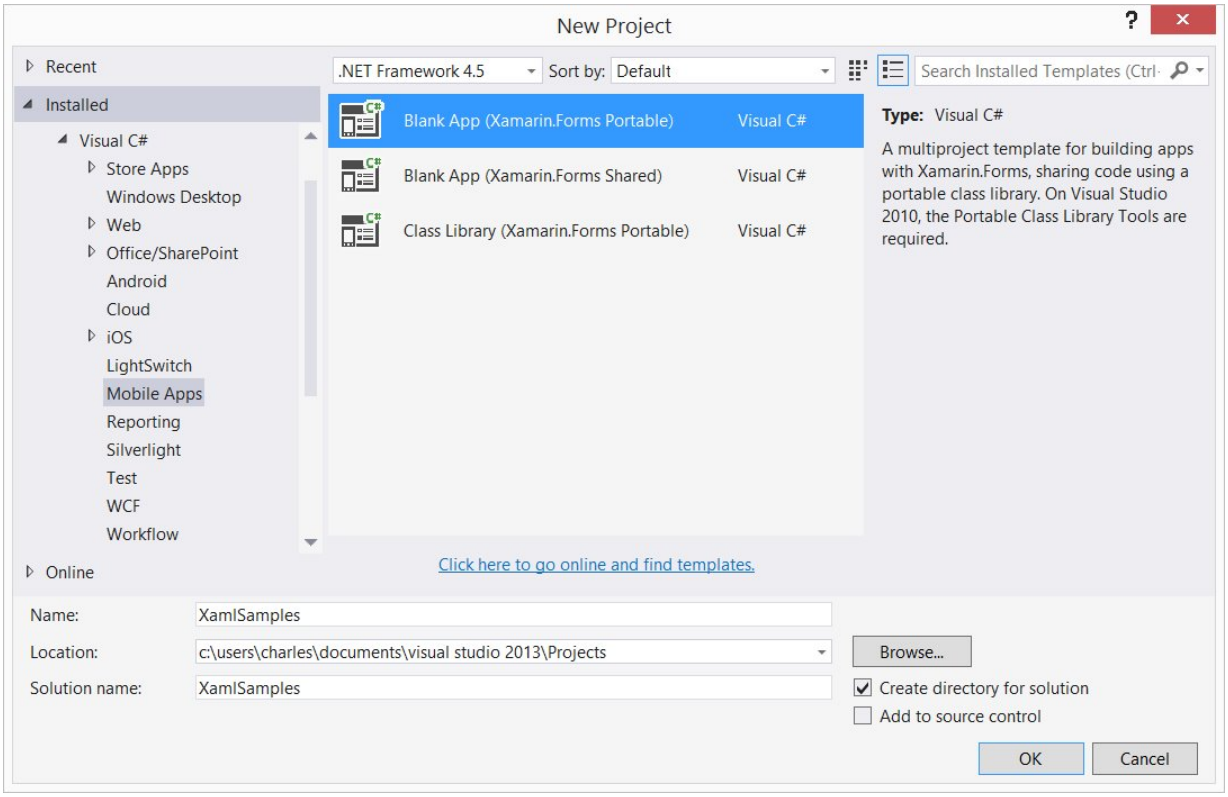
Part 1. Getting Started with XAML

Defining a Page with Elements and Attributes

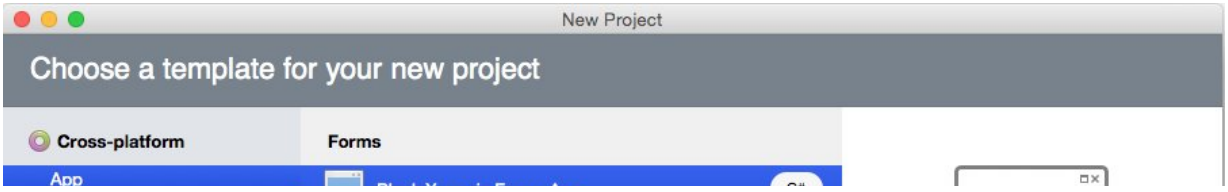
Creating the Page

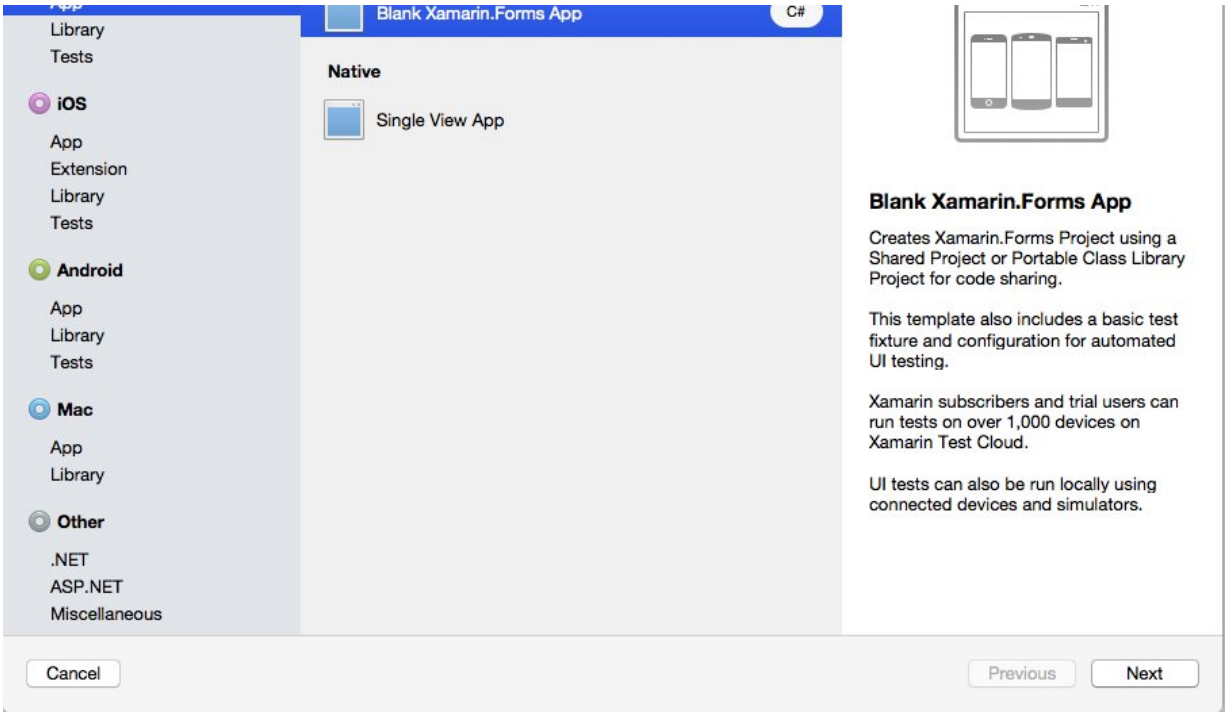
To begin editing your first XAML file, use Visual Studio or Xamarin Studio to create a new Xamarin.Forms solution.

In Visual Studio, select File > New > Project from the menu. In the New Project dialog, select Visual C# > Mobile Apps at the left, and then Blank App (Xamarin.Forms Portable) from the list in the center - this creates a PCL-based solution that supports XAML.

