Chapter 27

# Custom renderers

At the core of Xamarin.Forms is something that might seem like magic: the ability of a single element such as `Button` to appear as a native button under the iOS, Android, and Windows operating systems. In this chapter, you'll see how on all three platforms each element in Xamarin.Forms is supported by a special class known as a *renderer*. For example, the `Button` class in Xamarin.Forms is supported by several classes in the various platforms, each named `ButtonRenderer`.

The good news is that you can also write your own renderers, and this chapter will show you how. However, keep in mind that writing custom renderers is a big topic, and this chapter can only get you started.

Writing custom renderers is not quite as easy as writing a Xamarin.Forms application. You'll need to be familiar with the individual iOS, Android, and Windows Runtime platforms. But obviously it's a powerful technique. Indeed, some developers think of the ultimate value of Xamarin.Forms as providing a structured framework in which to write custom renderers.

## The complete class hierarchy

In Chapter 11, "The bindable infrastructure," you saw a program called **ClassHierarchy** that displays the Xamarin.Forms class hierarchy. However, that program only displays the types in the **Xamarin.Forms.Core** and **Xamarin.Forms.Xaml** assemblies, which are the types that a Xamarin.Forms application generally uses.

Xamarin.Forms also contains additional assemblies associated with each platform. These assemblies play a crucial role by providing the platform support for Xamarin.Forms, including all the renderers.

You're probably already familiar with the names of these assemblies from seeing them in the **Reference** sections of the various projects in your Xamarin.Forms solution:

- **Xamarin.Forms.Platform** (very small)
- **Xamarin.Forms.Platform.iOS**
- **Xamarin.Forms.Platform.Android**
- **Xamarin.Forms.Platform.UAP**
- **Xamarin.Forms.Platform.WinRT** (larger than the next two on this list)
- **Xamarin.Forms.Platform.WinRT.Tablet**

- **Xamarin.Forms.Platform.WinRT.Phone**

In this discussion, these will be referred to collectively as the *platform assemblies*.

Is it possible to write a Xamarin.Forms application that displays a class hierarchy of the types in these platform assemblies?

Yes! However, if you restrict yourself to examining only the assemblies normally loaded with an application—and this is certainly the simplest approach—then an application can only display the types in the assemblies that are part of that application. For example, you can only display the types in the **Xamarin.Forms.Platform.iOS** assembly with a Xamarin.Forms program running under iOS, and similarly for the other assemblies.

But there's still a problem: As you might recall, the original **ClassHierarchy** program began by obtaining .NET `Assembly` objects for the **Xamarin.Forms.Core** and **Xamarin.Forms.Xaml** assemblies based on two classes (`View` and `Extensions`) that it knew to be in those two assemblies:

```
typeof(View).GetTypeInfo().Assembly
typeof(Extensions).GetTypeInfo().Assembly
```

However, a Xamarin.Forms application's Portable Class Library doesn't have direct access to the platform assemblies. The platform assemblies are referenced only by the application projects. This means that a Xamarin.Forms Portable Class Library can't use similar code to get a reference to the platform assembly. This won't work:

```
typeof(ButtonRenderer).GetTypeInfo().Assembly
```

However, these platform assemblies are loaded when the application runs, so the PCL can instead obtain `Assembly` objects for the platform assemblies based on the assembly name. The **PlatformClassHierarchy** program begins like this:

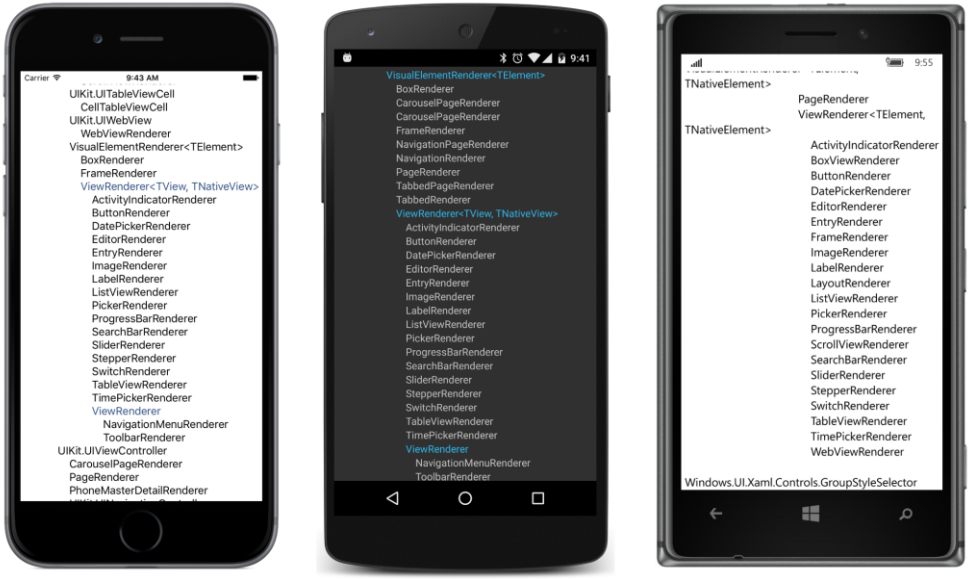
```
public partial class PlatformClassHierarchyPage : ContentPage
{
    public PlatformClassHierarchyPage()
    {
        InitializeComponent();

        List<TypeInformation> classList = new List<TypeInformation>();

        string[] assemblyNames = Device.OnPlatform(
            iOS: new string[] { "Xamarin.Forms.Platform.iOS" },
            Android: new string[] { "Xamarin.Forms.Platform.Android" },
            WinPhone: new string[] { "Xamarin.Forms.Platform.UAP",
                                     "Xamarin.Forms.Platform.WinRT",
                                     "Xamarin.Forms.Platform.WinRT.Tablet",
                                     "Xamarin.Forms.Platform.WinRT.Phone" }
            );

        foreach (string assemblyName in assemblyNames)
        {
            try
```

```
            {
                Assembly assembly = Assembly.Load(new AssemblyName(assemblyName));
                GetPublicTypes(assembly, classList);
            }
            catch
            {
            }
        }
        …
}
```

And from there the **PlatformClassHierarchy** program is the same as the original **ClassHierarchy** program.

As you can see, the `foreach` loop obtains the `Assembly` object from the static `Assembly.Load` method. However, there's not a straightforward way for the program to determine whether it's running under the Universal Windows Platform or one of the other Windows Runtime platforms, so if `Device.OnPlatform` indicates that it's the `WinPhone` device, the program tries all four assemblies and uses `try` and `catch` to just ignore the ones that don't work.

