NSFont

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	AppKit/NSFont.h

Class Description

NSFont objects represent PostScript fonts to an application, providing access to characteristics of the font and assistance in laying out glyphs relative to one another. Font objects are also used to establish the current font when drawing in an NSView, using the **set** method.

You don't create Font objects using the **alloc** and **init** methods. Instead, you use one of the **fontWithName:...** methods 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 getting standard system fonts, such as **systemFontOfSize:**, **userFontOfSize:**, and **messageFontOfSize:**.

Drawing Text with NSFonts

In most cases you draw text using an NSTextView object. You can also draw an NSString directly in an NSView using the methods **drawAtPoint:withAttributes:** and **drawInRect:withAttributes:**, which the Application Kit adds to NSString. These methods take an NSDictionary of attributes, as used by the NSAttributedString class, and apply them when drawing the string.

If you need to draw text using PostScript operators such as **show**, it's recommended that you set the current font using NSFont's **set** 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 class method **useFont:**. See the description of that method for more information.

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. See the various method descriptions for specific information. You can also get a complete dictionary of font metrics using the **afmDictionary** method, or retrieve the original contents of the metrics file using **afmFileContents**.

Calculating Glyph Layout

The OPENSTEP extended text system handles many complex aspects of laying glyphs out. If you need to calculate layout for your own purposes, you can use several methods defined by NSFont. There are three basic kinds of glyph layout: sequential, for running text; overstruck, for diacritics and other non-spacing marks; and stacked, for scripts such as Indic and Tibetan.

The first kind of layout is supported by the method **positionOfGlyph:precededByGlyph:isNominal:**. This method calculates the position of a glyph relative to glyph preceding it, using font metrics if they're available. This is the most straightforward kind of glyph layout.

Layout of overstruck glyphs is potentially the most complex, as it requires detailed information about placement of many kinds of modifying marks. When you know both the base glyph and the non-spacing mark being struck over it, you first use the method **positionOfGlyph:struckOverGlyph:metricsExist:** to calculate the location of the non-spacing mark. This method also indicates whether metrics are available for the two glyphs. If they're not, you must fall back to calculating the layout based on the bounding rectangle of the base glyph (which you can get using **boundingRectForGlyph:**). There are two methods for doing this. The first, **positionOfGlyph:struckOverRect:metricsExist:**, attempts to use metrics to place the non-spacing mark in a reasonable fashion relative to the base glyph's bounding box. If this method indicates that metrics aren't available (which NSFont's implementation always does, because current fonts don't provide for metrics relative to bare rectangles), you must resort to the last method, **positionOfGlyph:forCharacter:struckOverRect:**. This method attempts to place the non-spacing mark in a legible if not pleasing manner, treating it as a well-known character such as an acute accent, tilde, or other modifier.

An additional method for laying out a series of overstruck glyphs is **positionsForCompositeSequence:numberOfGlyphs:pointArray:**. This method accepts a C array containing the base glyph followed by all of its non-spacing marks, and calculates the positions for as many as it can. For those that it can't determine, you must resort to the individual methods described immediately above.

The final kind of layout, of stacked glyphs, is supported by the method **positionOfGlyph:withRelation:toBaseGlyph:totalAdvancement:metricsExist:**. 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 boxes of the two glyphs for approximate layout of the individual glyphs. Subclasses of NSFont can override this method to access any extra metrics information in order to provide more sophisticated layout of stacked glyphs.

Special Glyphs

NSFont defines two special glyphs. NSNullGlyph indicates no glyph at all, and is useful in some layout methods for calculating information that isn't relative to another glyph. For example, with

positionOfGlyph:precededByGlyph:isNominal:, you can specify NSNullGlyph as the first glyph to get the nominal advancement of the preceding glyph.

The other special glyph is NSControlGlyph, which the text system maps onto control characters 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.