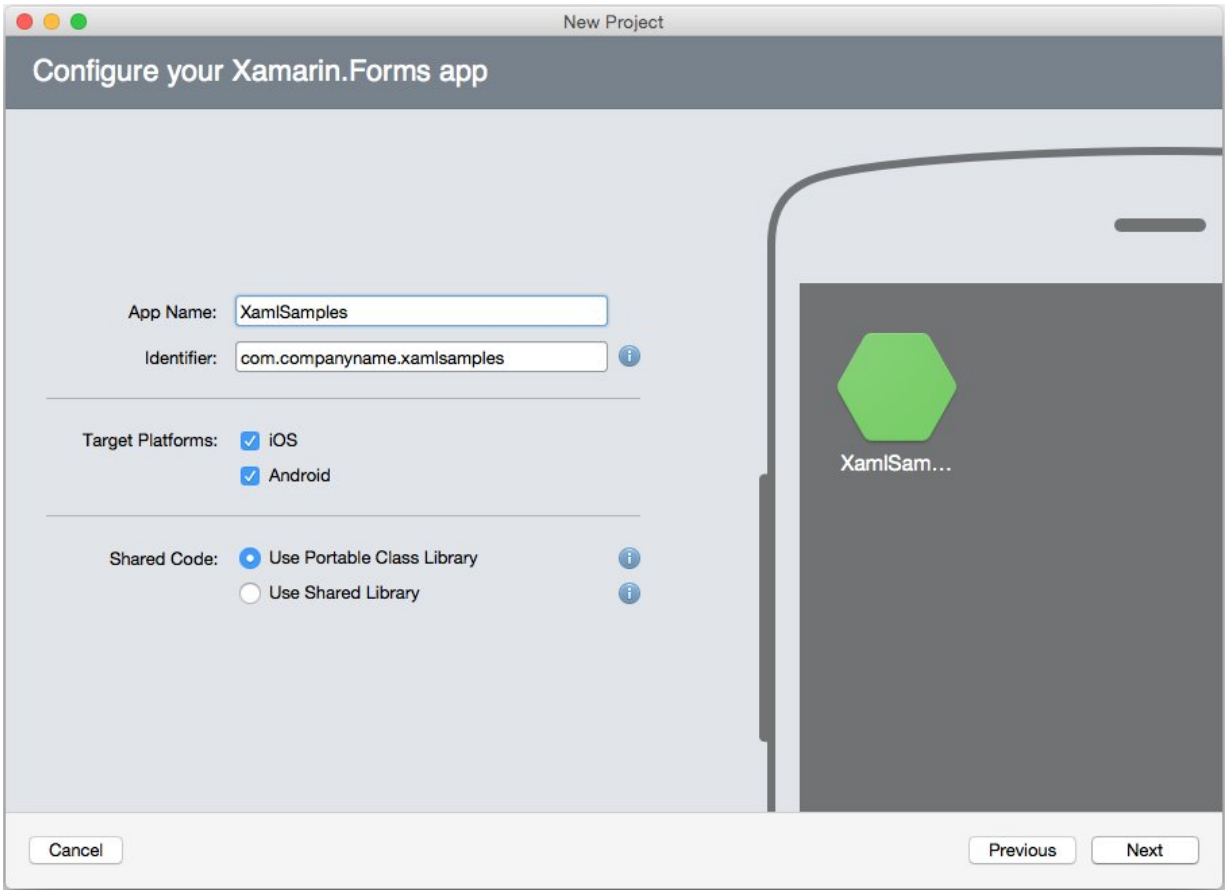


In Xamarin Studio, select File > New Solution from the menu. In the New Solution dialog, select Cross Platform > App at the left, and Blank Xamarin.Forms App from the template list.





On the following screen configure your Xamarin.Forms app by giving it a name, and select **Use Portable Class Library** for the shared code option - this creates a PCL-based solution that supports XAML:



Select a location for the solution and give it a name of XamlSamples (or whatever).

Visual Studio creates four projects: XamlSamples.Android, XamlSamples.iOS, XamlSamples.WinPhone, and a shared Portable Class Library (PCL) project named simply XamlSamples.

When using Xamarin Studio on the PC, only the XamlSamples.Android and XamlSamples projects are created. Xamarin Studio on the Mac also creates the XamlSamples.iOS project.

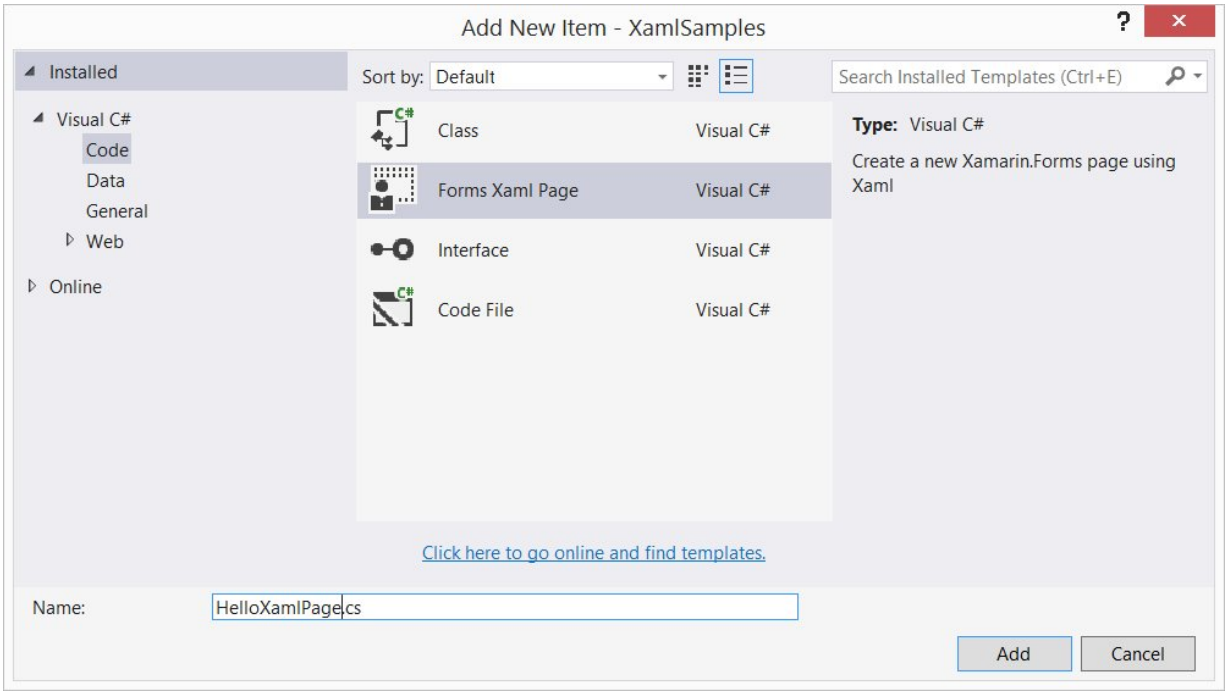
After creating the Xamarin.Forms solution, you might want to test your development environment by selecting the various platform projects as the solution startup project, and building and deploying the simple application created by the project template on either phone emulators or real devices.

