

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:

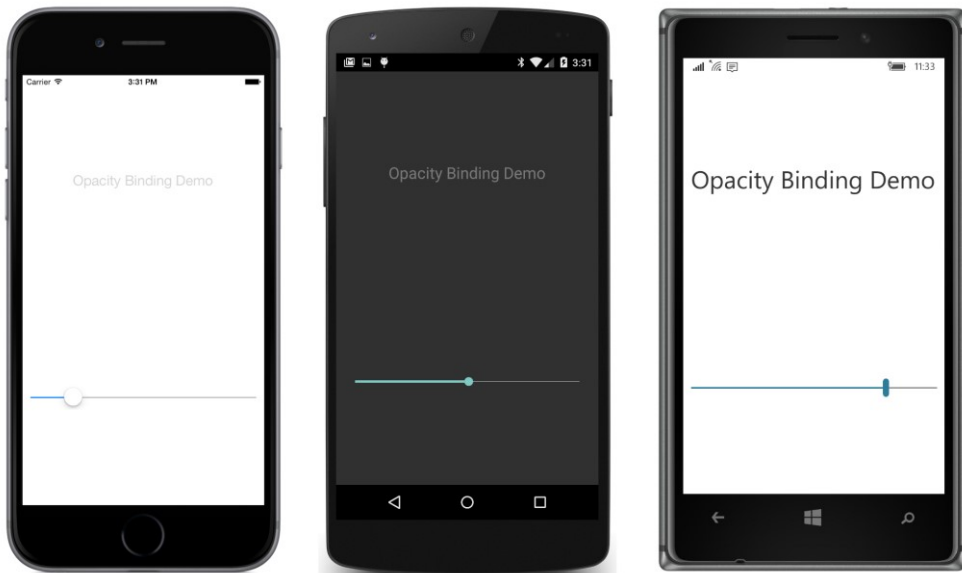
```
Label.BindingContext = slider;
```

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 `INotifyPropertyChanged` 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 **OpacityBindingCode** 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();
    }

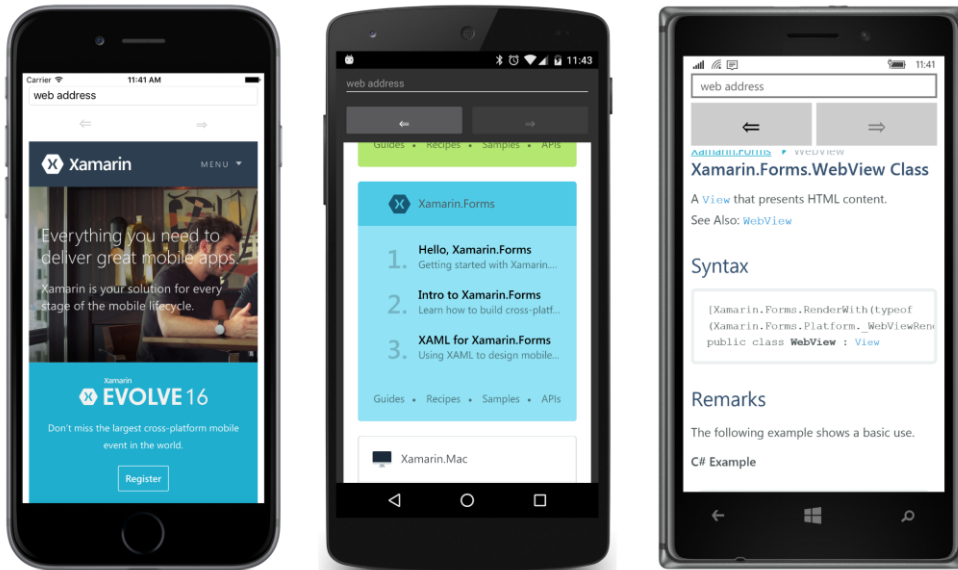
    void OnEntryCompleted(object sender, EventArgs args)
    {
        webView.Source = ((Entry)sender).Text;
    }

    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:



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 <code>TwoWay</code>
<code>Slider</code>	<code>Value</code>
<code>Stepper</code>	<code>Value</code>
<code>Switch</code>	<code>IsToggled</code>
<code>Entry</code>	<code>Text</code>
<code>Editor</code>	<code>Text</code>
<code>SearchBar</code>	<code>Text</code>
<code>DatePicker</code>	<code>Date</code>
<code>TimePicker</code>	<code>Time</code>

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>
  </StackLayout>
</ContentPage>
```

```

<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>
```

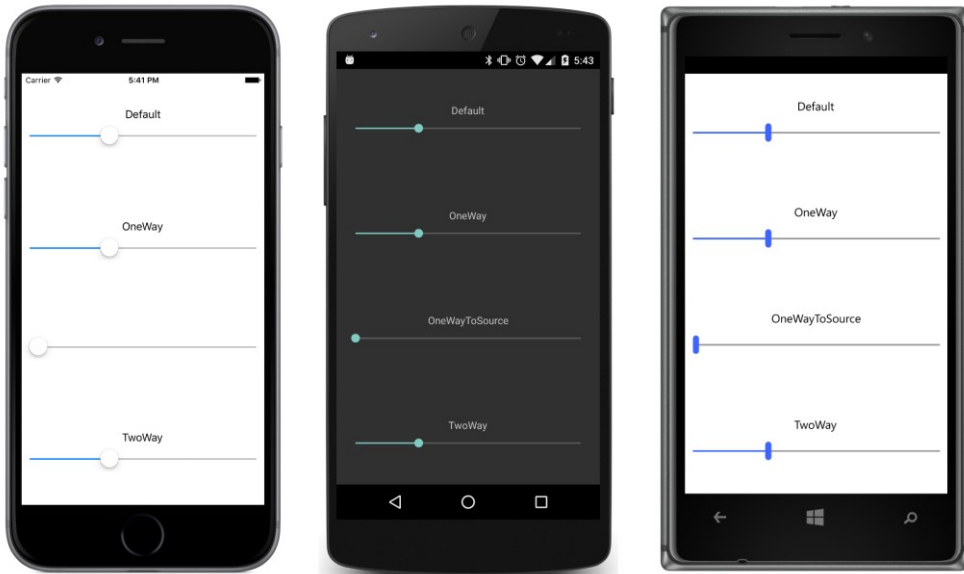
```
<StackLayout>
  <Label x:Name="Label2"
        Text="OneWay" />
  <Slider Maximum="50"
        Value="{Binding Source={x:Reference Label2},
                    Path=FontSize,
                    Mode=OneWay}" />
</StackLayout>

<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 `BindableProperty.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>

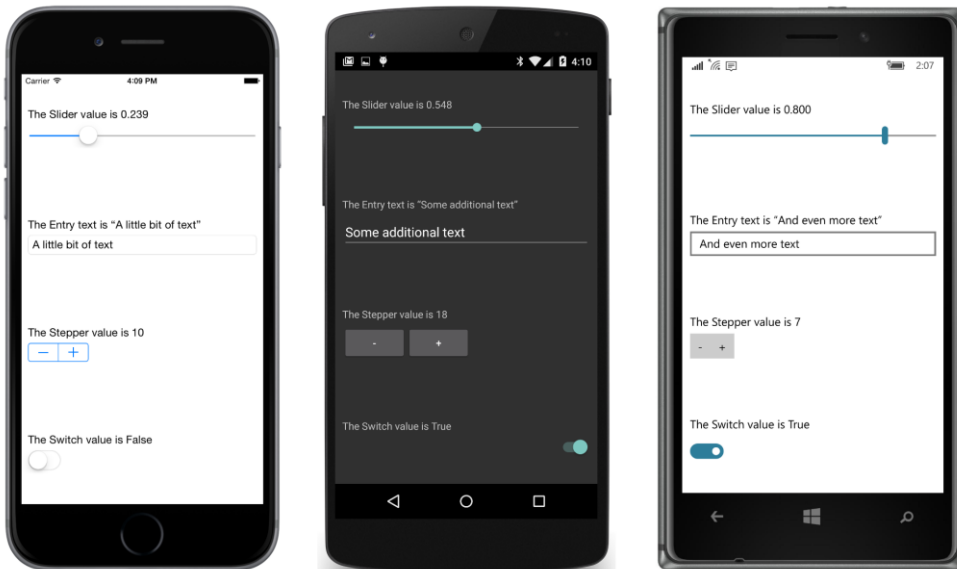
<StackLayout VerticalOptions="CenterAndExpand">
    <Label Text="{Binding Source={x:Reference stepper},
        Path=Value,
        StringFormat='The Stepper value is {0}'}" />
    <Stepper x:Name="stepper" />
</StackLayout>

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



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=mcorlib"
  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.

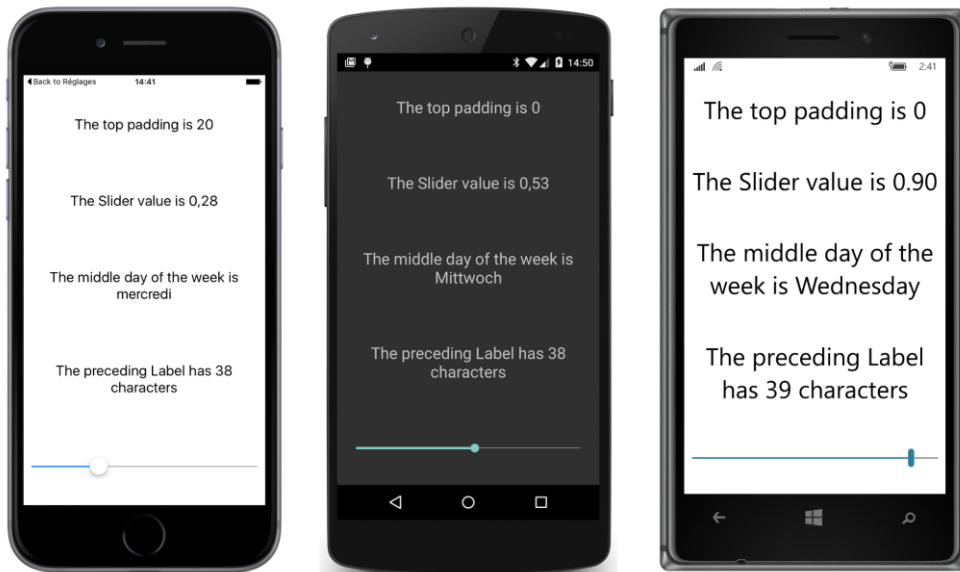
None of the classes in the `System.Globalization` namespace implement `INotifyPropertyChanged`, but that's okay because the values of these properties don't change at run time.

