

# Rich Text, Core Text

Rob Napier

[robnapier.net/cocoaconf](http://robnapier.net/cocoaconf)

Welcome to Rich Text, Core Text. My name is Rob Napier, and today we're going to dive into the iOS tools for text layout and style. iOS 6 made rich text much simpler by integrating it directly into UIKit, but to get the most out of it, you need to understand attributed strings.

- Understanding Rich Text
- Attributed Strings
- Core Text

We'll start with understanding rich text.

## What Makes Text “Rich?”

We all know what rich text is, right? Styled text. But really it's more than that. We've had “styled text” in iOS since the beginning.

- Understanding Rich Text
- Attributed Strings
- Core Text

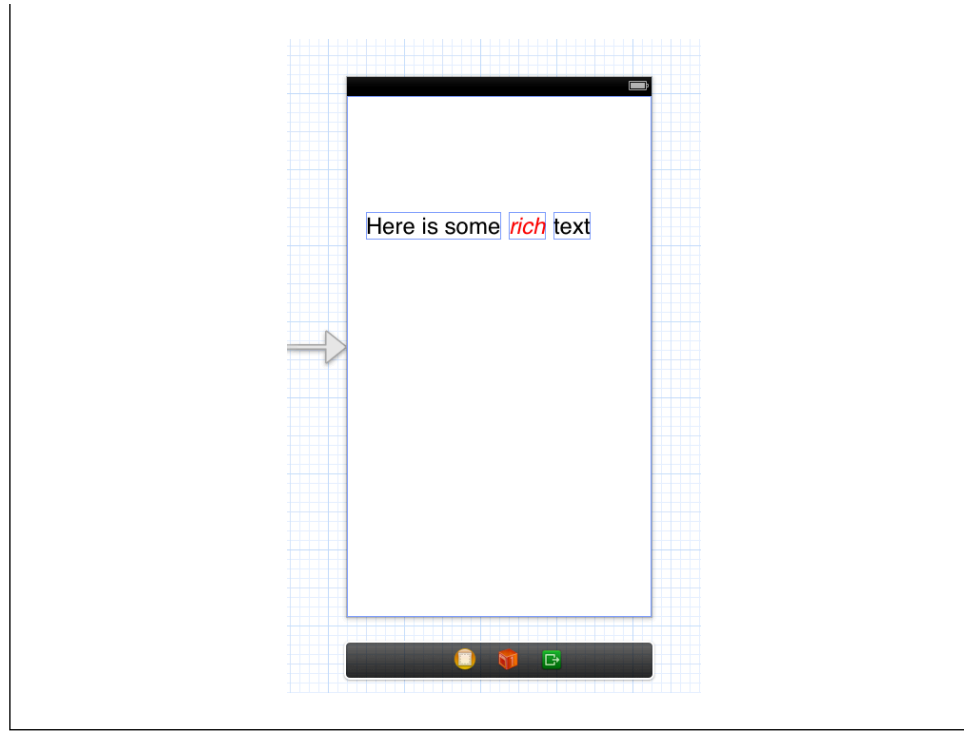
You should leave here today with a much deeper understanding of how rich text really works, the difference between a font and a decoration, between a character and glyph, the fundamentals of metrics, and basically just how to make heads or tails of the documentation. Incidentally, we'll also talk about heads and tails.

Once we cover the basics, we'll put them to practice, first with NSAttributedString and then with Core Text. By the time we're done, you should know how to build most normal kinds of formatting, and at least have a good idea of which part of the Core Text stack you need to use to attack even the most complicated layout problems. It's a really full session, so let's get to it.



After all, these are all “styled.” You can get italics and bold without “rich text.”

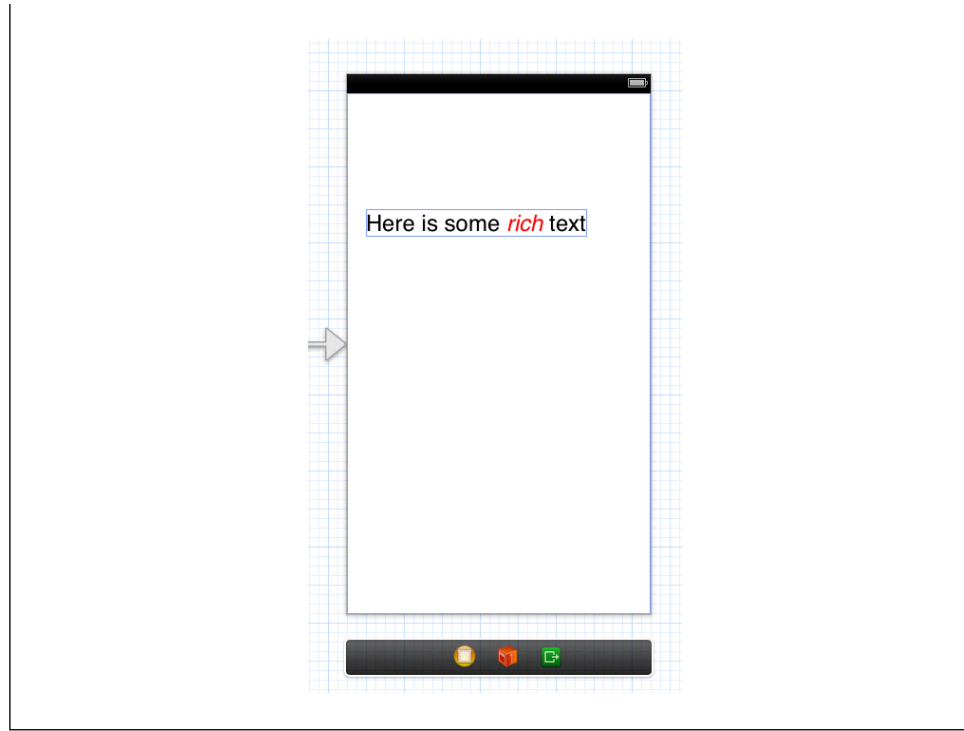
No. Rich text is more than just styling. Your next thought might be that it’s the mixing of styles, and that’s true. But you can mix styles without rich text.



You make three UILabels and stick them on the screen in the right places. Done. But this is a serious pain for two reasons: first, there's no easy data structure for managing the text along with its formatting, and second, it's up to you to layout the text.

The rich text features in iOS addresses both of these. First, it provides a data structure, NSAttributedString, that combines content and style in a single object. Then it supports automatically laying out these attributed strings.

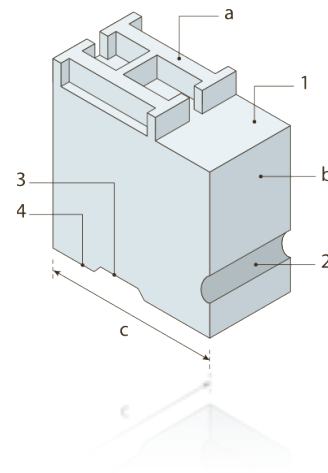
iOS has supported attributed string layout since 3.2, but it's been buried in Core Text. Core Text is great and we'll be discussing it later, but it's a very low-level tool. Once you start doing your own layout with Core Text, you get no help at all from UIKit. You can't use text fields or text views, you lose text selection, and cut and paste. You basically get a printing press rather than a word processor. Building an editable text view out of Core Text is a major undertaking.



But that changes in 6.0. Now, UIKit has its own support for attributed strings. That means you can put things like this in a single label. We'll discuss shortly how to build attributed strings like this, but first, a digression into the basics of fonts, decorations, and styles.

## Font:

(*noun*) a quantity of sorts composing a complete character set of a single size and style of a particular typeface.



