfSize(NSFont.systemFontSizeForControlSize(-1.0))</code>
Application font	<code>NSFont.userFontOfSize(-1.0)</code>
Application fixed-pitch font	<code>NSFont.userFixedPitchFontOfSize(-1.0)</code>

Font	Java methods
Label font	<code>NSFont.labelFontOfSize(-1.0)</code>

Characters and Glyphs

Characters are conceptual entities that correspond to units of written language. Examples of characters include the letters of the Roman alphabet, the Kanji ideographs used in Japanese, and symbols that indicate mathematical operations. Characters are represented as numbers in a computer's memory or on disk, and a character encoding defines the mapping between a numerical value and a specific character. For example, the ASCII and Unicode character encodings both assign the value 97 (decimal) to the character "a". The Cocoa text system uses the Unicode character encoding internally, although it can read and write other encodings on disk.

You can think of a glyph as the rendered image of a character. The words of this sentence are made visible through glyphs. A collection of glyphs that share certain graphic qualities is called a font.

The difference between a character and a glyph isn't immediately apparent in English since there's typically a one-to-one mapping between the two. But, in some Indic languages, for example, a single character can map to more than one glyph. And, in many languages, two or more characters may be needed to specify a single glyph.

To take a simple example, the glyph "ö" can be the result of two characters, one representing the base character "o" and the other representing the diacritical mark "¨". A user of a word processor can press an arrow key one time to move the insertion point from one side of the "ö" glyph to the other; however, the current position in the character stream must be incremented by two to account for the two characters that make up the single glyph.

Thus, the text system must manage two related but different streams of data: the stream of characters (and their attributes) and the stream of glyphs that are derived from these characters. The `NSTextStorage` object stores the attributed characters, and the `NSLayoutManager` object stores the derived glyphs. Finding the correspondence between these two streams is the responsibility of the layout manager.

For a given glyph, the layout manager can find the corresponding character or characters in the character stream. Similarly, for a given character, the layout manager can locate the associated glyph or glyphs. For example, when a user selects a range of text, the layout manager must determine which range of characters corresponds to the selection.

When characters are deleted, some glyphs may have to be redrawn. For example, if the user deletes the characters "ee" from the word "feel", the "f" and "l" can be represented by the "fl" ligature rather than the two glyphs "f" and "l". The `NSLayoutManager` object directs a glyph generator object to generate new glyphs as needed. Once the glyphs are regenerated, the text must be laid out and displayed. Working with the `NSTextContainer` object and other objects of the text system, the layout manager determines where each glyph appears in the `NSTextView`. Finally, the text view renders the text.

Because an `NSLayoutManager` object is central to the operation of the text system, it also serves as the repository of information shared by various components of the system. For more information about `NSLayoutManager`, refer to its reference documentation and to *Text Layout Programming Guide for Cocoa*.

Calculating Glyph Layout

The Cocoa text system handles many complex aspects of laying out glyphs. If you need to calculate layout for your own purposes, you can use methods defined by `NSFont`. There are three basic kinds of glyph layout, although Java supports only overstruck:

- Sequential, for running text
- Overstruck, for diacritics and other nonspacing marks
- Stacked, for certain non-Western scripts.

Sequential glyph layout

Sequential glyph layout is supported by the method `positionOfGlyph:precededByGlyph:isNominal:`, which is available only in Objective-C. This method calculates the position of a glyph relative to glyph preceding it, using the glyph's width and kerning information if they're available. This is the most straightforward kind of glyph layout.

Overstruck glyph layout

Overstruck glyph layout is the most complex, as it requires detailed information about placement of many kinds of modifying marks. Generally, you have two characters:

- A base glyph, which may be a character such as "a"
- A nonspacing mark, which may be a diacritical mark such as an acute accent (´) or a cedilla (,)

Cocoa gives you methods for combining the two characters, depending on whether the combination is a common one that the font has metrics for or whether the combination is an unusual one that you need to create on the fly. Try these methods in the following order, to get the best result:

- In Objective-C, to see if the font has metrics placing the nonspacing mark directly over the base glyph, use the method `positionOfGlyph:struckOverGlyph:metricsExist:` and check the value returned in the `metricsExist` argument.
- In Objective-C, to see if the font has metrics for placing the nonspacing mark over the base glyph's bounding rectangle, use the method `positionOfGlyph:struckOverRect:metricsExist:` and check the value returned in the `metricsExist` argument. Use the method `boundingRectForGlyph:` to get the bounding rect for the base glyph. Note that `NSFont` always sets `metricsExist` to `NO` and that this method is useful only if you're using a subclass of `NSFont` that overrides this method.

