

Chapter 3

Deeper into text

Despite how sophisticated graphical user interfaces have become, text remains the backbone of most applications. Yet text is potentially one of the most complex visual objects because it carries baggage of hundreds of years of typography. The primary consideration is that text must be readable. This requires that text not be too small, yet text mustn't be so large that it hogs a lot of space on the screen.

For these reasons, the subject of text is continued in several subsequent chapters, most notably Chapter 5, "Dealing with sizes." Very often, Xamarin.Forms programmers define font characteristics in styles, which are the subject of Chapter 12.

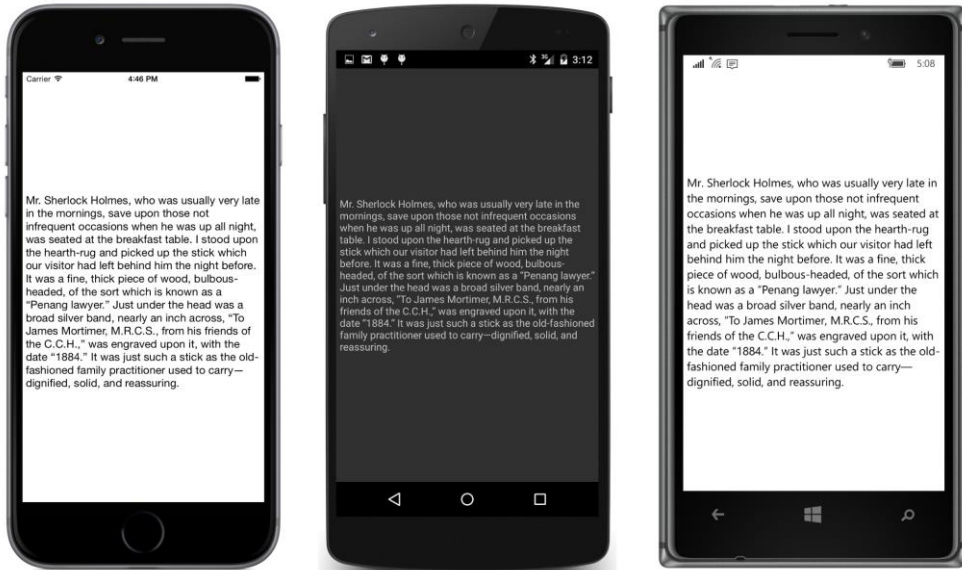
Wrapping paragraphs

Displaying a paragraph of text is as easy as displaying a single line of text. Just make the text long enough to wrap into multiple lines:

```
public class BaskervillesPage : ContentPage
{
    public BaskervillesPage()
    {
        Content = new Label
        {
            VerticalOptions = LayoutOptions.Center,
            Text =
                "Mr. Sherlock Holmes, who was usually very late in " +
                "the mornings, save upon those not infrequent " +
                "occasions when he was up all night, was seated at " +
                "the breakfast table. I stood upon the hearth-rug " +
                "and picked up the stick which our visitor had left " +
                "behind him the night before. It was a fine, thick " +
                "piece of wood, bulbous-headed, of the sort which " +
                "is known as a \u201CPenang lawyer.\u201D Just " +
                "under the head was a broad silver band, nearly an " +
                "inch across, \u201CTo James Mortimer, M.R.C.S., " +
                "from his friends of the C.C.H.,\u201D was engraved " +
                "upon it, with the date \u201C1884.\u201D It was " +
                "just such a stick as the old-fashioned family " +
                "practitioner used to carry\u2014dignified, solid, " +
                "and reassuring."
        };

        Padding = new Thickness(5, Device.OnPlatform(20, 5, 5), 5, 5);
    }
}
```

Notice the use of embedded Unicode codes for opened and closed “smart quotes” (\u201C and \u201D) and the em dash (\u2014). `Padding` has been set for 5 units around the page to avoid the text butting up against the edges of the screen, but the `VerticalOptions` property has been used as well to vertically center the entire paragraph on the page:



For this paragraph of text, setting `HorizontalOptions` to `Start`, `Center`, or `End` on iOS or Windows Phone will shift the entire paragraph horizontally slightly to the left, center, or right. (Android works a little differently for multiple lines of text.) The shifting is only slight because the width of the paragraph is the width of the longest line of text. Since word wrapping is governed by the page width (minus the padding), the paragraph likely occupies just slightly less width than the width available for it on the page.

But setting the `HorizontalTextAlignment` property of the `Label` has a much more profound effect: Setting this property affects the alignment of the individual lines. A setting of `TextAlignment.Center` will center all the lines of the paragraph, and `TextAlignment.Right` aligns them all at the right. You can use `HorizontalOptions` in addition to `HorizontalTextAlignment` to shift the entire paragraph slightly to the center or the right.

However, after you’ve set `VerticalOptions` to `Start`, `Center`, or `End`, any setting of `VerticalTextAlignment` has no effect.

