Chapter 16

Data binding

Events and event handlers are a vital part of the interactive interface of Xamarin.Forms, but often event handlers perform very rudimentary jobs. They transfer values between properties of different objects and in some cases simply update a `Label` to show the new value of a view.

You can automate such connections between properties of two objects with a powerful feature of Xamarin.Forms called *data binding*. Under the covers, a data binding installs event handlers and handles the transfer of values from one property to another so that you don't have to. In most cases you define these data bindings in the XAML file, so there's no code (or very little code) involved. The use of data bindings helps reduce the number of "moving parts" in the application.

Data bindings also play a crucial role in the Model-View-ViewModel (MVVM) application architecture. As you'll see in Chapter 18, "MVVM," data bindings provide the link between the View (the user interface often implemented in XAML) and the underlying data of the ViewModel and Model. This means that the connections between the user interface and underlying data can be represented in XAML along with the user interface.

Binding basics

Several properties, methods, and classes are involved in data bindings:

- The `Binding` class (which derives from `BindingBase`) defines many characteristics of a data binding.
- The `BindingContext` property is defined by the `BindableObject` class.
- The `SetBinding` method is also defined by the `BindableObject` class.
- The `BindableObjectExtensions` class defines two additional overloads of `SetBinding`.

Two classes support XAML markup extensions for bindings:

- The `BindingExtension` class, which is private to Xamarin.Forms, provides support for the `Binding` markup extension that you use to define a data binding in XAML.
- The `ReferenceExtension` class is also crucial to bindings.

Two interfaces also get involved in data binding. These are:

- `INotifyPropertyChanged` (defined in the `System.ComponentModel` namespace) is the standard interface that classes use when notifying external classes that a property has changed.

This interface plays a major role in MVVM.

- `IValueConverter` (defined in the `Xamarin.Forms` namespace) is used to define small classes that aid data binding by converting values from one type to another.

The most fundamental concept of data bindings is this: Data bindings always have a *source* and a *target*. The source is a property of an object, usually one that changes dynamically at run time. When that property changes, the data binding automatically updates the target, which is a property of another object.

Target ← Source

But as you'll see, sometimes the data flow between the source and target isn't in a constant direction. Even in those cases, however, the distinction between source and target is important because of one basic fact:

The target of a data binding must be backed by a `BindableProperty` *object.*

As you know, the `VisualElement` class derives from `BindableObject` by way of `Element`, and all the visual elements in Xamarin.Forms define most of their properties as bindable properties. For this reason, data-binding targets are almost always visual elements or—as you'll see in Chapter 19, "Collection views"—objects called *cells* that are translated to visual elements.

Although the target of a data binding must be backed by a `BindableProperty` object, there is no such requirement for a data-binding source. The source can be a plain old C# property. However, in all but the most trivial data bindings, a change in the source property causes a corresponding change in the target property. This means that the source object must implement some kind of notification mechanism to signal when the property changes. This notification mechanism is the `INotifyPropertyChanged` interface, which is a standard .NET interface involved in data bindings and used extensively for implementing the MVVM architecture.

The rule for a nontrivial data-binding source—that is, a data-binding source that can dynamically change value—is therefore:

The source of a nontrivial data binding must implement `INotifyPropertyChanged`.

Despite its importance, the `INotifyPropertyChanged` interface has the virtue of being very simple: it consists solely of one event, called `PropertyChanged`, which a class fires when a property has changed.

Very conveniently for our purposes, `BindableObject` implements `INotifyPropertyChanged`. Any property that is backed by a bindable property automatically fires a `PropertyChanged` event when that property changes. This automatic firing of the event extends to bindable properties you might define in your own classes.

This means that you can define data bindings between properties of visual objects. In the grand scheme of things, most data bindings probably link visual objects with underlying data, but for purposes of learning about data bindings and experimenting with them, it's nice to simply link properties of two views without defining data classes.

For the first few examples in this chapter, you'll see data bindings in which the source is the `Value` property of a `Slider` and the target is the `Opacity` property of a `Label`. As you manipulate the `Slider`, the `Label` changes from transparent to opaque. Both properties are of type `double` and range from 0 to 1, so they are a perfect match.

You already know how to do this little job with a simple event handler. Let's see how to do it with a data binding.

Code and XAML

Although most data bindings are defined in XAML, you should know how to do one in code. Here's one way (but not the only way) to set a data binding in code:

- Set the `BindingContext` property on the target object to refer to the source object.
- Call `SetBinding` on the target object to specify both the target and source properties.

The `BindingContext` property is defined by `BindableObject`. (It's the *only* property defined by `BindableObject`.) The `SetBinding` method is also defined by `BindableObject`, but there are two additional overloads of the `SetBinding` method in the `BindableObjectExtensions` class. The target property is specified as a `BindableProperty`; the source property is often specified as a string.

The **OpacityBindingCode** program creates two elements, a `Label` and a `Slider`, and defines a data binding that targets the `Opacity` property of the `Label` from the `Value` property of the `Slider`:

```
public class OpacityBindingCodePage : ContentPage
{
    public OpacityBindingCodePage()
    {
        Label label = new Label
        {
            Text = "Opacity Binding Demo",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        Slider slider = new Slider
        {
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        // Set the binding context: target is Label; source is Slider.
        label.BindingContext = slider;
        // Bind the properties: target is Opacity; source is Value.
        label.SetBinding(Label.OpacityProperty, "Value");

        // Construct the page.
        Padding = new Thickness(10, 0);
        Content = new StackLayout
        {
            Children = { label, slider }
        };
    }
}
```

Here's the property setting that connects the two objects:

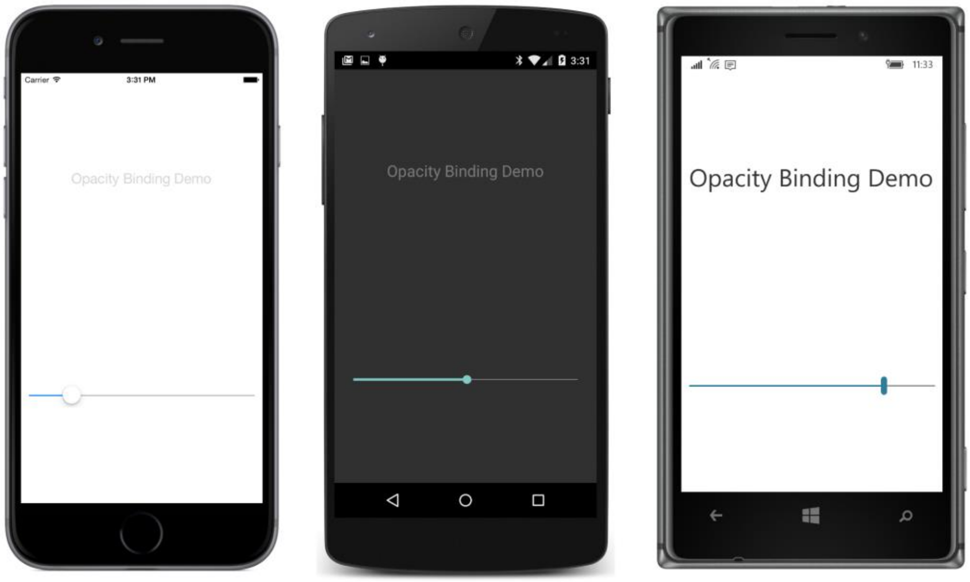
```
label.BindingContext = slider;
```

The `label` object is the target and the `slider` object is the source. Here's the method call that links the two properties:

```
label.SetBinding(Label.OpacityProperty, "Value");
```

The first argument to `SetBinding` is of type `BindableProperty`, and that's the requirement for the target property. But the source property is merely specified as a string. It can be any type of property.

The screenshot demonstrates that you don't need to set an event handler to use the `Slider` for controlling other elements on the page:



Of course, *somebody* is setting an event handler. Under the covers, when the binding initializes itself, it also performs initialization on the target by setting the `Opacity` property of the `Label` from the

Value property of the Slider. (As you discovered in the previous chapter, when you set an event handler yourself, this initialization doesn't happen automatically.) Then the internal binding code checks whether the source object (in this case the Slider) implements the INotifyProperty-Changed interface. If so, a PropertyChanged handler is set on the Slider. Whenever the Value property changes, the binding sets the new value to the Opacity property of the Label.

Reproducing the binding in XAML involves two markup extensions that you haven't seen yet:

- x:Reference, which is part of the XAML 2009 specification.
- Binding, which is part of Microsoft's XAML-based user interfaces.

The x:Reference binding extension is very simple, but the Binding markup extension is the most extensive and complex markup extension in all of Xamarin.Forms. It will be introduced incrementally over the course of this chapter.

Here's how you set the data binding in XAML:

- Set the BindingContext property of the target element (the Label) to an x:Reference markup extension that references the source element (the Slider).
- Set the target property (the Opacity property of the Label) to a Binding markup extension that references the source property (the Value property of the Slider).

The **OpacityBindingXaml** project shows the complete markup:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="OpacityBindingXaml.OpacityBindingXamlPage"
             Padding="10, 0">
    <StackLayout>
        <Label Text="Opacity Binding Demo"
               FontSize="Large"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center"
               BindingContext="{x:Reference Name=slider}"
               Opacity="{Binding Path=Value}" />

        <Slider x:Name="slider"
                VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

The two markup extensions for the binding are the last two attribute settings in the Label. The code-behind file contains nothing except the standard call to InitializeComponent.

When setting the BindingContext in markup, it is very easy to forget the x:Reference markup extension and simply specify the source name, but that doesn't work.

The Path argument of the Binding markup expression specifies the source property. Why is this argument called Path rather than Property? You'll see why later in this chapter.