- To place the nonspacing mark over the base glyph in a legible but not necessarily pleasing manner, use the method `positionOfGlyph:forCharacter:struckOverRect:` (`positionOfGlyphForCharacterStruckOverRect` in Java). This method handles all the common nonspacing marks, such as an acute accent, tilde, or cedilla, for Latin script. Use the method `boundingRectForGlyph:` (`boundingRectForGlyph` in Java) to get the bounding rectangle for the base glyph.
- To place a nonspacing mark over a base glyph of another font, also use the method `positionOfGlyph:forCharacter:struckOverRect:` (`positionOfGlyphForCharacterStruckOverRect` in Java). Use the method `boundingRectForGlyph:` (`boundingRectForGlyph` in Java) to get the bounding rectangle for the base glyph.

In Objective-C, if you need to place several nonspacing marks with respect to a base glyph, use the method `positionsForCompositeSequence:numberOfGlyphs:pointerArray:`. This method accepts a C array containing the base glyph followed by all of its nonspacing marks and calculates the positions for as many as of the marks as it can. To place the marks that this method can't handle, use the methods described above.

Stacked glyph layout

Stacked glyph layout is supported by the method `positionOfGlyph:withRelation:toBaseGlyph:totalAdvancement:metricsExist:`, which is available only in Objective-C. Stacked glyphs often have special compressed forms, which standard font metrics don't account for. NSFont's implementation of this method simply abuts the bounding rectangles of the two glyphs for approximate layout of the individual glyphs. Subclasses of NSFont can override this method to access any extra metrics information for more sophisticated layout of stacked glyphs.

Special Glyphs

NSFont defines two special glyphs. `NullGlyph` indicates no glyph at all and is useful in some layout methods for calculating information that isn't relative to another glyph. For example in Objective-C, with `positionOfGlyph:precededByGlyph:isNominal:`, you can specify `NSNullGlyph` as the first argument to get the nominal advancement of the preceding glyph.

The other special glyph is `ControlGlyph`, which the text system maps onto control functions such as linefeed and tab. This glyph has no graphic representation and has no inherent advancement of its own. Instead, the text system examines the control character underlying the glyph to determine what kind of special layout it needs to perform.

Recording the Font in a Selection

Any object that records fonts that the user can change should tell the font manager what the font of its selection is whenever it becomes the first responder and whenever its selection changes while it's the first responder. The object does so by sending the shared font manager a `setSelectedFont` message. It should pass in the first font of the selection, along with a flag indicating whether there's more than one font.

The font manager uses this information to update the Font panel and Font menu to reflect the font in the selection. For example, suppose the font is Helvetica Oblique 12.0 point. In this case, the Font panel selects that font and displays its name; the Font menu changes its Italic command to Unitalic; if there's no Bold variant of Helvetica available, the Bold menu item is disabled; and so on.

If you need to draw text using PostScript operators such as `show`, it's recommended that you set the current font using `NSFont`'s `setFont` method, rather than the PostScript operators `setFont` or `selectfont`. This allows the Application Kit printing mechanism to record the fonts used in the PostScript output. If you absolutely must set the font using a PostScript operator, you can record the font with the Application Kit using the `NSFont` static method `useFont`. See the description of that method for more information.

Initiating Font Changes

The user normally changes the font of the selection by manipulating the Font panel (also called the Fonts window) and the Font menu. These objects initiate the intended change by sending an action message to the font manager. There are four font-changing action methods:

```
addFontTrait
removeFontTrait
modifyFont
modifyFontViaPanel
```

The first three cause the font manager to query the sender of the message in order to determine which trait to add or remove, or how to modify the font. The last causes the font manager to use the settings in the Font panel to modify the font. The font manager records this information and uses it in later requests to convert fonts, as described in [“Responding to Font Changes”](#) (page 27).