Some of the class names—and particularly the fully qualified class names for classes outside the assembly—are a little too long for the portrait display and wrap awkwardly, but here's part of the display on the three platforms. Each has been scrolled to the part of the class hierarchy that begins with the generic `ViewRenderer` class. This is generally the class you'll derive from to create your own custom renderers:




Notice the generic parameters for the `ViewRenderer` class named either `TView` and `TNativeView`, or `TElement` and `TNativeElement`: As you'll see, `TView` or `TElement` is the Xamarin.Forms element,

such as `Button`, while `TNativeView` or `TNativeElement` is the native control for that `Button`.

Although the **PlatformClassHierarchy** program doesn't indicate this, the constraints for the `ViewRenderer` generic parameters are platform dependent:

- On iOS:
    - `TView` is constrained to `Xamarin.Forms.View`
    - `TNativeView` is constrained to `UIKit.UIView`
- On Android:
    - `TView` is constrained to `Xamarin.Forms.View`
    - `TNativeView` is constrained to `Android.Views.View`
- On the Windows platforms:
    - `TElement` is constrained to `Xamarin.Forms.View`
    - `TNativeElement` is constrained to `Windows.UI.Xaml.FrameworkElement`

To write a custom renderer, you derive a class from `ViewRenderer`. To accommodate all the platforms, you must implement the iOS renderer by using a class that derives from `UIView`, implement the Android renderer with a class that derives from `View`, and implement a renderer for the Windows platforms with a class that derives from `FrameworkElement`.

Let's try it!

## Hello, custom renderers!

The **HelloRenderers** program mostly demonstrates the overhead required to write simple renderers. The program defines a new `View` derivative named `HelloView` that is intended to display a simple fixed string of text. Here's the complete HelloView.cs file in the **HelloRenderers** Portable Class Library project:

```
using Xamarin.Forms;

namespace HelloRenderers
{
    public class HelloView : View
    {
    }
}
```

That's it! However, note that the class is defined as `public`. Even though you might think that this class is only referenced within the PCL, that's not the case. It must be visible to the platform assemblies.

The **HelloRenderers** PCL is so simple that it doesn't even bother with a page class. Instead, it instantiates and displays a `HelloView` object centered on the page right in the App.cs file:

```
namespace HelloRenderers
{
    public class App : Application
    {
        public App()
        {
            MainPage = new ContentPage
            {
                Content = new HelloView
                {
                    VerticalOptions = LayoutOptions.Center,
                    HorizontalOptions = LayoutOptions.Center
                }
            };
        }
        …
    }
}
```

Without any other code, this program runs fine, but you won't actually see the `HelloView` object on the screen because it's just a blank transparent view. What we need are some platform renderers for `HelloView`.

When a Xamarin.Forms application starts up, Xamarin.Forms uses .NET reflection to search through the various assemblies that comprise the application, looking for assembly attributes named `ExportRenderer`. An `ExportRenderer` attribute indicates the presence of a custom renderer that can supply support for a Xamarin.Forms element.

The **HelloRenderers.iOS** project contains the following HelloViewRenderer.cs file, shown in its entirety. Notice the `ExportRenderer` attribute right under the `using` directives. Because this is an assembly attribute, it must be outside a `namespace` declaration. This particular `ExportRenderer` attribute basically says "The `HelloView` class is supported by a renderer of type `HelloViewRenderer`":

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

using UIKit;

using HelloRenderers;
using HelloRenderers.iOS;

[assembly: ExportRenderer(typeof(HelloView), typeof(HelloViewRenderer))]

namespace HelloRenderers.iOS
{
    public class HelloViewRenderer : ViewRenderer<HelloView, UILabel>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<HelloView> args)
        {
```

```
            base.OnElementChanged(args);

            if (Control == null)
            {
                UILabel label = new UILabel
                {
                    Text = "Hello from iOS!",
                    Font = UIFont.SystemFontOfSize(24)
                };

                SetNativeControl(label);
            }
        }
    }
}
```

The definition of the `HelloViewRenderer` class follows the `ExportRenderer` attribute. The class must be public. It derives from the generic `ViewRenderer` class. The two generic parameters are named `TView`, which is the Xamarin.Forms class, and `TNativeView`, which is the class in this particular case that is native to iOS.

In iOS, a class that displays text is `UILabel` in the `UIKit` namespace, and that's what's used here. The two generic arguments to `ViewRenderer` basically say "A `HelloView` object is actually rendered as an iOS `UILabel` object."

The one essential job for a `ViewRenderer` derivative is to override the `OnElementChanged` method. This method is called when a `HelloView` object is created, and its job is to create a native control for rendering the `HelloView` object.

The `OnElementChanged` override begins by checking the `Control` property that the class inherits from `ViewRenderer`. This `Control` property is defined by `ViewRenderer` to be of type `TNativeView`, so in `HelloViewRenderer` it is of type `UILabel`. The first time that `OnElementChanged` is called, this `Control` property will be `null`. The `UILabel` object must be created. This is what the method does, assigning to it some text and a font size. That `UILabel` method is then passed to the `SetNativeControl` method. Thereafter, the `Control` property will be this `UILabel` object.

The `using` directives at the top of the file are divided into three groups:

- The `using` directive for the `Xamarin.Forms` namespace is required for the `ExportRenderer` attribute, while `Xamarin.Forms.Platform.iOS` is required for the `ViewRenderer` class.
- The iOS `UIKIt` namespace is required for `UILabel`.
- The `using` directives for `HelloRenderers` and `HelloRenderers.iOS` are required only for the `HelloView` and `HelloViewRenderer` references in the `ExportRenderer` attribute because the attribute must be outside the `HelloRenderer.iOS` namespace block.

Those last two `using` directives are particularly annoying because they're only required for a single purpose. If you'd like, you can get rid of those two `using` directives by fully qualifying the class names

within the `ExportRenderer` attribute.

This is done in the following renderer. Here's the complete HelloViewRenderer.cs file in the **Hello-Renderers.Droid** project. The Android widget for displaying text is `TextView` in the `Android.Widget` namespace:

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

using Android.Util;
using Android.Widget;

[assembly: ExportRenderer(typeof(HelloRenderers.HelloView),
                          typeof(HelloRenderers.Droid.HelloViewRenderer))]

namespace HelloRenderers.Droid
{
    public class HelloViewRenderer : ViewRenderer<HelloView, TextView>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<HelloView> args)
        {
            base.OnElementChanged(args);

            if (Control == null)
            {
                SetNativeControl(new TextView(Context)
                {
                    Text = "Hello from Android!"
                });

                Control.SetTextSize(ComplexUnitType.Sp, 24);
            }
        }
    }
}
```

This `HelloViewRenderer` class derives from the Android version of `ViewRenderer`. The generic arguments for `ViewRenderer` indicate that the `HelloView` class is supported by the Android `TextView` widget.