Adopted Protocols

NSCoding	– encodeWithCoder: – initWithCoder:
NSCopying	– copyWithZone:

Method Types

Creating arbitrary fonts	+ fontWithName:size: + fontWithName:matrix:
Creating user fonts	+ userFontOfSize: + userFixedPitchFontOfSize:
Creating system fonts	+ boldSystemFontOfSize: + menuFontOfSize: + messageFontOfSize: + paletteFontOfSize: + systemFontOfSize: + titleBarFontOfSize: + toolTipsFontOfSize:
Getting preferred fonts	+ setPreferredFontNames: + preferredFontNames
Using a font to draw	– set
Adding fonts to print operations	+ useFont:
Getting general font information	– encodingScheme – isBaseFont – isFixedPitch – mostCompatibleStringEncoding
Getting information about glyphs	– glyphIsEncoded: – glyphWithName:

Getting metrics information	– advancementForGlyph: – afmDictionary – afmFileContents – ascender – boundingRectForFont – boundingRectForGlyph: – capHeight – descender – italicAngle – matrix – maximumAdvancement – pointSize – underlinePosition – underlineThickness – widthOfString: – widths – xHeight
Getting font names	– displayName – familyName – fontName
Laying out sequential glyphs	– positionOfGlyph:precededByGlyph:isNominal: – positionsForCompositeSequence:numberOfGlyphs:pointArray:
Laying out overstruck glyphs	– positionOfGlyph:forCharacter:struckOverRect: – positionOfGlyph:struckOverGlyph:metricsExist: – positionOfGlyph:struckOverRect:metricsExist:
Laying out stacked glyphs	– positionOfGlyph:withRelation:toBaseGlyph:totalAdvancement: metricsExist:
Setting user fonts	+ setUserFont: + setUserFixedPitchFont:
Getting corresponding device fonts	– printerFont – screenFont

Class Methods

boldSystemFontOfSize:

+ (NSFont *)**boldSystemFontOfSize:**(float)*fontSize*

Returns the font used for standard interface items that are rendered in boldface type, in *fontSize*. This is equivalent to **titleBarFontOfSize:**.

See also: + **fontWithName:size:**

fontWithName:matrix:

+ (NSFont *)**fontWithName:**(NSString *)*typeface* **matrix:**(const float *)*fontMatrix*

Returns a font object for *typeface* and *fontMatrix*. A typeface is a fully specified family-face name, such as Helvetica-BoldOblique or Times-Roman (not a name as shown in the Font Panel). *fontMatrix* is a standard 6-element transformation matrix as used in the PostScript language, specifically with the **makefont** operator. In most cases, you can simply use **fontWithName:size:** to create standard scaled fonts.

You can use the defined value NSFontIdentityMatrix for [1 0 0 1 0 0]. Fonts created with a matrix other than NSFontIdentityMatrix *don't* automatically flip themselves in flipped views.

See also: – **isFlipped** (NSView)

fontWithName:size:

+ (NSFont *)**fontWithName:**(NSString *)*fontName* **size:**(float)*fontSize*

Returns a font object for *typeface* and *fontSize*. A typeface is a fully specified family-face name, such as Helvetica-BoldOblique or Times-Roman. *fontSize* is used to scale the font, and is equivalent to using a font matrix of [*fontSize* 0 0 *fontSize* 0 0] with **fontWithName:matrix:**. Fonts created with this method automatically flip themselves in flipped views. This method is the preferred means for creating fonts.

menuFontOfSize:

+ (NSFont *)**menuFontOfSize:**(float)*fontSize*

Returns the font used for menu items in *fontSize*.

See also: + **fontWithName:size:**

messageFontOfSize:

+ (NSFont *)**messageFontOfSize:**(float)*fontSize*

Returns the font used for standard interface items, such as button labels, menu items, and so on, in *fontSize*. This is equivalent to **systemFontOfSize:**.

See also: + **fontWithName:size:**

paletteFontOfSize:

+ (NSFont *)**paletteFontOfSize:**(float)*fontSize*

Returns the font used for palette window title bars.

See also: + **fontWithName:size:**, + **titleBarFontOfSize:**

preferredFontNames

+ (NSArray *)**preferredFontNames**

Returns the names of fonts that the Application Kit tries first when a character has no font specified, or when the font specified doesn’t have a glyph for that character. If none of these fonts provides a glyph, the remaining fonts on the system are searched for a glyph.