First, what is a font? Back in the day, a font was a big box of metal type like this one, called a sort, that you stuck into a letterpress, inked up, and pressed onto paper. Of course fonts aren't anything like that today, but it's important at times to remember the history because it plays into the terminology. A font is something that is cast by a foundry. And today, shops that make fonts, like Adobe, Bitstream, and Apple, are still called foundries. Since a font was a physical set of metal or wood blocks, a font had a specific size. So 10 point Bulmer was a completely different font from 12 point Bulmer.

Today, digital fonts are described by vectors, so size is irrelevant, and all sizes of Helvetica are the same font. But there are a lot of different fonts under the broad term "Helvetica."

Now, the industry is very inconsistent about how it classifies fonts, typefaces, families, and the like. Don't get too hung up on the specific terms. I'll try to match the Apple docs as best I can.



Baskerville 42pt

Baskerville 64pt

Baskerville 72pt

Baskerville 96pt

Baskerville 144pt

Let's start with a nice font like Baskerville. These are all the same font. I've just scaled them to different sizes.

Baskerville

*Baskerville Italic*

**Baskerville SemiBold**

**Baskerville Bold**

***Baskerville SemiBold Italic***

***Baskerville Bold Italic***

These, however, are six different fonts. They all belong to the same family, but they're completely different fonts. Notice how Baskerville Italic doesn't actually look that much like Baskerville. For instance, the a is a completely different shape.

So that brings me to the important point here. There's this font, Baskerville, that we might call the "regular" version of the font. It's often called the "Roman" version. But then, some font designer sat down and designed a completely different font called Baskerville Italic, and created a different font definition with different vector information. I want to emphasize this because if some font designer hadn't sat down and designed Baskerville Italic and created a new font definition for it, there wouldn't be an italic version of Baskerville. This is why sometimes you can't italicize text in some fonts. Try italicizing Lucida Grande in TextEdit and you'll see what I mean. There's no italic version, so Cmd-I doesn't do anything.

You'll also notice that there are both semibold and bold versions of this font. And Bold Italic is yet another font. So later, when we are trying to apply bold or italics to our text, don't be surprised that it requires searching for a new font.

Baskerville

Baskerville (Underlined)

Baskerville (Underlined-Keynote)

What about underline? Notice that the shape of the font is unchanged. Underlining is different from bold and italics. Underlining is a decoration like strikethrough or shadow. Even though Bold, Italics, and Underline are often put together as a group of three buttons, they're different kinds of operations.

Just a side note. The first line here is just a Keynote text box, but the second line is a screenshot from TextEdit. That's because TextEdit does the same kind of fancy underlining that iOS does. See how it breaks around the parentheses. Keynote doesn't do that. It underlines like this. <build>

These kinds of subtle differences in drawing engines can sometimes be a real problem, so if possible stick to one drawing engine.

# Characters and Glyphs

Just like in the old days, fonts are made up of a series of individual shapes. These shapes are related to characters, but they're not exactly the same.

# Characters

A	س	我
á	چ	你
!	¿	;

A character generally represents a unit of writing. These are letters, punctuation marks, and ideographs.

## Different Fonts Different Glyphs

a   **a**   a   *a*

我   我   我

A shape that represents a character is called a glyph. A font is mostly a collection of glyphs representing characters.

But glyphs don't map one to one with characters.

# Contextual Glyphs

Isolated	ه
Initial	هـ
Medial	هـ
Final	هـ

For instance, in Arabic, the shapes of most letters change based on their position in the word. So even within a single font, there may be more glyphs in a font than there are represented characters.

# Ligatures

Sometimes when letters are written together, they are combined into a single glyph called a ligature. So in a given string, there may be fewer glyphs drawn than there are characters.



## Ligatures

J	+	l	=	ŷ
f	+	i	=	fi
f	+	i	=	fi
<b>f</b>	+	<b>i</b>	=	<b>fi</b>
f	+	l	=	fl
<i>T</i>	+	<i>h</i>	=	<i>Th</i>

Some ligatures, like lem-aleph in Arabic, are part of the language and are required. But sometimes they're added to improve legibility.

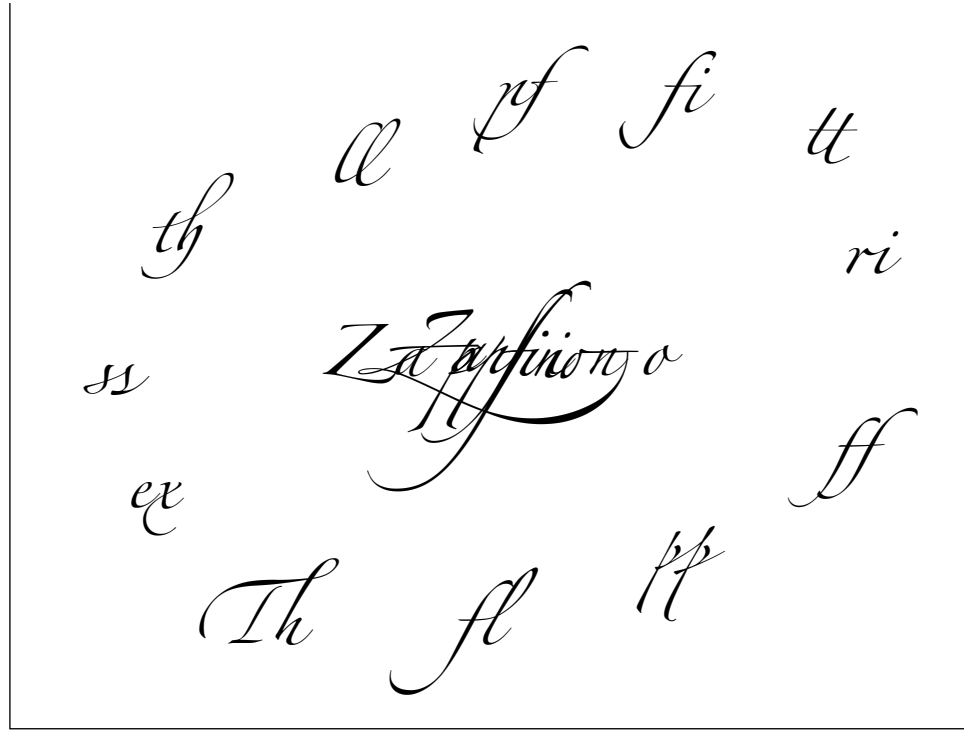
<build> The most common ligature in English is *f-i*.

<build> Different fonts may implement it differently.

<build> And it doesn't appear in all fonts.

<build> There are several other English ligatures that some fonts implement. For example *f-l* ...

<build> and *t-h*. Notice how dramatically the ligature can differ from the individual characters in some fonts.



Zapfino has some of my favorite ligatures, including a special 7-character glyph for its own name.

### <build>

Zapfino is actually a great font to explore if you're interested in the limits of modern typography. If this stuff interests you, look at Zapfino Extra Pro. I'll put a link to it on my site. It's pushing the limits of what it means to be a "font."

The point is that the number of glyphs in a string is only indirectly related to the number of characters in the string, and the number of glyphs may depend on the font and how the string happens to lay out. This'll be important later when we talk about Core Text.

# Diacritics

e	+	'	=	é
a	+	°	=	å
u	+	¨	=	ü
ك	+	´	=	ك´

There also may be more glyphs than characters. For instance, letters with a diacritic (“accent mark”) may be implemented by either a special glyph that includes the mark, or as two glyphs, one for the letter and another for the accent. This can depend on the font.

And Unicode can create the same glyphs from different sequences of characters.

For example, there’s a character “latin small letter e with acute,” as shown in the first line. But, you can also encode this as two characters, “latin small letter e” followed by “combining acute accent.” Same visual result, but whether these one or two characters map to one or two glyphs depends on a lot of things.