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:



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 `targetType` 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 `ConvertBack` 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 converters, 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 `ConverterParameter` 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" />
    </StackLayout>
</ContentPage>
```

```
<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.

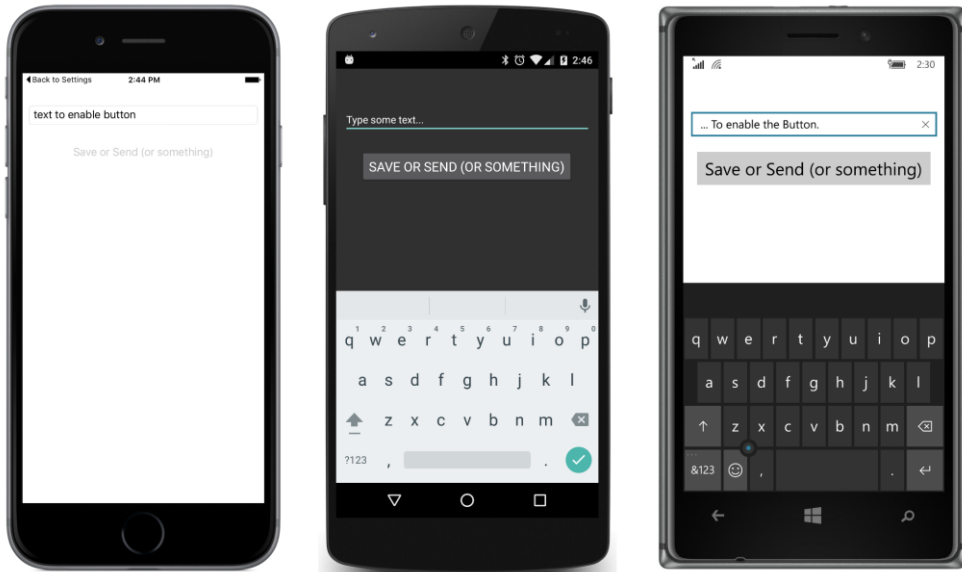
You might remember from previous chapters that a class in the **Xamarin.FormsBook.Toolkit** library that is referenced only in XAML is not sufficient to establish a link from the application to the library. For that reason, the `App` constructor in **ButtonEnabler** calls `Toolkit.Init`:

```
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:



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">

```



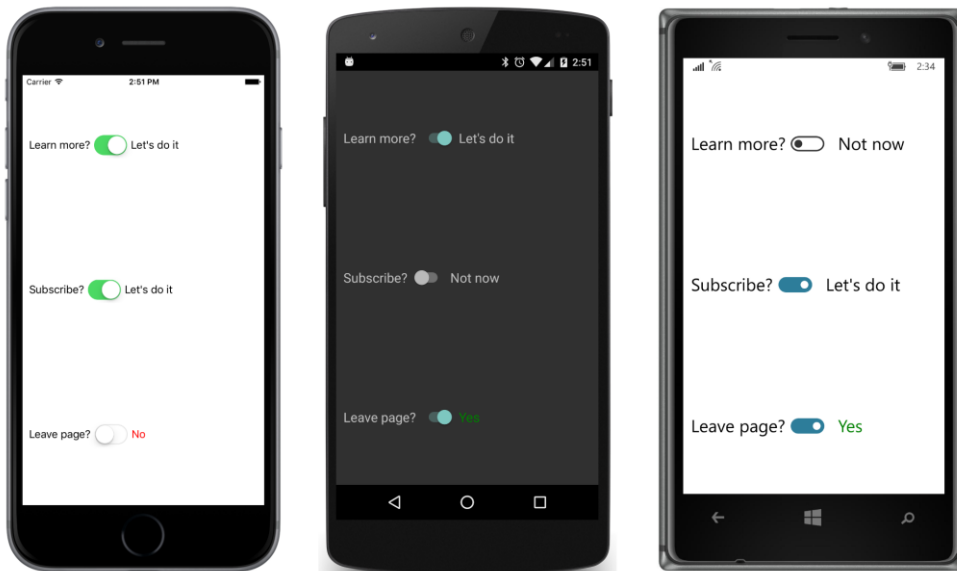
```

<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 `BindingContext`. 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" />

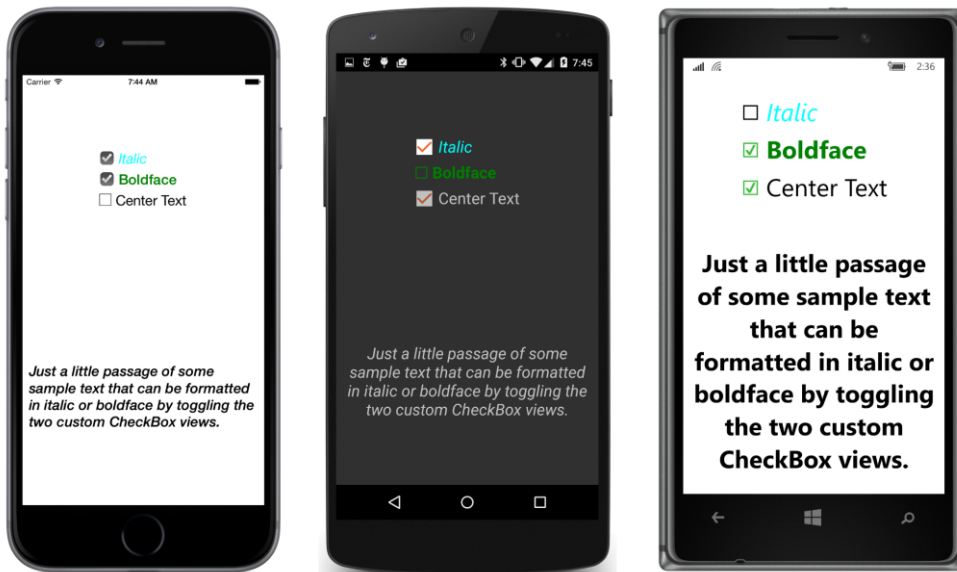
<toolkit:NewCheckBox Text="Boldface"
    FontSize="Large"
    TextColor="Green"
    FontAttributes="Bold"
    CheckedChanged="OnBoldCheckBoxChanged" />

<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:



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.

Chapter 17

Mastering the Grid

The `Grid` is a powerful layout mechanism that organizes its children into rows and columns of cells. At first, the `Grid` seems to resemble the HTML `table`, but there is a very important distinction: The HTML `table` is designed for presentation purposes, while the `Grid` is solely for layout. There is no concept of a heading in a `Grid`, for example, and no built-in feature to draw boxes around the cells or to separate rows and columns with divider lines. The strengths of the `Grid` are in specifying cell dimensions with three options of height and width settings.

As you've seen, the `StackLayout` is ideal for one-dimensional collections of children. Although it's possible to nest a `StackLayout` within a `StackLayout` to accommodate a second dimension and mimic a table, often the result can exhibit alignment problems. The `Grid`, however, is designed specifically for two-dimensional arrays of children. As you'll see toward the end of this chapter, the `Grid` can also be very useful for managing layouts that adapt to both portrait and landscape modes.

The basic Grid

A `Grid` can be defined and filled with children in either code or XAML, but the XAML approach is easier and clearer, and hence by far the more common.

The Grid in XAML

When defined in XAML, a `Grid` almost always has a fixed number of rows and columns. The `Grid` definition generally begins with two important properties, named `RowDefinitions` (which is a collection of `RowDefinition` objects) and `ColumnDefinitions` (a collection of `ColumnDefinition` objects). These collections contain one `RowDefinition` for every row in the `Grid` and one `ColumnDefinition` for every column, and they define the row and column characteristics of the `Grid`.

A `Grid` can consist of a single row or single column (in which case it doesn't need one of the two `Definitions` collections), or even just a single cell.

`RowDefinition` has a `Height` property of type `GridLength`, and `ColumnDefinition` has a `Width` property, also of type `GridLength`. The `GridLength` structure specifies a row height or a column width in terms of the `GridUnitType` enumeration, which has three members:

- `Absolute`—the width or height is a value in device-independent units (a number in XAML)
- `Auto`—the width or height is autosized based on the cell contents (“Auto” in XAML)
- `Star`—leftover width or height is allocated proportionally (a number with “*” in XAML)

Here's the first half of the XAML file in the **SimpleGridDemo** project:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SimpleGridDemo.SimpleGridDemoPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="100" />
            <RowDefinition Height="2*" />
            <RowDefinition Height="1*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        ...

    </Grid>
</ContentPage>
```

This `Grid` has four rows and two columns. The height of the first row is “Auto”—meaning that the height is calculated based on the maximum height of all the elements occupying that first row. The second row is 100 device-independent units in height.

The two `Height` settings using “*” (pronounced “star”) require some additional explanation: This particular `Grid` has an overall height that is the height of the page minus the `Padding` setting on iOS. Internally, the `Grid` determines the height of the first row based on the contents of that row, and it knows that the height of the second row is 100. It subtracts those two heights from its own height and allocates the remaining height proportionally among the third and fourth rows based on the number in the star setting. The third row is twice the height of the fourth row.

The two `ColumnDefinition` objects both set the `Width` equal to “*,” which is the same as “1*,” which means that the width of the screen is divided equally between the two columns.

You’ll recall from Chapter 14, “Absolute layout,” that the `AbsoluteLayout` class defines two attached bindable properties and four static `Set` and `Get` methods that allow a program to specify the position and size of a child of the `AbsoluteLayout` in code or XAML.

The `Grid` is quite similar. The `Grid` class defines four attached bindable properties for specifying the cell or cells that a child of the `Grid` occupies:

- `Grid.RowProperty`—the zero-based row; default value is 0

- `Grid.ColumnProperty`—the zero-based column; default value is 0
- `Grid.RowSpanProperty`—the number of rows that the child spans; default value is 1
- `Grid.ColumnSpanProperty`—the number of columns that the child spans; default value is 1

All four properties are defined to be of type `int`.

For example, to specify in code that a `Grid` child named `view` resides in a particular row and column, you can call:

```
view.SetValue(Grid.RowProperty, 2);
view.SetValue(Grid.ColumnProperty, 1);
```

Those are zero-based row and column numbers, so the child is assigned to the third row and the second column.