You can make the markup a little shorter. The public class that provides support for `Reference` is `ReferenceExtension`, which defines its content property to be `Name`. The content property of `BindingExtension` (which is not a public class) is `Path`, so you don't need the `Name` and `Path` arguments and equal signs:

```
<Label Text="Opacity Binding Demo"
       FontSize="Large"
       VerticalOptions="CenterAndExpand"
       HorizontalOptions="Center"
       BindingContext="{x:Reference slider}"
       Opacity="{Binding Value}" />
```

Or if you'd like to make the markup longer, you can break out the `BindingContext` and `Opacity` properties as property elements and set them by using regular element syntax for `x:Reference` and `Binding`:

```
<Label Text="Opacity Binding Demo"
       FontSize="Large"
       VerticalOptions="CenterAndExpand"
       HorizontalOptions="Center">

    <Label.BindingContext>
        <x:Reference Name="slider" />
    </Label.BindingContext>

    <Label.Opacity>
        <Binding Path="Value" />
    </Label.Opacity>
</Label>
```

As you'll see, the use of property elements for bindings is sometimes convenient in connection with the data binding.

Source and BindingContext

The `BindingContext` property is actually one of two ways to link the source and target objects. You can alternatively dispense with `BindingContext` and include a reference to the source object within the binding expression itself.

The **BindingSourceCode** project has a page class that is identical to the one in **OpacityBinding-Code** except that the binding is defined in two statements that don't involve the `BindingContext` property:

```
public class BindingSourceCodePage : ContentPage
{
    public BindingSourceCodePage()
    {
        Label label = new Label
        {
            Text = "Opacity Binding Demo",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        Slider slider = new Slider
        {
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        // Define Binding object with source object and property.
        Binding binding = new Binding
        {
            Source = slider,
            Path = "Value"
        };

        // Bind the Opacity property of the Label to the source.
        label.SetBinding(Label.OpacityProperty, binding);

        // Construct the page.
        Padding = new Thickness(10, 0);
        Content = new StackLayout
        {
            Children = { label, slider }
        };
    }
}
```

The target object and property are still specified in the call to the SetBinding method:

```
label.SetBinding(Label.OpacityProperty, binding);
```

However, the second argument references a Binding object that specifies the source object and property:

```
Binding binding = new Binding
{
    Source = slider,
    Path = "Value"
};
```

That is not the only way to instantiate and initialize a Binding object. An extensive Binding constructor allows for specifying many Binding properties. Here's how it could be used in the **BindingSourceCode** program:

```
Binding binding = new Binding("Value", BindingMode.Default, null, null, null, slider);
```

Or you can use a named argument to reference the slider object:

```
Binding binding = new Binding("Value", source: slider);
```

Binding also has a generic Create method that lets you specify the Path property as a Func object rather than as a string so that it's more immune from misspellings or changes in the property name. However, this Create method doesn't include an argument for the Source property, so you need to set it separately:

```
Binding binding = Binding.Create<Slider>(src => src.Value);
binding.Source = slider;
```

The BindableObjectExtensions class defines two overloads of SetBinding that allow you to avoid explicitly instantiating a Binding object. However, neither of these overloads includes the Source property, so they are restricted to cases where you're using the BindingContext.

The **BindingSourceXaml** program demonstrates how both the source object and source property can be specified in the Binding markup extension:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BindingSourceXaml.BindingSourceXamlPage"
             Padding="10, 0">
    <StackLayout>
        <Label Text="Binding Source Demo"
               FontSize="Large"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center"
               Opacity="{Binding Source={x:Reference Name=slider},
                                 Path=Value}" />

        <Slider x:Name="slider"
                VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

The Binding markup extension now has two arguments, one of which is another markup extension for x:Reference, so a pair of curly braces are nested within the main curly braces:

```
Opacity="{Binding Source={x:Reference Name=slider},
                  Path=Value}" />
```

For visual clarity, the two Binding arguments are vertically aligned within the markup extension, but that's not required. Arguments must be separated by a comma (here at the end of the first line), and no quotation marks must appear within the curly braces. You're not dealing with XML attributes within the markup extension. These are markup extension arguments.

You can simplify the nested markup extension by eliminating the Name argument name and equals sign in x:Reference because Name is the content property of the ReferenceExtension class:

```
Opacity="{Binding Source={x:Reference slider},
                  Path=Value}" />
```

However, you *cannot* similarly remove the Path argument name and equals sign. Even though BindingExtension defines Path as its content property, the argument name can be eliminated only when that argument is the first among multiple arguments. You need to switch around the arguments like so:

```
Opacity="{Binding Path=Value,
                  Source={x:Reference slider}}" />
```

And then you can eliminate the Path argument name, and perhaps move everything to one line:

```
Opacity="{Binding Value, Source={x:Reference slider}}" />
```

However, because the first argument is missing an argument name and the second argument has an argument name, the whole expression looks a bit peculiar, and it might be difficult to grasp the Binding arguments at first sight. Also, it makes sense for the Source to be specified *before* the Path because the particular property specified by the Path makes sense only for a particular type of object, and that's specified by the Source.

In this book, whenever the Binding markup extension includes a Source argument, it will be first, followed by the Path. Otherwise, the Path will be the first argument, and often the Path argument name will be eliminated.

You can avoid the issue entirely by expressing Binding in element form:

```
<Label Text="Binding Source Demo"
       FontSize="Large"
       VerticalOptions="CenterAndExpand"
       HorizontalOptions="Center">
    <Label.Opacity>
        <Binding Source="{x:Reference slider}"
                 Path="Value" />
    </Label.Opacity>
</Label>
```

The x:Reference markup extension still exists, but you can also express that in element form as well:

```
<Label Text="Binding Source Demo"
       FontSize="Large"
       VerticalOptions="CenterAndExpand"
       HorizontalOptions="Center">
    <Label.Opacity>
        <Binding Path="Value">
            <Binding.Source>
                <x:Reference Name="slider" />
            </Binding.Source>
        </Binding>
    </Label.Opacity>
</Label>
```

You have now seen two ways to specify the link between the source object with the target object:

- Use the BindingContext to reference the source object.
- Use the Source property of the Binding class or the Binding markup extension.

If you specify both, the Source property takes precedence over the BindingContext.

In the examples you've seen so far, these two techniques have been pretty much interchangeable. However, they have some significant differences. For example, suppose you have one object with two properties that are targets of two different data bindings involving two different source objects—for example, a Label with the Opacity property bound to a Slider and the IsVisible property bound to a Switch. You can't use BindingContext for both bindings because BindingContext applies to the whole target object and can only specify a single source. You must use the Source property of Binding for at least one of these bindings.

BindingContext is itself backed by a bindable property. This means that BindingContext can be set from a Binding markup extension. In contrast, you can't set the Source property of Binding to another Binding because Binding does not derive from BindableObject, which means Source is not backed by a bindable property and hence can't be the target of a data binding.

In this variation of the **BindingSourceXaml** markup, the BindingContext property of the Label is set to a Binding markup extension that includes a Source and Path.

```
<Label Text="Binding Source Demo"
       FontSize="Large"
       VerticalOptions="CenterAndExpand"
       HorizontalOptions="Center"
       BindingContext="{Binding Source={x:Reference Name=slider},
                                Path=Value}"
       Opacity="{Binding}" />
```

This means that the BindingContext for this Label is not the slider object as in previous examples but the double that is the Value property of the Slider. To bind the Opacity property to this double, all that's required is an empty Binding markup extension that basically says "use the BindingContext for the entire data-binding source."

Perhaps the most important difference between BindingContext and Source is a very special characteristic that makes BindingContext unlike any other property in all of Xamarin.Forms:

The binding context is propagated through the visual tree.

In other words, if you set BindingContext on a StackLayout, it applies to all the children of that StackLayout and their children as well. The data bindings within that StackLayout don't have to specify BindingContext or the Source argument to Binding. They inherit BindingContext from the StackLayout. Or the children of the StackLayout can override that inherited BindingContext with BindingContext settings of their own or with a Source setting in their bindings.

This feature turns out to be exceptionally useful. Suppose a StackLayout contains a bunch of visuals with data bindings set to various properties of a particular class. Set the BindingContext property of that StackLayout. Then, the individual data bindings on the children of the StackLayout don't require either a Source specification or a BindingContext setting. You could then set the BindingContext of the StackLayout to different instances of that class to display the properties for each instance. You'll see examples of this technique and other data-binding marvels in the chapters ahead, and particularly in Chapter 19.

Meanwhile, let's look at a much simpler example of `BindingContext` propagation through the visual tree.

The `WebView` is intended to embed a web browser inside your application. Alternatively, you can use `WebView` in conjunction with the `HtmlWebViewSource` class to display a chunk of HTML, perhaps saved as an embedded resource in the PCL.

For displaying webpages, you use `WebView` with the `UrlWebViewSource` class to specify an initial URL. However, `UrlWebViewSource` and `HtmlWebViewSource` both derive from the abstract class `WebViewSource`, and that class defines an implicit conversion of `string` and `Uri` to itself, so all you really need to do is set a string with a web address to the `Source` property of `WebView` to direct `WebView` to present that webpage.

`WebView` also defines two methods, named `GoBack` and `GoForward`, that internally implement the **Back** and **Forward** buttons typically found on web browsers. Your program needs to know when it can enable these buttons, so `WebView` also defines two get-only Boolean properties, named `CanGoBack` and `CanGoForward`. These two properties are backed by bindable properties, which means that any changes to these properties result in `PropertyChanged` events being fired, which further means that they can be used as data binding sources to enable and disable two buttons.