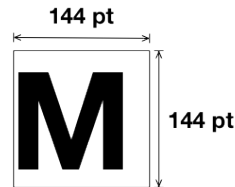
All this is to say that you shouldn’t assume that the number of glyphs is the same as the number of characters. Enough about that.



## Metrics

Let's talk about layout, which is what Core Text is all about. Line layout is guided by font metrics. Any time you want to layout lines manually, you're going to need these numbers. Core Text has functions to calculate them and we'll discuss those later. For now we'll just cover what the main ones mean. Most of these are given in points, which is 1/72nd of an inch.

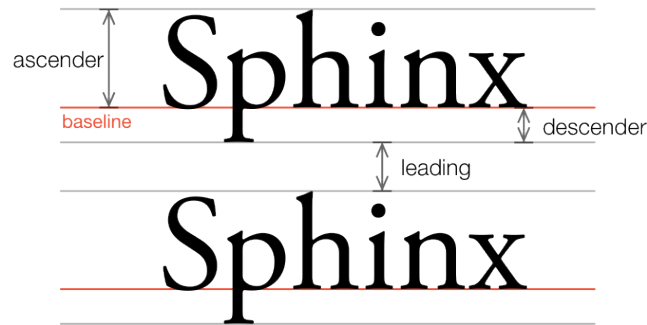
# The Em



1 pt = 1/72 inch

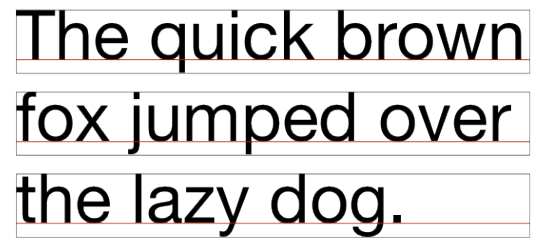
First there's the em. This defines the size of the font. When you say this is 144-point Helvetica, you mean Helvetica scaled to a 144-point em. The name comes from the letter *M*, but that doesn't mean that the letter *M* fills this box. In fact, it often doesn't. In fact, nothing in the font may fill this box. Glyphs might even draw outside the box. But it's used as a scaling factor. Two fonts with the same em should visually be at the same scale.

# Ascenders and Descenders



The most important metrics for layout are the ascender, descender, and leading. The ascender is how high the tallest glyph is above the baseline. Notice how this is often a little taller than the height of capital letters, or cap height. There are often extra little finials at the top of letters with long upstrokes like l and h that go above the cap height. The descender is the lowest point that glyphs reach below the baseline. The leading is extra space the font requires between lines. This is often zero. We'll discuss how line spacing is usually handled when we get to paragraph styles.

## Line Boxes

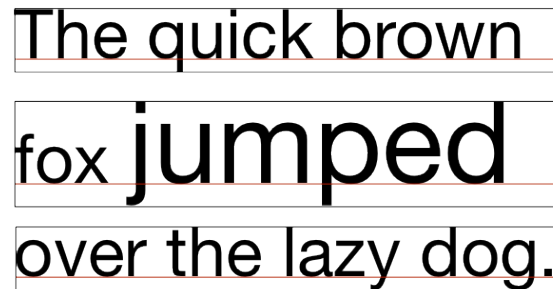


The quick brown  
fox jumped over  
the lazy dog.

The diagram shows three lines of text, each enclosed in a rectangular box. The boxes are stacked vertically. The text inside the boxes is 'The quick brown', 'fox jumped over', and 'the lazy dog.'. The boxes are drawn with thin black lines, and the text is centered within each box. The boxes are of equal height and width, illustrating how line boxes are used to define the vertical and horizontal space for each line of text.

These three numbers are enough to do all of the line layout. It doesn't matter what the actual characters are. All you need to know is the maximum ascender, maximum descender, and leading, and you can you build your typographic bounding boxes.

## Line Boxes



The quick brown  
fox jumped  
over the lazy dog.

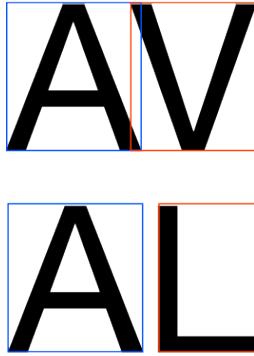
If you have a mixture of fonts, then you just expand your line box to match the largest ascender plus the largest descender. Decorations like underline go inside the box, so you don't have to worry about those for layout.

That really is most of line layout. Build a bunch of boxes based on the font metrics in the line, and then stack the boxes. For horizontal text, you stack the boxes top to bottom. For vertical text, everything is the same, except you stack the boxes right to left.

It's so simple that you can do it by hand with the help of a few Core Text calls to fetch the metrics. We'll walk through an example of that later.



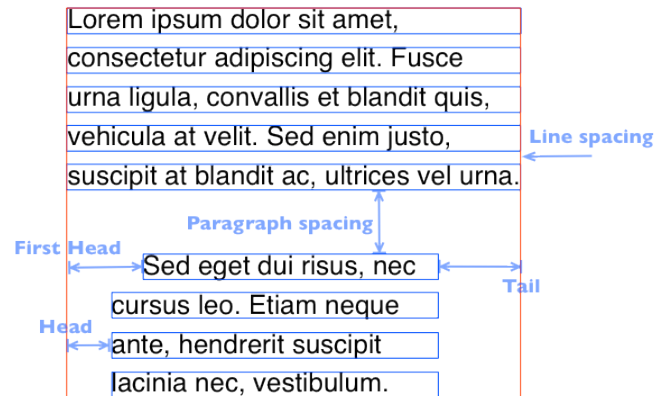
# Kerning



The same cannot be said for glyph layout. Figuring out where each glyph goes within a line is pretty complicated. Not only is each glyph a different width, but its positioning is based on which glyphs are around it. This is called kerning, which places characters like A and V closer to each other than A and L. As you can see, the box around A overlaps the box around V, but not the box around L. Horizontal layout is heavily dependent on the specific font and the specific characters. The line might also be justified, which can adjust spacing in even more complicated ways. In most cases, you want to leave glyph layout within a line to Apple and don't try to recreate any of that.

As a side note, iOS 6 has one really bad bug in its kerning. Currently, UILabels kern correctly if you ask them to. TextViews never kern. I have a radar against that, and hopefully it'll be fixed.

# Paragraphs



Lines are grouped into paragraphs. A paragraph collects style information that applies to a range of characters collectively. For instance, you can discuss the font of a single character, but line spacing only makes sense when talking about a range of characters. Line spacing doesn't tell you how far away one character is from another, but how rather far one group of characters is away from another group of characters. Similarly, you can only justify a line of text when you know all the characters in that line.

Paragraphs include a ruler, which in iOS mostly defines the margins, called the head and tail. The margins change the dimensions of the line's bounding box. And paragraphs include line spacing and paragraph spacing. So remember that the goal is to stack a bunch of lines together, and the paragraph tells the system how wide each line can be, and how closely they can be stacked together.

Paragraph styles also include justification, hyphenation, line breaking, and default writing direction. We'll see how to apply these when we discuss Attributed Strings.

# Quick Recap

- Fonts - Glyphs + Metrics
- Glyphs - Drawable units
- Lines - Glyphs + Positioning
- Paragraphs - Lines + Style

That sums up the basics of rich text, and especially the vocabulary that we'll be using for the rest of this session.

**<build>** There are fonts, which are a collection of individual glyphs, along with various metrics and rules for how to lay them out.

**<build>** Glyphs are the fundamental drawing unit, and there may be more or fewer glyphs than characters.

**<build>** Lines of glyphs are laid out according to their font metrics. Glyphs are laid out within a line based on their kerning rules and justification.

**<build>** Tying it all together are paragraph styles that decide how long each line is and how closely they stack.

With the basics out of the way, let's take a look at some concrete code examples. That means we need a data structure to store all this information. And that brings us to attributed strings.