`Label` defines a `LineBreakMode` property that you can set to a member of the `LineBreakMode` enumeration if you don’t want the text to wrap or to select truncation options.

There is no property to specify a first-line indent for the paragraph, but you can add one of your own with space characters of various types, such as the em space (Unicode \u2003).

You can display multiple paragraphs with a single `Label` view by ending each paragraph with one or more line feed characters (`\n`). However, a better approach is to use the string returned from the `Environment.NewLine` static property. This property returns `"\n"` on iOS and Android devices and `"\r\n"` on all Windows and Windows Phone devices. But rather than embedding line feed characters to create paragraphs, it makes more sense to use a separate `Label` view for each paragraph, as will be demonstrated in Chapter 4, "Scrolling the stack."

The `Label` class has lots of formatting flexibility. As you'll see shortly, properties defined by `Label` allow you to specify a font size or bold or italic text, and you can also specify different text formatting within a single paragraph.

`Label` also allows specifying color, and a little experimentation with color will demonstrate the profound difference between the `HorizontalOptions` and `VerticalOptions` properties and the `HorizontalTextAlignment` and `VerticalTextAlignment` properties.

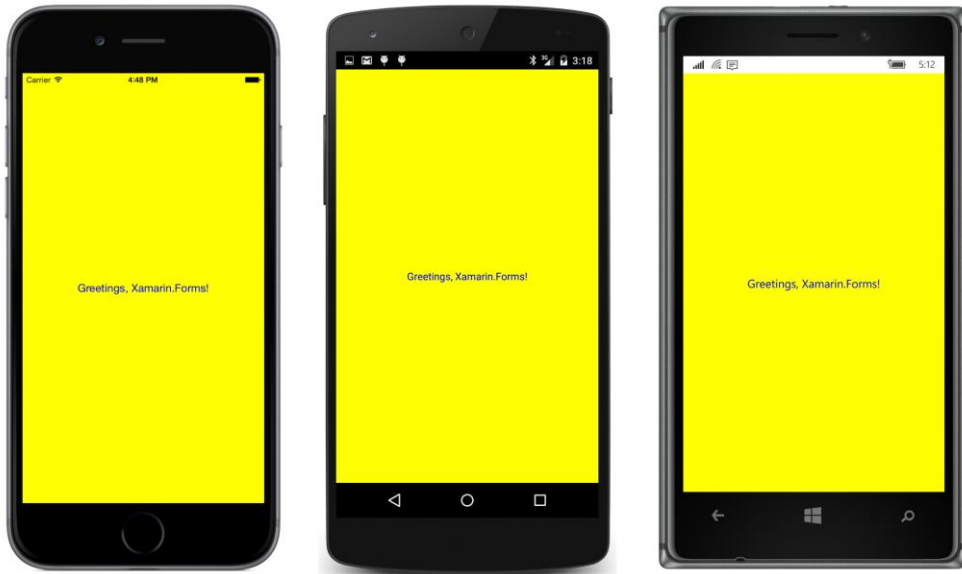
Text and background colors

As you've seen, the `Label` view displays text in a color appropriate for the device. You can override that behavior by setting two properties, named `TextColor` and `BackgroundColor`. `Label` itself defines `TextColor`, but it inherits `BackgroundColor` from `VisualElement`, which means that `Page` and `Layout` also have a `BackgroundColor` property.

You set `TextColor` and `BackgroundColor` to a value of type `Color`, which is a structure that defines 17 static fields for obtaining common colors. You can experiment with these properties with the **Greetings** program from the previous chapter. Here are two of these colors used in conjunction with `HorizontalTextAlignment` and `VerticalTextAlignment` to center the text:

```
public class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        Content = new Label
        {
            Text = "Greetings, Xamarin.Forms!",
            HorizontalTextAlignment = TextAlignment.Center,
            VerticalTextAlignment = TextAlignment.Center,
            BackgroundColor = Color.Yellow,
            TextColor = Color.Blue
        };
    }
}
```