The `Grid` class also defines eight static methods for streamlining the setting and getting of these properties in code:

- `Grid.SetRow` and `Grid.GetRow`
- `Grid.SetColumn` and `Grid.GetColumn`
- `Grid.SetRowSpan` and `Grid.GetRowSpan`
- `Grid.SetColumnSpan` and `Grid.GetColumnSpan`

Here's the equivalent of the two `SetValue` calls you just saw:

```
Grid.SetRow(view, 2);
Grid.SetColumn(view, 1);
```

As you learned in connection with `AbsoluteLayout`, such static `Set` and `Get` methods are implemented with `SetValue` and `GetValue` calls on the child of `Grid`. For example, here's how `SetRow` is very likely defined within the `Grid` class:

```
public static void SetRow(BindableObject bindable, int value)
{
    bindable.SetValue(Grid.RowProperty, value);
}
```

You cannot call these methods in XAML, so instead you use the following attributes for setting the attached bindable properties on a child of the `Grid`:

- `Grid.Row`
- `Grid.Column`
- `Grid.RowSpan`
- `Grid.ColumnSpan`

These XAML attributes are not actually defined by the `Grid` class, but the XAML parser knows that it must reference the associated attached bindable properties defined by `Grid`.

You don't need to set all these properties on every child of the `Grid`. If the child occupies just one cell, then don't set `Grid.RowSpan` or `Grid.ColumnSpan` because the default value is 1. The `Grid.Row` and `Grid.Column` properties have a default value of 0, so you don't need to set the values if the child occupies the first row or first column. However, for purposes of clarity, the code in this book will usually show the settings of these two properties. To save space, often these attributes will appear on the same line in the XAML listing.

Here's the complete XAML file for **SimpleGridDemo**:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SimpleGridDemo.SimpleGridDemoPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                   iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="100" />
            <RowDefinition Height="2*" />
            <RowDefinition Height="1*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Label Text="Grid Demo"
              Grid.Row="0" Grid.Column="0"
              FontSize="Large"
              HorizontalOptions="End" />

        <Label Text="Demo the Grid"
              Grid.Row="0" Grid.Column="1"
              FontSize="Small"
              HorizontalOptions="End"
              VerticalOptions="End" />

        <Image BackgroundColor="Gray"
              Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2">
            <Image.Source>
                <OnPlatform x:TypeArguments="ImageSource"
                           iOS="Icon-60.png"
                           Android="icon.png"
                           WinPhone="Assets/StoreLogo.png" />
            </Image.Source>
        </Image>
    </Grid>
</ContentPage>
```

```

        </Image.Source>
    </Image>

    <BoxView Color="Green"
            Grid.Row="2" Grid.Column="0" />

    <BoxView Color="Red"
            Grid.Row="2" Grid.Column="1" Grid.RowSpan="2" />

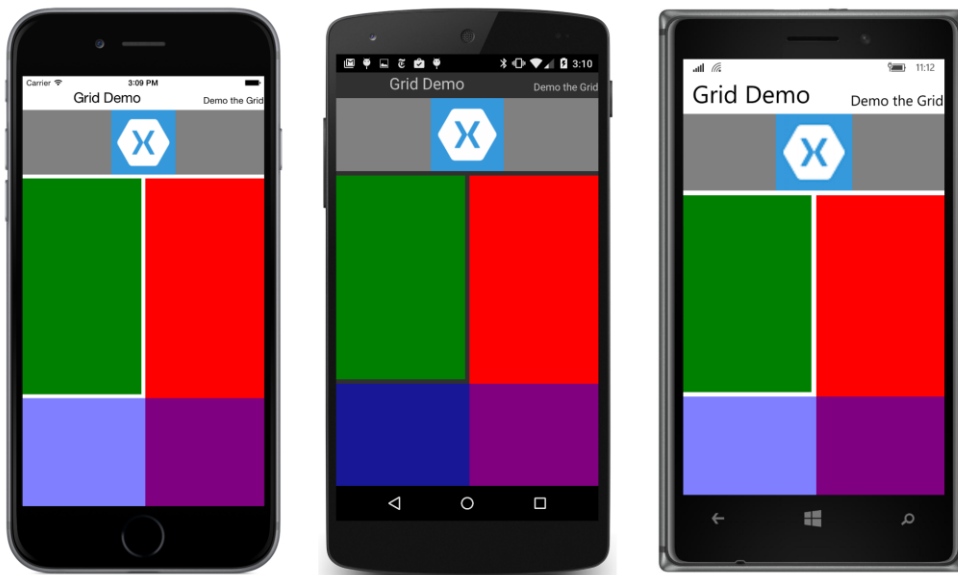
    <BoxView Color="Blue"
            Opacity="0.5"
            Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2" />
    </Grid>
</ContentPage>

```

Two `Label` elements with different `FontSize` settings occupy the two columns of the first row. The height of that row is governed by the tallest element. Settings of `HorizontalOptions` and `VerticalOptions` can position a child within the cell.

The second row has a height of 100 device-independent units. That row is occupied by an `Image` element displaying an application icon with a gray background. The `Image` element spans both columns of that row.

The bottom two rows are occupied by three `BoxView` elements, one that spans two rows, and another that spans two columns, and these overlap in the bottom right cell:



The screenshots confirm that the first row is sized to the height of the large `Label`; the second row is 100 device-independent units tall; and the third and fourth rows occupy all the remaining space. The third row is twice as tall as the fourth. The two columns are equal in width and divide the entire `Grid`

in half. The red and blue `BoxView` elements overlap in the bottom right cell, but the blue `BoxView` is obviously sitting on top of the red one because it has an `Opacity` setting of 0.5 and the result is purple.

The left half of the blue semitransparent `BoxView` is lighter on the iPhone and Windows 10 Mobile device than on the Android phone because of the white background.

As you can see, children of the `Grid` can share cells. The order that the children appear in the XAML file is the order that the children are put into the `Grid`, with later children seemingly sitting on top of (and obscuring) earlier children.

You'll notice that a little gap seems to separate the rows and columns where the background peeks through. This is governed by two `Grid` properties:

- `RowSpacing`—default value of 6
- `ColumnSpacing`—default value of 6

You can set these properties to 0 if you want to close up that space, and you can set the `BackgroundColor` property of the `Grid` if you want the color peeking through to be something different. You can also add space on the inside of the `Grid` around its perimeter with a `Padding` setting on the `Grid`.

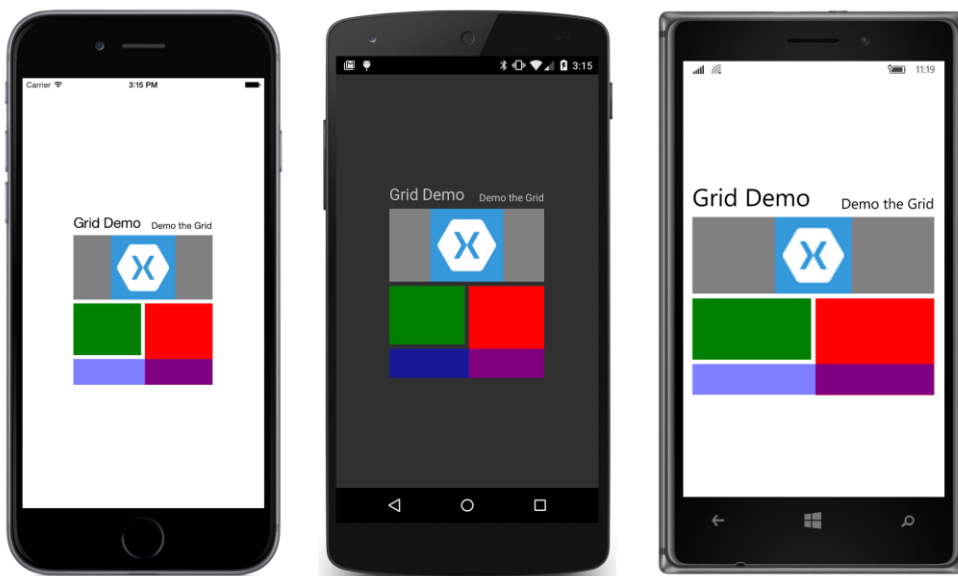
You have now been introduced to all the public properties and methods defined by `Grid`.

Before moving on, let's perform a couple of experiments with **SimpleGridDemo**. First, comment out or delete the entire `RowDefinitions` and `ColumnDefinitions` section near the top of the `Grid`, and then redeploy the program. Here's what you'll see:



When you don't define your own `RowDefinition` and `ColumnDefinition` objects, the `Grid` generates them automatically as views are added to the `Children` collection. However, the default `RowDefinition` and `ColumnDefinition` is "*" (star), meaning that the four rows now equally divide the screen in quarters, and each cell is one-eighth of the total `Grid`.

Here's another experiment. Restore the `RowDefinitions` and `ColumnDefinitions` sections and set the `HorizontalOptions` and `VerticalOptions` properties on the `Grid` itself to `Center`. By default these two properties are `Fill`, which means that the `Grid` fills its container. Here's what happens with `Center`:



The third row is still twice the height of the bottom row, but now the bottom row's height is based on the default `HeightRequest` of `BoxView`, which is 40.

You'll see a similar effect when you put a `Grid` in a `StackLayout`. You can also put a `StackLayout` in a `Grid` cell, or another `Grid` in a `Grid` cell, but don't get carried away with this technique: The deeper you nest `Grid` and other layouts, the more the nested layouts will impact performance.

The Grid in code

It is also possible to define a `Grid` entirely in code, but usually without the clarity or orderliness of the XAML definition. The **GridCodeDemo** program demonstrates the code approach by reproducing the layout of **SimpleGridDemo**.

To specify the height of a `RowDefinition` and the width of the `ColumnDefinition`, you use values of the `GridLength` structure, often in combination with the `GridUnitType` enumeration. The row definitions toward the top of the `GridCodeDemoPage` class demonstrate the variations of

`GridLength`. The column definitions aren't included because they are the same as those generated by default:

```
public class GridCodeDemoPage : ContentPage
{
    public GridCodeDemoPage()
    {
        Grid grid = new Grid
        {
            RowDefinitions =
            {
                new RowDefinition { Height = GridLength.Auto },
                new RowDefinition { Height = new GridLength(100) },
                new RowDefinition { Height = new GridLength(2, GridUnitType.Star) },
                new RowDefinition { Height = new GridLength(1, GridUnitType.Star) }
            }
        };

        // First Label (row 0 and column 0).
        grid.Children.Add(new Label
        {
            Text = "Grid Demo",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            HorizontalOptions = LayoutOptions.End
        });

        // Second Label.
        grid.Children.Add(new Label
        {
            Text = "Demo the Grid",
            FontSize = Device.GetNamedSize(NamedSize.Small, typeof(Label)),
            HorizontalOptions = LayoutOptions.End,
            VerticalOptions = LayoutOptions.End
        },
        1,          // left
        0);        // top