- Understanding Rich Text
- **Attributed Strings**
- Core Text

Attributed strings combine text and attributes into a single data structure. “Attributes” doesn’t have to mean style information. You can attach any key/value pairs you like to a range of characters in an attributed string. But typically the keys you’ll use are from a list that define styles. Let’s take a concrete example...

Be **Bold!** And a little color wouldn't hurt, either.

This sentence has six distinct ranges in it...

Be **Bold!** And a little color wouldn't hurt, either.

Each range has a different set of attributes. Let's look at the code.

Be **Bold!** And a little color wouldn't hurt, either.

```
NSString *string = @"Be Bold! And a little color wouldn't hurt either.";
NSDictionary *attrs = @{ NSFontAttributeName : [UIFont systemFontOfSize:36]};
NSMutableAttributedString *
as = [[NSMutableAttributedString alloc] initWithString:string attributes:attrs];

[as addAttribute:NSFontAttributeName
  value:[UIFont boldSystemFontOfSize:36]
  range:[string rangeOfString:@"Bold!"]];

[as addAttribute:NSForegroundColorAttributeName
  value:[UIColor blueColor]
  range:[string rangeOfString:@"little color"]];

[as addAttribute:NSFontAttributeName
  value:[UIFont systemFontOfSize:18]
  range:[string rangeOfString:@"little"]];

self.label.attributedText = as;
```

Start with a string and apply a base set of attributes to create a mutable attributed string. You'll generally create a mutable attributed string rather than a normal attributed string, since that's the only way to apply different styles to different ranges. In this case, we just assign a font as the base attribute. Then we add attributes for each range. Notice how you can layer attributes, one on top of another.

Attributed strings keep track of how attributes combine over a range. So you can determine the effective attributes at any point, and you can work out ranges that have the same attributes, or the same value for a given attribute.

# Finding Fonts

```
[UIFont fontWithName:@"Helvetica" size:14.0];

[UIFont systemFontOfSize:14.0];
[UIFont boldSystemFontOfSize:14.0];
[UIFont italicSystemFontOfSize:14.0];

[UIFont systemFontOfSize:[UIFont systemFontOfSize]];
[UIFont systemFontOfSize:[UIFont smallSystemFontSize]];
[UIFont systemFontOfSize:[UIFont labelFontSize]];
```

One of the most common attributes you'll set is the font. If you know the specific font you want, it's very easy to set.

<build> You can just use `fontWithName:size:` and you're done.

If you want a system font, that's easy, too.

<build> You can get the system font in any size you want, in normal, bold, or italics.

If you're not certain of the correct size, you can ask for the system default sizes.

<build> If you're not certain of the correct size, you can ask for the system default sizes.

But what if you have an existing string, and you want to bold whatever font happens to be there?



# UIKit Toggles

```
// these methods are not implemented in NSObject
@interface NSObject(UIResponderStandardEditActions)

- (void)cut:(id)sender NS_AVAILABLE_IOS(3_0);
- (void)copy:(id)sender NS_AVAILABLE_IOS(3_0);
- (void)paste:(id)sender NS_AVAILABLE_IOS(3_0);
- (void)select:(id)sender NS_AVAILABLE_IOS(3_0);
- (void)selectAll:(id)sender NS_AVAILABLE_IOS(3_0);
- (void)delete:(id)sender NS_AVAILABLE_IOS(3_2);
- (void)makeTextWritingDirectionLeftToRight:(id)sender NS_AVAILABLE_IOS(5_0);
- (void)makeTextWritingDirectionRightToLeft:(id)sender NS_AVAILABLE_IOS(5_0);

- (void)toggleBoldface:(id)sender NS_AVAILABLE_IOS(6_0);
- (void)toggleItalics:(id)sender NS_AVAILABLE_IOS(6_0);
- (void)toggleUnderline:(id)sender NS_AVAILABLE_IOS(6_0);

@end
```

If you're trying to bold selected text in a text field or text view, then you can use the `toggleBoldface:` method. They really buried this in the docs as an informal protocol on `NSObject`. There are also `toggleItalics:` and `toggleUnderline:` methods. In practice, if you're using the standard controls, then you probably never have to care about this anyway, since the standard controls create their own menus for bold, italics, and underline, and handle the requests automatically, so this doesn't come up much.

But what about a less standard situation? Say you have an attributed string with fonts you don't know in advance. You want to bold all of it. How do you go about that?

```
NSMutableAttributedString *as = ...  
[as enumerateAttribute:...  
    inRange:...  
    options:...  
    usingBlock:^(id value, NSRange range, BOOL *stop)  
    {  
        ...  
    }];
```

The first tool you'll want is a way to walk through the string, picking out ranges that have the same font. You can do that with `enumerateAttribute:inRange:options:usingBlock:.`

You pass it an attribute name you're interested in, ...

```
NSMutableAttributedString *as = ...  
[as enumerateAttribute:NSFontAttributeName  
    inRange:...  
    options:...  
    usingBlock:^(id value, NSRange range, BOOL *stop)  
    {  
        ...  
    }];
```

the range to enumerate, ...

```
NSMutableAttributedString *as = ...  
[as enumerateAttribute:NSFontAttributeName  
    inRange:NSMakeRange(0, as.length)  
    options:...  
    usingBlock:^(id value, NSRange range, BOOL *stop)  
    {  
        ...  
    }];
```

some options, ...

```
NSMutableAttributedString *as = ...

[as enumerateAttribute:NSFontAttributeName
    inRange:NSMakeRange(0, as.length)
    options:NSAttributedStringEnumerationLongestEffectiveRangeNotRequired
    usingBlock:^(id value, NSRange range, BOOL *stop)
    {
        ...
    }];
```

and a block to execute. ...

```
NSMutableAttributedString *as = ...

[as enumerateAttribute:NSFontAttributeName
    inRange:NSMakeRange(0, as.length)
    options:NSAttributedStringEnumerationLongestEffectiveRangeNotRequired
    usingBlock:^(id value, NSRange range, BOOL *stop)
    {
        UIFont *font = value;
        UIFont *boldFont = RNGetBoldFontForFont(font);
        if (boldFont) {
            [as addAttribute:NSFontAttributeName value:boldFont range:range];
        }
    }];
```

The block will be run once for each range that has the same value for the attribute. In this case a range of characters with the same font. There's a related method that will do this for all attributes if you need that.

Unlike most enumeration methods, this one actually allows you to modify the attributed string inside the block. You can even add and remove characters within the range you were given. We'll be using that ability to mutate the attributes here.

First let's look at the options...

```
NSMutableAttributedString *as = ...

[as enumerateAttribute:NSFontAttributeName
    inRange:NSMakeRange(0, as.length)
    options:NSAttributedStringEnumerationLongestEffectiveRangeNotRequired
    usingBlock:^(id value, NSRange range, BOOL *stop)
    {
        UIFont *font = value;
        UIFont *boldFont = RNGetBoldFontForFont(font);
        if (boldFont) {
            [as addAttribute:NSFontAttributeName value:boldFont range:range];
        }
    }
];
```

First let's look at the options. LongestEffectiveRangeNotRequired lets the enumerator use a slightly faster algorithm that isn't guaranteed to be the longest possible range. NSAttributedString has several methods that offer this optimization. It just means that the block may be called more times than necessary for complex attributed strings.

```
NSMutableAttributedString *as = ...

[as enumerateAttribute:NSFontAttributeName
    inRange:NSMakeRange(0, as.length)
    options:NSAttributedStringEnumerationLongestEffectiveRangeNotRequired
    usingBlock:^(id value, NSRange range, BOOL *stop)
    {
        UIFont *font = value;
        UIFont *boldFont = RNGetBoldFontForFont(font);
        if (boldFont) {
            [as addAttribute:NSFontAttributeName value:boldFont range:range];
        }
    }];
```