Once again, on the first call to `OnElementChanged`, the `Control` property will be `null`. The method must create a native Android `TextView` widget and call `SetNativeControl`. To save a little space, the newly instantiated `TextView` object is passed directly to the `SetNativeControl` method. Notice that the `TextView` constructor requires the Android `Context` object. This is available as a property of `OnElementChanged`.

After the call to `SetNativeControl`, the `Control` property defined by `ViewRenderer` is the native Android widget, in this case the `TextView` object. The method uses this `Control` property to call `SetTextSize` on the `TextView` object. In Android, text sizes can be scaled in a variety of ways. The `ComplexUnitType.Sp` enumeration member indicates "scaled pixels," which is compatible with how

Xamarin.Forms handles font sizes for `Label` in Android.

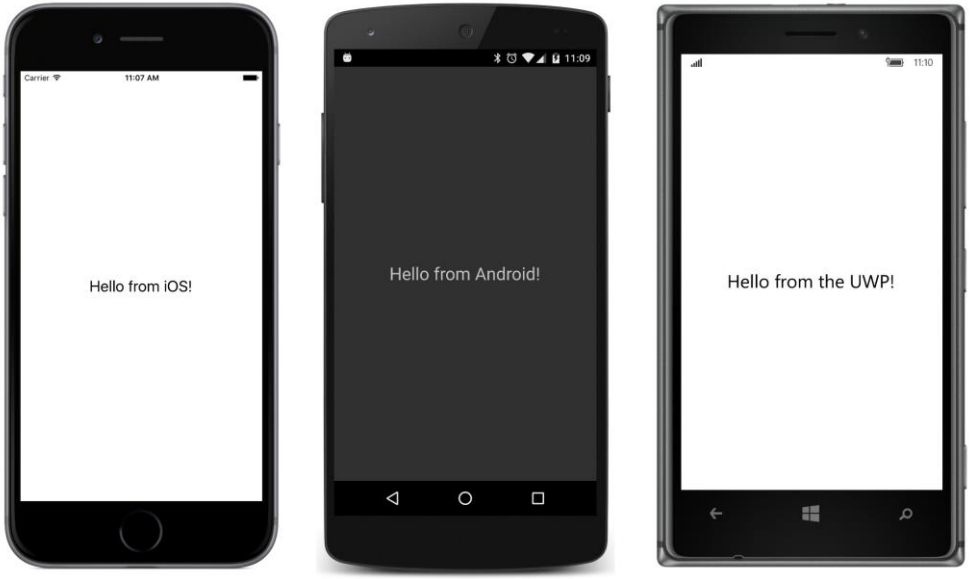
Here's the UWP version of `HelloViewRenderer` in the **HelloRenderers.UWP** project:

```
using Xamarin.Forms.Platform.UWP;

using Windows.UI.Xaml.Controls;

[assembly: ExportRenderer (typeof(HelloRenderers.HelloView),
                           typeof(HelloRenderers.UWP.HelloViewRenderer))]

namespace HelloRenderers.UWP
{
    public class HelloViewRenderer : ViewRenderer<HelloView, TextBlock>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<HelloView> args)
        {
            base.OnElementChanged(args);

            if (Control == null)
            {
                SetNativeControl(new TextBlock
                {
                    Text = "Hello from the UWP!",
                    FontSize = 24,
                });
            }
        }
    }
}
```

In all the Windows platforms, the `HelloView` object is rendered by a Windows Runtime `TextBlock` in the `Windows.UI.Xaml.Controls` namespace.

The `HelloViewRenderer` classes in the **HelloRenderers.Windows** and **HelloRenderers.WinPhone** projects are mostly the same except for namespaces and the text used to set the `Text` property of `TextBlock`.

Here's the program running on the three standard platforms:




Notice how the text is properly centered through the use of the normal `HorizontalOptions` and `VerticalOptions` properties set on the `HelloView` object. However, you can't set the `HorizontalTextAlignment` and `VerticalTextAlignment` properties on `HelloView`. Those properties are defined by `Label` and not by `HelloView`.

To turn `HelloView` into a full-fledged view for displaying text, you'd need to start adding properties to the `HelloView` class. Let's examine how properties are added to renderers with a different example.

## Renderers and properties

Xamarin.Forms includes a `BoxView` element for displaying rectangular blocks of color. Have you ever wished you had something similar for drawing a circle, or to make it more generalized, an ellipse?

That's the purpose of `EllipseView`. However, because you might want to use `EllipseView` in multiple applications, it is implemented in the **Xamarin.FormsBook.Platform** libraries, introduced in Chapter 20, "Async and file I/O."

`BoxView` defines one property on its own—a `Color` property of type `Color`—and `EllipseView` can do the same. It doesn't need properties to set the width and height of the ellipse because it inherits its `WidthRequest` and `HeightRequest` from `VisualElement`.

So here's `EllipseView` as defined in the **Xamarin.FormsBook.Platform** library project:

```
namespace Xamarin.FormsBook.Platform
{
```

```
    public class EllipseView : View
    {
        public static readonly BindableProperty ColorProperty =
            BindableProperty.Create(
                "Color",
                typeof(Color),
                typeof(EllipseView),
                Color.Default);

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

        protected override SizeRequest OnSizeRequest(double widthConstraint,
                                                     double heightConstraint)
        {
            return new SizeRequest(new Size(40, 40));
        }
    }
}
```

The `Color` property simply involves a basic definition of a bindable property with no property-changed handler. The property is defined, but it doesn't seem to be doing anything. Somehow, the `Color` property defined in `EllipseView` has to be linked up with a property on the object that the renderer is rendering.

The only other code in `EllipseView` is an override of `OnSizeRequest` to set a default size of the ellipse, the same as `BoxView`.

Let's begin with the Windows platform. It turns out that a Windows renderer for `EllipseView` is simpler than the iOS and Android renderers.

As you'll recall, the **Xamarin.FormsBook.Platform** solution created in Chapter 20 has a facility to allow sharing code among the various Windows platforms: The **Xamarin.FormsBook.Platform.UWP** library, the **Xamarin.FormsBook.Platform.Windows** library, and the **Xamarin.FormsBook.Platform.WinPhone** library all have references to the **Xamarin.FormsBook.Platform.WinRT** library, which is not a library at all but actually a shared project. This shared project is where the `EllipseViewRenderer` class for all the Windows platforms can reside.