Unless you need to write platform-specific code, the shared XamlSamples PCL project is where you'll be spending virtually all of your programming time, and this article will not venture outside of that project.

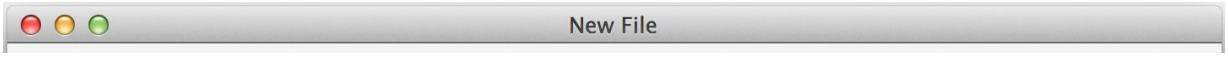
XAML can play a role in a Xamarin.Forms application in several ways, but undoubtedly the most common is for defining the visual contents of an entire page, which is usually a class derived from `ContentPage`.

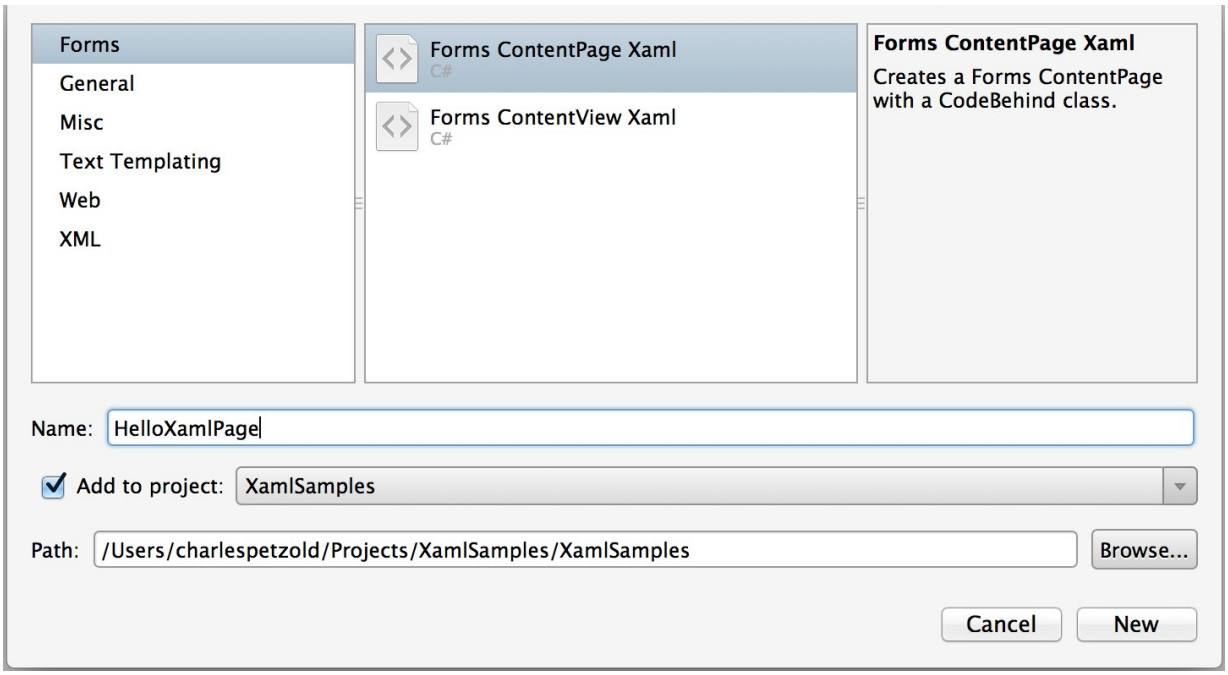
You'll now want to add a XAML-based `ContentPage` to the XamlSamples project.