The block itself is simple. For each font, get the bold version. If there is a bold version, then apply it to the range.



```
NSMutableAttributedString *as = ...

[as enumerateAttribute:NSFontAttributeName
    inRange:NSMakeRange(0, as.length)
    options:NSAttributedStringEnumerationLongestEffectiveRangeNotRequired
    usingBlock:^(id value, NSRange range, BOOL *stop)
    {
        UIFont *font = value;
        UIFont *boldFont = RNGetBoldFontForFont(font);
        if (boldFont) {
            [as addAttribute:NSFontAttributeName value:boldFont range:range];
        }
    }];
```

But you'll notice this `RNGetBoldFontForFont`. You'd think that'd be builtin, but it isn't. `UIFont` doesn't have many methods for dealing with non-system fonts. You have to dive down into Core Text. But Core Text has a completely different font object than UIKit, and the two aren't compatible. So we're going to have convert our UIKit font into a Core Text font, use that to find a bold version, if there is one, and then convert the Core Text font back to a `UIFont` and apply it to our attributed string. Let's walk through it.

```
// Returns the bold version of a font. May return nil if there is no bold version.
UIFont *RNGetBoldFontForFont(UIFont *font) {
    UIFont *result = nil;

    CTFontRef ctFont = CTFontCreateWithName((__bridge CFStringRef)(font.fontName),
                                              font.pointSize, NULL);
    if (ctFont) {

        CFRelease(ctFont);
    }
    return result;
}
```

First we find a Core Text font that has the same name and size as the UIKit font. This should always succeed, but there have been some hiccups in the past, and you should check for failures. When the retina display came out, it added a special private font called “dot HelveticaNeueUI”. With a leading period. It’s not like other fonts. In particular, it doesn’t show up in some lists of available fonts, which can lead to crashes if you assume that the current font always does. The point is, fonts and font names can be a little weird, and can definitely change over time, so code this stuff very defensively.

```
// Returns the bold version of a font. May return nil if there is no bold version.
UIFont *RNGetBoldFontForFont(UIFont *font) {
    UIFont *result = nil;

    CTFontRef ctFont = CTFontCreateWithName((__bridge CFStringRef)(font.fontName),
                                              font.pointSize, NULL);
    if (ctFont) {
        if ((CTFontGetSymbolicTraits(ctFont) & kCTFontTraitBold) == 0) {

        }
        CFRelease(ctFont);
    }
    return result;
}
```

OK, now that we have our CTFont, we can check whether it's already bolded. This particular example doesn't require this step, but it's good to show how it's done.

Bold is considered a symbolic trait. It's technically a special case of the weight trait, which is a number from -1 to 1. Unless you really want that kind of fine-grain control, it's usually best just to use the symbolic traits. The symbolic traits are a bitfield, so we use the bitwise “and” to check the current state.

```
// Returns the bold version of a font. May return nil if there is no bold version.
UIFont *RNGetBoldFontForFont(UIFont *font) {
    UIFont *result = nil;

    CTFontRef ctFont = CTFontCreateWithName((__bridge CFStringRef)(font.fontName),
                                              font.pointSize, NULL);
    if (ctFont) {
        if ((CTFontGetSymbolicTraits(ctFont) & kCTFontTraitBold) == 0) {
            CTFontRef ctBoldFont = CTFontCreateCopyWithSymbolicTraits(ctFont,
                                                                        font.pointSize,
                                                                        NULL,
                                                                        kCTFontBoldTrait,
                                                                        kCTFontBoldTrait);

            CFRelease(ctFont);
        }
        return result;
    }
}
```

And then we make a new font with the bold trait turned on. The fourth parameter is a value and the second parameter is a mask, so you can use this to turn on and off various traits at the same time.

```

// Returns the bold version of a font. May return nil if there is no bold version.
UIFont *RNGetBoldFontForFont(UIFont *font) {
    UIFont *result = nil;

    CTFontRef ctFont = CTFontCreateWithName((__bridge CFStringRef)(font.fontName),
                                             font.pointSize, NULL);
    if (ctFont) {
        if ((CTFontGetSymbolicTraits(ctFont) & kCTFontTraitBold) == 0) {
            CTFontRef ctBoldFont = CTFontCreateCopyWithSymbolicTraits(ctFont,
                                                                        font.pointSize,
                                                                        NULL,
                                                                        kCTFontBoldTrait,
                                                                        kCTFontBoldTrait);

            if (ctBoldFont) {
                NSString *fontName = CFBridgingRelease(CTFontCopyPostScriptName(ctBoldFont));
                result = [UIFont fontWithName:fontName size:font.pointSize];
                CFRelease(ctBoldFont);
            }
        }
        CFRelease(ctFont);
    }
    return result;
}

```

And finally, we create our new font and return it. I'm using the PostScript name here, and that's the one I recommend as a unique identifier. There's another function called `CTFontCopyName`, but it's really a more general information function that can return the copyright, the font family, the font style name, and a bunch of other information, as well as the PostScript name.

```
/*!
    @defined    kCTFontUniqueNameKey
    @abstract    The name specifier for the unique name.
    @discussion  Note that this name is often not unique and should not be
                  assumed to be truly unique.
*/
extern const CFStringRef kCTFontUniqueNameKey
CT_AVAILABLE_STARTING( __MAC_10_5, __IPHONE_3_2);
```

My favorite thing you can get back from `CTFontCopyName` is the unique name. Check the description.

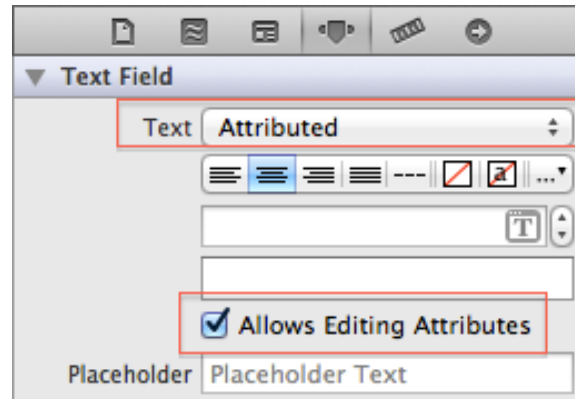
That's right. The “unique name” is often not unique, so please don't use it as a unique identifier. Use the postscript name.

```
NSAttributedString *as = [[NSAttributedString alloc] initWithString:@"Here is some text!"];  
[as drawAtPoint:CGPointZero];  
[as drawInRect:CGRectMake(0, 100, 100, CGFLOAT_MAX)];
```

Once you've created an attributed string, actually using them in UIKit is trivial. Most controls now support attributed strings, usually with a property called “attributed something.” Like UILabel has a text property for regular text. Now it has an attributedText property for attributed text. Same for buttons, text fields, text views, and picker views.

You can also draw them into a context, just like regular strings, using drawAtPoint: or drawInRect:.

# Switching to Attributed



In Interface Builder, for UITextField and UITextView, you need to set the Text type to “Attributed” and select “Allows Editing Attributes.”



## B/U Selector



When you do this, then users can select text and apply bold, italics and underlining.

In the current version of Xcode. Setting “Allows Editing Attributes” for a text view doesn’t actually do anything, though it does work for a text field. You have to set this in `viewDidLoad` or the like. I’ve opened a radar against that, and hopefully this will be fixed soon.

# Paragraph Styles

Most styles apply specifically to the range you attach them to. Fonts, colors, strike-through, underline. They all impact individual characters. Paragraph styles are a little different, since they apply to things beyond the current character. What does it mean to change the margin in the middle of a line?

The only paragraph style that matters is one applied to the first character of a paragraph. Paragraphs are delimited by newlines.