When the font manager receives an `addFontTrait` or `removeFontTrait` message, it queries the sender with a `tag` message, interpreting the return value as a trait mask for use with `convertFontToHaveTrait` or `convertFontToNotHaveTrait`, as described in [“Converting Fonts Manually”](#) (page 29). The Font menu commands *Italic* and *Bold*, for example, have trait mask values of `ItalicMask` and `BoldMask`, respectively. See the section “Constants” in the `NSFontManager` reference documentation for a list of trait mask values.

When the font manager receives a `modifyFont` message, it queries the sender with a `tag` message and interprets the return value as a particular kind of conversion to perform, via the various conversion methods described in [“Converting Fonts Manually”](#) (page 29). For example, a button whose tag value is `SizeUpFontAction` causes the font manager’s `convertFont` method to increase the size of the `NSFont` passed as the argument. See the `NSFontManager` method `modifyFont` for a list of conversion tag values.

For `modifyFontViaPanel`, the font manager sends the application’s Font panel a `panelConvertFont` message. The Font panel in turn uses the font manager to convert the font provided according to the user’s choices. For example, if the user selects only the font family in the Font panel (perhaps Helvetica), then for whatever fonts are provided to `panelConvertFont`, only the family is changed: Courier Medium 10.0 point becomes Helvetica Medium 10.0 point, and Times Italic 12.0 point becomes Helvetica Oblique 12.0 point.

Creating a Font Manager

You normally set up a font manager and the Font menu using Interface Builder. However, you can also do so programmatically by getting the shared font manager instance and having it create the standard Font menu at runtime, as in this Objective-C example:

```
NSFontManager *fontManager = [NSFontManager sharedFontManager];  
NSMenu *fontMenu = [fontManager fontMenu:YES];
```

You can then add the Font menu to your application's menus. Once the Font menu is installed, your application automatically gains the functionality of both the Font menu and the Font panel.

Responding to Font Changes

The font manager responds to a font-changing action method by sending a `changeFont` action message up the responder chain. A text-bearing object that receives this message should have the font manager convert the fonts in its selection by invoking `convertFont` for each font and using the `NSFont` object returned. The `convertFont` method uses the information recorded by the font-changing action method, such as `addFontTrait`, modifying the font provided appropriately. (There's no way to explicitly set the font-changing action or trait; instead, you use the methods described in ["Converting Fonts Manually"](#) (page 29).)

This simple Objective-C example assumes there's only one font in the selection:

```
- (void)changeFont:(id)sender
{
    NSFont *oldFont = [self selectionFont];
    NSFont *newFont = [sender convertFont:oldFont];
    [self setSelectionFont:newFont];
    return;
}
```

Most text-bearing objects have to scan the selection for ranges with different fonts and invoke `convertFont` for each one.

Converting Fonts Manually

NSFontManager defines a number of methods for explicitly converting particular traits and characteristics of a font. Table 1 lists these conversion methods.

Table 1 Font conversion methods

Objective-C Methods	Java Methods
<code>convertFont:toFace:</code>	<code>convertFontToFace</code>
<code>convertFont:toFamily:</code>	<code>convertFontToFamily</code>
<code>convertFont:toHaveTrait:</code>	<code>convertFontToFamily</code>
<code>convertFont:toNotHaveTrait:</code>	<code>convertFontToNotHaveTrait</code>
<code>convertFont:toSize:</code>	<code>convertFontToSize</code>
<code>convertWeight:ofFont:</code>	<code>convertWeight</code>

Each method returns a transformed version of the font provided, or the original font if it can't be converted. `convertFont:toFace:` and `convertFont:toFamily:` both alter the basic design of the font provided. The first method requires a fully specified typeface name, such as "Times-Roman" or "Helvetica-BoldOblique", while the second expects only a family name, such as "Times" or "Helvetica".

The `convertFont:toHaveTrait:` and `convertFont:toNotHaveTrait:` methods use trait masks to add or remove a single trait such as Italic, Bold, Condensed, or Extended.

The `convertFont:toSize:` method returns a font of the requested size, with all other characteristics the same as those of the original font.

The `convertWeight:ofFont:` method either increases or decreases the weight of the font provided, according to a Boolean flag. Font weights are typically indicated by a series of names, which can vary from font to font. Some go from Light to Medium to Bold, while others have Book, SemiBold, Bold, and Black. This method offers a uniform way of incrementing and decrementing any font's weight.