        // Image element.
        grid.Children.Add(new Image
        {
            BackgroundColor = Color.Gray,
            Source = Device.OnPlatform("Icon-60.png",
                                     "icon.png",
                                     "Assets/StoreLogo.png")
        },
        0,          // left
        2,          // right
        1,          // top
        2);        // bottom

        // Three BoxView elements.
        BoxView boxView1 = new BoxView { Color = Color.Green };
        Grid.SetRow(boxView1, 2);
        Grid.SetColumn(boxView1, 0);
    }
}
```



```

grid.Children.Add(boxView1);

BoxView boxView2 = new BoxView { Color = Color.Red };
Grid.SetRow(boxView2, 2);
Grid.SetColumn(boxView2, 1);
Grid.SetRowSpan(boxView2, 2);
grid.Children.Add(boxView2);

BoxView boxView3 = new BoxView
{
    Color = Color.Blue,
    Opacity = 0.5
};
Grid.SetRow(boxView3, 3);
Grid.SetColumn(boxView3, 0);
Grid.SetColumnSpan(boxView3, 2);
grid.Children.Add(boxView3);

Padding = new Thickness(0, Device.OnPlatform(20, 0, 0), 0, 0);
Content = grid;
}
}

```

The program shows several different ways to add children to the `Grid` and specify the cells in which they reside. The first `Label` is in row 0 and column 0, so it only needs to be added to the `Children` collection of the `Grid` to get default row and column settings:

```

grid.Children.Add(new Label
{
    ...
});

```

The `Grid` redefines its `Children` collection to be of type `IGridList<View>`, which includes several additional `Add` methods. One of these `Add` methods lets you specify the row and column:

```

grid.Children.Add(new Label
{
    ...
},
1,          // left
0);        // top

```

As the comments indicate, the arguments are actually named `left` and `top` rather than `column` and `row`. These names make more sense when you see the syntax for specifying row and column spans:

```

grid.Children.Add(new Image
{
    ...
},
0,          // left
2,          // right
1,          // top
2);        // bottom

```

What this means is that the child element goes in the column starting at `left` but ending before `right`—in other words, columns 0 and 1. It occupies the row starting at `top` but ending before `bottom`, which is row 1. The `right` argument must always be greater than `left`, and the `bottom` argument must be greater than `top`. If not, the `Grid` throws an `ArgumentOutOfRangeException`.

The `IGridList<View>` interface also defines `AddHorizontal` and `AddVertical` methods to add children to a single row or single column `Grid`. The `Grid` expands in columns or rows as these calls are made, as well as automatically assigning `Grid.Column` or `Grid.Row` settings on the children. You'll see a use for this facility in the next section.

When adding children to a `Grid` in code, it's also possible to make explicit calls to `Grid.SetRow`, `Grid.SetColumn`, `Grid.SetRowSpan`, and `Grid.SetColumnSpan`. It doesn't matter whether you make these calls before or after you add the child to the `Children` collection of the `Grid`:

```
BoxView boxView1 = new BoxView { ... };
Grid.SetRow(boxView1, 2);
Grid.SetColumn(boxView1, 0);
grid.Children.Add(boxView1);

BoxView boxView2 = new BoxView { ... };
Grid.SetRow(boxView2, 2);
Grid.SetColumn(boxView2, 1);
Grid.SetRowSpan(boxView2, 2);
grid.Children.Add(boxView2);

BoxView boxView3 = new BoxView
{
    ...
};
Grid.SetRow(boxView3, 3);
Grid.SetColumn(boxView3, 0);
Grid.SetColumnSpan(boxView3, 2);
grid.Children.Add(boxView3);
```

The Grid bar chart

The `AddVertical` and `AddHorizontal` methods defined by the `Children` collection of the `Grid` have the capability to add an entire collection of views to the `Grid` in one shot. By default, the new rows or columns get a height or width of `"*"` (star), so the resultant `Grid` consists of multiple rows or columns, each with the same size.

Let's use the `AddHorizontal` method to make a little bar chart that consists of 50 `BoxView` elements with random heights. The XAML file for the **GridBarChart** program defines an `AbsoluteLayout` that is parent to both a `Grid` and a `Frame`. This `Frame` serves as an overlay to display information about a particular bar in the bar chart. It has its `Opacity` set to 0, so it is initially invisible:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="GridBarChart.GridBarChartPage">
```

```

<AbsoluteLayout>

  <!-- Grid occupying entire page. -->
  <Grid x:Name="grid"
        ColumnSpacing="1"
        AbsoluteLayout.LayoutBounds="0, 0, 1, 1"
        AbsoluteLayout.LayoutFlags="All" />

  <!-- Overlay in center of screen. -->
  <Frame x:Name="overly"
        OutlineColor="Accent"
        BackgroundColor="#404040"
        Opacity="0"
        AbsoluteLayout.LayoutBounds="0.5, 0.5, AutoSize, AutoSize"
        AbsoluteLayout.LayoutFlags="PositionProportional">

    <Label x:Name="label"
           TextColor="White"
           FontSize="Large" />

  </Frame>
</AbsoluteLayout>
</ContentPage>

```

The code-behind file creates 50 `BoxView` elements with a random `HeightRequest` property between 0 and 300. In addition, the `StyleId` property of each `BoxView` is assigned a string that consists of alternated random consonants and vowels to resemble a name (perhaps of someone from another planet). All these `BoxView` elements are accumulated in a generic `List` collection and then added to the `Grid`. That job is the bulk of the code in the constructor:

```

public partial class GridBarChartPage : ContentPage
{
    const int COUNT = 50;
    Random random = new Random();

    public GridBarChartPage()
    {
        InitializeComponent();

        List<View> views = new List<View>();
        TapGestureRecognizer tapGesture = new TapGestureRecognizer();
        tapGesture.Tapped += OnBoxViewTapped;

        // Create BoxView elements and add to List.
        for (int i = 0; i < COUNT; i++)
        {
            BoxView boxView = new BoxView
            {
                Color = Color.Accent,
                HeightRequest = 300 * random.NextDouble(),
                VerticalOptions = LayoutOptions.End,
                StyleId = RandomNameGenerator()
            };

```

```

        boxView.GestureRecognizers.Add(tapGesture);
        views.Add(boxView);
    }

    // Add whole List of BoxView elements to Grid.
    grid.Children.AddHorizontal(views);

    // Start a timer at the frame rate.
    Device.StartTimer(TimeSpan.FromMilliseconds(15), OnTimerTick);
}

// Arrays for Random Name Generator.
string[] vowels = { "a", "e", "i", "o", "u", "ai", "ei", "ie", "ou", "oo" };
string[] consonants = { "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
                        "n", "p", "q", "r", "s", "t", "v", "w", "x", "z" };

string RandomNameGenerator()
{
    int numPieces = 1 + 2 * random.Next(1, 4);
    StringBuilder name = new StringBuilder();

    for (int i = 0; i < numPieces; i++)
    {
        name.Append(i % 2 == 0 ?
                    consonants[random.Next(consonants.Length)] :
                    vowels[random.Next(vowels.Length)]);
    }
    name[0] = Char.ToUpper(name[0]);
    return name.ToString();
}

// Set text to overlay Label and make it visible.
void OnBoxViewTapped(object sender, EventArgs args)
{
    BoxView boxView = (BoxView)sender;
    label.Text = String.Format("The individual known as {0} " +
                               "has a height of {1} centimeters.",
                               boxView.StyleId, (int)boxView.HeightRequest);

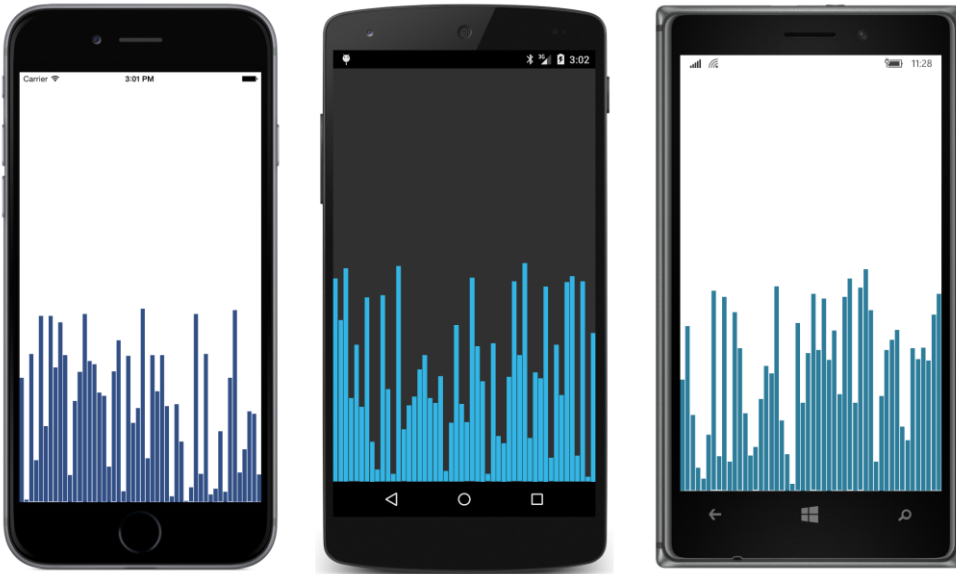
    overlay.Opacity = 1;
}

// Decrease visibility of overlay.
bool OnTimerTick()
{
    overlay.Opacity = Math.Max(0, overlay.Opacity - 0.0025);
    return true;
}
}

```

The `AddHorizontal` method of the `Children` collection adds the multiple `BoxView` elements to the `Grid` and gives them sequential `Grid.Column` settings. Each column by default has a width of "*" (star), so the width of each `BoxView` is the same while the height is governed by the `HeightRequest`

settings. The `Spacing` value of 1 set to the `Grid` in the XAML file provides a little separation between the bars of the bar chart:

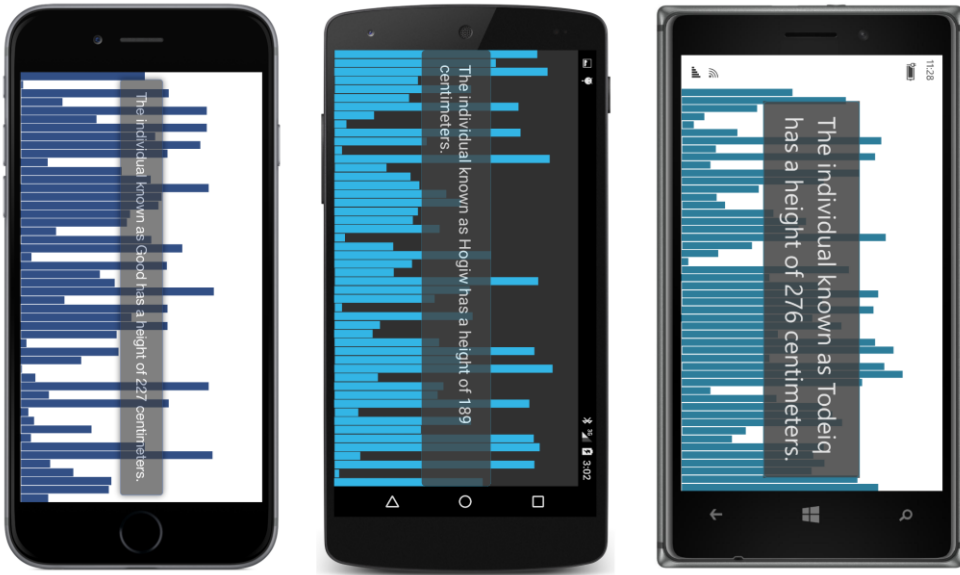


The bars are more distinct when you turn the phone sideways to give them more width:



This program has another feature: When you tap on one of the bars, the overlay is made visible and displays information about that tapped bar—specifically, the interplanetary visitor's name from the

`StyleId` and the height of the bar. But a timer set in the constructor continuously decreases the `Opacity` value on the overlay, so this information gradually fades from view:



Even without a native graphics system, Xamarin.Forms is able to display something that looks quite a lot like graphics.