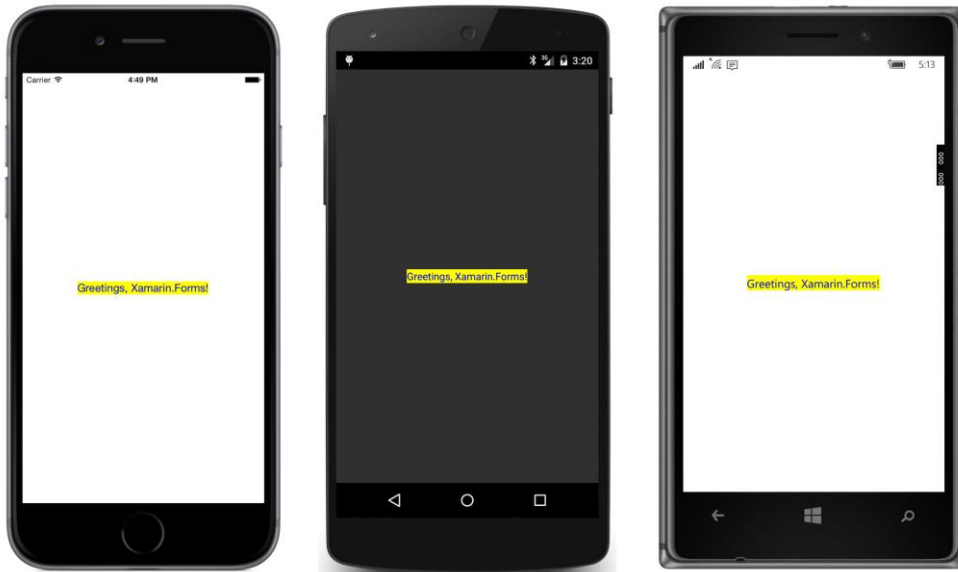
The result might surprise you. As these screenshots illustrate, the `Label` actually occupies the entire area of the page (including underneath the iOS status bar), and the `HorizontalTextAlignment` and `VerticalTextAlignment` properties position the text within that area:



In contrast, here's some code that colors the text the same but instead centers the text using the `HorizontalOptions` and `VerticalOptions` properties:

```
public class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        Content = new Label
        {
            Text = "Greetings, Xamarin.Forms!",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            BackgroundColor = Color.Yellow,
            TextColor = Color.Blue
        };
    }
}
```

Now the `Label` occupies only as much space as required for the text, and that's what's positioned in the center of the page:



The default value of `HorizontalOptions` and `VerticalOptions` is not `LayoutOptions.Start`, as the default appearance of the text might suggest. The default value is instead `LayoutOptions.Fill`. This is the setting that causes the `Label` to fill the page. The default `HorizontalTextAlignment` and `VerticalTextAlignment` value of `TextAlignment.Start` is what caused the text to be positioned at the upper-left in the first version of the **Greetings** program in the previous chapter.

You can combine various settings of `HorizontalOptions`, `VerticalOptions`, `HorizontalTextAlignment`, and `VerticalTextAlignment` for different effects.

You might wonder: What are the default values of the `TextColor` and `BackgroundColor` properties, because the default values result in different colors for the different platforms?

The default value of `TextColor` and `BackgroundColor` is actually a special color value named `Color.Default`, which does not represent a real color but instead is used to reference the text and background colors appropriate for the particular platform.

Let's explore color in more detail.

The Color structure

Internally, the `Color` structure stores colors in two different ways:

- As red, green, and blue (RGB) values of type `double` that range from 0 to 1. Read-only properties named `R`, `G`, and `B` expose these values.

- As hue, saturation, and luminosity values of type `double`, which also range from 0 to 1. These values are exposed with read-only properties named `Hue`, `Saturation`, and `Luminosity`.

The `Color` structure also supports an alpha channel for indicating degrees of opacity. A read-only property named `A` exposes this value, which ranges from 0 for transparent to 1 for opaque.

All the properties that define a color are read-only. In other words, once a `Color` value is created, it is immutable.

You can create a `Color` value in one of several ways. The three constructors are the easiest:

- `new Color(double grayShade)`
- `new Color(double r, double g, double b)`
- `new Color(double r, double g, double b, double a)`

Arguments can range from 0 to 1. `Color` also defines several static creation methods, including:

- `Color.FromRgb(double r, double g, double b)`
- `Color.FromRgb(int r, int g, int b)`
- `Color.FromRgba(double r, double g, double b, double a)`
- `Color.FromRgba(int r, int g, int b, int a)`
- `Color.FromHsla(double h, double s, double l, double a)`

The two static methods with integer arguments assume that the values range from 0 to 255, which is the customary representation of RGB colors. Internally, the constructor simply divides the integer values by 255.0 to convert to `double`.

Watch out! You might think that you're creating a red color with this call:

















```
Color.FromRgb(1, 0, 0)
```

However, the C# compiler will assume that these arguments are integers. The integer `FromRgb` method will be invoked, and the first argument will be divided by 255.0, with a result that is nearly zero. If you want to invoke the method that has `double` arguments, be explicit:

```
Color.FromRgb(1.0, 0, 0)
```

`Color` also defines static creation methods for a packed `uint` format and a hexadecimal format in a string, but these are used less frequently.