In Visual Studio, right-click the XamlSamples project and select Add > New Item. In the Add New Item dialog, select Visual C# > Code at the left, and Forms Xaml Page from the list.



In Xamarin Studio, from the XamlSamples drop-down menu, select Add > New File. In the New File dialog, select Forms at the left, and Forms ContentPage Xaml—not Forms ContentView Xaml—from the list.





Name it **HelloXamlPage**.

Two new files are created in the XamlSamples project: The first is a XAML file named **HelloXamlPage.xaml**. The second file is displayed indented underneath it; this is a C# code file with the unusual name **HelloXamlPage.xaml.cs**. The names of these two files reveal that they are intimately related. The C# file is often referred to as the *code-behind* file of the XAML file.

Both **HelloXamlPage.xaml** and **HelloXamlPage.xaml.cs** contribute to the definition of a class named `HelloXamlPage` that derives from `ContentPage`.

Anatomy of a XAML Class

In HelloXamlPage.xaml, the first thing you'll want to do is delete anything that appears between the start and end tags so the file looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.HelloXamlPage">
</ContentPage>
```

The two XML namespace (`xmlns`) declarations refer to URIs, the first seemingly on Xamarin's web site and the second on Microsoft's. Don't bother checking what those URIs point to. There's nothing there. They

are simply URIs owned by Xamarin and Microsoft, and they basically function as version identifiers.

The first XML namespace declaration means that tags defined within the XAML file with no prefix refer to classes in Xamarin.Forms, for example `ContentPage`. The second namespace declaration defines a prefix of `x`. This is used for several elements and attributes that are intrinsic to XAML itself and which (in theory) are supported by all implementations of XAML. However, these elements and attributes are slightly different depending on the year embedded in the URI. Xamarin.Forms supports the 2009 XAML specification, but not all of it.

Immediately after the `x` prefix is declared, that prefix is used for an attribute named `Class`. Because the use of this `x` prefix is pretty much universal in XAML files, XAML attributes such as `Class` are almost always referred to as `x:Class`.

The `x:Class` attribute specifies a fully qualified .NET class name: the `HelloXamlPage` class in the `XamlSamples` namespace. This means that this XAML file defines a new class named `HelloXamlPage` in the `XamlSamples` namespace that derives from `ContentPage`—the tag in which the `x:Class` attribute appears.

The `x:Class` attribute can only appear in the root element of a XAML file to define a derived C# class. This is the only new class defined in the XAML file. Everything else that appears in the XAML file is instead simply instantiated and initialized.

The **HelloXamlPage.xaml.cs** code-behind file looks like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace XamlSamples
{
    public partial class HelloXamlPage
    {
        public HelloXamlPage()
        {
            InitializeComponent();
        }
    }
```

```
}
```

Notice the `partial` class definition. Although it's not indicated explicitly in this class definition, `HelloXamlPage` derives from `ContentPage`.

But there seems to be something missing. Shouldn't there be another C# file with another partial class definition for `HelloXamlPage`? And what is that `InitializeComponent` method? You'll see those shortly.

Look for the `App` class in the XamlSamples project, you'll want to remove some of the existing code and use the `App` constructor to set `MainPage` to an instance of `HelloXamlPage`:

```
namespace XamlSamples
{
    public class App : Xamarin.Forms.Application
    {
        public App ()
        {
            MainPage = new HelloXamlPage();
        }
    }
}
```

The project can now be compiled for any of the three platforms, but the page is entirely blank.

During the build-deploy-run cycle, the XAML file is parsed twice. It is parsed first during the build process. The entire XAML file is also bound into the Portable Code Library DLL, and it is parsed again at runtime.

During the build step, a C# code file is generated from the XAML file. If you look in the **XamlSamples\XamlSamples\obj\Debug** directory, you'll find a file named **HelloXamlPage.xaml.g.cs**. The 'g' stands for generated. Here is that file (but without the comments normally at the top indicating that the file shouldn't be changed):

```
namespace XamlSamples {
    using System;
    using Xamarin.Forms;
    using Xamarin.Forms.Xaml;

    public partial class HelloXamlPage : ContentPage {
```

```
        private void InitializeComponent() {
            this.LoadFromXaml(typeof(HelloXamlPage));
        }
    }
}
```

This is the other partial class definition of `HelloXamlPage`, and it explicitly indicates that the base class is `ContentPage`. This class definition also contains the definition of the `InitializeComponent` method called from the `HelloXamlPage` constructor. During the build process, this code file is first generated from the XAML file, and then the two partial class definitions of `HelloXamlPage` are compiled together.