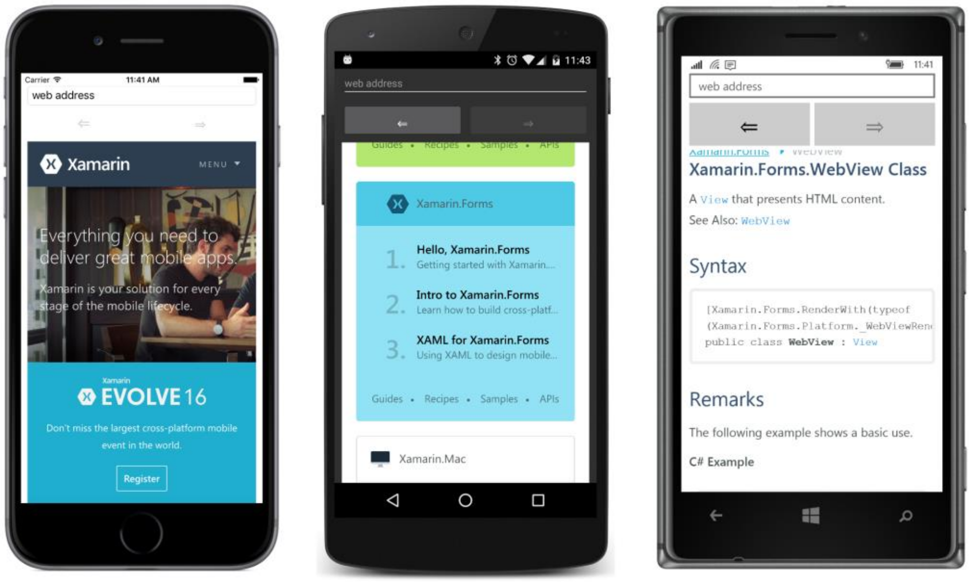
Here's the XAML file for **WebViewDemo**. Notice that the nested `StackLayout` containing the two `Button` elements has its `BindingContext` property set to the `WebView`. The two `Button` children in that `StackLayout` inherit the `BindingContext`, so the buttons can have very simple `Binding` expressions on their `IsEnabled` properties that reference only the `CanGoBack` and `CanGoForward` properties:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="WebViewDemo.WebViewDemoPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="10, 20, 10, 0"
                    Android="10, 0"
                    WinPhone="10, 0" />
    </ContentPage.Padding>

    <StackLayout>
        <Entry Keyboard="Url"
               Placeholder="web address"
               Completed="OnEntryCompleted" />

        <StackLayout Orientation="Horizontal"
                     BindingContext="{x:Reference webView}">

            <Button Text="&#x21D0;"
                    FontSize="Large"
                    HorizontalOptions="FillAndExpand"
                    IsEnabled="{Binding CanGoBack}"
                    Clicked="OnGoBackClicked" />

            <Button Text="&#x21D2;"
                    FontSize="Large"
                    HorizontalOptions="FillAndExpand"
                    IsEnabled="{Binding CanGoForward}"
                    Clicked="OnGoForwardClicked" />
        </StackLayout>

        <WebView x:Name="webView"
                 VerticalOptions="FillAndExpand"
                 Source="https://xamarin.com" />
    </StackLayout>
</ContentPage>
```

The code-behind file needs to handle the `Clicked` events for the **Back** and **Forward** buttons as well as the `Completed` event for the `Entry` that lets you enter a web address of your own:

```
public partial class WebViewDemoPage : ContentPage
{
    public WebViewDemoPage()
    {
        InitializeComponent();
    }

    void OnEntryCompleted(object sender, EventArgs args)
    {
        webView.Source = ((Entry)sender).Text;
    }

    void OnGoBackClicked(object sender, EventArgs args)
    {
        webView.GoBack();
    }

    void OnGoForwardClicked(object sender, EventArgs args)
    {
        webView.GoForward();
    }
}
```

You don't need to enter a web address when the program starts up because the XAML file is hard-coded to go to your favorite website, and you can navigate around from there:


web address
Xamarin
MENU
Everything you need to deliver great mobile apps.
Xamarin is your solution for every stage of the mobile lifecycle.
EVOLVE 16
Don't miss the largest cross-platform mobile event in the world.
Register
web address
Xamarin.Forms
Hello, Xamarin.Forms
Intro to Xamarin.Forms
XAML for Xamarin.Forms
Guides • Recipes • Samples • APIs
Xamarin.Mac
web address
Xamarin.Forms.WebView Class
A View that presents HTML content.
See Also: WebView
Syntax
Remarks
The following example shows a basic use.
C# Example

The binding mode

Here is a `Label` whose `FontSize` property is bound to the `Value` property of a `Slider`:

```
<Label FontSize="{Binding Source={x:Reference slider},
                          Path=Value}" />
<Slider x:Name="slider"
        Maximum="100" />
```

That should work, and if you try it, it will work. You'll be able to change the `FontSize` of the `Label` by manipulating the `Slider`.

But here's a `Label` and `Slider` with the binding reversed. Instead of the `FontSize` property of the `Label` being the target, now `FontSize` is the source of the data binding, and the target is the `Value` property of the `Slider`:

```
<Label x:Name="label" />
<Slider Maximum="100"
        Value="{Binding Source={x:Reference label},
                        Path=FontSize}" />
```

That doesn't seem to make any sense. But if you try it, it will work just fine. Once again, the `Slider` will manipulate the `FontSize` property of the `Label`.

The second binding works because of something called the *binding mode*.

You've learned that a data binding sets the value of a target property from the value of a source property, but sometimes the data flow is not so clear cut. The relationship between target and source is defined by members of the `BindingMode` enumeration:

- `Default`
- `OneWay` — changes in the source affect the target (normal).
- `OneWayToSource` — changes in the target affect the source.
- `TwoWay` — changes in the source and target affect each other.

This `BindingMode` enumeration plays a role in two different classes:

When you create a `BindableProperty` object by using one of the static `Create` or `CreateReadOnly` static methods, you can specify a default `BindingMode` value to use when that property is the target of a data binding.

If you don't specify anything, the default binding mode is `OneWay` for bindable properties that are readable and writeable, and `OneWayToSource` for read-only bindable properties. If you specify `BindingMode.Default` when creating a bindable property, the default binding mode for the property is set to `OneWay`. (In other words, the `BindingMode.Default` member is not intended for defining bindable properties.)

You can override that default binding mode for the target property when you define a binding either in code or XAML. You override the default binding mode by setting the `Mode` property of `Binding` to one of the members of the `BindingMode` enumeration. The `Default` member means that you want to use the default binding mode defined for the target property.

When you set the `Mode` property to `OneWayToSource` you are *not* switching the target and the source. The target is still the object on which you've set the `BindingContext` and the property on which you've called `SetBinding` or applied the `Binding` markup extension. But the data flows in a different direction—from target to source.

Most bindable properties have a default binding mode of `OneWay`. However, there are some exceptions. Of the views you've encountered so far in this book, the following properties have a default mode of `TwoWay`:

Class	**Property that is TwoWay**
`Slider`	`Value`
`Stepper`	`Value`
`Switch`	`IsToggled`
`Entry`	`Text`
`Editor`	`Text`
`SearchBar`	`Text`
`DatePicker`	`Date`
`TimePicker`	`Time`

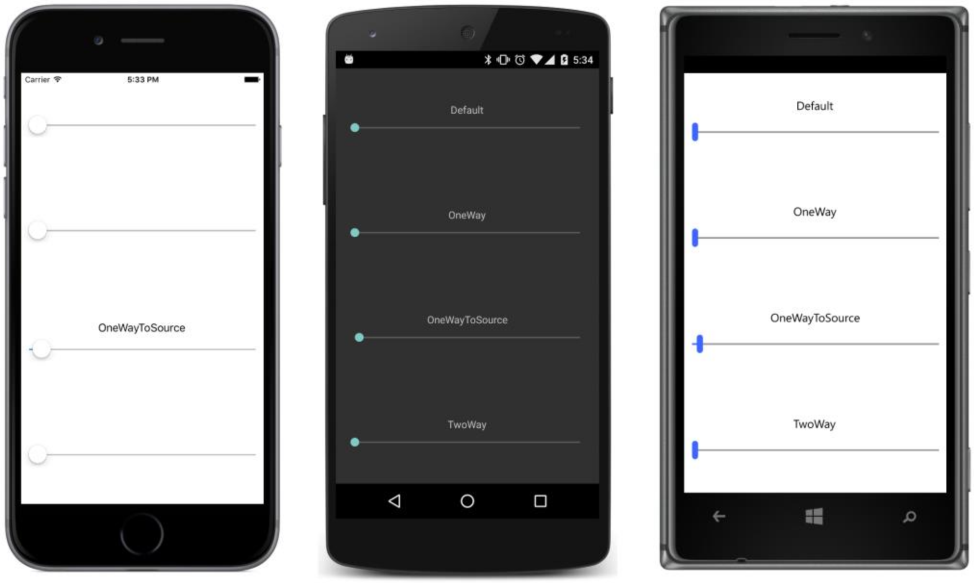
The properties that have a default binding mode of `TwoWay` are those most likely to be used with underlying data models in an MVVM scenario. With MVVM, the binding targets are visual objects and the binding sources are data objects. In general, you want the data to flow both ways. You want the visual objects to display the underlying data values (from source to target), and you want the interactive visual objects to cause changes in the underlying data (target to source).

The **BindingModes** program connects four `Label` elements and four `Slider` elements with "normal" bindings, meaning that the target is the `FontSize` property of the `Label` and the source is the `Value` property of the `Slider`:

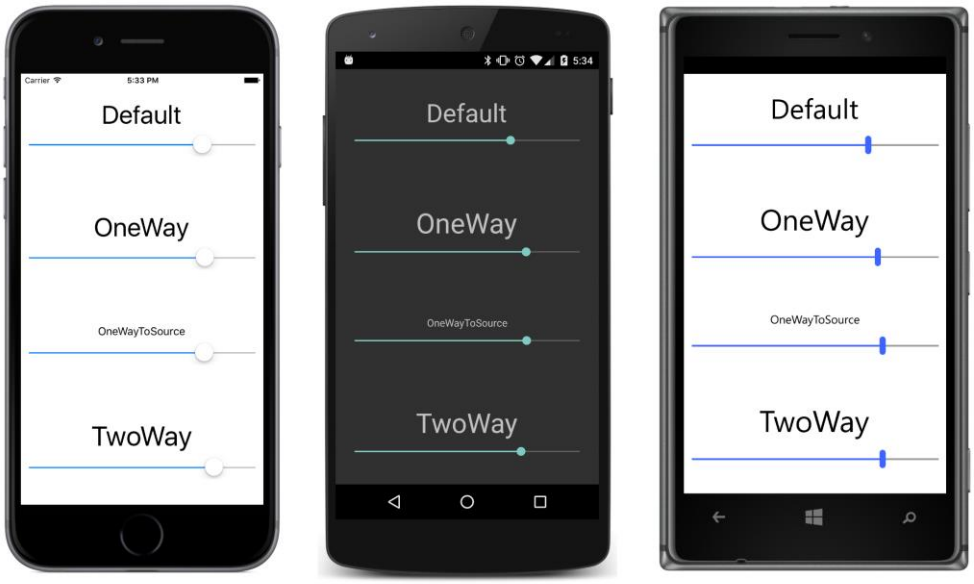
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BindingModes.BindingModesPage"
             Padding="10, 0">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="StackLayout">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="HorizontalOptions" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout VerticalOptions="Fill">
        <StackLayout>
            <Label Text="Default"
                   FontSize="{Binding Source={x:Reference slider1},
                                      Path=Value}" />
            <Slider x:Name="slider1"
                    Maximum="50" />
        </StackLayout>

        <StackLayout>
            <Label Text="OneWay"
                   FontSize="{Binding Source={x:Reference slider2},
                                      Path=Value,
                                      Mode=OneWay}" />
            <Slider x:Name="slider2"
                    Maximum="50" />
        </StackLayout>

        <StackLayout>
            <Label Text="OneWayToSource"
                   FontSize="{Binding Source={x:Reference slider3},
                                      Path=Value,
                                      Mode=OneWayToSource}" />
            <Slider x:Name="slider3"
                    Maximum="50" />
        </StackLayout>

        <StackLayout>
```

```
            <Label Text="TwoWay"
                   FontSize="{Binding Source={x:Reference slider4},
                                      Path=Value,
                                      Mode=TwoWay}" />
            <Slider x:Name="slider4"
                    Maximum="50" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

The `Text` of the `Label` indicates the binding mode. When you first run this program, all the `Slider` elements are initialized at zero, except for the third one, which is slightly nonzero:



By manipulating each `Slider`, you can change the `FontSize` of the `Label`, but it doesn't work for the third one because the `OneWayToSource` mode indicates that changes in the target (the `FontSize` property of the `Label`) affect the source (the `Value` property of the `Slider`):



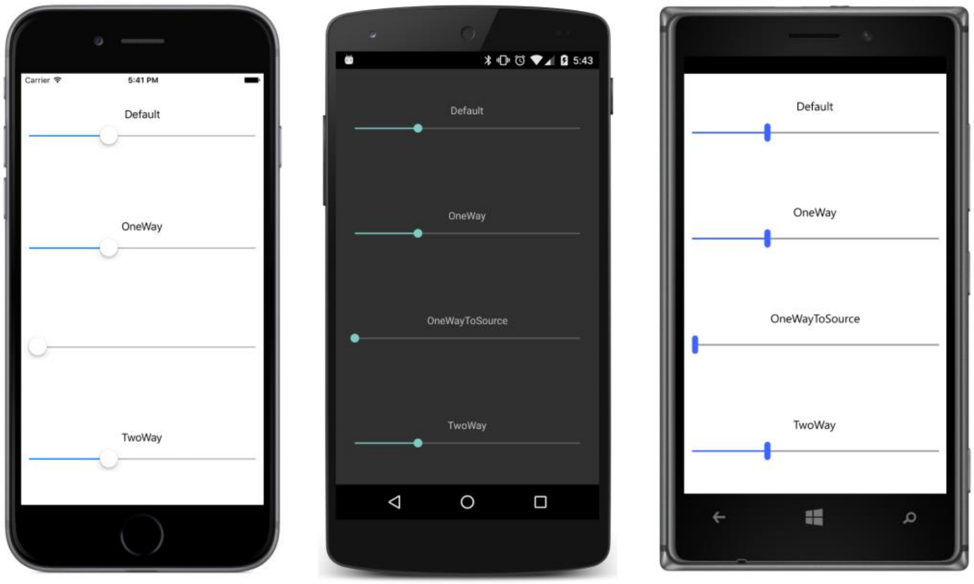
Although it’s not quite evident here, the default binding mode is `OneWay` because the binding is set on the `FontSize` property of the `Label`, and that’s the default binding mode for the `FontSize` property.

The **ReverseBinding** program sets the bindings on the `Value` property of the `Slider`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ReverseBinding.ReverseBindingPage"
             Padding="10, 0">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="StackLayout">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="HorizontalOptions" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout VerticalOptions="Fill">
        <StackLayout>
            <Label x:Name="label1"
                   Text="Default" />
            <Slider Maximum="50"
                    Value="{Binding Source={x:Reference label1},
                                    Path=FontSize}" />
        </StackLayout>
        <StackLayout>
            <Label x:Name="label2"
                   Text="OneWay" />
            <Slider Maximum="50"
                    Value="{Binding Source={x:Reference label2},
                                    Path=FontSize,
                                    Mode=OneWay}" />
        </StackLayout>

        <StackLayout>
            <Label x:Name="label3"
                   Text="OneWayToSource" />
            <Slider Maximum="50"
                    Value="{Binding Source={x:Reference label3},
                                    Path=FontSize,
                                    Mode=OneWayToSource}" />
        </StackLayout>

        <StackLayout>
            <Label x:Name="label4"
                   Text="TwoWay" />
            <Slider Maximum="50"
                    Value="{Binding Source={x:Reference label4},
                                    Path=FontSize,
                                    Mode=TwoWay}" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

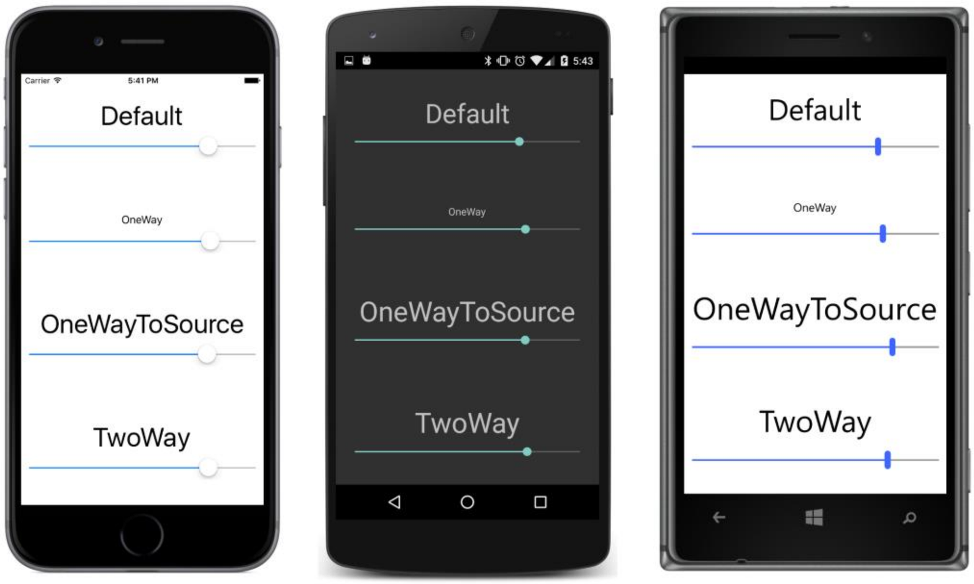
The default binding mode on these bindings is `TwoWay` because that's the mode set in the `Binda-bleProperty.Create` method for the `Value` property of the `Slider`.

What's interesting about this approach is that for three of the cases here, the `Value` property of the `Slider` is initialized from the `FontSize` property of the `Label`:



It doesn't happen for `OneWayToSource` because for that mode, changes to the `Value` property of the `Slider` affect the `FontSize` property of the `Label` but not the other way around.

Now let's start manipulating these sliders:



Now the `OneWayToSource` binding works because changes to the `Value` property of the `Slider` affect the FontSize property of the Label, but the OneWay binding does not work because that indicates that the Value property of the Slider is only affected by changes in the FontSize property of the Label.

Which binding works the best? Which binding initializes the Value property of the Slider to the FontSize property of the Label, but also allows Slider manipulations to change the FontSize? It's the reverse binding set on the Slider with a mode of TwoWay, which is the default mode.

This is exactly the type of initialization you want to see when a Slider is bound to some data. For that reason, when using a Slider with MVVM, the binding is set on the Slider to both display the data value and to manipulate the data value.

String formatting

Some of the sample programs in the previous chapter used event handlers to display the current values of the Slider and Stepper views. If you try defining a data binding that targets the Text property of a Label from the Value property of a Slider, you'll discover that it works, but you don't have much control over it. In general, you'll want to control any type conversion or value conversion required in data bindings. That's discussed later in this chapter.

String formatting is special, however. The Binding class has a StringFormat property that allows you to include an entire .NET formatting string. Almost always, the target of such a binding is the Text property of a Label, but the binding source can be of any type.

The .NET formatting string that you supply to StringFormat must be suitable for a call to the String.Format static method, which means that it should contain a placeholder of "{0}" with or without a formatting specification suitable for the source data type—for example "{0:F3}" to display a double with three decimal places.

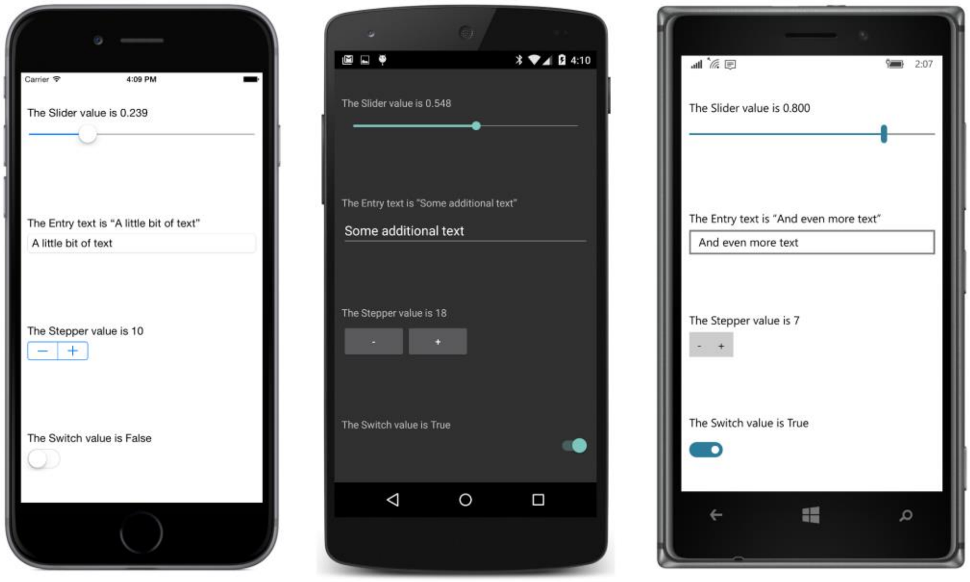
In XAML, this placeholder is a bit of a problem because the curly braces can be mistaken for the curly braces used to delimit markup extensions. The easiest solution is to put the entire formatting string in single quotation marks.

The **ShowViewValues** program contains four examples that display the current values of a Slider, Entry, Stepper, and Switch. The hexadecimal codes in the formatting string used for displaying the Entry contents are Unicode IDs for "smart quotes":

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ShowViewValues.ShowViewValuesPage"
             Padding="10, 0">

    <StackLayout>
        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="{Binding Source={x:Reference slider},
                                  Path=Value,
                                  StringFormat='The Slider value is {0:F3}'}" />
            <Slider x:Name="slider" />
        </StackLayout>

        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="{Binding Source={x:Reference entry},
                                  Path=Text,
                                  StringFormat='The Entry text is &#x201C;{0}&#x201D;'}" />
            <Entry x:Name="entry" />
        </StackLayout>

        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="{Binding Source={x:Reference stepper},
                                  Path=Value,
                                  StringFormat='The Stepper value is {0}'}" />
            <Stepper x:Name="stepper" />
        </StackLayout>

        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="{Binding Source={x:Reference switch},
                                  Path=IsToggled,
                                  StringFormat='The Switch value is {0}'}" />
            <Switch x:Name="switch" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

When using `StringFormat` you need to pay particular attention to the placement of commas, single quotation marks, and curly braces.

Here's the result:

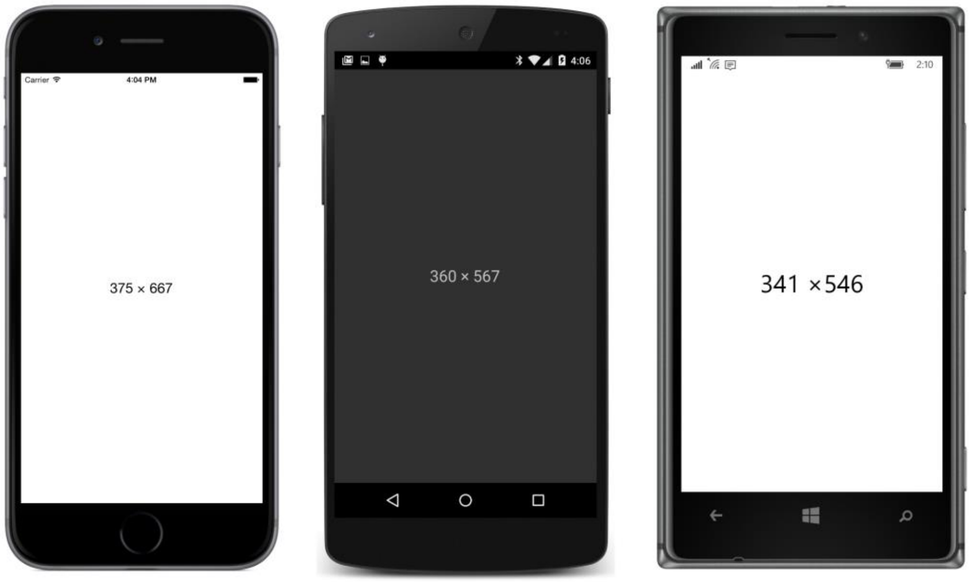

Carrier
4:09 PM
The Slider value is 0.239
The Entry text is "A little bit of text"
A little bit of text
The Stepper value is 10
The Switch value is False
4:10
The Slider value is 0.548
The Entry text is "Some additional text"
Some additional text
The Stepper value is 18
The Switch value is True
2:07
The Slider value is 0.800
The Entry text is "And even more text"
And even more text
The Stepper value is 7
The Switch value is True

You might recall the **WhatSize** program from Chapter 5, "Dealing with sizes." That program used a SizeChanged event handler on the page to display the current width and height of the screen in device-independent units.

The **WhatSizeBindings** program does the whole job in XAML. First it adds an x:Name attribute to the root tag to give the WhatSizeBindingsPage object a name of page. Three Label views share a horizontal StackLayout in the center of the page, and two of them have bindings to the Width and Height properties. The Width and Height properties are get-only, but they are backed by bindable properties, so they fire PropertyChanged events when they change:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="WhatSizeBindings.WhatSizeBindingsPage"
             x:Name="page">

    <StackLayout Orientation="Horizontal"
                 Spacing="0"
                 HorizontalOptions="Center"
                 VerticalOptions="Center">

        <StackLayout.Resources>
            <ResourceDictionary>
                <Style TargetType="Label">
                    <Setter Property="FontSize" Value="Large" />
                </Style>
            </ResourceDictionary>
        </StackLayout.Resources>

        <Label Text="{Binding Source={x:Reference page},
                              Path=Width,
                              StringFormat='{0:F0}'}" />

        <!-- Multiplication sign. -->
        <Label Text=" &#x00D7; " />

        <Label Text="{Binding Source={x:Reference page},
                              Path=Height,
                              StringFormat='{0:F0}'}" />
    </StackLayout>
</ContentPage>
```