Alignment, line spacing, margins, line breaking, and default text direction are bundled into an `NSParagraphStyle` object, and are collectively one attribute. So when you change the margins on a paragraph, you need to be careful not to remove any other paragraph styles that might have already been on it. It's easiest to show this in an example.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi ac urna id augue volutpat tempus at vitae libero. Donec venenatis faucibus erat, et feugiat nunc dignissim sed. Curabitur egestas quam ut ante tincidunt dignissim. Nunc non nulla eros. Donec elementum auctor augue, dapibus blandit augue sollicitudin in. Proin dictum, neque sit amet venenatis facilisis, orci ligula vestibulum dolor, non pulvinar ipsum magna quis leo.

Integer eget enim at erat rhoncus volutpat a sit amet ligula. Suspendisse potenti. Curabitur faucibus vulputate nibh vel condimentum. Mauris tortor arcu, tincidunt sit amet auctor nec, consectetur vel ipsum. Proin vitae magna risus, non aliquam tortor. Nunc ut purus eu diam semper laoreet.

Suspendisse potenti. Ut eu mi elit, eu tincidunt mauris. Nullam ut egestas ante. Proin suscipit convallis nisi sed convallis. Nullam convallis posuere venenatis. Curabitur in mauris nulla, in tempus est. Etiam augue metus, viverra eget cursus interdum, malesuada eu nisl. Nunc vel nibh sit amet purus blandit malesuada. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

We want to layout something like this. There are three paragraphs. They all have a paragraph spacing attribute, they're all justified, and they all have a first-line head indent. (We say head and tail rather than left and right because the text might be vertical.) We want to give the second paragraph different margins than the others.

```

// Set up the base paragraph style
NSMutableParagraphStyle *wholeDocStyle = [[NSMutableParagraphStyle alloc] init];
[wholeDocStyle setParagraphSpacing:34.0];
[wholeDocStyle setFirstLineHeadIndent:10.0];
[wholeDocStyle setAlignment:NSTextAlignmentJustified];

NSMutableAttributedString *
pas = [[NSMutableAttributedString alloc] initWithString:paragraphs
        attributes:
            @{NSParagraphStyleAttributeName: wholeDocStyle}];

// Find the second paragraph and copy the current style
NSUInteger secondParagraphStart = NSMaxRange([pas.string rangeOfString:@"\n"]);

NSMutableParagraphStyle *secondParagraphStyle = [[pas attribute:NSParagraphStyleAttributeName
        atIndex:secondParagraphStart
        effectiveRange:NULL] mutableCopy];

// Adjust the second paragraph's style
secondParagraphStyle.headIndent += 50.0;
secondParagraphStyle.firstLineHeadIndent += 50.0;
secondParagraphStyle.tailIndent -= 50.0;

// Apply the style to the start of the second paragraph
[pas addAttribute:NSParagraphStyleAttributeName
        value:secondParagraphStyle
        range:NSMakeRange(secondParagraphStart, 1)];

```

First, let's set up the base paragraph style and apply it to the whole document.

<build> For the second paragraph, we want to merge some style information into what's already there. First we need to find the second paragraph and make a copy of its current style. Of course in this case we actually know the style, since we just applied it, but assume we don't know the style and need to look it up, so we search for a newline, and then get the paragraph style at that point.

<build> Now we can adjust the style. Notice we're adding and deleting, not setting. headIndent applies to all lines but the first one, so you need to set both headIndent and firstLineHeadIndent. Tail indent is negative since it's towards the left.

<build> And that's it. We can just apply it now.

You don't need to worry about finding the end of the paragraph, since only the first character matters.

And there you go, paragraph styles.

noitceriD txeT

# Text Direction

# Text Direction

With John and Samir (سمير)

```
self.label.text = [NSString stringWithFormat:@"Hello, %@. How are you?", name];
```

Hello, John. How are you?

Hello, سمير. How are you?

```
self.label.text = [NSString stringWithFormat:@"%s@, how are you?", name];
```

John, how are you?

?how are you سمير,

That leaves just one more wrinkle. Text direction. Dealing with right-to-left languages can be one of the trickiest parts of text layout, particularly when you start mixing directions in the same string. Let's look at an example in a UILabel. We want to say hello to our user, so we have code like this.

<build>No problems.

<build>We get "Hello, John. How are you?" and "Hello, Samir. How are you?"

<build>But let's change the string a little bit.

<build>Looks good. And for Samir:

<build>What? That's a mess. What's going on here? iOS must be broken. Open a radar... wait. Maybe this isn't a bug. Maybe it's doing the right thing. Let's think this through. The default text layout direction is called "natural." That means it bases the direction on the string itself.

Let's go back to the first Arabic example.

# Text Direction

With John and Samir (سمير)

Hello, سميير. How are you?

Most unicode characters have an associated direction. The letter “H” has a left-to-right direction. So when iOS sees the “H” in “Hello,” it marks the string as left-to-right. When it gets to the seen in Samir, it switches to right-to left until it sees the period. Punctuation marks don’t have a direction, they rely on the direction of the characters around them. If the characters disagree, then the default direction wins. So the string says “H, so start left to right”, now e-l-l-o-comma-space, seen, hey this is right to left, meem, reh, hmm, a period, need to keep looking, a space, keep looking, an “H”. Between reh and H, H wins because this string is left-to-right, so we lay out the punctuation left-to-right. And so on.

Let’s see how that works out in the second example.



# Text Direction

With John and Samir (سمير)

English w/  
Arabic

Hello, سمير. How are you?

?how are you, سمير

Arabic w/  
English

The first letter is seen, so we start right-to-left. Then we lay out meem and reh. The comma and space are punctuation so we look at the next character, h. It disagrees with reh, so we use the default and keep going right-to-left for the punctuation and switch to left-to-right for the h and so on until we get to the question mark. The end of the string is treated like it's in the default direction, so that forces the question mark to be laid out right-to-left.

<build> The easiest way to think of this is that the first example is an English string with embedded Arabic. The second example is an Arabic string with embedded English.

Fine, fine. It does what it's supposed to do. But that's definitely not what we wanted it to do. So what now?

Well, the quickest solution is to mark the label's direction as left-to-right rather than natural. That's ok, but then when you translate your UI to Arabic, you need to mark the label right-to-left or it'll layout backwards.

Another approach is to always start your strings with a character that goes in the correct direction. The first example showed how to do that, but there's a better way.

## Unicode Implicit Directional Marks

Abbr.	Code	Name	Description
LRM	U+200E	LEFT-TO-RIGHT MARK	Left-to-right zero-width character
RLM	U+200F	RIGHT-TO-LEFT MARK	Right-to-left zero-width character

You can add a unicode implicit directional mark at the beginning of your strings that might start with unknown content. This is nice because it can be handled entirely in your strings file. Both are zero-width, in other words invisible. They behave as though they were a left-to-right or right-to-left character at that position, which makes them very simple to use. There are other Unicode characters that explicitly change the direction of the text, but they're more complicated and error-prone. I strongly recommend using the implicit directional marks unless you have a very specialized problem.

# Using Implicit Directional Marks

@"%@, how are you?" → @"\u200e%@, how are you?"

So, consider a format string like this.

In English, you would just localize that as:

**<build>**

That will force the string to be left-to-right, no matter what the substitution is. You only need to do this when your localized string begins with a substitution. That might not happen in every language, but it's easy to watch out for while localizing.

Learning the rules of text direction will save you a lot of headaches when faced with multi-directional text. The rules aren't that complicated, and they make a lot of sense once you get to know them.

# References

Understanding Bidirectional (BIDI) Text in Unicode  
Cal Henderson

<http://www.iamcal.com/understanding-bidirectional-text/>

Unicode Standard Annex #9  
Unicode Bidirectional Algorithm

<http://www.unicode.org/reports/tr9>

I highly recommend Cal Henderson, article, “Understanding Bidirectional Text in Unicode.” If you want all the gory details, you can read the Unicode Standard’s bidirectional algorithm.