The `Color` structure also defines 17 public static read-only fields of type `Color`. In the table below, the integer RGB values that the `Color` structure uses internally to define these fields are shown together with the corresponding `Hue`, `Saturation`, and `Luminosity` values, somewhat rounded for purposes of clarity:

Color Fields	Color	Red	Green	Blue	Hue	Saturation	Luminosity
White		255	255	255	0	0	1.00
Silver		192	192	192	0	0	0.75
Gray		128	128	128	0	0	0.50
Black		0	0	0	0	0	0
Red		255	0	0	1.00	1	0.50
Maroon		128	0	0	1.00	1	0.25
Yellow		255	255	0	0.17	1	0.50
Olive		128	128	0	0.17	1	0.25
Lime		0	255	0	0.33	1	0.50
Green		0	128	0	0.33	1	0.25
Aqua		0	255	255	0.50	1	0.50
Teal		0	128	128	0.50	1	0.25
Blue		0	0	255	0.67	1	0.50
Navy		0	0	128	0.67	1	0.25
Pink		255	102	255	0.83	1	0.70
Fuchsia		255	0	255	0.83	1	0.50
Purple		128	0	128	0.83	1	0.25

With the exception of `Pink`, you might recognize these as the color names supported in HTML. An 18th public static read-only field is named `Transparent`, which has `R`, `G`, `B`, and `A` properties all set to zero.

When people are given an opportunity to interactively formulate a color, the HSL color model is often more intuitive than RGB. The `Hue` cycles through the colors of the visible spectrum (and the rainbow) beginning with red at 0, green at 0.33, blue at 0.67, and back to red at 1.

The `Saturation` indicates the degree of the hue in the color, ranging from 0, which is no hue at all and results in a gray shade, to 1 for full saturation.

The `Luminosity` is a measure of lightness, ranging from 0 for black to 1 for white.

Color-selection programs in Chapter 15, “The interactive interface,” let you explore the RGB and HSL models more interactively.

The `Color` structure includes several interesting instance methods that allow creating new colors that are modifications of existing colors:

- `AddLuminosity(double delta)`
- `MultiplyAlpha(double alpha)`
- `WithHue(double newHue)`
- `WithLuminosity(double newLuminosity)`
- `WithSaturation(double newSaturation)`

Finally, `Color` defines two special static read-only properties of type `Color`:

- `Color.Default`
- `Color.Accent`

The `Color.Default` property is used extensively within `Xamarin.Forms` to define the default color of views. The `VisualElement` class initializes its `BackgroundColor` property to `Color.Default`, and the `Label` class initializes its `TextColor` property as `Color.Default`.

However, `Color.Default` is a `Color` value with its `R`, `G`, `B`, and `A` properties all set to `-1`, which means that it's a special "mock" value that means nothing in itself but indicates that the actual value is platform specific.

For `Label` and `ContentPage` (and most classes that derive from `VisualElement`), the `BackgroundColor` setting of `Color.Default` means transparent. The background color you see on the screen is the background color of the page. The `BackgroundColor` property of the page has a default setting of `Color.Default`, but that value means something different on the various platforms. The meaning of `Color.Default` for the `TextColor` property of `Label` is also device dependent.

Here are the default color schemes implied by the `BackgroundColor` of the page and the `TextColor` of the `Label`:

Platform	Color Scheme
iOS	Dark text on a light background
Android	Light text on a dark background
UWP	Dark text on a light background
Windows 8.1	Light text on a dark background
Windows Phone 8.1	Light text on a dark background

On Android, Windows, and Windows Phone devices, you can change this color scheme for your application. See the next section.

You have a couple of possible strategies for working with color: You can choose to do your `Xamarin.Forms` programming in a very platform-independent manner and avoid making any assumptions about the default color scheme of any phone. Or, you can use your knowledge about the color schemes of the various platforms and use `Device.OnPlatform` to specify platform-specific colors.

But don't try to just ignore all the platform defaults and explicitly set all the colors in your application to your own color scheme. This probably won't work as well as you hope because many views use other colors that relate to the color theme of the operating system but that are not exposed through `Xamarin.Forms` properties.

One straightforward option is to use the `Color.Accent` property for an alternative text color. On the iPhone and Android platforms, this is a color that is visible against the default background but is not the default text color. On the Windows platforms, it's a color selected by the user as part of the color theme.

You can make text semitransparent by setting `TextColor` to a `Color` value with an `A` property less than 1. However, if you want a semitransparent version of the default text color, use the `Opacity` property of the `Label` instead. This property is defined by the `VisualElement` class and has a default value of 1. Set it to values less than 1 for various degrees of transparency.

Changing the application color scheme

When targeting your application for Android, Windows, and Windows Phone, it is possible to change the color scheme for the application. In this case, the settings of `Color.Default` for the `BackgroundColor` of the `ContentPage` and the `TextColor` property of the `Label` will have different meanings.

There are several ways to set color schemes in Android, but the simplest requires only a single attribute setting in the `AndroidManifest.xml` file in the **Properties** folder of the Android project. That file normally looks like this:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
  <uses-sdk android:minSdkVersion="15" />
  <application>
  </application>
</manifest>
```

Add the following attribute to the `application` tag:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
  <uses-sdk android:minSdkVersion="15" />
  <application android:theme="@style/android:Theme.Holo.Light">
  </application>
</manifest>
```

Now your Android application will display dark text on a light background.