Here's the result for the devices used for this book:



The display changes as you turn the phone between portrait and landscape modes.

Alternatively, the `BindingContext` on the `StackLayout` could be set to an `x:Reference` markup extension referencing the `page` object, and the `Source` settings on the bindings wouldn't be necessary.

Why is it called "Path"?

The `Binding` class defines a property named `Path` that you use to set the source property name. But why is it called `Path`? Why isn't it called `Property`?

The `Path` property is called what it's called because it doesn't need to be one property. It can be a stack of properties, subproperties, and even indexers connected with periods.

Using `Path` in this way can be tricky, so here's a program called **BindingPathDemos** that has four `Binding` markup extensions, each of which sets the `Path` argument to a string of property names and indexers:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:globe="clr-namespace:System.Globalization;assembly=mscorlib"
             x:Class="BindingPathDemos.BindingPathDemosPage"
             x:Name="page">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="10, 20, 10, 0"
                    Android="10, 0"
                    WinPhone="10, 0" />
    </ContentPage.Padding>

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>

            <Style TargetType="Label" BasedOn="{StaticResource baseStyle}">
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>

            <Style TargetType="Slider" BasedOn="{StaticResource baseStyle}" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout BindingContext="{x:Reference page}">
        <Label Text="{Binding Path=Padding.Top,
                              StringFormat='The top padding is {0}'}" />

        <Label Text="{Binding Path=Content.Children[4].Value,
                              StringFormat='The Slider value is {0:F2}'}" />

        <Label Text="{Binding Source={x:Static globe:CultureInfo.CurrentCulture},
                              Path=DateTimeFormat.DayNames[3],
                              StringFormat='The middle day of the week is {0}'}" />

        <Label Text="{Binding Path=Content.Children[2].Text.Length,
                              StringFormat='The preceding Label has {0} characters'}" />
        <Slider />
    </StackLayout>
</ContentPage>
```

Only one element here has an `x:Name`, and that's the page itself. The `BindingContext` of the `StackLayout` is that page, so all the bindings within the `StackLayout` are relative to the page (except for the binding that has an explicit `Source` property set).

The first `Binding` looks like this:

```
<Label Text="{Binding Path=Padding.Top,
                      StringFormat='The top padding is {0}'}" />
```

The `Path` begins with the `Padding` property of the page. That property is of type `Thickness`, so it's possible to access a property of the `Thickness` structure with a property name such as `Top`. Of course, `Thickness` is a structure and therefore does not derive from `BindableObject`, so `Top` can't be a `BindableProperty`. The binding infrastructure can't set a `PropertyChanged` handler on that property, but it will set a `PropertyChanged` handler on the `Padding` property of the page, and if that changes, the binding will update the target.

The second `Binding` references the `Content` property of the page, which is the `StackLayout`. That `StackLayout` has a `Children` property, which is a collection, so it can be indexed:

```
<Label Text="{Binding Path=Content.Children[4].Value,
                      StringFormat='The Slider value is {0:F2}'}" />
```

The view at index 4 of the Children collection is a Slider (down at the bottom of the markup, with no attributes set), which has a Value property, and that's what's displayed here.

The third Binding overrides its inherited BindingContext by setting the Source argument to a static property using x:Static. The globe prefix is defined in the root tag to refer to the .NET System.Globalization namespace, and the Source is set to the CultureInfo object that encapsulates the culture of the user's phone:

```
<Label Text="{Binding Source={x:Static globe:CultureInfo.CurrentCulture},
                      Path=DateTimeFormat.DayNames[3],
                      StringFormat='The middle day of the week is {0}'}" />
```

One of the properties of CultureInfo is DateTimeFormat, which is a DateTimeFormatInfo object that contains information about date and time formatting, including a property named DayNames that is an array of the seven days of the week. The index 3 picks out the middle one.

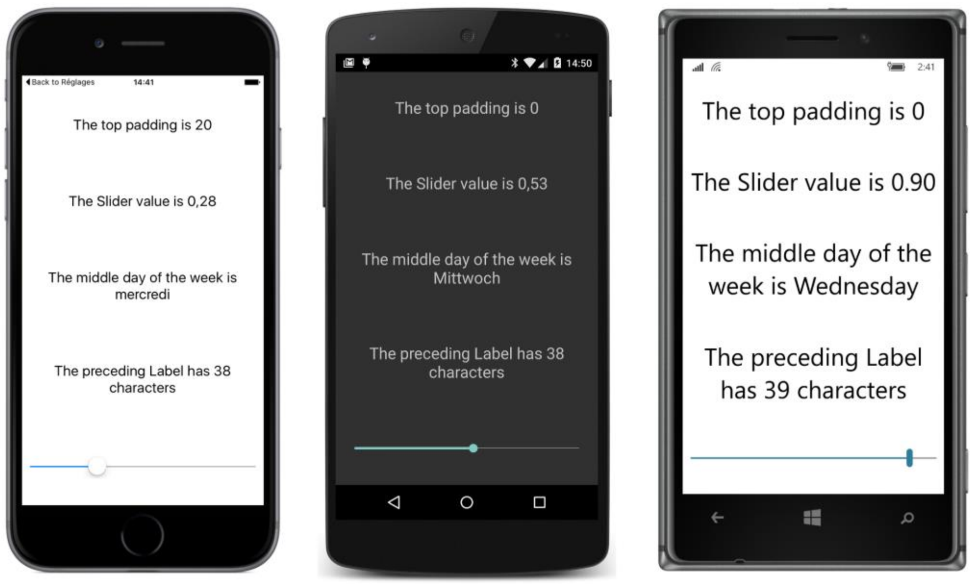
None of the classes in the System.Globalization namespace implement INotifyPropertyChanged, but that's okay because the values of these properties don't change at run time.

The final Binding references the child of the StackLayout with a child index of 2. That's the previous Label. It has a Text property, which is of type string, and string has a Length property:

```
<Label Text="{Binding Path=Content.Children[2].Text.Length,
                      StringFormat='The preceding Label has {0} characters'}" />
```

The binding system installs a property-changed handler for the Text property of the Label, so if it changes, the binding will get the new length.

For the following screenshots, the iOS phone was switched to French, and the Android phone was switched to German. This affects the formatting of the Slider value—notice the comma rather than a period for the decimal divider—and the name of the middle day of the week:



The top padding is 20

The Slider value is 0,28

The middle day of the week is mercredi

The preceding Label has 38 characters

The top padding is 0

The Slider value is 0,53

The middle day of the week is Mittwoch

The preceding Label has 38 characters

The top padding is 0

The Slider value is 0.90

The middle day of the week is Wednesday

The preceding Label has 39 characters

These `Path` specifications can be hard to configure and debug. Keep in mind that class names do not appear in the `Path` specifications—only property names and indexers. Also keep in mind that you can build up a `Path` specification incrementally, testing each new piece with a placeholder of "{0}" in `StringFormat`. This will often display the fully qualified class name of the type of the value set to the last property in the `Path` specification, and that can be very useful information.

You'll also want to keep an eye on the **Output** window in Visual Studio or Xamarin Studio when running your program under the debugger. You'll see messages there relating to run-time errors encountered by the binding infrastructure.

Binding value converters

You now know how to convert any binding source object to a string by using `StringFormat`. But what about other data conversions? Perhaps you're using a `Slider` for a binding source but the target is expecting an integer rather than a double. Or maybe you want to display the value of a `Switch` as text, but you want "Yes" and "No" rather than "True" and "False".

The tool for this job is a class—often a very tiny class—informally called a *value converter* or (sometimes) a *binding converter*. More formally, such a class implements the `IValueConverter` interface. This interface is defined in the `Xamarin.Forms` namespace, but it is similar to an interface available in Microsoft's XAML-based environments.

An example: Sometimes applications need to enable or disable a `Button` based on the presence of text in an `Entry`. Perhaps the `Button` is labeled **Save** and the `Entry` is a filename. Or the `Button` is labeled **Send** and the `Entry` contains a mail recipient. The `Button` shouldn't be enabled unless the `Entry` contains at least one character of text.

There are a couple of ways to do this job. In a later chapter, you'll see how a data trigger can do it (and can also perform validity checks of the text in the `Entry`). But for this chapter, let's do it with a value converter.

The data-binding target is the `IsEnabled` property of the `Button`. That property is of type `bool`. The binding source is the `Text` property of an `Entry`, or rather the `Length` property of that `Text` property. That `Length` property is of type `int`. The value converter needs to convert an `int` equal to 0 to a `bool` of `false` and a positive `int` to a `bool` of `true`. The code is trivial. We just need to wrap it in a class that implements `IValueConverter`.

Here is that class in the **Xamarin.FormsBook.Toolkit** library, complete with `using` directives. The `IValueConverter` interface consists of two methods, named `Convert` and `ConvertBack`, with identical parameters. You can make the class as generalized or as specialized as you want:

```
using System;
using System.Globalization;
using Xamarin.Forms;

namespace Xamarin.FormsBook.Toolkit
{
    public class IntToBoolConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            return (int)value != 0;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return (bool)value ? 1 : 0;
        }
    }
}
```

When you include this class in a data binding—and you'll see how to do that shortly—the `Convert` method is called whenever a value passes from the source to the target.

The `value` argument to `Convert` is the value from the data binding source to be converted. You can use `GetType` to determine its type, or you can assume that it's always a particular type. In this example, the `value` argument is assumed to be of type `int`, so casting to an `int` won't raise an exception. More sophisticated value converters can perform more validity checks.

The `targetType` is the type of the data-binding target property. Versatile value converters can use this argument to tailor the conversion for different target types. The `Convert` method should return an object or value that matches this targetType. This particular Convert method assumes that target-Type is bool.

The parameter argument is an optional conversion parameter that you can specify as a property to the Binding class. (You'll see an example in Chapter 18, "MVVM.")

Finally, if you need to perform a culture-specific conversion, the last argument is the CultureInfo object that you should use.

The body of this particular Convert method assumes that value is an int, and the method returns a bool that is true if that integer is nonzero.

The ConvertBack method is called only for TwoWay or OneWayToSource bindings. For the Con-vertBack method, the value argument is the value from the target and the targetType argument is actually the type of the source property. If you know that the ConvertBack method will never be called, you can simply ignore all the arguments and return null or 0 from it. With some value convert-ers, implementing a ConvertBack body is virtually impossible, but sometimes it's fairly simple (as in this case).

When you use a value converter in code, you set an instance of the converter to the Converter property of Binding. You can optionally pass an argument to the value converter by setting the Con-verterParameter property of Binding.

If the binding also has a StringFormat, the value that is returned by the value converter is the value that is formatted as a string.

Generally, in a XAML file you'll want to instantiate the value converter in a Resources dictionary and then reference it in the Binding expression by using StaticResource. The value converter shouldn't maintain state and can thus be shared among multiple bindings.

Here's the **ButtonEnabler** program that uses the value converter:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="ButtonEnabler.ButtonEnablerPage"
             Padding="10, 50, 10, 0">

    <ContentPage.Resources>
        <ResourceDictionary>
            <toolkit:IntToBoolConverter x:Key="intToBool" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Spacing="20">
        <Entry x:Name="entry"
               Text=""
               Placeholder="text to enable button" />
        <Button Text="Save or Send (or something)"
                FontSize="Medium"
                HorizontalOptions="Center"
                IsEnabled="{Binding Source={x:Reference entry},
                                    Path=Text.Length,
                                    Converter={StaticResource intToBool}}" />
    </StackLayout>
</ContentPage>
```

The IntToBoolConverter is instantiated in the Resources dictionary and referenced as a nested markup extension in the Binding that is set on the IsEnabled property of the Button.

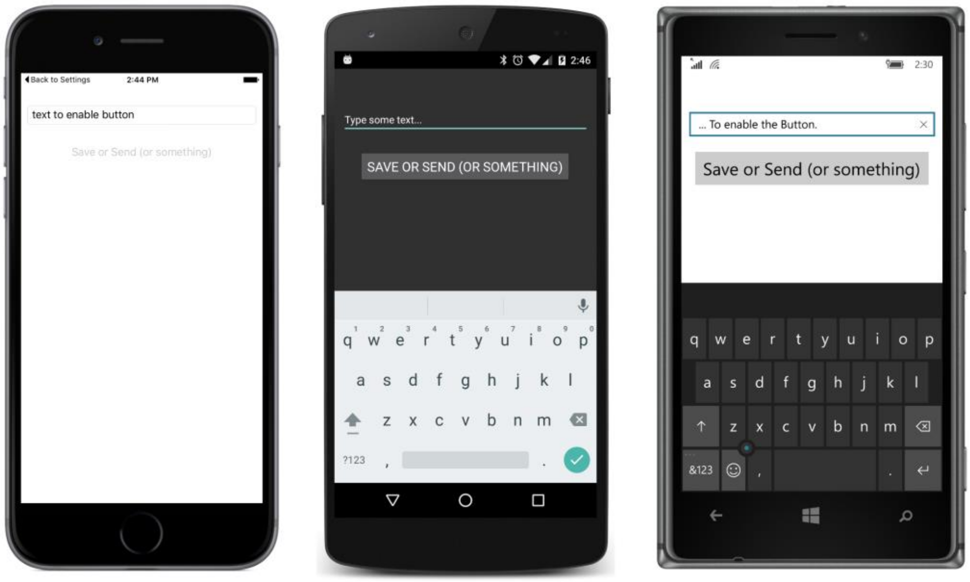
Notice that the Text property is explicitly initialized in the Entry tag to an empty string. By default, the Text property is null, which means that the binding Path setting of Text.Length doesn't result in a valid value.