Be **Bold!** And a little color wouldn't hurt either.

```
NSString *string = @"Be Bold! And a little color wouldn't hurt either.";
NSDictionary *attrs = @{ NSFontAttributeName : [UIFont systemFontOfSize:36]};
NSMutableAttributedString *
as = [[NSMutableAttributedString alloc] initWithString:string attributes:attrs];

[as addAttribute:NSFontAttributeName
 value:[UIFont boldSystemFontOfSize:36]
 range:[string rangeOfString:@"Bold!"]];

[as addAttribute:NSForegroundColorAttributeName
 value:[UIColor blueColor]
 range:[string rangeOfString:@"little color"]];

[as addAttribute:NSFontAttributeName
 value:[UIFont systemFontOfSize:18]
 range:[string rangeOfString:@"little"]];

self.label.attributedText = as;
```

Before leaving attributed strings, as you've seen, the dictionaries for attributed strings can sometimes be a bit of a pain. You need to work out all the ranges, which you can either hard-code as numbers, or you have to do `rangeOfString:` searches, both of which are fragile and it's very hard to localize. So while `NSAttributedString` is really powerful, it's also a real pain.

There are several solutions to that.

Be **Bold!** And a little color wouldn't hurt either.

```
NSData *html =
[@"Be <strong>Bold!</strong> And a <span style='color:blue'>"
@"<span style='font-size:18'>little</span> color</span> wouldn't hurt either."
dataUsingEncoding:NSUTF8StringEncoding];

NSAttributedString *as = [[NSAttributedString alloc] initWithHTMLData:html
options:@{
    DTDefaultFontFamily:@"Helvetica",
    DTDefaultFontSize: @36,
    DTUseiOS6Attributes: @YES,
    documentAttributes:nil};
```

<https://github.com/Cocoanetics/DTCoreText>

Cocoanetics has a CoreText wrapper called DTCoreText that is really powerful and reads HTML. And if you have a lot of HTML to display, or you need a full rich text editor, you really should go check out DTCoreText. I'm very impressed with his work. He even provides UI elements so that you can display attributed strings on iOS 4.

But for this example, DTCoreText is a bit of overkill. HTML is really overkill. I don't need the full power of bulleted lists and images and clickable links and CSS and all the other things that DTCoreText offers. I just want some of my text to be blue, and I'm using iOS 6.

Be **Bold!** And a little color wouldn't hurt either.

```
// One-time setup
MTStringAttributes *attributes = [[MTStringAttributes alloc] init];
attributes.font = [UIFont systemFontOfSize:36];

MTStringParser *parser = [MTStringParser new];
[parser setDefaultAttributes:attributes];
[parser addStyleWithTagName:@"bold" font:[UIFont boldSystemFontOfSize:36]];
[parser addStyleWithTagName:@"blue" font:nil color:[UIColor blueColor]];
[parser addStyleWithTagName:@"small" font:[UIFont systemFontOfSize:18]];

// Then strings like this:
NSString *markup =
@"Be <bold>Bold!</bold> And a <blue><small>little</small> color</blue> wouldn't hurt either.";

self.label.attributedText = [parser attributedStringFromMarkup:markup];
```

<https://github.com/mysterioustrousers/MTStringAttributes>

<https://github.com/chrisdevereux/Slash>

That brings me to MTStringAttributes, and the Slash project it's based on. It's a very simple tag/attribute system. You define tags like “bold” or “small” or whatever you want, and then you assign attributes to those. Or you can use MTStringAttributes as just a short-cut for creating NSAttributedString attribute dictionaries. Very simple stuff, really, but very useful. If you have an iOS 6 project, and don't need a rich text editor, I would look at MTStringAttributes first. If you need more power than that, go with DTCoreText.

- Understanding Rich Text
- Attributed Strings
- Core Text

That wraps up our introduction to attributed strings. We've seen how they combine content with style information in a single data structure, and we've seen how to create, modify, and draw them. We've learned about paragraph styles, and how to assign text direction. And just briefly, we touched on Core Text and how it integrates, and doesn't integrate, with UIKit. Now it's time to leave UIKit behind and dive headfirst into Core Text.



# Core Text

- Framesetter
- Typesetter
- Line
- Run
- Glyph

Why do we even care about Core Text now that UIKit makes displaying rich text so much easier? The simple answer is layout. UIKit is great at displaying rich text in simple ways, but if you want complicated layout, Core Text is still a great tool. There are some disadvantages with Core Text, however. It's not as simple to use as UIKit. You'll need to be comfortable with Core Foundation and ARC won't memory manage anything for you. There also still aren't any good ways to integrate Core Text with text selection. It's more possible in iOS 6, but my initial experiments still require a huge amount of code.

That said, Core Text is a very good tool to have available. Let's learn our way around. Today we'll cover framesetters, typesetters, lines, run, and glyphs.



The highest abstraction layer in Core Text is the framesetter. You create a framesetter with an attributed string. Then you give it a series of boxes, and the framesetter will fill the boxes with text, which you can then draw.

Let's look at a simple example of column layout. In this example, we have an existing array of CGPaths, as well as an attributed string. We're just focusing here on the drawRect: code here.

```

// Initialize the text matrix
CGContextRef context = UIGraphicsGetCurrentContext();
CGContextSetTextMatrix(context, CGAffineTransformIdentity);

// Create the framesetter
CTFramesetterRef
framesetter = CTFramesetterCreateWithAttributedString(attrString);

// For each path, create a frame filled with text
CFArrayRef paths = [self copyPaths];
CFIndex pathCount = CFArrayGetCount(paths);
CFIndex charIndex = 0;
for (CFIndex pathIndex = 0; pathIndex < pathCount; ++pathIndex) {
    CGPathRef path = CFArrayGetValueAtIndex(paths, pathIndex);

    CTFrameRef
    frame = CTFramesetterCreateFrame(framesetter,
                                     CFRangeMake(charIndex, 0),
                                     path,
                                     NULL);

    // Draw the text frame
    CTFrameDraw(frame, context);

    // Find the starting point for the next frame
    CFRange frameRange = CTFrameGetVisibleStringRange(frame);
    charIndex += frameRange.length;
    CFRelease(frame);
}

```

First, before drawing with Core Text you should always initialize your text matrix. The text matrix is a transform applied to text drawing. For some reason it's not initialized for you before `drawRect:` is called, and it's not saved in the graphics state. So for safety, you should initialize it.

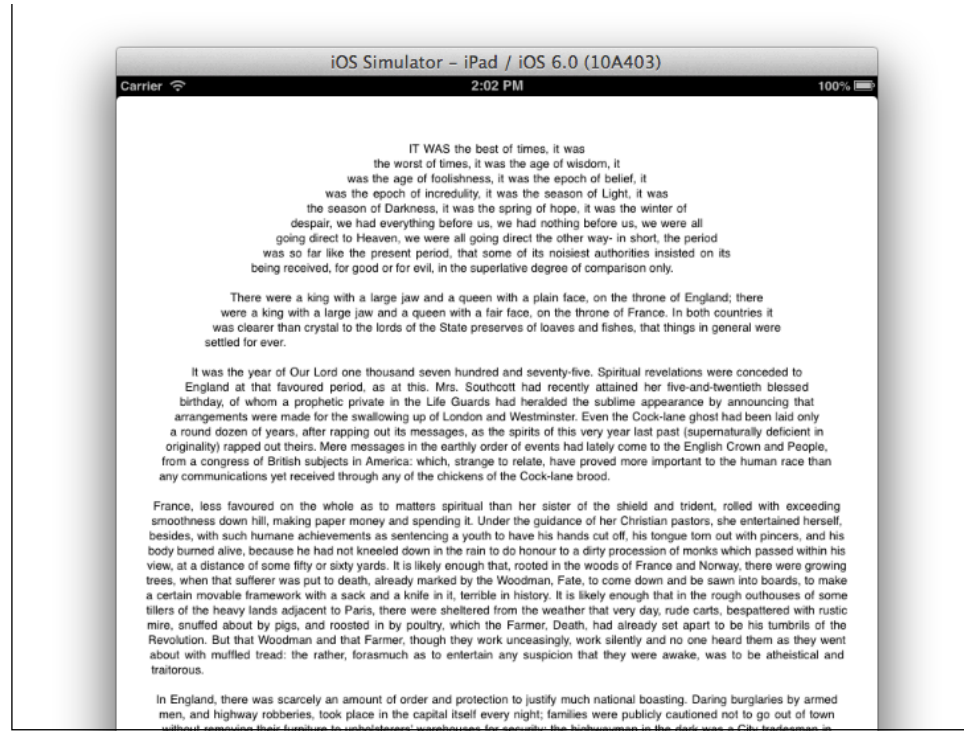
<build> Then we'll create a framesetter for the attributed string.