On the Windows platforms, an `EllipseView` can be rendered by a native Windows element called `Ellipse` in the `Windows.UI.Xaml.Shapes` namespace, because `Ellipse` satisfies the criteria of deriving from `Windows.UI.Xaml.FrameworkElement`.

The `Ellipse` is specified as the second generic argument to the `ViewRenderer` class. Because this file is shared by all the Windows platforms, it needs some preprocessing directives to include the correct namespace for the `ExportRendererAttribute` and `ViewRenderer` classes:

```
using System.ComponentModel;
```

```
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;

#if WINDOWS_UWP
using Xamarin.Forms.Platform.UWP;
#else
using Xamarin.Forms.Platform.WinRT;
#endif

[assembly: ExportRenderer(typeof(Xamarin.FormsBook.Platform.EllipseView),
                          typeof(Xamarin.FormsBook.Platform.WinRT.EllipseViewRenderer))]

namespace Xamarin.FormsBook.Platform.WinRT
{
    public class EllipseViewRenderer : ViewRenderer<EllipseView, Ellipse>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<EllipseView> args)
        {
            base.OnElementChanged(args);

            if (Control == null)
            {
                SetNativeControl(new Ellipse());
            }

            if (args.NewElement != null)
            {
                SetColor();
            }
        }
        …
    }
}
```

As you might expect by now, the `OnElementChanged` override first checks whether the `Control` property is `null`, and if so, it creates the native object, in this case an `Ellipse`, and passes it to `SetNativeControl`. Thereafter, the `Control` property is set to this `Ellipse` object.

This `OnElementChanged` override also contains some additional code involving the `ElementChangedEventArgs` argument. This requires a little explanation:

Each renderer instance—in this example, an instance of this `EllipseViewRenderer` class—maintains a single instance of a native object, in this example an `Ellipse`.

However, the rendering infrastructure has a facility both to attach a renderer instance to a Xamarin.Forms element and to detach it and attach another Xamarin.Forms element to the same renderer. Perhaps Xamarin.Forms needs to re-create the element or substitute another element for the one already associated with the renderer.

Changes of this sort are communicated to the renderer with calls to `OnElementChanged`. The `ElementChangedEventArgs` argument includes two properties, `OldElement` and `NewElement`, both of

the type indicated in the generic argument to `ElementChangedEventArgs`, in this case `EllipseView`. In many cases, you don't have to worry about different Xamarin.Forms elements being attached and detached from a single renderer instance. But in some cases you might want to use the opportunity to clean up or free some resources that your renderer uses.

In the simplest and most common case, each renderer instance will get one call to `OnElementChanged` for the Xamarin.Forms view that uses that renderer. You'll use the call to `OnElementChanged` to create the native element and pass it to `SetNativeControl`, as you've already seen. After that call to `SetNativeControl`, the `Control` property defined by `ViewRenderer` is the native object, in this case the `Ellipse`.

At the time you get that call to `OnElementChanged`, the Xamarin.Forms object (in this case an `EllipseView`) has probably already been created and it might also have some properties set. (In other words, the element might be initialized with a few property settings by the time the renderer is required to display the element.) But the system is designed so that this is not necessarily the case. It's possible that a subsequent call to `OnElementChanged` indicates that an `EllipseView` has been created.

What's important is the `NewElement` property of the event arguments. If that property is not `null` (which is the normal case), that property is the Xamarin.Forms element, and you should transfer property settings from that Xamarin.Forms element to the native object. That's the purpose of the call to the `SetColor` method shown above. You'll see the body of that method shortly.

The `ViewRenderer` defines a property named `Element` that it sets to the Xamarin.Forms element, in this case an `EllipseView`. If the most recent call to `OnElementChanged` contained a non-`null` `NewElement` property, then `Element` is that same object.

In summary, these are the two essential properties that you can use throughout your renderer class:

- `Element`—the Xamarin.Forms element, valid if the most recent `OnElementChanged` call had a non-`null` `NewElement` property.
- `Control`—the native view, or widget, or control object, valid after a call to `SetNativeView`.

As you know, properties of Xamarin.Forms elements can change. For example, the `Color` property of `EllipseView` might be animated. If a property such as `Color` is backed by a bindable property, any change to that property causes a `PropertyChanged` event to be fired.

The renderer is also notified of that property change. Any change to a bindable property in a Xamarin.Forms element attached to a renderer also causes a call to the protected virtual `OnElementPropertyChanged` method in the `ViewRenderer` class. In this particular example, any change to *any* bindable property in `EllipseView` (including the `Color` property) generates a call to `OnElementPropertyChanged`. Your renderer should override that method and check for which property has changed:

```
namespace Xamarin.FormsBook.Platform.WinRT
{
    public class EllipseViewRenderer : ViewRenderer<EllipseView, Ellipse>
```

```
    {
        …
        protected override void OnElementPropertyChanged(object sender,
                                                         PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == EllipseView.ColorProperty.PropertyName)
            {
                SetColor();
            }
        }
        …
    }
}
```

If the `Color` property has changed, the `PropertyName` property of the event argument is "Color," the text name specified when the `EllipseView.ColorProperty` bindable property was created. But to avoid misspelling the name, the `OnElementPropertyChanged` method checks the actual string value in the bindable property. The renderer must respond by transferring that new setting of the `Color` property to the native object, in this case the Windows `Ellipse` object.

This `SetColor` method is called from only two places—the `OnElementChanged` override and the `OnElementPropertyChanged` override. Don't think you can skip the call in `OnElementChanged` under the assumption that the property hasn't changed prior to the call to `OnElementChanged`. It is very often the case that `OnElementChanged` is called *after* an element has been initialized with property settings.

However, `SetColor` can make some valid assumptions about the existence of the Xamarin.Forms element and the native control: When `SetColor` is called from `OnElementChanged`, the native control has been created and `NewElement` is non-`null`. This means that both the `Control` and `Element` properties are valid. The `Element` property is also valid when `OnElementPropertyChanged` is called because that's the object whose property has just changed.