The default implementation of font conversion is very conservative, making a change only if no other trait or aspect is affected. For example, if you try to convert Helvetica Oblique 12.0 point by adding the Bold trait, and only Helvetica Bold is available, the font isn't converted. You can create a subclass of NSFontManager and override the conversion methods to perform less conservative conversion, perhaps using Helvetica Bold in this case and losing the Oblique trait.

In addition to the font-conversion methods, NSFontManager defines `fontWithFamily:traits:weight:size:` to construct a font with a given set of characteristics. If you don't care to make a subclass of NSFontManager, you can use this method to perform approximate font conversions yourself.

Examining Fonts

In addition to converting fonts, `NSFontManager` provides information on which fonts are available to the application and on the characteristics of any given font. The `availableFonts` method returns an array of the names of all fonts available. The `availableFontNamesWithTraits:` method filters the available fonts based on a font trait mask.

There are three methods for examining individual fonts. The `fontName:HasTraits:` method returns `true` if the font matches the trait mask provided. The `traitsOfFont:` method returns a trait mask for a given font. The `weightOfFont:` method returns an approximate ranking of a font's weight on a scale of 0–15, where 0 is the lightest possible weight, 5 is Normal or Book weight, 9 is the equivalent of Bold, and 15 is the heaviest possible (often called Black or Ultra Black).

Customizing the Font Conversion System

In Objective-C only, if you need to customize the font conversion system by creating subclasses of `NSFontManager` or `NSFontPanel`, you must inform the `NSFontManager` class of this change with a `setFontManagerFactory:` or `setFontPanelFactory:` message, before either the shared font manager or shared Font panel is created. These methods record your class as the one to instantiate the first time the font manager or Font panel is requested. You may be able to avoid using subclasses if all you need is to add some custom controls to the Font panel. In this case, you can invoke the `NSFontPanel` `setAccessoryView:` method to add an `NSView` below its font browser.

In Java, to add some custom controls to the Font panel, invoke `NSFontPanel`'s `setAccessoryView` to add an `NSView` below its font browser.

If you provide your own Font menu, you should register it with the font manager using the `setFontMenu` method. The font manager is responsible for validating Font menu items and changing their titles and tags according to the font of the selection. For example, when the selected font is Italic, the font manager changes the Italic Font menu item to Unitalic and changes its tag to `UnitalicMask`. Your Font menu's items should use the appropriate action methods and tags. Here are some examples:

Font menu item	Action	Tag
Italic	<code>addFontTrait</code>	<code>ItalicMask</code>
Bold	<code>addFontTrait</code>	<code>BoldMask</code>
Heavier	<code>modifyFont</code>	<code>HeavierFontAction</code>
Larger	<code>modifyFont</code>	<code>SizeUpFontAction</code>

Document Revision History

This table describes the changes to *Font Handling*.

Date	Notes
2008-02-08	Corrected typo in "Responding to Font Changes" code example.
2004-08-31	Added " Characters and Glyphs " (page 15) article, removed "Drawing Text With NSFonts" article, added references to related documents to introduction, and made editorial revisions throughout.
2004-02-12	Rewrote introduction and added an index.
2003-03-26	Fixed method names (Java and Objective-C names ran together), rewrote introduction, and made minor edits.
2002-11-12	Revision history added to existing document.

Index

A

`addFontTrait` method [23, 27](#)
`advancement of glyphs` [11, 19](#)
`advancementForGlyph:` method [11](#)
Aqua standard fonts [9, 13](#)
`ascender` method [11](#)
ascent of fonts [11](#)
`availableFontNamesWithTraits:` method [31](#)
`availableFonts` method [31](#)

B

base glyphs [17](#)
`boldSystemFontOfSize` method (Java) [13](#)
`boldSystemFontOfSize:` method [13](#)
bounding rectangle of glyphs [11, 18](#)
`boundingRectForFont` method [11](#)
`boundingRectForGlyph:` method [11, 17, 18](#)

C

cap height of fonts [11](#)
`capHeight` method [11](#)
`changeFont` method [27](#)
character encodings
 defined [15](#)
characters
 defined [15](#)
control glyphs [19](#)
`convertFont` method [23, 27](#)
`convertFont:toFace:` method [29](#)
`convertFont:toFamily:` method [29](#)
`convertFont:toHaveTrait:` method [29](#)
`convertFont:toNotHaveTrait:` method [29](#)
`convertFont:toSize:` method [29](#)
`convertFontToFace` method [29](#)
`convertFontToFamily` method [29](#)