See also: + **setPreferredFontNames:**, – **glyphIsEncoded:**

setPreferredFontNames:

+ (void)**setPreferredFontNames:**(NSArray *)*fontNames*

Sets the list of preferred font names to *fontNames*, and records them in the user defaults database for all applications. The Application Kit tries these fonts first when a character has no font specified, or when the font specified doesn’t have a glyph for that character. If none of these fonts provides a glyph, the remaining fonts on the system are searched for a glyph.

This method is useful for optimizing glyph rendering for uncommon scripts, by guaranteeing that appropriate fonts are searched first.

See also: + **preferredFontNames**

setUserFixedPitchFont:

+ (void)**setUserFixedPitchFont:**(NSFont *)*aFont*

Sets the font used by default for documents and other text under the user's control, when that font should be fixed-pitch, to *aFont*, and records the font in the user defaults database for all applications.

See also: + **setUserFont:**, + **userFixedPitchFontOfSize:**

setUserFont:

+ (void)**setUserFont:**(NSFont *)*aFont*

Sets the font used by default for documents and other text under the user's control to *aFont*, and records the font in the user defaults database for all applications.

See also: + **setUserFixedPitchFont:**, + **userFontOfSize:**

systemFontOfSize:

+ (NSFont *)**systemFontOfSize:**(float)*fontSize*

Returns the font used for standard interface items, such as button labels, menu items, and so on, in *fontSize*. This is equivalent to **messageFontOfSize:**.

See also: + **boldSystemFontOfSize:**, + **userFontOfSize:**, + **userFixedPitchFontOfSize:**, + **fontWithName:size:**

titleBarFontOfSize:

+ (NSFont *)**titleBarFontOfSize:**(float)*fontSize*

Returns the font used for window title bars, in *fontSize*. This is equivalent to **boldSystemFontOfSize:**.

See also: + **paletteFontOfSize:**

toolTipsFontOfSize:

+ (NSFont *)**toolTipsFontOfSize:**(float)*fontSize*

Returns the font used for tool-tips labels, in *fontSize*.

See also: + **fontWithName:size:**

useFont:

+ (void)**useFont:**(NSString *)*fontName*

Records *fontName* as one used in the current print operation.

The NSFont class object keeps track of the fonts used in an NSView by recording each one that receives a **set** message. When the view is called upon to generate conforming PostScript language output (such as during printing), the NSFont class provides the list of fonts required for the **%%DocumentFonts** comment, as required by Adobe's Document Structuring Conventions.

useFont: augments this system by providing a way to register fonts that are included in the document but not set using NSFont's **set** method. For example, you might set a font by executing the **setfont** operator within a function created by the **pswrap** utility. In such a case, be sure to pair the use of the font with a **useFont:** message to register the font for listing in the document comments.

userFixedPitchFontOfSize:

+ (NSFont *)**userFixedPitchFontOfSize:**(float)*fontSize*

Returns the font used by default for documents and other text under the user's control (that is, text whose font the user can normally change), when that font should be fixed-pitch.

See also: + **userFontOfSize:**, + **fontWithName:size:**, + **setUserFixedPitchFont:**

userFontOfSize:

+ (NSFont *)**userFontOfSize:**(float)*fontSize*

Returns the font used by default for documents and other text under the user's control (that is, text whose font the user can normally change).

See also: + **userFixedPitchFontOfSize:**, + **fontWithName:size:**, + **setUserFont:**

Instance Methods

advancementForGlyph:

– (NSSize)**advancementForGlyph:**(NSGlyph)*aGlyph*

Returns the nominal spacing for *aGlyph*—the distance that the current point moves after showing the glyph—accounting for the receiving font's size. This spacing is given according to the glyph's movement direction, which is either strictly horizontal or strictly vertical.