This means that the `SetColor` method can simply transfer a color from `Element` (the Xamarin.Forms element) to `Control`, the native object. To avoid namespace clashes, this `SetColor` method fully qualifies all references to any structure named `Color`:

```
namespace Xamarin.FormsBook.Platform.WinRT
{
    public class EllipseViewRenderer : ViewRenderer<EllipseView, Ellipse>
    {
        …
        void SetColor()
        {
            if (Element.Color == Xamarin.Forms.Color.Default)
            {
                Control.Fill = null;
            }
            else
```

```
            {
                Xamarin.Forms.Color color = Element.Color;

                global::Windows.UI.Color winColor =
                    global::Windows.UI.Color.FromArgb((byte)(color.A * 255),
                                                      (byte)(color.R * 255),
                                                      (byte)(color.G * 255),
                                                      (byte)(color.B * 255));

                Control.Fill = new SolidColorBrush(winColor);
            }
        }
    }
}
```

The Windows `Ellipse` object has a property named `Fill` of type `Brush`. By default, this property is `null`, and that's what the `SetColor` method sets it to if the `Color` property of `EllipseView` is `Color.Default`. Otherwise, the Xamarin.Forms `Color` must be converted to a Windows `Color`, which is then passed to the `SolidColorBrush` constructor. The `SolidColorBrush` objects is set to the `Fill` property of `Ellipse`.

That's the Windows version, but when it comes time to create iOS and Android renderers for `EllipseView`, you might feel a little stymied. Here again are the constraints for the second generic parameter to `ViewRenderer`:

- iOS: `TNativeView` is constrained to `UIKit.UIView`
- Android: `TNativeView` is constrained to `Android.View.Views`
- Windows: `TNativeElement` is constrained to `Windows.UI.Xaml.FrameworkElement`

This means that to make an `EllipseView` renderer for iOS, you need a `UIView` derivative that displays an ellipse. Does something like that exist? No, it does not. Therefore, you must make one yourself. This is the first step to making the iOS renderer.

For that reason, the **Xamarin.FormsBook.Platform.iOS** library contains a class named `EllipseUIView` that derives from `UIView` for the sole purpose of drawing an ellipse:

```
using CoreGraphics;
using UIKit;

namespace Xamarin.FormsBook.Platform.iOS
{
    public class EllipseUIView : UIView
    {
        UIColor color = UIColor.Clear;

        public EllipseUIView()
        {
            BackgroundColor = UIColor.Clear;
        }
```

```
        public override void Draw(CGRect rect)
        {
            base.Draw(rect);

            using (CGContext graphics = UIGraphics.GetCurrentContext())
            {
                //Create ellipse geometry based on rect field.
                CGPath path = new CGPath();
                path.AddEllipseInRect(rect);
                path.CloseSubpath();

                //Add geometry to graphics context and draw it.
                color.SetFill();
                graphics.AddPath(path);
                graphics.DrawPath(CGPathDrawingMode.Fill);
            }
        }

        public void SetColor(UIColor color)
        {
            this.color = color;
            SetNeedsDisplay();
        }
    }
}
```

The class overrides the `OnDraw` method to create a graphics path of an ellipse and then to draw it on the graphics context. The color it uses is stored as a field and is initially set to `UIColor.Clear`, which is transparent. However, you'll notice a `SetColor` method at the bottom. This delivers new color to the class and then calls `SetNeedsDisplay`, which invalidates the drawing surface and generates another call to `OnDraw`.

Notice also that the `BackgroundColor` of the `UIView` is set in the constructor to `UIColor.Clear`. Without that setting, the view has a black background in the area not covered by the ellipse.

Now that the `EllipseUIView` class exists for iOS, the `EllipseViewRenderer` can be written using `EllipseUIView` as the native control. Structurally, this class is virtually identical to the Windows renderer:

```
using System.ComponentModel;

using UIKit;

using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer(typeof(Xamarin.FormsBook.Platform.EllipseView),
                          typeof(Xamarin.FormsBook.Platform.iOS.EllipseViewRenderer))]

namespace Xamarin.FormsBook.Platform.iOS
{
    public class EllipseViewRenderer : ViewRenderer<EllipseView, EllipseUIView>
```

```
    {
        protected override void OnElementChanged(ElementChangedEventArgs<EllipseView> args)
        {
            base.OnElementChanged(args);

            if (Control == null)
            {
                SetNativeControl(new EllipseUIView());
            }

            if (args.NewElement != null)
            {
                SetColor();
            }
        }

        protected override void OnElementPropertyChanged(object sender,
                                                         PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == EllipseView.ColorProperty.PropertyName)
            {
                SetColor();
            }
        }

        void SetColor()
        {
            if (Element.Color != Color.Default)
            {
                Control.SetColor(Element.Color.ToUIColor());
            }
            else
            {
                Control.SetColor(UIColor.Clear);
            }
        }
    }
}
```

The only real differences between this renderer and the Windows version is that the `Control` property is set to an instance of `ColorUIView`, and the body of the `SetColor` method at the bottom is different. It now calls the `SetColor` method in `ColorUIView`. This `SetColor` method is also able to make use of a public extension method in the **Xamarin.Forms.Platform.iOS** library called `ToUIColor` to convert a Xamarin.Forms color to an iOS color.

You might have noticed that neither the Windows renderer nor the iOS renderer had to worry about sizing. As you'll see shortly, an `EllipseView` can be set to a variety of sizes, and the size calculated in the Xamarin.Forms layout system becomes the size of the native control.

This unfortunately turned out *not* to be the case with the Android renderer. The Android renderer

needs some sizing logic. Like iOS, Android is also missing a native control that renders an ellipse. Therefore, the **Xamarin.FormsBook.Platform.Android** library contains a class named `EllipseDrawableView` that derives from `View` and draws an ellipse:

```
using Android.Content;
using Android.Views;
using Android.Graphics.Drawables;
using Android.Graphics.Drawables.Shapes;
using Android.Graphics;

namespace Xamarin.FormsBook.Platform.Android
{
    public class EllipseDrawableView : View
    {
        ShapeDrawable drawable;

        public EllipseDrawableView(Context context) : base(context)
        {
            drawable = new ShapeDrawable(new OvalShape());
        }

        protected override void OnDraw(Canvas canvas)
        {
            base.OnDraw(canvas);
            drawable.Draw(canvas);
        }

        public void SetColor(Xamarin.Forms.Color color)
        {
            drawable.Paint.SetARGB((int)(255 * color.A),
                                   (int)(255 * color.R),
                                   (int)(255 * color.G),
                                   (int)(255 * color.B));
            Invalidate();
        }

        public void SetSize(double width, double height)
        {
            float pixelsPerDip = Resources.DisplayMetrics.Density;
            drawable.SetBounds(0, 0, (int)(width * pixelsPerDip),
                                     (int)(height * pixelsPerDip));
            Invalidate();
        }
    }
}
```

Structurally, this is similar to the `EllipseUIView` class defined for iOS, except that the constructor creates a `ShapeDrawable` object for an ellipse, and the `OnDraw` override renders it.