At runtime, code in the particular platform project calls the static `App.GetMainPage` method to get the initial page. This method instantiates `HelloXamlPage`. The constructor of that class calls `InitializeComponent`, which then calls the `LoadFromXaml` method that extracts the entire XAML file from the Portable Class Library and parses it, instantiates and initializes all the objects defined in the XAML file, connects them all together in parent-child relationships, attaches event handlers defined in code to events set in the XAML file, and sets the resultant tree of objects as the content of the page.

Although you normally don't need to spend much time with generated code files, sometimes runtime exceptions are raised on code in the generated files, so you should be familiar with them.

The parsing of the XAML file during the build process is rudimentary compared with the later parsing at runtime. The parsing at build time reveals XML syntax errors but not incorrectly spelled elements or attributes. Problems of that sort will only be detected at runtime. Fortunately, the runtime exceptions usually provide sufficient information to locate and fix the problem.

Xamarin Studio often displays the exception message in a popup window. In Visual Studio, when running an iOS or Android application, XAML exceptions are generally raised in the generated code file in the `LoadFromXaml` call. Check the Local window for an `Instance` object of type `Xamarin.Forms.Xaml.XamlParseException`. The `Message` property usually indicates the problem with line and column numbers, but you might need to check an inner exception to get this information.

When running the program on Windows Phone, a XAML exception generally causes the App.xaml.cs file in the Windows Phone project to break in the `RootFrame_NavigationFailed` call. The `$exception` object indicates the problem.

Setting Page Content

A single-page Xamarin.Forms application generally contains a class derived from `ContentPage`. The

`Content` property of this class is generally set to a single view or a layout with child views. In XAML, a view (for example a `Label`) can be implicitly set to the `Content` property of the page by being placed between the start and end `ContentPage` tags:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.HelloXamlPage"
             Title="Hello XAML Page"
             Padding="10, 40, 10, 10">

  <Label Text="Hello, XAML!"
         VerticalOptions="Start"
         HorizontalTextAlignment="Center"
         Rotation="-15"
         IsVisible="true"
         FontSize="Large"
         FontAttributes="Bold"
         TextColor="Aqua" />

</ContentPage>
```

At this time, the relationship between classes, properties, and XML should be obvious: A Xamarin.Forms class (such as `ContentPage` or `Label`) appears in the XAML file as an XML element, and properties of that class—including `Title` and `Padding` on `ContentPage` and seven properties of `Label`—appear as XML attributes.

Many shortcuts exist to set the values of these properties. Some properties are basic data types: For example, the `Title` and `Text` properties are of type `String`, `Rotation` is of type `Double`, and `IsVisible` (which is `true` by default and is set here only for illustration) is of type `Boolean`.

The `HorizontalTextAlignment` property is of type `TextAlignment`, which is an enumeration. For a property of any enumeration type, all you need supply is a member name.

For properties of more complex types, however, converters are used for parsing the XAML. These are classes in Xamarin.Forms that derive from `TypeConverter`. Many are public classes but some are not. For this particular XAML file, several of these classes play a role behind the scenes:

- `ThicknessTypeConverter` for the `Padding` property

- `LayoutOptionsConverter` for the `VerticalOptions` property
- `FontSizeConverter` for the `FontSize` property
- `ColorTypeConverter` for the `TextColor` property

These converters essentially govern the allowable syntax of the property settings.

The `ThicknessTypeConverter` can handle one, two, or four numbers separated by commas. If one number is supplied, it applies to all four sides. With two numbers, the first is left and right padding, and the second is top and bottom. Four numbers are in the order left, top, right, and bottom.

The `LayoutOptionsConverter` can convert the names of public static fields of the `LayoutOptions` structure to values of type `LayoutOptions`.

The `FontSizeConverter` can handle a `NamedSize` member or a numeric font size.

The `ColorTypeConverter` accepts the names of public static fields of the `Color` structure or hexadecimal RGB values, with or without an alpha channel, preceded by a number sign (#). Here's the syntax without an alpha channel:

```
TextColor="#rrggbb"
```

Each of the little letters is a hexadecimal digit. Here is how an alpha channel is included:

```
TextColor="#aarrggbb">
```

For the alpha channel, keep in mind that FF is fully opaque and 00 is fully transparent.

Two other formats allow you to specify only a single hexadecimal digit for each channel:

```
TextColor="#rgb" TextColor="#argb"
```

In these cases, the digit is repeated to form the value. For example, #CF3 is the RGB color CC-FF-33.

Here's the resultant page on the iPhone, Android, and Windows Phone:

If you need to embed any Unicode characters into the text, you can use the standard XML syntax. For example, to put the greeting in smart quotes, use:

```
<Label Text="&amp;#x201C;Hello, XAML!&amp;#x201D;" … />
```

Here's what it looks like:

These screenshots are the result of instantiating and returning `HelloXamlPage` directly from the `App.GetMainPage` method. The downloadable XamlSamples solution includes all the sample XAML pages from this series of articles on XAML, and each one is navigated to from the home page. The `HelloXamlPage` in that program looks a little different on iPhone and Android as a result of the navigation architecture. The screenshots in the remainder of this article were taken directly from the downloadable XamlSamples solution and display the navigation user interface at the top on the iPhone and Android.

XAML and Code Interactions

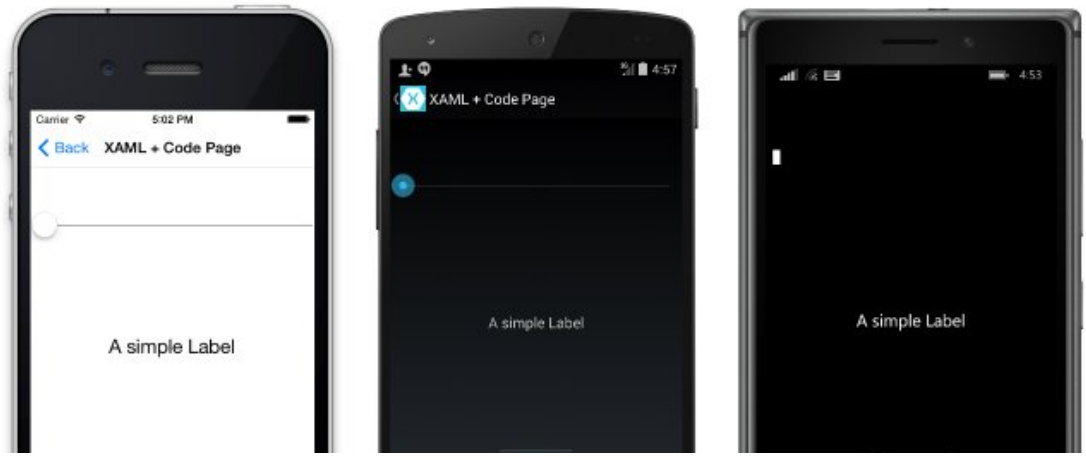
The HelloXamlPage sample contains only a single `Label` on the page, but this is very unusual. Most `ContentPage` derivatives set the `Content` property to a layout of some sort, such as a `StackLayout`. The `Children` property of the `StackLayout` is defined to be of type `IList<View>` but it's actually an object of type `ElementCollection<View>`, and that collection can be populated with multiple views or other layouts. In XAML, these parent-child relationships are established with normal XML hierarchy. Here's a XAML file for a class named XamlPlusCodePage:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.XamlPlusCodePage"
             Title="XAML + Code Page">
  <StackLayout>
    <Slider VerticalOptions="CenterAndExpand" />

    <Label Text="A simple Label"
           Font="Large"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />

    <Button Text="Click Me!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
  </StackLayout>
</ContentPage>
```

This XAML file is syntactically complete and here's what it looks like:



However, it is probably deficient in functionality. It is very likely that manipulating the `Slider` is supposed to cause the `Label` to display the current value, and the `Button` is probably intended to do something within the program.

As you'll see in Part 4. Data Binding Basics, the job of displaying a `Slider` value using a `Label` can be handled entirely in XAML with a data binding. But it is useful to see the code solution first. Even so, handling the `Button` click definitely requires code. This means that the code-behind file for `XamlPlusCodePage` must contain handlers for the `ValueChanged` event of the `Slider` and the `Clicked` event of the `Button`. Let's add them:

```
namespace XamlSamples
{
    public partial class XamlPlusCodePage
    {
        public XamlPlusCodePage()
        {
            InitializeComponent();
        }

        void OnSliderValueChanged(object sender,
                                  ValueChangedEventArgs args)
        {

        }

        void OnButtonClicked(object sender, EventArgs args)
        {

        }
    }
}
```

These event handlers do not need to be public.