See also: – **boundingRectForGlyph**, – **maximumAdvancement**

afmDictionary

– (NSDictionary *)**afmDictionary**

Returns the receiving font’s AFM information in dictionary form. It contains the following information under these keys, with all values as strings:

NSAFMFamilyName	NSAFMCapHeight
NSAFMFontName	NSAFMXHeight
NSAFMFormatVersion	NSAFMAscender
NSAFMFullName	NSAFMDescender
NSAFMNotice	NSAFMUnderlinePosition
NSAFMVersion	NSAFMUnderlineThickness
NSAFMWeight	NSAFMItalicAngle
NSAFMEncodingScheme	NSAFMMappingScheme
NSAFMCharacterSet	

For any other items, use the AFM file contents, as returned by **afmFileContents**.

afmFileContents

– (NSString *)**afmFileContents**

Returns the receiving font’s AFM file as a string object.

ascender

– (float)**ascender**

Returns the top y coordinate of the receiving font’s longest ascender.

See also: – **descender**, – **capHeight**, – **xHeight**

boundingRectForFont

– (NSRect)**boundingRectForFont**

Returns the receiving font’s bounding rectangle, scaled to the font’s size. The bounding rectangle is the union of the bounding rectangles of every glyph in the font.

See also: – **boundingRectForGlyph:**

boundingRectForGlyph:

– (NSRect)**boundingRectForGlyph:**(NSGlyph)*aGlyph*

Returns the bounding rectangle for *aGlyph*, scaled to the receiving font's size.

See also: – **boundingRectForFont**

capHeight

– (float)**capHeight**

Returns the receiving font's cap height.

See also: – **ascender**, – **descender**, – **xHeight**

descender

– (float)**descender**

Returns the bottom y coordinate of the receiving font's longest descender.

displayName

– (NSString *)**displayName**

Returns the name used to represent the receiving font in the user interface, typically localized for the user's language.

encodingScheme

– (NSString *)**encodingScheme**

Returns the name of the receiving font's encoding scheme, such as "AdobeStandardEncoding", "ISOLatin1Encoding", "FontSpecific", and so on.

familyName

– (NSString *)**familyName**

Returns the receiving font's family name; for example, "Times" or "Helvetica".

See also: – **fontName**

fontName

– (NSString *)**fontName**

Returns the receiver’s full font name, as used in PostScript language code; for example, “Times-Roman” or “Helvetica-Oblique”.

See also: – **familyName**

glyphIsEncoded:

– (BOOL)**glyphIsEncoded:**(NSGlyph)*aGlyph*

Returns YES if the receiving font encodes *aGlyph*, NO if it doesn’t contain it.

glyphWithName:

– (NSGlyph)**glyphWithName:**(NSString *)*glyphName*

Returns the encoded glyph named *glyphName*, or –1 if the receiving font contains no such glyph. Also returns –1 if the glyph named *glyphName* isn’t encoded.

isBaseFont

– (BOOL)**isBaseFont**

Returns YES if the receiver is a PostScript base font, NO if it’s a PostScript composite font composed of other base fonts.

isFixedPitch

– (BOOL)**isFixedPitch**

Returns YES if all glyphs in the receiving font have the same advancement, NO if any advancements differ.

See also: – **advancementForGlyph:**

italicAngle

– (float)**italicAngle**

Returns the receiving font’s italic angle, as read from its AFM file.

matrix

– (const float *)**matrix**

Returns the receiver’s font matrix, a standard 6-element transformation matrix as used in the PostScript language, specifically with the **makefont** operator. In most cases, with a font of *fontSize*, this matrix is [*fontSize* 0 0 *fontSize* 0 0].

See also: + **fontWithName:matrix:**

maximumAdvancement

– (NSSize)**maximumAdvancement**

Returns the greatest advancement of any of the receiving font’s glyphs. This is always either strictly horizontal or strictly vertical.

See also: – **advancementForGlyph:**

mostCompatibleStringEncoding

– (NSStringEncoding)**mostCompatibleStringEncoding**

Returns the string encoding that works best with the receiving font, where there are the fewest possible unmatched characters in the string encoding and glyphs in the font. You can use NSString’s **dataUsingEncoding:** or **dataUsingEncoding:allowLossyConversion:** method to convert the string to this encoding.

Note: This method works heuristically using well-known font encodings, so for nonstandard encodings it may not in fact return the optimal string encoding.