This class has two methods to set properties of this ellipse. The `SetColor` method converts a Xamarin.Forms color to set the `Paint` property of the `ShapeDrawable` object, and the `SetSize` method converts a size in device-independent units to pixels for setting the bounds of the `ShapeDrawable`

object. Both `SetColor` and `SetSize` conclude with a call to `Invalidate` to invalidate the drawing surface and generate another call to `OnDraw`.

The Android renderer makes use of the `EllipseDrawableView` class as its native object:

```
using System.ComponentModel;

using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

[assembly: ExportRenderer(typeof(Xamarin.FormsBook.Platform.EllipseView),
                          typeof(Xamarin.FormsBook.Platform.Android.EllipseViewRenderer))]

namespace Xamarin.FormsBook.Platform.Android
{
    public class EllipseViewRenderer : ViewRenderer<EllipseView, EllipseDrawableView>
    {
        double width, height;

        protected override void OnElementChanged(ElementChangedEventArgs<EllipseView> args)
        {
            base.OnElementChanged(args);

            if (Control == null)
            {
                SetNativeControl(new EllipseDrawableView(Context));
            }

            if (args.NewElement != null)
            {
                SetColor();
                SetSize();
            }
        }

        protected override void OnElementPropertyChanged(object sender,
                                                         PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == VisualElement.WidthProperty.PropertyName)
            {
                width = Element.Width;
                SetSize();
            }
            else if (args.PropertyName == VisualElement.HeightProperty.PropertyName)
            {
                height = Element.Height;
                SetSize();
            }
            else if (args.PropertyName == EllipseView.ColorProperty.PropertyName)
            {
                SetColor();
            }
```

```
        }

        void SetColor()
        {
            Control.SetColor(Element.Color);
        }

        void SetSize()
        {
            Control.SetSize(width, height);
        }
    }
}
```

Notice that the `OnElementPropertyChanged` method needs to check for changes to both the `Width` and `Height` properties and save them in fields so they can be combined into a single `Bounds` setting for the `SetSize` call to `EllipseDrawableView`.

With all the renderers in place, it's time to see whether it works. The **EllipseDemo** solution also contains links to the various projects of the **Xamarin.FormsBook.Platform** solution, and each of the projects in **EllipseDemo** contains a reference to the corresponding library project in **Xamarin.FormsBook.Platform**.

Each of the projects in **EllipseDemo** also contains a call to the `Toolkit.Init` method in the corresponding library project. This is not always necessary. But keep in mind that the various renderers are not directly referenced by any code in any of the projects, and some optimizations can cause the code not to be available at run time. The call to `Toolkit.Init` avoids that.

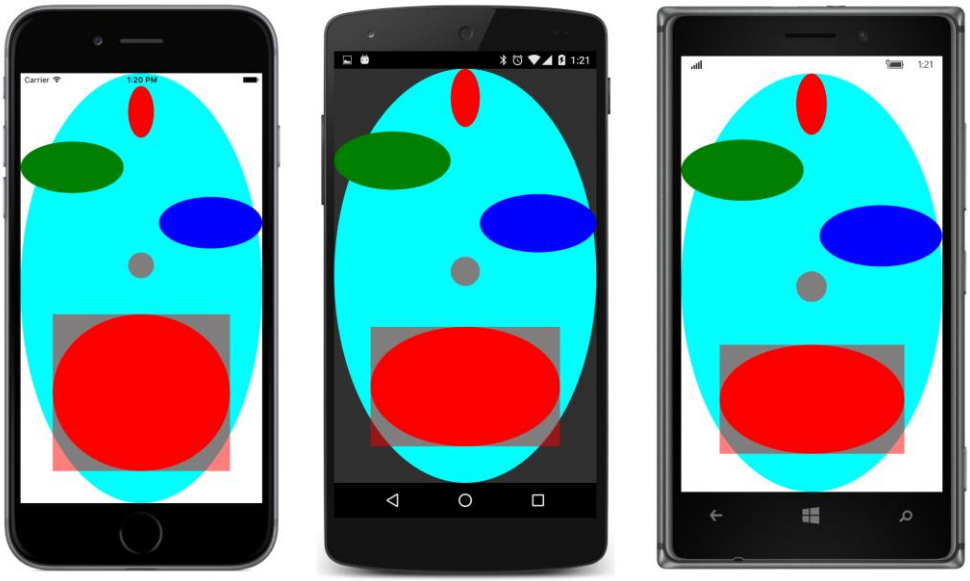
The XAML file in **EllipseDemo** creates several `EllipseView` objects with different colors and sizes, some constrained in size while others are allowed to fill their container:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:platform=
                 "clr-namespace:Xamarin.FormsBook.Platform;assembly=Xamarin.FormsBook.Platform"
             x:Class="EllipseDemo.EllipseDemoPage">
    <Grid>
        <platform:EllipseView Color="Aqua" />

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

            <platform:EllipseView Color="Red"
                                  WidthRequest="40"
                                  HeightRequest="80"
                                  HorizontalOptions="Center" />

            <platform:EllipseView Color="Green"
```

```
                                WidthRequest="160"
                                HeightRequest="80"
                                HorizontalOptions="Start" />

            <platform:EllipseView Color="Blue"
                                WidthRequest="160"
                                HeightRequest="80"
                                HorizontalOptions="End" />

            <platform:EllipseView Color="#80FF0000"
                                HorizontalOptions="Center" />

            <ContentView Padding="50"
                         VerticalOptions="FillAndExpand">

                <platform:EllipseView Color="Red"
                                      BackgroundColor="#80FF0000" />

            </ContentView>
        </StackLayout>
    </Grid>
</ContentPage>
```

Take note in particular of the penultimate `EllipseView` that gives itself a half-opaque red color. Against the `Aqua` of the large ellipse filling the page, this should render as medium gray.

The last `EllipseView` gives itself a `BackgroundColor` setting of half-opaque red. Again, this should render as gray against the large `Aqua` ellipse, but as a light red against a white background and dark red against a black background. Here they are:



Once you have an `EllipseView`, of course you'll want to write a bouncing-ball program. The **BouncingBall** solution also includes links to all the projects in the **Xamarin.FormsBook.Platform** solution, and all the application projects have references to the corresponding library projects. The **BouncingBall** PCL also has a reference to the **Xamarin.FormsBook.Toolkit** library for a structure called `Vector2`, a two-dimensional vector.

The XAML file positions an `EllipseView` in the center of the page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:platform=
                 "clr-namespace:Xamarin.FormsBook.Platform;assembly=Xamarin.FormsBook.Platform"
             x:Class="BouncingBall.BouncingBallPage">

    <platform:EllipseView x:Name="ball"
                          WidthRequest="100"
                          HeightRequest="100"
                          HorizontalOptions="Center"
                          VerticalOptions="Center" />

</ContentPage>
```