<build> Now, for each path in our list, we'll create a `CTFrame` with `CTFramesetterCreateFrame`. A frame is a block of drawable text. We pass a framesetter, a range within the string, and a path to fill, along with some options.

<build> It passes us back a frame object filled with glyphs, which we can then draw.

<build> We need to know where to start our next loop, so we can ask the frame what part of the string it drew, and adjust our offset accordingly.

And that's pretty much it. Framesetters are pretty easy to use if you already have a bunch of paths to fill.



They can even handle non-rectangular paths.

But there are a few tricky bits.

First, and I skipped over this to keep things simple, but framesetters draw upside down on iOS. It's not just that they lay out their coordinates from the lower-left corner. It's that the code I just showed you will actually draw the text upside down. There are a few ways to fix this.

# Flipping the Context

```
CGAffineTransform transform = CGAffineTransformMakeScale(1, -1);
CGAffineTransformTranslate(transform, 0, -self.bounds.size.height);
self.transform = transform;
```

```
#import <QuartzCore/QuartzCore.h>
...
[self.layer.flippedGeometry = YES;]
```

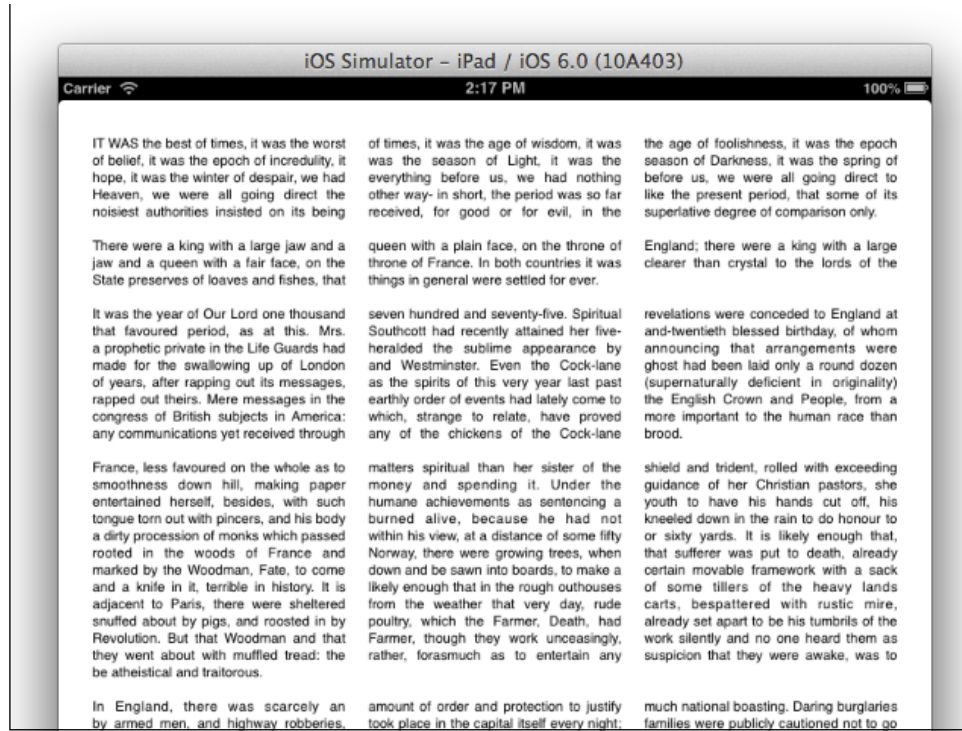
```
CGContextTranslateCTM(context, 0, ([self bounds]).size.height );
CGContextScaleCTM(context, 1.0, -1.0);
```

The easiest is to just flip the whole layer in `initWithFrame`:

<build> If you link with QuartzCore, you can just set `flippedGeometry`.

<build> You also could apply the transform in `drawRect`: if you wanted to. Just remember that `framesetter` moves down the Y axis as it lays out lines. This is very important to remember if you ever want to do hit testing or figure out where a specific character is on the screen. On iOS, you're going to have to spend a lot of time thinking upside down. Of course, on Mac, it all works the same.

OK, thinking back to the column example, I created three separate paths and handed them to the `framesetter` one at a time. What if I'd made one path that drew all three boxes with `moveTo`? ...



As you see, the framesetter fills the first column, then continues along to fill the second column and the third column before moving down a row. This is actually really useful if you want to cut holes in your text, but keep it in mind if you mean to create columns. You can't create columns in a single path.

Framesetters are just the highest level abstraction. They rely on other pieces to do a lot of the work. To demonstrate each piece, we're going to build up an example I call PinchText.

## PinchText Demo

It lays out the text, and wherever you touch, it pinches the characters towards it. To do this, we'll need to work out the location for every glyph, and that's definitely a job for Core Text.

**<DEMO: See PinchText>**

# Core Text

- Framesetter
- Typesetter
- Line
- Run
- Glyph

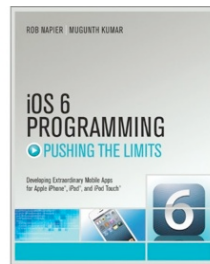
I know that was a bit of a whirlwind tour of Core Text. All the source is on my site, if you want to dig deeper. Hopefully it's introduced all the major pieces so you'll feel comfortable knowing what you should be looking for in the docs. You should recognize the full stack now: Framesetters for one-stop text layout into frames. Typesetters that handle line breaks. Lines that collect runs of glyphs.

Core Text is a really powerful system, and the fact that you can do things like this last example, letting it do most of the hard work while you just tweak it a little, is really fantastic. The main thing to remember is that Core Text is just a layout engine. It doesn't provide text editing, or text selection, or anything like that. It just lays out text and draws it.

The next lesson is to use the highest abstraction you can. If framesetter does what you need, use that. If you need to dig deeper, remember that each level can generally draw itself. So if you need to play around with lines, you can still use `CTLineDraw` to draw them rather than diving all the way down to hand-drawing glyphs. Hand-drawing glyphs is a last resort and can be a lot of work to handle all the possible cases. But it's really nice that it's available.



robnapier.net/cocoaconf  
robnapier@gmail.com  
@cocoaphony  
αrobnapier



iOS 6 Programming Pushing The Limits  
Chapter 26  
Dec 17, 2012

iosptl.com

That wraps up my talk on Rich Text, Core Text. We've covered the system from top to bottom, from UILabel to CGGlyph, and hopefully some of it will stick with you. You should now understand the language of rich text, from fonts and decorations to characters and glyphs. You should know your way around NSAttributedString, be able to tell a framesetter from a typesetter, and manage right to left text. It's been a full session, and I hope you enjoyed it.

All the example code is available at [robnapier.net](http://robnapier.net), along with links to other resources.

Any questions?