You might remember from previous chapters that a class in the **Xamarin.FormsBook.Toolkit** library that is referenced only in XAML is not sufficient to establish a link from the application to the library. For that reason, the App constructor in **ButtonEnabler** calls Toolkit.Init:

```
public class App : Application
{
    public App()
    {
        Xamarin.FormsBook.Toolkit.Toolkit.Init();

        MainPage = new ButtonEnablerPage();
    }
    …
}
```

Similar code appears in all the programs in this chapter that use the **Xamarin.FormsBook.Toolkit** library.

The screenshots confirm that the Button is not enabled unless the Entry contains some text:


Back to Settings
2:44 PM
text to enable button
Save or Send (or something)
2:46
Type some text...
SAVE OR SEND (OR SOMETHING)
2:30
... To enable the Button.
Save or Send (or something)

If you're using only one instance of a value converter, you don't need to store it in the `Resources` dictionary. You can instantiate it right in the `Binding` tag with the use of property-element tags for the target property and for the `Converter` property of `Binding`:

```
<Button Text="Save or Send (or something)"
        FontSize="Large"
        HorizontalOptions="Center">
    <Button.IsEnabled>
        <Binding Source="{x:Reference entry}"
                 Path="Text.Length">
            <Binding.Converter>
                <toolkit:IntToBoolConverter />
            </Binding.Converter>
        </Binding>
    </Button.IsEnabled>
</Button>
```

Sometimes it's convenient for a value converter to define a couple of simple properties. For example, suppose you want to display some text for the two settings of a `Switch` but you don't want to use "True" and "False", and you don't want to hard-code alternatives into the value converter. Here's a `BoolToStringConverter` with a pair of public properties for two text strings:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class BoolToStringConverter : IValueConverter
    {
        public string TrueText { set; get; }

        public string FalseText { set; get; }
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            return (bool)value ? TrueText : FalseText;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return false;
        }
    }
}
```

The body of the `Convert` method is trivial: it just selects between the two strings based on the Boolean `value` argument.

A similar value converter converts a Boolean to one of two colors:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class BoolToColorConverter : IValueConverter
    {
        public Color TrueColor { set; get; }

        public Color FalseColor { set; get; }

        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            return (bool)value ? TrueColor : FalseColor;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return false;
        }
    }
}
```

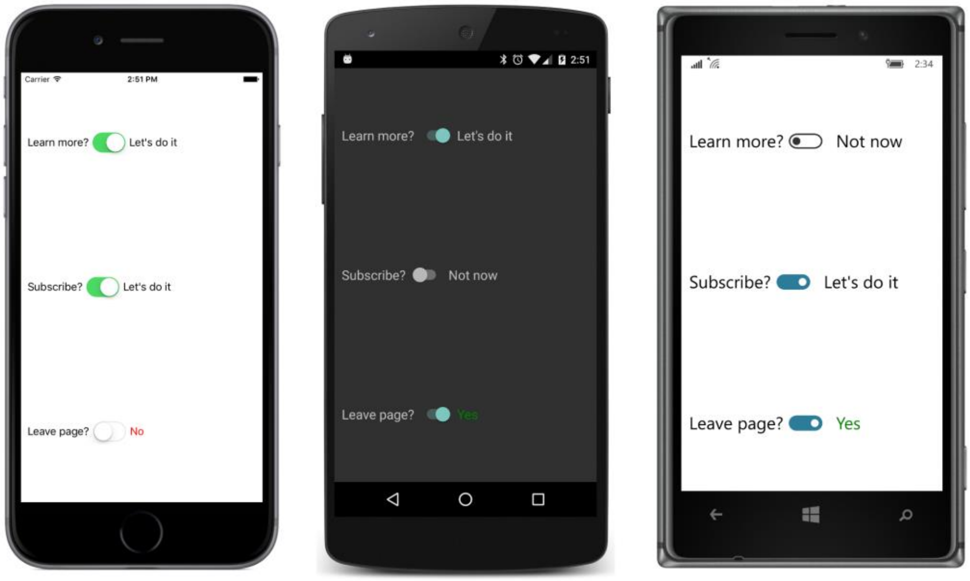
The **SwitchText** program instantiates the `BoolToStringConverter` converter twice for two different pairs of strings: once in the `Resources` dictionary, and then within `Binding.Converter` property-element tags. Two properties of the final `Label` are subjected to the `BoolToStringConverter` and the `BoolToColorConverter` based on the same `IsToggled` property from the `Switch`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="SwitchText.SwitchTextPage"
             Padding="10, 0">
```

```
    <ContentPage.Resources>
        <ResourceDictionary>
            <toolkit:BoolToStringConverter x:Key="boolToString"
                                           TrueText="Let's do it"
                                           FalseText="Not now" />

            <Style TargetType="Label">
                <Setter Property="FontSize" Value="Medium" />
                <Setter Property="VerticalOptions" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <!-- First Switch with text. -->
        <StackLayout Orientation="Horizontal"
                     VerticalOptions="CenterAndExpand">
            <Label Text="Learn more?" />

            <Switch x:Name="switch1"
                    VerticalOptions="Center" />

            <Label Text="{Binding Source={x:Reference switch1},
                                  Path=IsToggled,
                                  Converter={StaticResource boolToString}}"
                   HorizontalOptions="FillAndExpand" />
        </StackLayout>

        <!-- Second Switch with text. -->
        <StackLayout Orientation="Horizontal"
                     VerticalOptions="CenterAndExpand">
            <Label Text="Subscribe?" />

            <Switch x:Name="switch2"
                    VerticalOptions="Center" />

            <Label Text="{Binding Source={x:Reference switch2},
                                  Path=IsToggled,
                                  Converter={StaticResource boolToString}}"
                   HorizontalOptions="FillAndExpand" />
        </StackLayout>

        <!-- Third Switch with text and color. -->
        <StackLayout Orientation="Horizontal"
                     VerticalOptions="CenterAndExpand">
            <Label Text="Leave page?" />

            <Switch x:Name="switch3"
                    VerticalOptions="Center" />

            <Label HorizontalOptions="FillAndExpand">
                <Label.Text>
                    <Binding Source="{x:Reference switch3}"
                             Path="IsToggled">
```

```
                        <Binding.Converter>
                            <toolkit:BoolToStringConverter TrueText="Yes"
                                                           FalseText="No" />
                        </Binding.Converter>
                    </Binding>
                </Label.Text>

                <Label.TextColor>
                    <Binding Source="{x:Reference switch3}"
                             Path="IsToggled">
                        <Binding.Converter>
                            <toolkit:BoolToColorConverter TrueColor="Green"
                                                          FalseColor="Red" />
                        </Binding.Converter>
                    </Binding>
                </Label.TextColor>
            </Label>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

With the two fairly trivial binding converters, the `Switch` can now display whatever text you want for the two states and can color that text with custom colors:



Now that you've seen a `BoolToStringConverter` and a `BoolToColorConverter`, can you generalize the technique to objects of any type? Here is a generic `BoolToObjectConverter` also in the **Xamarin.FormsBook.Toolkit** library:

```
public class BoolToObjectConverter<T> : IValueConverter
{
    public T TrueObject { set; get; }
    public T FalseObject { set; get; }

    public object Convert(object value, Type targetType,
                          object parameter, CultureInfo culture)
    {
        return (bool)value ? this.TrueObject : this.FalseObject;
    }

    public object ConvertBack(object value, Type targetType,
                              object parameter, CultureInfo culture)
    {
        return ((T)value).Equals(this.TrueObject);
    }
}
```

The next sample uses this class.

Bindings and custom views

In Chapter 15, "The interactive interface," you saw a custom view named `CheckBox`. This view defines a `Text` property for setting the text of the `CheckBox` as well as a `FontSize` property. It could also have defined all the other text-related properties—`TextColor`, `FontAttributes`, and `FontFamily`—but it did not, mostly because of the work involved. Each property requires a `BindableProperty` definition, a CLR property definition, and a property-changed handler that transfers the new setting of the property to the `Label` views that comprise the visuals of the `CheckBox`.