`convertFontToHaveTrait` method [23](#)
`convertFontToNotHaveTrait` method [23, 29](#)
`convertFontToSize` method [29](#)
`convertWeight` method [29](#)
`convertWeight:ofFont:` method [29](#)

D

`defaultLineHeightForFont` method [11](#)
`descender` method [11](#)
descent of fonts [11](#)
diacritical marks [17](#)

F

first responder
 font manager and [21](#)
font manager
 creating [25](#)
 font changes and [23, 27](#)
 recording fonts [21](#)
 subclassing [33](#)
Font menu [21, 25, 33](#)
font metrics [11, 17](#)
font objects [9](#)
Font panel
 changing fonts and [23](#)
 relation to Font menu [25](#)
 updating [21](#)
`fontNamed:HasTraits:` method [31](#)
fonts
 changing [23, 27](#)
 converting traits [29, 33](#)
 defined [15](#)
 examining [31](#)
 metrics of [11](#)
 recording [21, 23](#)
 standard system [9, 13](#)
`fontWithFamily:traits:weight:size:` method [29](#)

fontWithName:matrix: [method 9](#)
fontWithName:size: [method 9](#)

G

glyph layout
 overstruck [17](#)
 sequential [17](#)
 stacked [18](#)
glyphs
 control [19](#)
 defined [15](#)
 drawing of [15](#)
 null [19](#)

I

Interface Builder
 to set up font manager [25](#)
italic angle of fonts [12](#)
italicAngle [method 12](#)

L

LabelFontOfSize [method \(Java\) 14](#)
LabelFontOfSize: [method 13](#)
LabelFontSize [method 11](#)
ligatures [15](#)
line height of fonts [11](#)

M

maximumAdvancement [method 11](#)
messageFontOfSize: [method 9](#)
metrics
 font [11, 17](#)
modifyFont [method 23](#)
modifyFontViaPanel [method 23](#)

N

nonspacing marks [17](#)
NSFont class [13, 17](#)
NSFontManager class [29, 31](#)
NSLayoutManager class [15](#)

NSTextStorage class [15](#)
null glyphs [19](#)

O

overstruck glyph layout [17](#)

P

panelConvertFont [method 23](#)
pointSize [method 11](#)
positionOfGlyph:forCharacter:struckOverRect:
 [method 18](#)
positionOfGlyph:precededByGlyph:isNominal:
 [method 17, 19](#)
positionOfGlyph:struckOverGlyph:metricsExist:
 [method 17](#)
positionOfGlyph:struckOverRect:metricsExist:
 [method 17](#)
positionOfGlyph:withRelation:toBaseGlyph:
 totalAdvancement:metricsExist: [method 18](#)
positionsForCompositeSequence:numberOfGlyphs:
 pointArray: [method 18](#)
PostScript operators
 drawing text using [21](#)

R

removeFontTrait [method 23](#)

S

sequential glyph layout [17](#)
setAccessoryView: [method 33](#)
setFontManagerFactory: [method 33](#)
setFontMenu [method 33](#)
setFontPanelFactory: [method 33](#)
setSelectedFont [method 21](#)
smallSystemFontSize [method 11](#)
smallSystemFontSize [method \(Java\) 13](#)
stacked glyph layout [18](#)
system fonts [9, 13](#)
systemFontOfSize [method \(Java\) 13](#)
systemFontOfSize: [method 9, 13](#)
systemFontSize [method 11](#)
systemFontSizeForControlSize [method \(Java\) 13](#)
systemFontSizeForControlSize: [method 11, 13](#)

T

traitsOfFont: [method 31](#)

U

Unicode

as used in the Cocoa text system [15](#)

useFont [method 21](#)

userFixedPitchFontOfSize [method \(Java\) 13](#)

userFixedPitchFontOfSize: [method 13](#)

userFontOfSize [method \(Java\) 13](#)

userFontOfSize: [method 9, 13](#)

W

weightOfFont: [method 31](#)

X

x-height [11](#)

xHeight [method 11](#)