Alignment in the Grid

A `Grid` row with a `Height` property of `Auto` constrains the height of elements in that row in the same way as a vertical `StackLayout`. Similarly, a column with a `Width` of `Auto` works much like a horizontal `StackLayout`.

As you've seen earlier in this chapter, you can set the `HorizontalOptions` and `VerticalOptions` properties of children of the `Grid` to position them within the cell. Here's a program called **GridAlignment** that creates a `Grid` with nine equal-size cells and then puts six `Label` elements all in the center cell but with different alignment settings:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="GridAlignment.GridAlignmentPage">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
```

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="*" />
  <ColumnDefinition Width="*" />
  <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>

<Label Text="Upper Left"
  Grid.Row="1" Grid.Column="1"
  VerticalOptions="Start"
  HorizontalOptions="Start" />

<Label Text="Upper Right"
  Grid.Row="1" Grid.Column="1"
  VerticalOptions="Start"
  HorizontalOptions="End" />

<Label Text="Center Left"
  Grid.Row="1" Grid.Column="1"
  VerticalOptions="Center"
  HorizontalOptions="Start" />

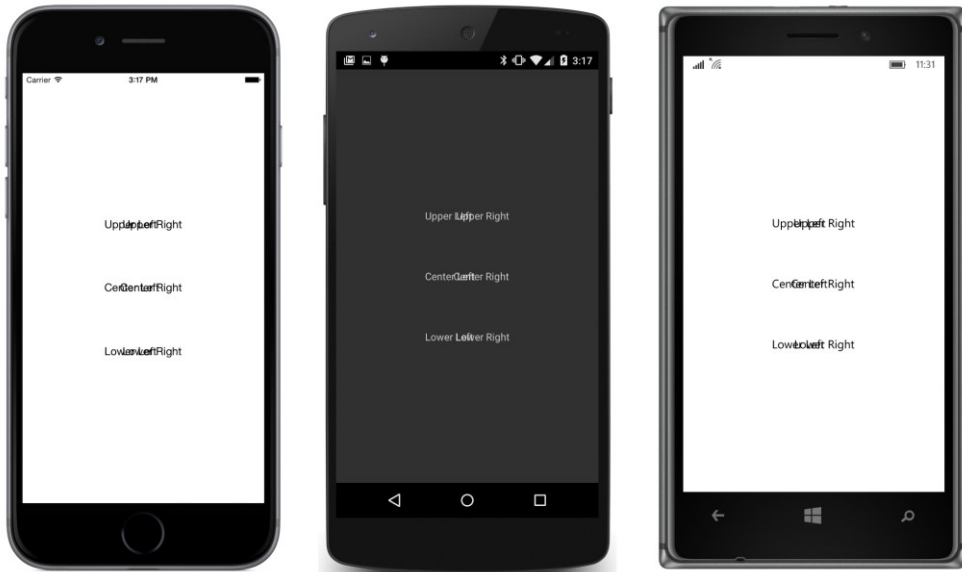
<Label Text="Center Right"
  Grid.Row="1" Grid.Column="1"
  VerticalOptions="Center"
  HorizontalOptions="End" />

<Label Text="Lower Left"
  Grid.Row="1" Grid.Column="1"
  VerticalOptions="End"
  HorizontalOptions="Start" />

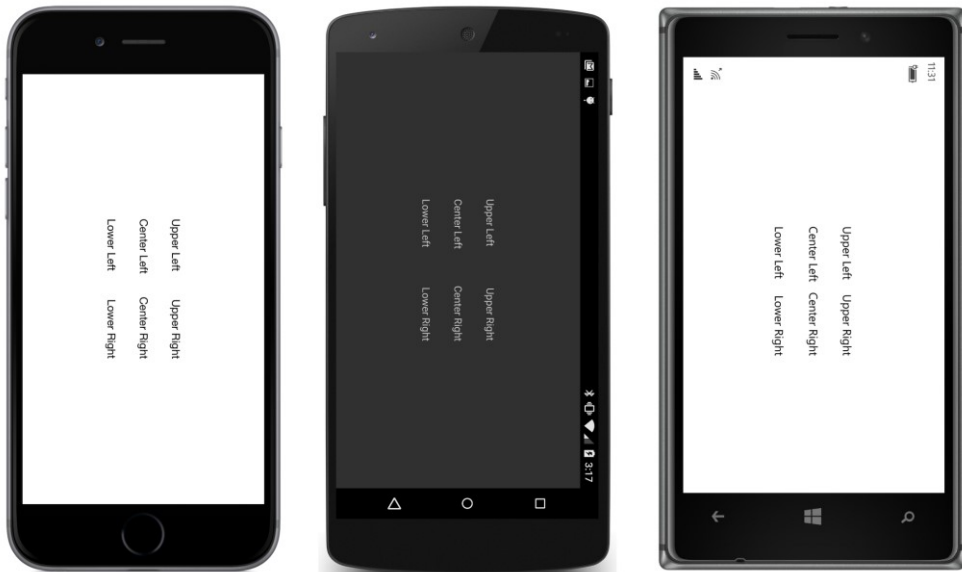
<Label Text="Lower Right"
  Grid.Row="1" Grid.Column="1"
  VerticalOptions="End"
  HorizontalOptions="End" />

</Grid>
</ContentPage>
```

As you can see, some of the text overlaps:



But if you turn the phone sideways, the cells resize and the text doesn't overlap:



Although you can use `HorizontalOptions` and `VerticalOptions` on children of a `Grid` to set the child's alignment, you cannot use the `Expands` flag. Strictly speaking, you actually *can* use the `Expands` flag, but it has no effect on children of a `Grid`. The `Expands` flag only affects children of a `StackLayout`.

Often you've seen programs that use the `Expands` flag for children of a `StackLayout` to provide

extra space to surround elements within the layout. For example, if two `Label` children of a `StackLayout` both have their `VerticalOptions` properties set to `CenterAndExpand`, then all the extra space is divided equally between the two slots in the `StackLayout` allocated for these children.

In a `Grid`, you can perform similar layout tricks by using cells sized with the "*" (star) specification together with `HorizontalOptions` and `VerticalOptions` settings on the children. You can even create empty rows or empty columns just for spacing purposes.

The **SpacingButtons** program equally spaces three vertical buttons and three horizontal buttons. The first three buttons occupy a three-row `Grid` that takes up much of the page, and the three horizontal buttons are in a three-column `Grid` down at the bottom of the page. The two grids are in a `StackLayout`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SpacingButtons.SpacingButtonsPage">
  <StackLayout>
    <Grid VerticalOptions="FillAndExpand">
      <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
      </Grid.RowDefinitions>

      <Button Text="Button 1"
              Grid.Row="0"
              VerticalOptions="Center"
              HorizontalOptions="Center" />

      <Button Text="Button 2"
              Grid.Row="1"
              VerticalOptions="Center"
              HorizontalOptions="Center" />

      <Button Text="Button 3"
              Grid.Row="2"
              VerticalOptions="Center"
              HorizontalOptions="Center" />
    </Grid>

    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>

      <Button Text="Button 4"
              Grid.Column="0"
              HorizontalOptions="Center" />