See also: – **widthOfString:**

pointSize

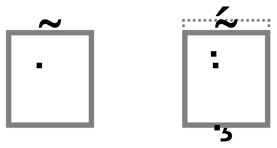
– (float)**pointSize**

Returns the receiving font’s point size, or the vertical point size for a font with a nonstandard matrix.

positionOfGlyph:forCharacter:struckOverRect:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
forCharacter:(unichar)*aChar*
struckOverRect:(NSRect)*aRect*

Calculates and returns a suitable location for *aGlyph* to be drawn as a diacritic or non-spacing mark relative to *aRect*, assuming that *aGlyph* represents *aChar*. Returns NSZeroPoint if the location can't be calculated. The nature of *aChar* as one appearing above or below its base character determines the location returned. For example, in the first figure below, the gray tilde and box represent *aGlyph* and *aRect*, and the black dot is the point returned (defined relative to the origin of the *aRect*).



To place multiple glyphs over a rectangle, work from the innermost ones to the outermost. As you calculate the position of each glyph, enlarge the rectangle to include the bounding rectangle of the glyph in preparation for the next glyph. The second figure shows a tilde, acute accent, and cedilla all placed over a rectangle, with the acute accent placed relative to bounding box of the base rectangle and the tilde.

This method is the last fallback mechanism for performining minimally legible typography when metrics aren't available, and should be used when **positionOfGlyph:struckOverGlyph:metricsExist:** indicates that metrics don't exist for the base glyph specified. It can account for the layout and placement of most Latin, Greek, and Cyrillic non-spacing marks. You should draw the glyph at the returned location, even if it's NSZeroRect.

positionOfGlyph:precededByGlyph:isNominal:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
precededByGlyph:(NSGlyph)*prevGlyph*
isNominal:(BOOL *)*flag*

Calculates and returns the location of *aGlyph* relative to *prevGlyph*, assuming that *prevGlyph* precedes it in the layout (*not* necessarily in the character stream). The point returned should be used relative to whatever location is used for *prevGlyph*. If *flag* is non-NULL, it's filled with NO if font metrics are available and were used in the calculation, YES if the default spacing for the glyphs was used.

Returns NSZeroPoint is either *aGlyph* or *prevGlyph* is NSControlGlyph or is invalid. Returns the nominal advancement of *prevGlyph* if *aGlyph* is NSNullGlyph.

This method is useful for sequential glyph placement when glyphs aren't drawn with a single PostScript operation.

positionOfGlyph:struckOverGlyph:metricsExist:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
  struckOverGlyph:(NSGlyph)*baseGlyph*
  metricsExist:(BOOL *)*flag*

Calculates and returns a suitable location for *aGlyph* to be drawn as a diacritic or non-spacing mark relative to *baseGlyph*. The point returned should be used relative to whatever location is used for *baseGlyph*. If *flag* is non-NULL it's filled with YES if font metrics are available, NO if they're not. If *flag* is returned as NO, the result isn't valid and shouldn't be used. In that case, use **positionOfGlyph:struckOverRect:metricsExist:** or **positionOfGlyph:forCharacter:struckOverRect:** to calculate a reasonable offset.

See also: – **positionsForCompositeSequence:numberOfGlyphs:pointArray:**,
– **positionOfGlyph:structOverRect:metricsExist:**

positionOfGlyph:struckOverRect:metricsExist:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
  struckOverRect:(NSRect)*aRect*
  metricsExist:(BOOL *)*flag*

Overridden by subclasses to calculate and return a suitable location for *aGlyph* to be drawn as a diacritic or non-spacing mark relative to *aRect*, provided metrics exist. Returns NSZeroRect if the location can't be determined. If *flag* is non-NULL it's filled with YES if font metrics are available, NO if they're not. If *flag* is returned as NO, the result isn't valid and shouldn't be used. In that case, use **positionOfGlyph:forCharacter:struckOverRect:** to calculate a reasonable offset.