Data bindings can help simplify this process for some properties by eliminating the property-changed handlers. Here's the code-behind file for a new version of `CheckBox` called `NewCheckBox`. Like the earlier class, it's part of the **Xamarin.FormsBook.Toolkit** library. The file has been reorganized a bit so that each `BindableProperty` definition is paired with its corresponding CLR property definition. You might prefer this type of source-code organization of the properties, or perhaps not.

```
namespace Xamarin.FormsBook.Toolkit
{
    public partial class NewCheckBox : ContentView
    {
        public event EventHandler<bool> CheckedChanged;

        public NewCheckBox()
        {
            InitializeComponent();
        }

        // Text property.
        public static readonly BindableProperty TextProperty =
            BindableProperty.Create(
                "Text",
                typeof(string),
                typeof(NewCheckBox),
                null);

        public string Text
        {
            set { SetValue(TextProperty, value); }
            get { return (string)GetValue(TextProperty); }
        }

        // TextColor property.
        public static readonly BindableProperty TextColorProperty =
            BindableProperty.Create(
                "TextColor",
                typeof(Color),
                typeof(NewCheckBox),
                Color.Default);

        public Color TextColor
        {
            set { SetValue(TextColorProperty, value); }
            get { return (Color)GetValue(TextColorProperty); }
        }

        // FontSize property.
        public static readonly BindableProperty FontSizeProperty =
            BindableProperty.Create(
                "FontSize",
                typeof(double),
                typeof(NewCheckBox),
                Device.GetNamedSize(NamedSize.Default, typeof(Label)));

        [TypeConverter(typeof(FontSizeConverter))]
        public double FontSize
        {
            set { SetValue(FontSizeProperty, value); }
            get { return (double)GetValue(FontSizeProperty); }
        }

        // FontAttributes property.
        public static readonly BindableProperty FontAttributesProperty =
            BindableProperty.Create(
                "FontAttributes",
                typeof(FontAttributes),
                typeof(NewCheckBox),
                FontAttributes.None);

        public FontAttributes FontAttributes
        {
            set { SetValue(FontAttributesProperty, value); }
            get { return (FontAttributes)GetValue(FontAttributesProperty); }
        }

        // IsChecked property.
        public static readonly BindableProperty IsCheckedProperty =
            BindableProperty.Create(
                "IsChecked",
                typeof(bool),
                typeof(NewCheckBox),
                false,
                propertyChanged: (bindable, oldValue, newValue) =>
                {
                    // Fire the event.
                    NewCheckBox checkbox = (NewCheckBox)bindable;
                    EventHandler<bool> eventHandler = checkbox.CheckedChanged;
                    if (eventHandler != null)
                    {
                        eventHandler(checkbox, (bool)newValue);
                    }
                });

        public bool IsChecked
        {
            set { SetValue(IsCheckedProperty, value); }
            get { return (bool)GetValue(IsCheckedProperty); }
        }

        // TapGestureRecognizer handler.
        void OnCheckBoxTapped(object sender, EventArgs args)
        {
            IsChecked = !IsChecked;
        }
    }
}
```

Besides the earlier `Text` and `FontSize` properties, this code file now also defines `TextColor` and `FontAttributes` properties. However, the only property-changed handler is for the `IsChecked` handler to fire the `CheckedChanged` event. Everything else is handled by data bindings in the XAML file:

```
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit="clr-namespace:Xamarin.FormsBook.Toolkit"
             x:Class="Xamarin.FormsBook.Toolkit.NewCheckBox"
             x:Name="checkbox">

    <StackLayout Orientation="Horizontal"
                 BindingContext="{x:Reference checkbox}">

        <Label x:Name="boxLabel" Text="&#x2610;"
                                 TextColor="{Binding TextColor}"
                                 FontSize="{Binding FontSize}">
            <Label.Text>
                <Binding Path="IsChecked">
                    <Binding.Converter>
                        <toolkit:BoolToStringConverter TrueText="&#x2611;"
                                                       FalseText="&#x2610;" />
                    </Binding.Converter>
                </Binding>
            </Label.Text>
        </Label>

        <Label x:Name="textLabel" Text="{Binding Path=Text}"
                                  TextColor="{Binding TextColor}"
                                  FontSize="{Binding FontSize}"
                                  FontAttributes="{Binding FontAttributes}" />
    </StackLayout>

    <ContentView.GestureRecognizers>
        <TapGestureRecognizer Tapped="OnCheckBoxTapped" />
    </ContentView.GestureRecognizers>
</ContentView>
```

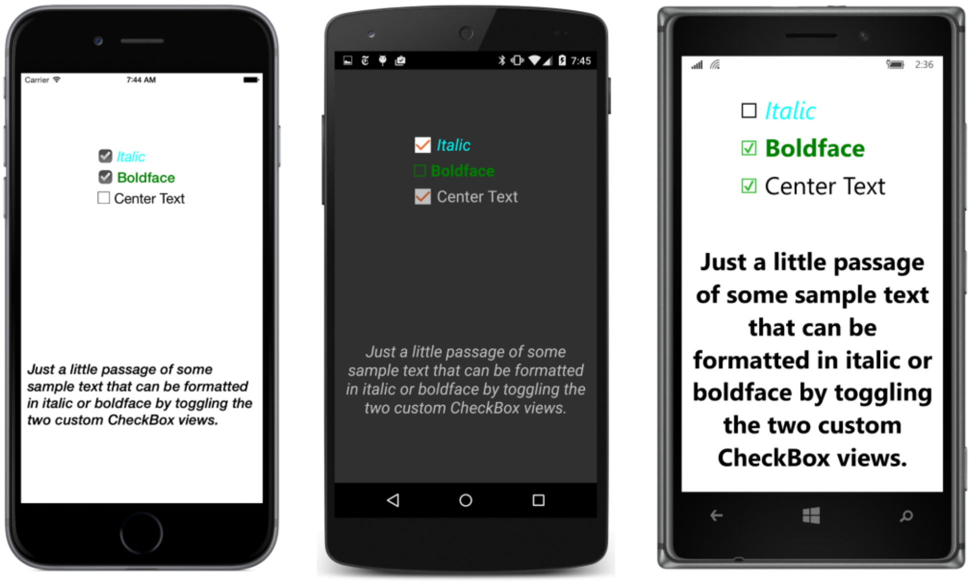
The root element is given a name of checkbox, and the StackLayout sets that as its Binding-Context. All the data bindings within that StackLayout can then refer to properties defined by the code-behind file. The first Label that displays the box has its TextColor and FontSize properties bound to the values of the underlying properties, while the Text property is targeted by a binding that uses a BoolToStringConverter to display an empty box or a checked box based on the IsChecked property. The second Label is more straightforward: the Text, TextColor, FontSize, and FontAttributes properties are all bound to the corresponding properties defined in the code-behind file.

If you'll be creating several custom views that include Text elements and you need definitions of all the text-related properties, you'll probably want to first create a code-only class (named CustomViewBase, for example) that derives from ContentView and includes only those text-based property definitions. You can then derive other classes from CustomViewBase and have Text and all the text-related properties readily available.

Let's write a little program called **NewCheckBoxDemo** that demonstrates the NewCheckBox view. Like the earlier **CheckBoxDemo** program, these check boxes control the bold and italic formatting of a paragraph of text. But to demonstrate the new properties, these check boxes are given colors and font attributes, and to demonstrate the BoolToObjectConverter, one of the check boxes controls the horizontal alignment of that paragraph:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="NewCheckBoxDemo.NewCheckBoxDemoPage">

    <StackLayout Padding="10, 0">
        <StackLayout HorizontalOptions="Center"
                     VerticalOptions="CenterAndExpand">

            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style TargetType="toolkit:NewCheckBox">
                        <Setter Property="FontSize" Value="Large" />
                    </Style>
                </ResourceDictionary>
            </StackLayout.Resources>
            <toolkit:NewCheckBox Text="Italic"
                                 TextColor="Aqua"
                                 FontSize="Large"
                                 FontAttributes="Italic"
                                 CheckedChanged="OnItalicCheckBoxChanged" />

            <toolkit:NewCheckBox Text="Boldface"
                                 FontSize="Large"
                                 TextColor="Green"
                                 FontAttributes="Bold"
                                 CheckedChanged="OnBoldCheckBoxChanged" />

            <toolkit:NewCheckBox x:Name="centerCheckBox"
                                 Text="Center Text" />
        </StackLayout>

        <Label x:Name="label"
               Text=
"Just a little passage of some sample text that can be formatted
in italic or boldface by toggling the two custom CheckBox views."
               FontSize="Large"
               VerticalOptions="CenterAndExpand">
            <Label.HorizontalTextAlignment>
                <Binding Source="{x:Reference centerCheckBox}"
                         Path="IsChecked">
                    <Binding.Converter>
                        <toolkit:BoolToObjectConverter x:TypeArguments="TextAlignment"
                                                       TrueObject="Center"
                                                       FalseObject="Start" />
                    </Binding.Converter>
                </Binding>
            </Label.HorizontalTextAlignment>
        </Label>
    </StackLayout>
</ContentPage>
```

Notice the `BoolToObjectConverter` between the `Binding.Converter` tags. Because it's a generic class, it requires an `x:TypeArguments` attribute that indicates the type of the `TrueObject` and `FalseObject` properties and the type of the return value of the `Convert` method. Both `TrueObject` and `FalseObject` are set to members of the `TextAlignment` enumeration, and the converter selects one to be set to the `HorizontalTextAlignment` property of the `Label`, as the following screenshots demonstrate:


Carrier
7:44 AM
Italic
Boldface
Center Text
Just a little passage of some sample text that can be formatted in italic or boldface by toggling the two custom CheckBox views.
7:45
Italic
Boldface
Center Text
Just a little passage of some sample text that can be formatted in italic or boldface by toggling the two custom CheckBox views.
2:36
Italic
Boldface
Center Text
Just a little passage of some sample text that can be formatted in italic or boldface by toggling the two custom CheckBox views.

However, this program still needs a code-behind file to manage applying the italic and boldface attributes to the block of text. These methods are identical to those in the early **CheckBoxDemo** program:

```
public partial class NewCheckBoxDemoPage : ContentPage
{
    public NewCheckBoxDemoPage()
    {
        InitializeComponent();
    }

    void OnItalicCheckBoxChanged(object sender, bool isChecked)
    {
        if (isChecked)
        {
            label.FontAttributes |= FontAttributes.Italic;
        }
        else
        {
            label.FontAttributes &= ~FontAttributes.Italic;
        }
    }

    void OnBoldCheckBoxChanged(object sender, bool isChecked)
    {
        if (isChecked)
        {
            label.FontAttributes |= FontAttributes.Bold;
        }
        else
        {
            label.FontAttributes &= ~FontAttributes.Bold;
        }
    }
}
```

Xamarin.Forms does not support a “multi-binding” that might allow multiple binding sources to be combined to change a single binding target. Bindings can do a lot, but without some additional code support, they can’t do everything.

There’s still a role for code.