      <Button Text="Button 5"
```

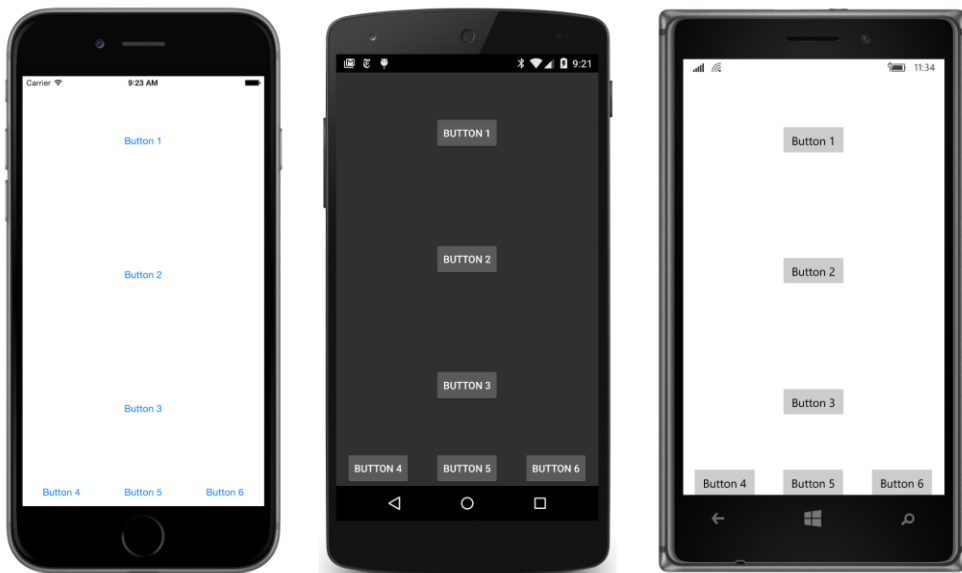
```

        Grid.Column="1"
        HorizontalOptions="Center" />

        <Button Text="Button 6"
            Grid.Column="2"
            HorizontalOptions="Center" />
    </Grid>
</StackLayout>
</ContentPage>

```

The second `Grid` has a default `VerticalOptions` value of `Fill`, while the first `Grid` has an explicit setting for `VerticalOptions` to `FillAndExpand`. This means that the first `Grid` will occupy all the area of the screen not occupied by the second `Grid`. The three `RowDefinition` objects of the first `Grid` divide that area into thirds. Within each cell, the `Button` is horizontal and vertically centered:



The second `Grid` divides its area into three equally spaced columns, and each `Button` is horizontally centered within that area.

Although the `Expands` flag of `LayoutOptions` can assist in equally spacing visual objects within a `StackLayout`, the technique breaks down when the visual objects are not a uniform size. The `Expands` option allocates leftover space equally among all the slots in the `StackLayout`, but the total size of each slot depends on the size of the individual visual objects. The `Grid`, however, allocates space equally to the cells, and then the visual objects are aligned within that space.

Cell dividers and borders

The `Grid` doesn't have any built-in cell dividers or borders. But if you'd like some, you can add them yourself. The `GridCellDividers` program defines a `GridLength` value in its `Resources` dictionary

named `dividerThickness`. This is used for the height and width of every other row and column in the `Grid`. The idea here is that these rows and columns are for the dividers, while the other rows and columns are for regular content:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="GridCellDividers.GridCellDividersPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                   iOS="0, 20, 0, 0"
                   Android="0"
                   WinPhone="0" />
    </ContentPage.Padding>

    <Grid>
        <Grid.Resources>
            <ResourceDictionary>
                <GridLength x:Key="dividerThickness">2</GridLength>

                <Style TargetType="BoxView">
                    <Setter Property="Color" Value="Accent" />
                </Style>

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

        <Grid.RowDefinitions>
            <RowDefinition Height="{StaticResource dividerThickness}" />
            <RowDefinition Height="*" />
            <RowDefinition Height="{StaticResource dividerThickness}" />
            <RowDefinition Height="*" />
            <RowDefinition Height="{StaticResource dividerThickness}" />
            <RowDefinition Height="*" />
            <RowDefinition Height="{StaticResource dividerThickness}" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="{StaticResource dividerThickness}" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="{StaticResource dividerThickness}" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="{StaticResource dividerThickness}" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="{StaticResource dividerThickness}" />
        </Grid.ColumnDefinitions>

        <BoxView Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="7" />
        <BoxView Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="7" />
        <BoxView Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="7" />
    </Grid>
</ContentPage>
```

```

<BoxView Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="7" />

<BoxView Grid.Row="0" Grid.Column="0" Grid.RowSpan="7" />
<BoxView Grid.Row="0" Grid.Column="2" Grid.RowSpan="7" />
<BoxView Grid.Row="0" Grid.Column="4" Grid.RowSpan="7" />
<BoxView Grid.Row="0" Grid.Column="6" Grid.RowSpan="7" />

<Label Text="Grid"
      Grid.Row="1" Grid.Column="1" />

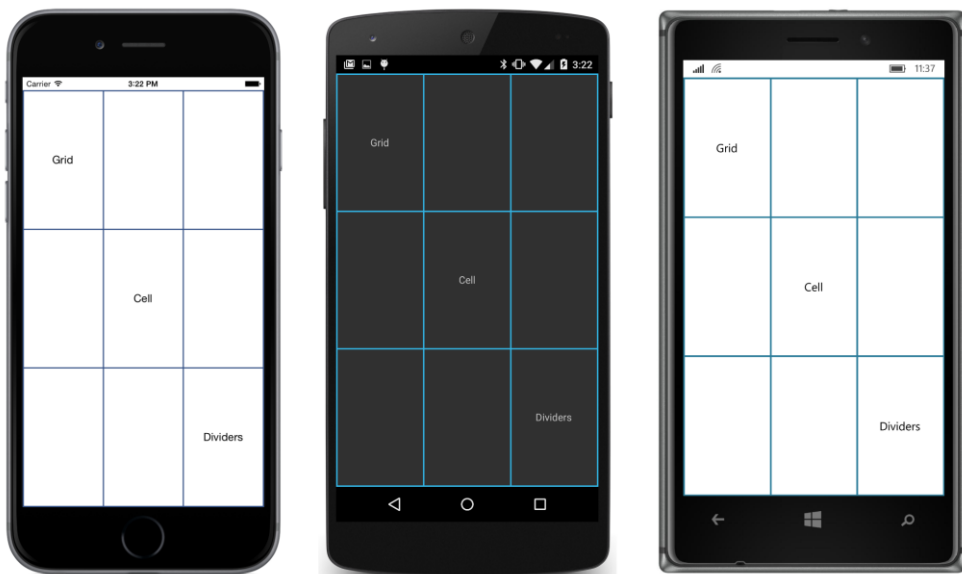
<Label Text="Cell"
      Grid.Row="3" Grid.Column="3" />

<Label Text="Dividers"
      Grid.Row="5" Grid.Column="5" />
</Grid>
</ContentPage>

```

Each row and column for the dividers is occupied by a `BoxView` colored with the `Accent` color from an implicit style. For the horizontal dividers, the height is set by the `RowDefinition` and the width is governed by the `Grid.ColumnSpan` attached bindable property; a similar approach is applied for the vertical dividers.

The `Grid` also contains three `Label` elements just to demonstrate how regular content fits in with these dividers:



It is not necessary to allocate entire rows and columns to these dividers. Keep in mind that visual objects can share cells, so it's possible to add a `BoxView` (or two or three or four) to a cell and set the horizontal and vertical options so that it hugs the wall of the cell and resembles a border.

Here's a similar program, called **GridCellBorders**, that displays content in the same three cells as **GridCellDividers**, but those three cells are also adorned with borders.

The `Resources` dictionary contains no fewer than seven styles that target `BoxView`! The base style sets the color, two more styles set the `HeightRequest` and `WidthRequest` for the horizontal and vertical borders, and then four more styles set the `VerticalOptions` to `Start` or `End` for the top and bottom borders and `HorizontalOptions` to `Start` and `End` for the left and right borders. The `borderThickness` dictionary entry is a `double` because it's used to set `WidthRequest` and `HeightRequest` properties of the `BoxView` elements:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="GridCellBorders.GridCellBordersPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                   iOS="10, 20, 10, 10"
                   Android="10"
                   WinPhone="10" />
    </ContentPage.Padding>

    <Grid>
        <Grid.Resources>
            <ResourceDictionary>
                <x:Double x:Key="borderThickness">1</x:Double>

                <Style x:Key="baseBorderStyle" TargetType="BoxView">
                    <Setter Property="Color" Value="Accent" />
                </Style>

                <Style x:Key="horzBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource baseBorderStyle}">
                    <Setter Property="HeightRequest" Value="{StaticResource borderThickness}" />
                </Style>

                <Style x:Key="topBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource horzBorderStyle}">
                    <Setter Property="VerticalOptions" Value="Start" />
                </Style>

                <Style x:Key="bottomBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource horzBorderStyle}">
                    <Setter Property="VerticalOptions" Value="End" />
                </Style>

                <Style x:Key="vertBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource baseBorderStyle}">
                    <Setter Property="WidthRequest" Value="{StaticResource borderThickness}" />
                </Style>

                <Style x:Key="leftBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource vertBorderStyle}">

```

```

        <Setter Property="HorizontalOptions" Value="Start" />
    </Style>

    <Style x:Key="rightBorderStyle" TargetType="BoxView"
        BasedOn="{StaticResource vertBorderStyle}">
        <Setter Property="HorizontalOptions" Value="End" />
    </Style>

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

<Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>

<Label Text="Grid"
    Grid.Row="0" Grid.Column="0" />

<BoxView Style="{StaticResource topBorderStyle}"
    Grid.Row="0" Grid.Column="0" />

<BoxView Style="{StaticResource bottomBorderStyle}"
    Grid.Row="0" Grid.Column="0" />

<BoxView Style="{StaticResource leftBorderStyle}"
    Grid.Row="0" Grid.Column="0" />

<BoxView Style="{StaticResource rightBorderStyle}"
    Grid.Row="0" Grid.Column="0" />

<Grid Grid.Row="1" Grid.Column="1">
    <Label Text="Cell" />
    <BoxView Style="{StaticResource topBorderStyle}" />
    <BoxView Style="{StaticResource bottomBorderStyle}" />
    <BoxView Style="{StaticResource leftBorderStyle}" />
    <BoxView Style="{StaticResource rightBorderStyle}" />
</Grid>

<Grid Grid.Row="2" Grid.Column="2">
    <Label Text="Borders" />
    <BoxView Style="{StaticResource topBorderStyle}" />
    <BoxView Style="{StaticResource bottomBorderStyle}" />

```

```

        <BoxView Style="{StaticResource leftBorderStyle}" />
        <BoxView Style="{StaticResource rightBorderStyle}" />
    </Grid>
</Grid>
</ContentPage>

```

In the cell in the upper-left corner, the `Label` and four `BoxView` elements each gets its `Grid.Row` and `Grid.Column` attributes set to 0. However, for the middle `Grid` and the bottom-right `Grid`, a rather easier approach is taken: Another `Grid` with a single cell occupies the cell, and that single-cell `Grid` contains the `Label` and four `BoxView` elements. The simplicity results from setting `Grid.Row` and `Grid.Column` only on the single-cell `Grid`:

```

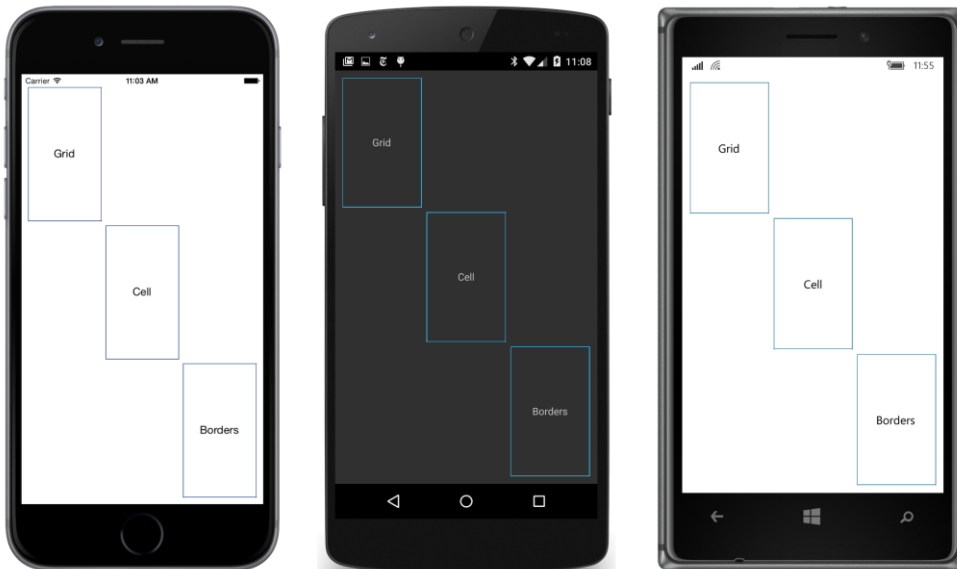
<Grid Grid.Row="1" Grid.Column="1">
    <Label Text="Cell" />
    <BoxView Style="{StaticResource topBorderStyle}" />
    <BoxView Style="{StaticResource bottomBorderStyle}" />
    <BoxView Style="{StaticResource leftBorderStyle}" />
    <BoxView Style="{StaticResource rightBorderStyle}" />
</Grid>

```

When nesting a `Grid` inside another `Grid`, the use of the `Grid.Row` and `Grid.Column` attributes can be confusing. This single-cell `Grid` occupies the second row and second column of its parent, which is the `Grid` that occupies the entire page.

Also, keep in mind that when a `Grid` is laying itself out, it looks only at the `Grid.Row` and `Grid.Column` settings of its children, and never its grandchildren or other descendants in the visual tree.

Here's the result:



It might be a little disconcerting that the corners of the borders don't meet, but that's due to the default row and column spacing of the `Grid`. Set the `RowSpacing` and `ColumnSpacing` attributes to 0, and the corners will meet although the lines will still seem somewhat discontinuous because the borders are in different cells. If this is unacceptable, use the technique shown in **GridCellDividers**.

If you want all the rows and columns shown with dividers as in **GridCellDividers**, another technique is to set the `BackgroundColor` property of the `Grid` and use the `RowSpacing` and `ColumnSpacing` properties to let that color peek through the spaces between the cells. But all the cells must contain content that has an opaque background for this technique to be visually convincing.

Almost real-life Grid examples

We are now ready to rewrite the **XamlKeypad** program from Chapter 8 to use a `Grid`. The new version is called **KeypadGrid**. The use of a `Grid` not only forces the `Button` elements that make up the keypad to be all the same size, but also allows components of the keypad to span cells.

The `Grid` that makes up the keypad is centered on the page with `HorizontalOptions` and `VerticalOptions` settings. It has five rows and three columns but the `RowDefinitions` and `ColumnDefinitions` collections don't need to be explicitly constructed because every cell has a "*" (star) height and width.

Moreover, the entire `Grid` is given a platform-specific `WidthRequest` and `HeightRequest`, where the width is three-fifths of the height. (The difference for Windows Phone is based on the somewhat larger size of the `Large` font size used for the `Button`.) This causes every cell in the `Grid` to be square:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="KeypadGrid.KeypadGridPage">

    <Grid RowSpacing="2"
          ColumnSpacing="2"
          VerticalOptions="Center"
          HorizontalOptions="Center">
        <Grid.WidthRequest>
            <OnPlatform x:TypeArguments="x:Double"
                       iOS="180"
                       Android="180"
                       WinPhone="240" />
        </Grid.WidthRequest>

        <Grid.HeightRequest>
            <OnPlatform x:TypeArguments="x:Double"
                       iOS="300"
                       Android="300"
                       WinPhone="400" />
        </Grid.HeightRequest>

        <Grid.Resources>
```



```
<ResourceDictionary>
  <Style TargetType="Button">
    <Setter Property="FontSize" Value="Large" />
    <Setter Property="BorderWidth" Value="1" />
  </Style>
</ResourceDictionary>
</Grid.Resources>

<Label x:Name="displayLabel"
  Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
  FontSize="Large"
  LineBreakMode="HeadTruncation"
  VerticalOptions="Center"
  HorizontalTextAlignment="End" />

<Button x:Name="backspaceButton"
  Text="&#x21E6;"
  Grid.Row="0" Grid.Column="2"
  IsEnabled="False"
  Clicked="OnBackspaceButtonClicked" />

<Button Text="7" StyleId="7"
  Grid.Row="1" Grid.Column="0"
  Clicked="OnDigitButtonClicked" />

<Button Text="8" StyleId="8"
  Grid.Row="1" Grid.Column="1"
  Clicked="OnDigitButtonClicked" />

<Button Text="9" StyleId="9"
  Grid.Row="1" Grid.Column="2"
  Clicked="OnDigitButtonClicked" />

<Button Text="4" StyleId="4"
  Grid.Row="2" Grid.Column="0"
  Clicked="OnDigitButtonClicked" />

<Button Text="5" StyleId="5"
  Grid.Row="2" Grid.Column="1"
  Clicked="OnDigitButtonClicked" />

<Button Text="6" StyleId="6"
  Grid.Row="2" Grid.Column="2"
  Clicked="OnDigitButtonClicked" />

<Button Text="1" StyleId="1"
  Grid.Row="3" Grid.Column="0"
  Clicked="OnDigitButtonClicked" />

<Button Text="2" StyleId="2"
  Grid.Row="3" Grid.Column="1"
  Clicked="OnDigitButtonClicked" />

<Button Text="3" StyleId="3"
```

```

        Grid.Row="3" Grid.Column="2"
        Clicked="OnDigitButtonClicked" />

<Button Text="0" StyleId="0"
        Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2"
        Clicked="OnDigitButtonClicked" />

<Button Text="." StyleId="."
        Grid.Row="4" Grid.Column="2"
        Clicked="OnDigitButtonClicked" />
</Grid>
</ContentPage>

```

The `Label` and the backspace button occupy the top row, but the `Label` spans two columns and the backspace button is in the third column. Similarly, the bottom row of the `Grid` contains the zero button and the decimal-point button, but the zero button spans two columns as is typical on computer keypads.

The code-behind file is the same as the **XamlKeypad** program. In addition, the program saves entries when the program is put to sleep and then restores them when the program starts up again. A border has been added to the `Button` in an implicit style so that it looks more like a real keypad on iOS:



As you might recall, the `OnDigitButtonClicked` handler in the code-behind file uses the `StyleId` property to append a new character to the text string. But as you can see in the XAML file, for each of the buttons with this event handler, the `StyleId` is set to the same character as the `Text` property of the `Button`. Can't the event handler use that instead?

Yes, it can. But suppose you decide that the decimal point in the `Button` doesn't show up very well.

You might prefer to use a heavier and more central dot, such as `\u00B7` (called Middle Dot) or `\u22C5` (the mathematical Dot Operator) or even `\u2022` (the Bullet). Perhaps you'd also like different styles of numbers for these other buttons, such as the set of encircled numbers that begin at `\u2460` in the Unicode standard, or the Roman numerals that begin at `\u2160`. You can replace the `Text` property in the XAML file without touching the code-behind file:



The `styleId` is one of the tools to keep the visuals and mechanics of the user interface restricted to markup and separated from your code. You'll see more tools to structure your program in the next chapter, which covers the Model-View-ViewModel application architecture. That chapter also presents a variation of the keypad program turned into an adding machine.

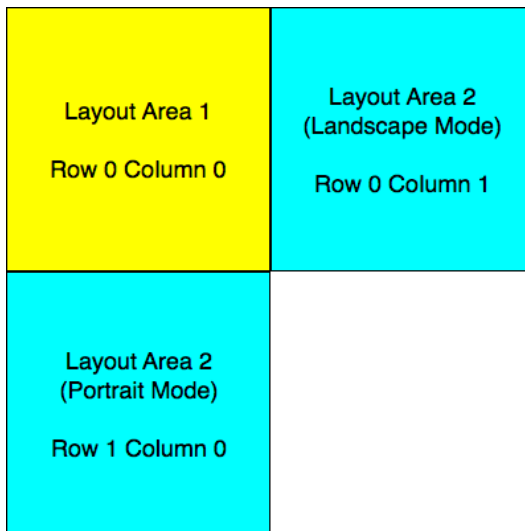
Responding to orientation changes

The layout of an application's page is usually tied fairly closely to a particular form factor and aspect ratio. Sometimes, an application will require that it be used only in portrait or landscape mode. But usually an application will attempt to move things around on the screen when the phone changes orientation.

A `Grid` can help an application accommodate itself to orientation changes. The `Grid` can be defined in XAML with certain allowances for both portrait and landscape modes, and then a little code can make the proper adjustments within a `SizeChanged` handler for the page.

This job is easiest if you can divide the entire layout of your application into two large areas that can be arranged vertically when the phone is oriented in portrait mode or horizontally for landscape mode. Put each of these areas in separate cells of a `Grid`. When the phone is in portrait mode, the `Grid` has two rows, and when it's in landscape mode, it has two columns. In the following diagram, the first area

is always at the top or the left. The second area can be in either the second row for portrait mode or the second column for landscape mode:



To keep things reasonably simple, you'll want to define the `Grid` in XAML with two rows and two columns, but in portrait mode, the second column has a width of zero, and in landscape mode the second row has a zero height.

The **GridRgbSliders** program demonstrates this technique. It is similar to the **RgbSliders** program from Chapter 15, "The interactive interface," except that the layout uses a combination of a `Grid` and a `StackLayout`, and the `Label` elements display the current values of the `Slider` elements by using data bindings with a value converter and a value converter parameter. (More on this later.) Setting the `Color` property of the `BoxView` based on the three `Slider` elements still requires code because the `R`, `G`, and `B` properties of the `Color` struct are not backed by bindable properties, and these properties cannot be individually changed anyway because they do not have public `set` accessors. (However, in the next chapter, on MVVM, you'll see a way to eliminate this logic in the code-behind file.)