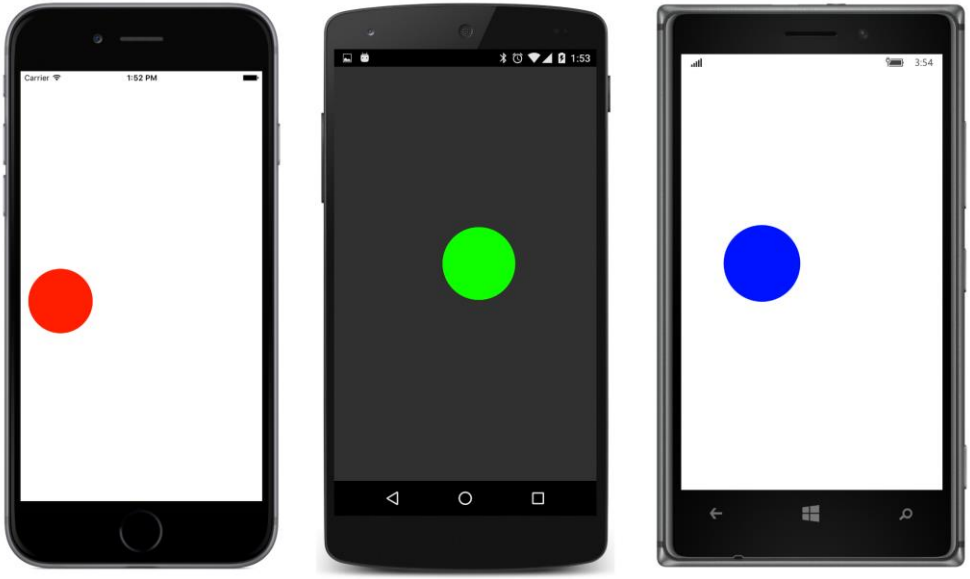
The code-behind file starts up two animations that run "forever." The first animation is defined in the constructor and animates the `Color` property of the bouncing ball to take it through the colors of the rainbow every 10 seconds.

The second animation bounces the ball on the four "walls" of the screen. For each cycle through the `while` loop, the code first determines which wall it will hit first and the distance to that wall in device-independent units. The new calculation of `center` toward the end of the `while` loop is the position of the ball as it strikes a wall. The new calculation of `vector` determines a deflection vector based on an existing vector and a vector that is perpendicular to the surface that it's hitting (called a *normal* vector):

```
public partial class BouncingBallPage : ContentPage
{
    public BouncingBallPage()
    {
        InitializeComponent();

        // Color animation: cycle through rainbow every 10 seconds.
        new Animation(callback: v => ball.Color = Color.FromHsla(v, 1, 0.5),
                      start: 0,
                      end: 1
                      ).Commit(owner: this,
                               name: "ColorAnimation",
                               length: 10000,
                               repeat: () => true);

        BounceAnimationLoop();
    }

    async void BounceAnimationLoop()
```

```
    {
        // Wait until the dimensions are good.
        while (Width == -1 && Height == -1)
        {
            await Task.Delay(100);
        }

        // Initialize points and vectors.
        Point center = new Point();
        Random rand = new Random();
        Vector2 vector = new Vector2(rand.NextDouble(), rand.NextDouble());
        vector = vector.Normalized;
        Vector2[] walls = { new Vector2(1, 0), new Vector2(0, 1),       // left, top
                            new Vector2(-1, 0), new Vector2(0, -1) };   // right, bottom

        while (true)
        {
            // The locations of the four "walls" (taking ball size into account).
            double right = Width / 2 - ball.Width / 2;
            double left = -right;
            double bottom = Height / 2 - ball.Height / 2;
            double top = -bottom;

            // Find the number of steps till a wall is hit.
            double nX = Math.Abs(((vector.X > 0 ? right : left) - center.X) / vector.X);
            double nY = Math.Abs(((vector.Y > 0 ? bottom : top) - center.Y) / vector.Y);
            double n = Math.Min(nX, nY);

            // Find the wall that's being hit.
            Vector2 wall = walls[nX < nY ? (vector.X > 0 ? 2 : 0) : (vector.Y > 0 ? 3 : 1)];

            // New center and vector after animation.
            center += n * vector;
            vector -= 2 * Vector2.DotProduct(vector, wall) * wall;

            // Animate at 3 msec per unit.
            await ball.TranslateTo(center.X, center.Y, (uint)(3 * n));
        }
    }
}
```

Of course, a still photograph can't possibly capture the exciting action of the animation:




## Renderers and events

Most Xamarin.Forms elements are interactive. They respond to user input by firing events. If you implement an event in your Xamarin.Forms custom element, you probably also need to define an event handler in the renderers for the corresponding event that the native control fires. This section will show you how.

The `StepSlider` element was inspired by a problem with the Xamarin.Forms implementation of the Windows `Slider` element. By default, the Xamarin.Forms `Slider` when running on the Windows platforms has only 10 steps from 0 through 1, so it is only capable of `Value` values of 0, 0.1, 0.2, and so forth up to 1.0.

Like the regular Xamarin.Forms `Slider`, the `StepSlider` element has `Minimum`, `Maximum`, and `Value` properties, but it also defines a `Step` property to specify the number of steps between `Minimum` and `Maximum`. For example, if `Minimum` is set to 5, `Maximum` is set to 10, and `Step` is set to 20, then the possible values of the `Value` property are 5.00, 5.25, 5.50, 5.75, 6.00, and so forth up to 10. The number of possible `Value` values is equal to the `Step` value plus 1.

Interestingly, implementing this `Step` property turned out to require a different approach on all three platforms, but the primary purpose of this exercise is to demonstrate how to implement events.