Because current font metrics don't include support for generic placement relative to rectangles, NSFont's implementation of this method always returns NSZeroPoint and returns *flag* as NO.

positionOfGlyph:withRelation:toBaseGlyph:totalAdvancement:metricsExist:

– (NSPoint)**positionOfGlyph:**(NSGlyph)*aGlyph*
  withRelation:(NSGlyphRelation)*relation*
  toBaseGlyph:(NSGlyph)*baseGlyph*
  totalAdvancement:(NSSize *)*offset*
  metricsExist:(BOOL *)*flag*

Calculates and returns a suitable location for *aGlyph* to be drawn relative to *baseGlyph*, where *relation* is NSGlyphBelow or NSGlyphAbove. The point returned should be used relative to whatever location is used for *baseGlyph*. This method is useful for calculating the layout of stacked glyphs, such as those in Indic and Tibetan scripts.

If *offset* is non-NULL, it’s filled with the larger of the two glyphs’ advancements, allowing for reasonable layout of following glyphs.

If *flag* is non-NULL it’s filled with YES if font metrics are available, NO if they’re not. If metrics aren’t available, the location is calculated as a simple stacking with no gap between *baseGlyph* and *aGlyph*.

Note: *This method only supports horizontally laid-out base glyp*hs.

positionsForCompositeSequence:numberOfGlyphs:pointArray:

– (int)**positionsForCompositeSequence:**(NSGlyph *)*glyphs*
numberOfGlyphs:(int)*numGlyphs*
pointArray:(NSPoint *)*points*

Calculates and fills *points* with the locations for *glyphs*, assuming that the first glyph is a base character and those following are non-spacing marks. These points should all be interpreted as relative to the location of the first glyph in *glyphs*. Returns the number of points that could be calculated.

If the number of points calculated is less than *numGlyphs*, the number of glyphs provided, you can use **positionOfGlyph:structOverRect:metricsExist:** to determine the positions for the remaining glyphs. When using that method, calculate the base rectangle for each glyph from the bounding rectangles and positions of all preceding glyphs.

printerFont

– (NSFont *)**printerFont**

When sent to a font object representing a scalable PostScript font, returns **self**. When sent to a font object representing a bitmapped screen font, returns its corresponding scalable PostScript font.

See also: – **screenFont**

screenFont

– (NSFont *)**screenFont**

When sent to a font object representing a bitmapped screen font, returns **self**. When sent to a font object representing a scalable PostScript font, returns a bitmapped screen font matching the receiver in typeface and matrix (or size), or **nil** if such a font can’t be found.

Note: Screen fonts are for direct use with the Window Server only. Never use them with Application Kit objects, such as in **setFont:** methods.

See also: – **printerFont**

set

– (void)**set**

Establishes the receiving font as the current font for PostScript **show** and other text-drawing operators. During a print operation, also records the font as used in the PostScript code emitted.

See also: + **useFont:**

underlinePosition

– (float)**underlinePosition**

Returns the baseline offset that should be used when drawing underlines with the receiving font, as determined by the font’s AFM file. This value is usually negative, which must be considered when drawing in a flipped coordinate system.

See also: – **underlineThickness**

underlineThickness

– (float)**underlineThickness**

Returns the thickness that should be used when drawing underlines with the receiving font, as determined by the font’s AFM file.

See also: – **underlinePosition**

widthOfString:

– (float)**widthOfString:**(NSString *)*aString*

Returns the *x* axis offset of the current point when *aString* is drawn with a PostScript **show** operator in the receiving font. This method performs lossy conversion of *aString* to the most compatible encoding for the receiving font.

Use this method only when you’re sure all of *aString* can be rendered with the receiving font. In general, it’s better to use the Application Kit’s string-drawing methods, as described under NSString Additions.

See also: – **mostCompatibleStringEncoding**

widths

– (float *)**widths**

Returns a C array of 256 **float**s, giving the unscaled width of each glyph in the font. This data is useful only for simple fonts without non-spacing marks, and doesn’t account for Unicode-related issues at all. You should avoid using this method wherever possible; use **advancementForGlyph:** instead.

xHeight

– (float)**xHeight**

Returns the x-height of the receiving font.

See also: – **ascender**, – **descender**