As you can see in the following listing, the `Grid` named `mainGrid` does indeed have two rows and two columns. However, it is initialized for portrait mode, so the second column has a width of zero. The top row of the `Grid` contains the `BoxView`, and that's made as large as possible with a "*" (star) setting, while the bottom row contains a `StackLayout` with all the interactive controls. This is given a height of `Auto`:

```
<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="GridRgbSliders.GridRgbSlidersPage"
             SizeChanged="OnPageSizeChanged">
    <ContentPage.Padding>
```

```

        <OnPlatform x:TypeArguments="Thickness"
            iOS="0, 20, 0, 0" />
    </ContentPane.Padding>

    <ContentPane.Resources>
        <ResourceDictionary>
            <toolkit:DoubleToIntConverter x:Key="doubleToInt" />

            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPane.Resources>

    <Grid x:Name="mainGrid">
        <!-- Initialized for portrait mode. -->
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="0" />
        </Grid.ColumnDefinitions>

        <BoxView x:Name="boxView"
            Grid.Row="0" Grid.Column="0" />

        <StackLayout x:Name="controlPanelStack"
            Grid.Row="1" Grid.Column="0"
            Padding="10, 5">

            <StackLayout VerticalOptions="CenterAndExpand">
                <Slider x:Name="redSlider"
                    ValueChanged="OnSliderValueChanged" />

                <Label Text="{Binding Source={x:Reference redSlider},
                    Path=Value,
                    Converter={StaticResource doubleToInt},
                    ConverterParameter=255,
                    StringFormat='Red = {0:X2}}'" />
            </StackLayout>

            <StackLayout VerticalOptions="CenterAndExpand">
                <Slider x:Name="greenSlider"
                    ValueChanged="OnSliderValueChanged" />

                <Label Text="{Binding Source={x:Reference greenSlider},
                    Path=Value,
                    Converter={StaticResource doubleToInt},
                    ConverterParameter=255,
                    StringFormat='Green = {0:X2}}'" />
            </StackLayout>
        </StackLayout>
    </Grid>

```

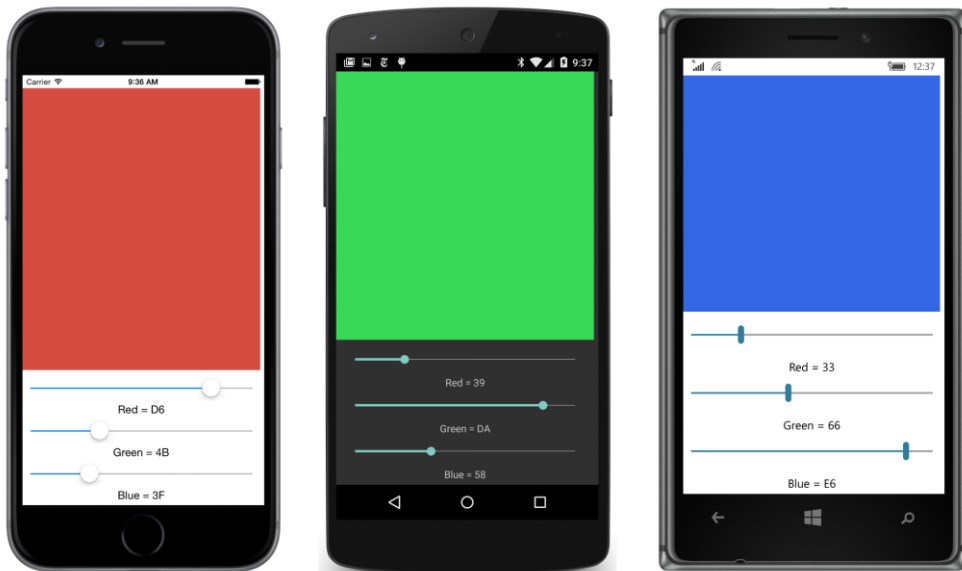
```

<StackLayout VerticalOptions="CenterAndExpand">
  <Slider x:Name="blueSlider"
    ValueChanged="OnSliderValueChanged" />

  <Label Text="{Binding Source={x:Reference blueSlider},
    Path=Value,
    Converter={StaticResource doubleToInt},
    ConverterParameter=255,
    StringFormat='Blue = {0:X2}'}" />
</StackLayout>
</StackLayout>
</Grid>
</ContentPage>

```

And here's the portrait view:



The layout in the XAML file is prepared for landscape mode in a couple of ways. First, the `Grid` already has a second column. This means that to switch to landscape mode, the code-behind file needs to change the height of the second row to zero and the width of the second column to a nonzero value.

Secondly, the `StackLayout` containing all the `Slider` and `Label` elements is accessible from code because it has a name, specifically `controlPanelStack`. The code-behind file can then make `Grid.SetRow` and `Grid.SetColumn` calls on this `StackLayout` to move it from row 1 and column 0 to row 0 and column 1.

In portrait mode, the `BoxView` has a height of "*" (star) and the `StackLayout` has a height of `Auto`. Does this mean that the width of the `StackLayout` should be `Auto` in landscape mode? That wouldn't

be wise because it would shrink the widths of the `Slider` elements. A better solution for landscape mode is to give both the `BoxView` and the `StackLayout` a width of "*" (star) to divide the screen in half.

Here's the code-behind file showing the `SizeChanged` handler on the page responsible for switching between portrait and landscape mode, as well as the `ValueChanged` handler for the `Slider` elements that sets the `BoxView` color:

```
public partial class GridRgbSlidersPage : ContentPage
{
    public GridRgbSlidersPage()
    {
        // Ensure link to Toolkit library.
        new Xamarin.Forms.Book.Toolkit.DoubleToIntConverter();

        InitializeComponent();
    }

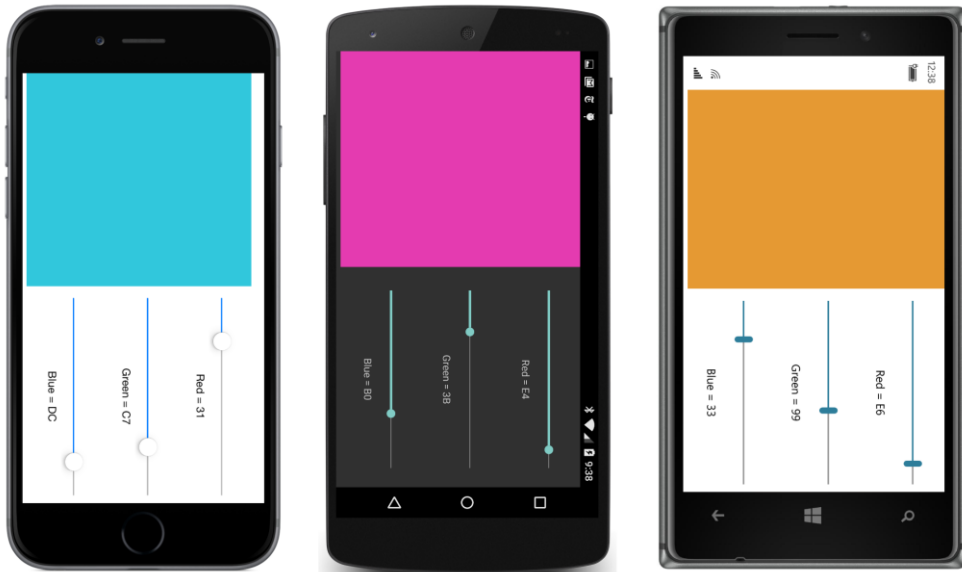
    void OnPageSizeChanged(object sender, EventArgs args)
    {
        // Portrait mode.
        if (Width < Height)
        {
            mainGrid.RowDefinitions[1].Height = GridLength.Auto;
            mainGrid.ColumnDefinitions[1].Width = new GridLength(0, GridUnitType.Absolute);

            Grid.SetRow(controlPanelStack, 1);
            Grid.SetColumn(controlPanelStack, 0);
        }
        // Landscape mode.
        else
        {
            mainGrid.RowDefinitions[1].Height = new GridLength(0, GridUnitType.Absolute);
            mainGrid.ColumnDefinitions[1].Width = new GridLength(1, GridUnitType.Star);

            Grid.SetRow(controlPanelStack, 0);
            Grid.SetColumn(controlPanelStack, 1);
        }
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        boxView.Color = new Color(redSlider.Value, greenSlider.Value, blueSlider.Value);
    }
}
```

And here's the landscape layout, displayed sideways as usual:



Notice, particularly on the iOS and Android displays, how each pair of `Slider` and `Label` elements is grouped together. This results from a third way that the XAML file is prepared to accommodate landscape mode. Each pair of `Slider` and `Label` elements is grouped in a nested `StackLayout`. This is given a `VerticalOptions` setting of `CenterAndExpand` to perform this spacing.

A little thought was given to arranging the `BoxView` and the control panel: In portrait mode, the fingers manipulating the `Slider` elements won't obscure the result in the `BoxView`, and in landscape mode, the fingers of right-handed users won't obscure the `BoxView` either. (Of course, left-handed users will probably insist on a program option to swap the locations!)

The screenshots show the `Slider` values displayed in hexadecimal. This is done with a data binding, and that would normally be a problem. The `Value` property of the `Slider` is of type `double`, and if you attempt to format a `double` with "X2" for hexadecimal, an exception will be raised. A type converter (named `DoubleToIntConverter`, for example) must convert the source `double` to an `int` for the string formatting. However, the `Slider` elements are set up for a range of 0 to 1, while integer values formatted as hexadecimal must range from 0 to 255.

A solution is to make use of the `ConverterParameter` property of `Binding`. Whatever is set to this property is passed as the third argument to the `Convert` and `ConvertBack` methods in the value converter. Here's the `DoubleToIntConverter` class in the **Xamarin.FormsBook.Toolkit** library:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class DoubleToIntConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
            object parameter, CultureInfo culture)
        {

```



```

    string strParam = parameter as string;
    double multiplier = 1;

    if (!String.IsNullOrEmpty(strParam))
    {
        Double.TryParse(strParam, out multiplier);
    }

    return (int)Math.Round((double)value * multiplier);
}

public object ConvertBack(object value, Type targetType,
    object parameter, CultureInfo culture)
{
    string strParam = parameter as string;
    double divider = 1;

    if (!String.IsNullOrEmpty(strParam))
    {
        Double.TryParse(strParam, out divider);
    }

    return (int)value / divider;
}
}
}

```

The `Convert` and `ConvertBack` methods assume that the `parameter` argument is a string and, if so, attempt to convert it to a `double`. This value is then multiplied by the `double` value being converted, and then the product is cast to an `int`.

The combination of the value converter, the converter parameter, and the string formatting converts values ranging from 0 to 1 coming from the `Slider` to integers in the range of 0 to 255 that are then formatted as two hexadecimal digits:

```

<Label Text="{Binding Source={x:Reference redSlider},
    Path=Value,
    Converter={StaticResource doubleToInt},
    ConverterParameter=255,
    StringFormat='Red = {0:X2}'}" />

```

Of course, if you were defining the `Binding` in code, you would probably set the `ConverterParameter` property to the numeric value of 255 rather than a string of "255", and the logic in the `DoubleToIntConverter` would fail. Simple value converters are usually simpler than they should be for complete bulletproofing.

Can a program like **GridRgbSliders** be entirely realized without the `Slider` event handlers in the code-behind file? Code will certainly still be required, but some of it will be moved away from the user-interface logic. That's the main objective of the Model-View-ViewModel architecture explored in the next chapter.