For the three Windows and Windows Phone projects, you'll need to change the `App.xaml` file located in the particular project.

In the **UWP** project, the default `App.xaml` file looks like this:

```
<Application
  x:Class="Baskervilles.UWP.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Baskervilles.UWP"
  RequestedTheme="Light">

</Application>
```

That `RequestedTheme` attribute is what gives the UWP application a color scheme of dark text on a light background. Change it to `Dark` for light text on a dark background. Remove the `RequestedTheme` attribute entirely to allow the user's setting to determine the color scheme.

The `App.xaml` file for the Windows Phone 8.1 and Windows 8.1 projects is similar, but the `RequestedTheme` attribute is not included by default. Here's the `App.xaml` file in the **WinPhone** project:

```
<Application
  x:Class="Baskervilles.WinPhone.App"
```

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
xmlns:local="using:Baskervilles.WinPhone">
```

```
</Application>
```

By default, the color scheme is determined by the user's setting. You can include a `RequestedTheme` attribute and set it to `Light` or `Dark` to override the user's preference and take control of the color scheme.

By setting `RequestedTheme` on your Windows Phone and Windows projects, your application should have complete knowledge of the underlying color schemes on all the platforms.

Font sizes and attributes

By default, the `Label` uses a system font defined by each platform, but `Label` also defines several properties that you can use to change this font. `Label` is one of only two classes with these font-related properties; `Button` is the other.

The properties that let you change this font are:

- `FontFamily` of type `string`
- `FontSize` of type `double`
- `FontAttributes` of type `FontAttributes`, an enumeration with three members: `None`, `Bold`, and `Italic`.

There is also a `Font` property and corresponding `Font` structure, but this is deprecated and should not be used.

The hardest of these to use is `FontFamily`. In theory you can set it to a font family name such as "Times Roman," but it will work only if that particular font family is supported on the particular platform. For this reason, you'll probably use `FontFamily` in connection with `Device.OnPlatform`, and you'll need to know each platform's supported font family names.

The `FontSize` property is a little awkward as well. You need a number that roughly indicates the height of the font, but what numbers should you use? This is a thorny issue, and for that reason, it's relegated to Chapter 5, "Dealing with sizes," when the tools to pick a good font size will become available.

Until then, however, the `Device` class helps out with a static method called `GetNamedSize`. This method requires a member of the `NamedSize` enumeration:

- `Default`
- `Micro`

- Small
- Medium
- Large

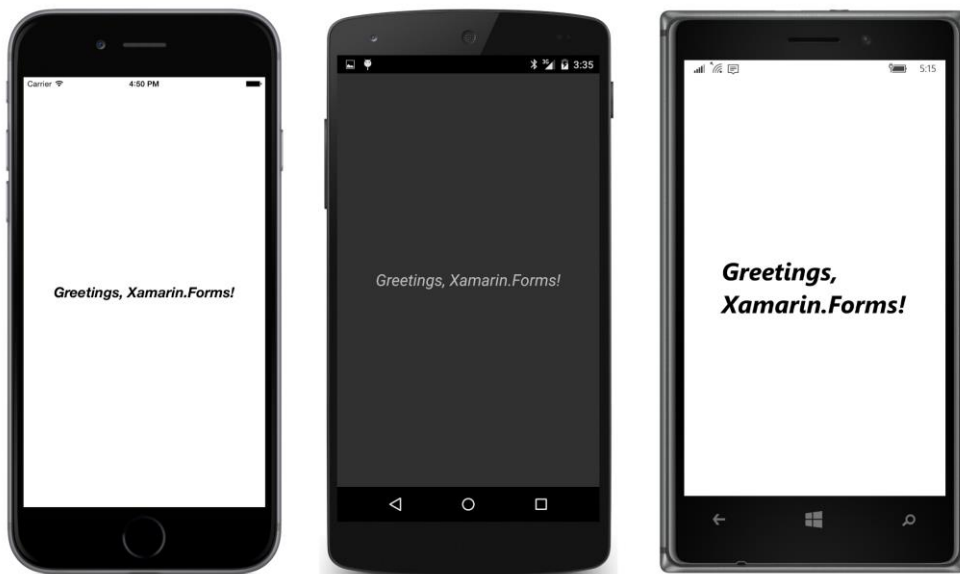
`GetNamedSize` also requires the type of the class that you're sizing with this font size, and that argument will be either `typeof(Label)` or `typeof(Button)`. You can also use an instance of `Label` or `Button` itself rather than the `Type`, but this option is often less convenient.

As you'll see later in this chapter, the `NamedSize.Medium` member does not necessarily return the same size as `NamedSize.Default`.

`FontAttributes` is the least complicated of the three font-related properties to use. You can specify `Bold` or `Italic` or both, as this little snippet of code (adapted from the **Greetings** program from the previous chapter) demonstrates:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        Content = new Label
        {
            Text = "Greetings, Xamarin.Forms!",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            FontAttributes = FontAttributes.Bold | FontAttributes.Italic
        };
    }
}
```

Here it is on the three platforms:



The Windows 10 Mobile screen is not quite wide enough to display the text in a single line.