Back in the XAML file, the `Slider` and `Button` tags need to include attributes for the `ValueChanged` and `Clicked` events that reference these handlers:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.XamlPlusCodePage"
             Title="XAML + Code Page">
  <StackLayout>
    <Slider VerticalOptions="CenterAndExpand"
            ValueChanged="OnSliderValueChanged" />

    <Label Text="A simple Label"
           Font="Large"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />

    <Button Text="Click Me!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            Clicked="OnButtonClicked" />
  </StackLayout>
</ContentPage>
```

Notice that assigning a handler to an event has the same syntax as assigning a value to a property.

If the handler for the `ValueChanged` event of the `Slider` will be using the `Label` to display the current value, the handler needs to reference that object from code. The `Label` needs a name, which is specified with the `x:Name` attribute.

```
<Label x:Name="valueLabel"
       Text="A simple Label"
       Font="Large"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand" />
```

The `x` prefix of the `x:Name` attribute indicates that this attribute is intrinsic to XAML.

The name you assign to the `x:Name` attribute has the same rules as C# variable names. For example, it

must begin with a letter or underscore and contain no embedded spaces.

Now the `ValueChanged` event handler can set the `Label` to display the new `Slider` value. The new value is available from the event arguments:

```
void OnSliderValueChanged(object sender,
                          ValueChangedEventArgs args)
{
    valueLabel.Text = args.NewValue.ToString("F3");
}
```

Or, the handler could obtain the `Slider` object that is generating this event from the sender argument and obtain the `Value` property from that:

```
void OnSliderValueChanged(object sender,
                          ValueChangedEventArgs args)
{
    valueLabel.Text = ((Slider)sender).Value.ToString("F3");
}
```

When you first run the program, the `Label` doesn't display the `Slider` value because the `ValueChanged` event hasn't yet fired. But any manipulation of the `Slider` causes the value to be displayed:

Now for the `Button`. Let's simulate a response to a `Clicked` event by displaying an alert with the `Text` of

the button. The event handler can safely cast the `sender` argument to a `Button` and then access its properties:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    Button button = (Button)sender;
    await DisplayAlert("Clicked!",
        "The button labeled '" + button.Text + "' has been clicked",
        "OK");
}
```

The method is defined as `async` because the `DisplayAlert` method is asynchronous and should be prefaced with the `await` operator, which returns when the method completes. Because this method obtains the `Button` firing the event from the `sender` argument, the same handler could be used for multiple buttons.

You've seen that an object defined in XAML can fire an event that is handled in the code-behind file, and that the code-behind file can access an object defined in XAML using the name assigned to it with the `x:Name` attribute. These are the two fundamental ways that code and XAML interact.

Some additional insights into how XAML works can be gleaned by examining the newly generated XamlPlusCode.xaml.g.cs file, which now includes any name assigned to any `x:Name` attribute as a private field:

```
public partial class XamlPlusCodePage : ContentPage {

    private Label valueLabel;

    private void InitializeComponent() {
        this.LoadFromXaml(typeof(XamlPlusCodePage));
        valueLabel = this.FindByName<Label>("valueLabel");
    }
}
```

The declaration of this field allows the variable to be freely used anywhere within the `XamlPlusCodePage` partial class file under your jurisdiction. At runtime, the field is assigned after the XAML has been parsed. This means that the `valueLabel` field is `null` when the `XamlPlusCodePage` constructor begins but valid after `InitializeComponent` is called.

After `InitializeComponent` returns control back to the constructor, the visuals of the page have been

constructed just as if they had been instantiated and initialized in code. The XAML file no longer plays any role in the class. You can manipulate these objects on the page in any way you want, for example, by adding views to the `StackLayout`, or setting the `Content` property of the page to something else entirely. You can "walk the tree" by examining the `Content` property of the page and the items in the `Children` collections of layouts. You can set properties on views accessed in this way, or assign event handlers to them dynamically.

Feel free. It's your page, and XAML is only a tool to build its content.

Summary

With this introduction, you've seen how a XAML file and code file are constituents of a new class definition that includes initialization, and how the XAML and code files interact. But XAML also has its own unique syntactical features that allow it to be used in a very flexible manner. You can begin exploring these in Part 2. Essential XAML Syntax.