Formatted text

As you've seen, `Label` has a `Text` property that you can set to a string. But `Label` also has an alternative `FormattedText` property that constructs a paragraph with nonuniform formatting.

The `FormattedText` property is of type `FormattedString`, which has a `Spans` property of type `IList`, a collection of `Span` objects. Each `Span` object is a uniformly formatted chunk of text that is governed by six properties:

- `Text`
- `FontFamily`
- `FontSize`
- `FontAttributes`
- `ForegroundColor`
- `BackgroundColor`

Here's one way to instantiate a `FormattedString` object and then add `Span` instances to its `Spans` collection property:

```
public class VariableFormattedTextPage : ContentPage
{
```

```

public VariableFormattedTextPage()
{
    FormattedString formattedString = new FormattedString();

    formattedString.Spans.Add(new Span
    {
        Text = "I "
    });

    formattedString.Spans.Add(new Span
    {
        Text = "love",
        FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
        FontAttributes = FontAttributes.Bold
    });

    formattedString.Spans.Add(new Span
    {
        Text = " Xamarin.Forms!"
    });

    Content = new Label
    {
        FormattedText = formattedString,
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.Center,
        FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label))
    };
}
}

```

As each `Span` is created, it is directly passed to the `Add` method of the `Spans` collection. Notice that the `Label` is given a `FontSize` of `NamedSize.Large`, and the `Span` with the `Bold` setting is also explicitly given that same size. When a `Span` is given a `FontAttributes` setting, it does not inherit the `FontSize` setting of the `Label`.

Alternatively, it's possible to initialize the contents of the `Spans` collection by following it with a pair of curly braces. Within these curly braces, the `Span` objects are instantiated. Because no method calls are required, the entire `FormattedString` initialization can occur within the `Label` initialization:

```

public class VariableFormattedTextPage : ContentPage
{
    public VariableFormattedTextPage()
    {
        Content = new Label
        {
            FormattedText = new FormattedString
            {
                Spans =
                {
                    new Span
                    {
                        Text = "I "
                    }
                }
            }
        }
    }
}

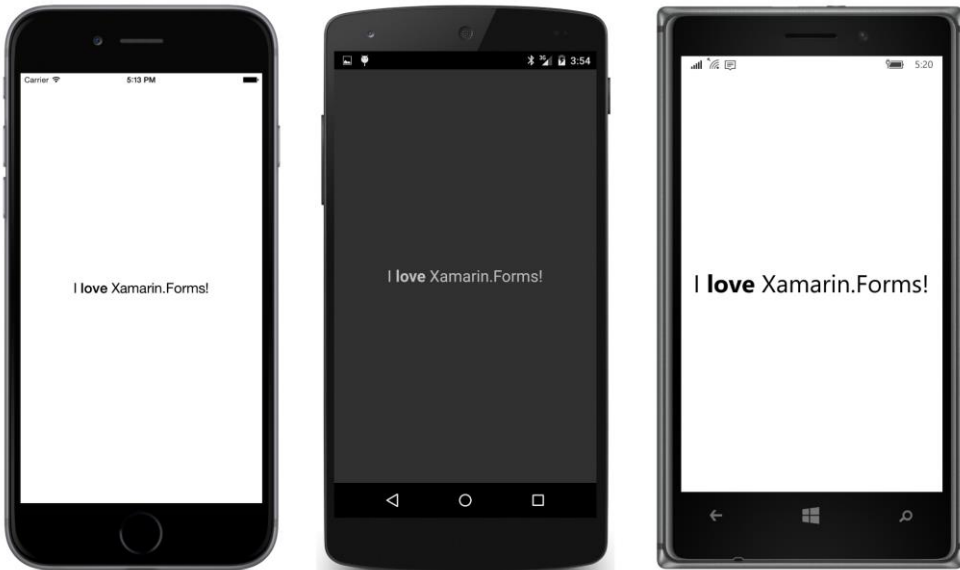
```

```

    },
    new Span
    {
        Text = "love",
        FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
        FontAttributes = FontAttributes.Bold
    },
    new Span
    {
        Text = " Xamarin.Forms!"
    }
}
},
HorizontalOptions = LayoutOptions.Center,
VerticalOptions = LayoutOptions.Center,
FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label))
};
}
}

```

This is the version of the program that you'll see in the collection of sample code for this chapter. Regardless of which approach you use, here's what it looks like:



You can also use the `FormattedText` property to embed italic or bold words within an entire paragraph, as the **VariableFormattedParagraph** program demonstrates:

```

public class VariableFormattedParagraphPage : ContentPage
{
    public VariableFormattedParagraphPage()
    {

```

```

Content = new Label
{
    FormattedText = new FormattedString
    {
        Spans =
        {
            new Span
            {
                Text = "\u2003There was nothing so "
            },
            new Span
            {
                Text = "very",
                FontAttributes = FontAttributes.Italic
            },
            new Span
            {
                Text = " remarkable in that; nor did Alice " +
                    "think it so "
            },
            new Span
            {
                Text = "very",
                FontAttributes = FontAttributes.Italic
            },
            new Span
            {
                Text = " much out of the way to hear the " +
                    "Rabbit say to itself \u2018Oh " +
                    "dear! Oh dear! I shall be too late!" +
                    "\u2019 (when she thought it over " +
                    "afterwards, it occurred to her that " +
                    "she ought to have wondered at this, " +
                    "but at the time it all seemed quite " +
                    "natural); but, when the Rabbit actually "
            },
            new Span
            {
                Text = "took a watch out of its waistcoat-pocket",
                FontAttributes = FontAttributes.Italic
            },
            new Span
            {
                Text = ", and looked at it, and then hurried on, " +
                    "Alice started to her feet, for it flashed " +
                    "across her mind that she had never before " +
                    "seen a rabbit with either a waistcoat-" +
                    "pocket, or a watch to take out of it, " +
                    "and, burning with curiosity, she ran " +
                    "across the field after it, and was just " +
                    "in time to see it pop down a large " +
                    "rabbit-hole under the hedge."
            }
        }
    }
}

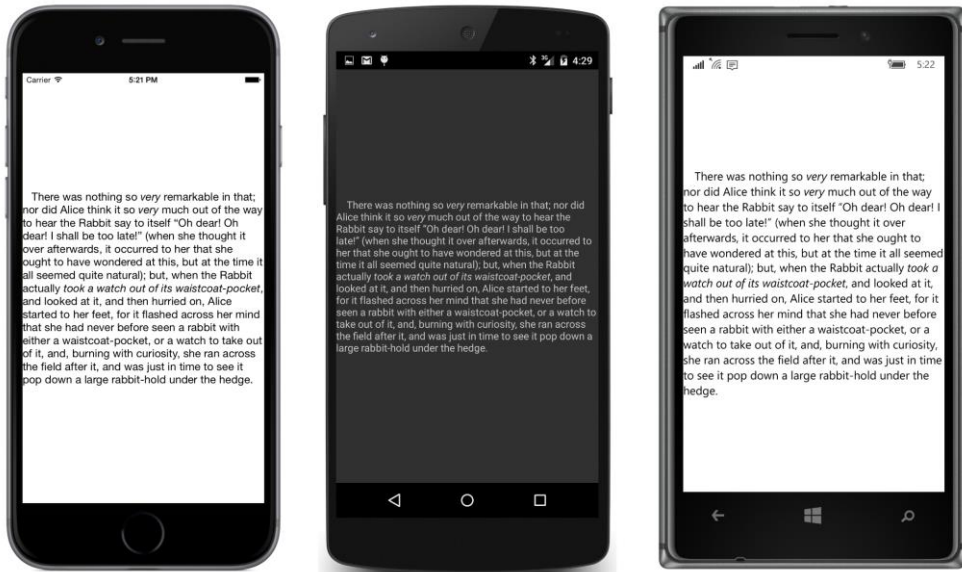
```

```

    },
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center
};
}
}
}

```

The paragraph begins with an em space (Unicode \u2003) and contains so-called smart quotes (\u201C and \u201D), and several words are italicized:



You can persuade a single `Label` to display multiple lines or paragraphs with the insertion of end-of-line characters. This is demonstrated in the **NamedFontSizes** program. Multiple `Span` objects are added to a `FormattedString` object in a `foreach` loop. Each `Span` object uses a different `NamedFont` value and also displays the actual size returned from `Device.GetNamedSize`:

```

public class NamedFontSizesPage : ContentPage
{
    public NamedFontSizesPage()
    {
        FormattedString formattedString = new FormattedString();
        NamedSize[] namedSizes =
        {
            NamedSize.Default, NamedSize.Micro, NamedSize.Small,
            NamedSize.Medium, NamedSize.Large
        };
        foreach (NamedSize namedSize in namedSizes)
        {
            double fontSize = Device.GetNamedSize(namedSize, typeof(Label));

```



```

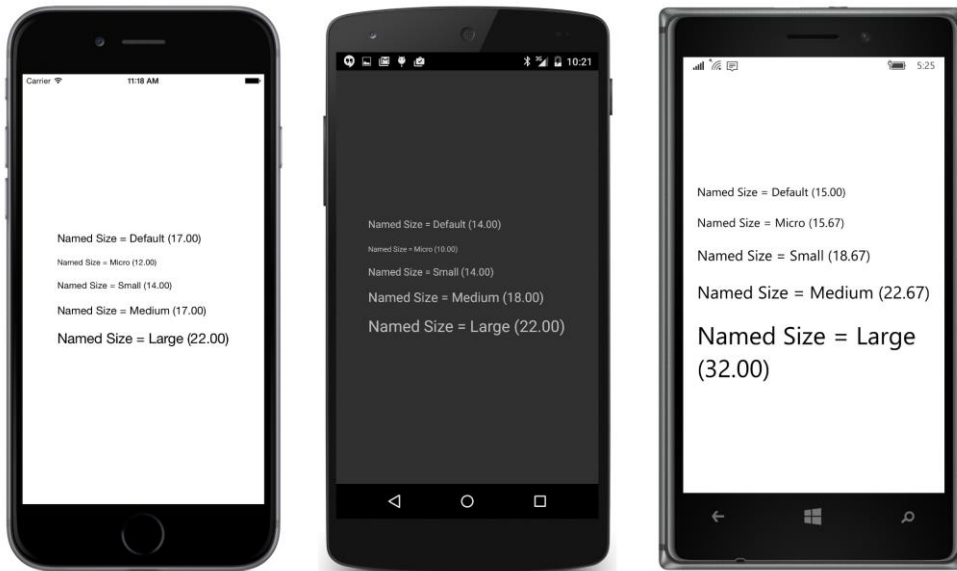
formattedString.Spans.Add(new Span
{
    Text = String.Format("Named Size = {0} ({1:F2})",
        namedSize, fontSize),
    FontSize = fontSize
});

if (namedSize != namedSizes.Last())
{
    formattedString.Spans.Add(new Span
    {
        Text = Environment.NewLine + Environment.NewLine
    });
}
}

Content = new Label
{
    FormattedText = formattedString,
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center
};
}
}

```

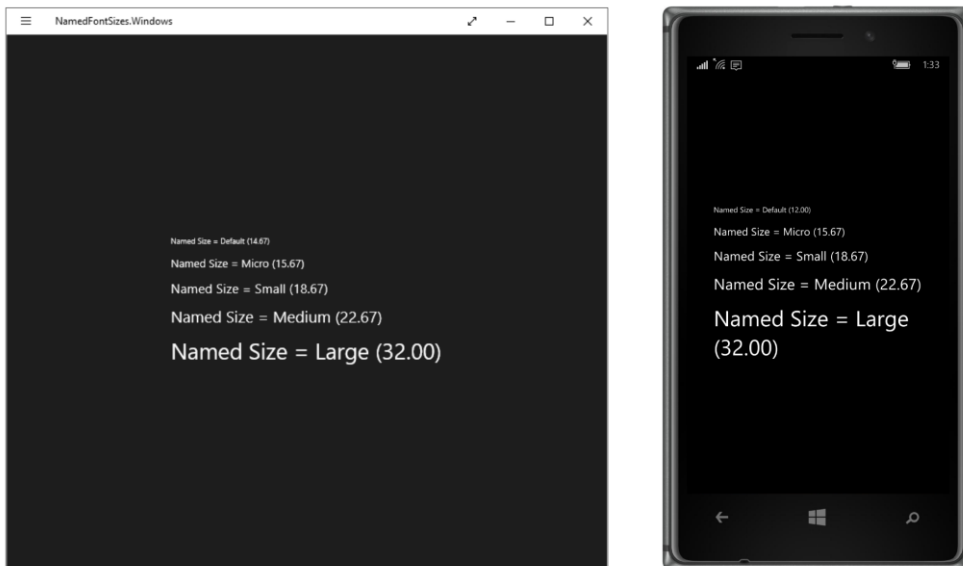
Notice that a separate `Span` contains the two platform-specific end-of-line strings to space the individual lines. This ensures that the line spacing is based on the default font size rather than the font size just displayed:



These are not pixel sizes! As with the height of the iOS status bar, it's best to refer to these sizes only vaguely as some kind of "units." Some additional clarity is coming in Chapter 5.

The `Default` size is generally chosen by the operating system, but the other sizes were chosen by the Xamarin.Forms developers. On iOS, `Default` is the same as `Medium`, but on Android `Default` is the same as `Small`, and on Windows 10 Mobile, `Default` is smaller than `Micro`.

The sizes on the iPad and Windows 10 are the same as the iPhone and Windows 10 Mobile, respectively. However, the sizes on the Windows 8.1 and Windows Phone 8.1 platforms show more of discrepancy:



Of course, the use of multiple `Span` objects in a single `Label` is not a good way to render multiple paragraphs of text. Moreover, text often has so many paragraphs that it must be scrolled. This is the job for the next chapter and its exploration of `StackLayout` and `ScrollView`.