Here is the `StepSlider` class in the **Xamarin.FormsBook.Platform** library. Notice the definition of the `ValueChanged` event at the top and the firing of that event by changes in the `Value` property. Much of the bulk of the bindable property definitions are devoted to the `validateValue` methods,

which ensure that the property is within allowable bounds, and the `coerceValue` methods, which ensure that the properties are consistent among themselves:

```
namespace Xamarin.FormsBook.Platform
{
    public class StepSlider : View
    {
        public event EventHandler<ValueChangedEventArgs> ValueChanged;

        public static readonly BindableProperty MinimumProperty =
            BindableProperty.Create(
                "Minimum",
                typeof(double),
                typeof(StepSlider),
                0.0,
                validateValue: (obj, min) => (double)min < ((StepSlider)obj).Maximum,
                coerceValue: (obj, min) =>
                {
                    StepSlider stepSlider = (StepSlider)obj;
                    stepSlider.Value = stepSlider.Coerce(stepSlider.Value,
                                                         (double)min,
                                                         stepSlider.Maximum);
                    return min;
                });

        public static readonly BindableProperty MaximumProperty =
            BindableProperty.Create(
                "Maximum",
                typeof(double),
                typeof(StepSlider),
                100.0,
                validateValue: (obj, max) => (double)max > ((StepSlider)obj).Minimum,
                coerceValue: (obj, max) =>
                {
                    StepSlider stepSlider = (StepSlider)obj;
                    stepSlider.Value = stepSlider.Coerce(stepSlider.Value,
                                                         stepSlider.Minimum,
                                                         (double)max);
                    return max;
                });

        public static readonly BindableProperty StepsProperty =
            BindableProperty.Create(
                "Steps",
                typeof(int),
                typeof(StepSlider),
                100,
                validateValue: (obj, steps) => (int)steps > 1);

        public static readonly BindableProperty ValueProperty =
            BindableProperty.Create(
                "Value",
                typeof(double),
                typeof(StepSlider),
```

```
                0.0,
                BindingMode.TwoWay,
                coerceValue: (obj, value) =>
                {
                    StepSlider stepSlider = (StepSlider)obj;
                    return stepSlider.Coerce((double)value,
                                             stepSlider.Minimum,
                                             stepSlider.Maximum);
                },
                propertyChanged: (obj, oldValue, newValue) =>
                {
                    StepSlider stepSlider = (StepSlider)obj;
                    EventHandler<ValueChangedEventArgs> handler = stepSlider.ValueChanged;
                    if (handler != null)
                        handler(obj, new ValueChangedEventArgs((double)oldValue,
                                                               (double)newValue));
                });

        public double Minimum
        {
            set { SetValue(MinimumProperty, value); }
            get { return (double)GetValue(MinimumProperty); }
        }

        public double Maximum
        {
            set { SetValue(MaximumProperty, value); }
            get { return (double)GetValue(MaximumProperty); }
        }

        public int Steps
        {
            set { SetValue(StepsProperty, value); }
            get { return (int)GetValue(StepsProperty); }
        }

        public double Value
        {
            set { SetValue(ValueProperty, value); }
            get { return (double)GetValue(ValueProperty); }
        }

        double Coerce(double value, double min, double max)
        {
            return Math.Max(min, Math.Min(value, max));
        }
    }
}
```

The `StepSlider` class fires the `ValueChanged` property when the `Value` property changes, but there's nothing in this class that changes the `Value` property when the user manipulates the platform renderer for `StepSlider`. That's left to the renderer class.

Once again, let's first look at the Windows implementation of `StepSliderRenderer` in the **Xamarin.FormsBook.Platform.WinRT** shared project because it's a little more straightforward. The renderer uses the `Windows.UI.Xaml.Controls.Slider` for the native control. To avoid a namespace clash between the Windows `Slider` and the Xamarin.Forms `Slider`, a `using` directive defines the `win` prefix to refer to the Windows namespace and uses that to reference the Windows `Slider`:

```
using System.ComponentModel;

using Xamarin.Forms;

using Win = Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;

#if WINDOWS_UWP
using Xamarin.Forms.Platform.UWP;
#else
using Xamarin.Forms.Platform.WinRT;
#endif

[assembly: ExportRenderer(typeof(Xamarin.FormsBook.Platform.StepSlider),
                          typeof(Xamarin.FormsBook.Platform.WinRT.StepSliderRenderer))]

namespace Xamarin.FormsBook.Platform.WinRT
{
    public class StepSliderRenderer : ViewRenderer<StepSlider, Win.Slider>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<StepSlider> args)
        {
            base.OnElementChanged(args);

            if (Control == null)
            {
                SetNativeControl(new Win.Slider());
            }

            if (args.NewElement != null)
            {
                SetMinimum();
                SetMaximum();
                SetSteps();
                SetValue();

                Control.ValueChanged += OnWinSliderValueChanged;
            }
            else
            {
                Control.ValueChanged -= OnWinSliderValueChanged;
            }
        }
        …
    }
}
```

The big difference between this renderer and the one you've seen earlier is that this one sets an event handler on the `ValueChanged` event of the native Windows `Slider`. (You'll see the event handler shortly.) If `args.NewElement` becomes `null`, however, that means that there is no longer a Xamarin.Forms element attached to the renderer and that the event handler is no longer needed. Moreover, you'll see soon that the event handler refers to the `Element` property inherited from the `ViewRenderer` class, and that property will also be `null` if `args.NewElement` is `null`.

For that reason, `OnElementChanged` detaches the event handler when `args.NewElement` becomes `null`. Likewise, any resources you've allocated for the renderer should be freed whenever `args.NewElement` becomes `null`.

The override of the `OnElementPropertyChanged` method checks for changes in the four properties that `StepSlider` defines:

```
namespace Xamarin.FormsBook.Platform.WinRT
{
    public class StepSliderRenderer : ViewRenderer<StepSlider, Win.Slider>
    {
        …
        protected override void OnElementPropertyChanged(object sender,
                                                         PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == StepSlider.MinimumProperty.PropertyName)
            {
                SetMinimum();
            }
            else if (args.PropertyName == StepSlider.MaximumProperty.PropertyName)
            {
                SetMaximum();
            }
            else if (args.PropertyName == StepSlider.StepsProperty.PropertyName)
            {
                SetSteps();
            }
            else if (args.PropertyName == StepSlider.ValueProperty.PropertyName)
            {
                SetValue();
            }
        }
        …
    }
}
```

The Windows `Slider` defines `Minimum`, `Maximum`, and `Value` properties just like the Xamarin.Forms `Slider` and the new `StepSlider`. But it doesn't define a `Steps` property. Instead, it defines a `StepFrequency` property, which is the opposite of a `Steps` property. To reproduce the earlier example (`Minimum` set to 5, `Maximum` set to 10, and `Steps` set to 20), you'd set `StepFrequency` to 0.25. The conversion is fairly simple:

```
namespace Xamarin.FormsBook.Platform.WinRT
{
    public class StepSliderRenderer : ViewRenderer<StepSlider, Win.Slider>
    {
        …
        void SetMinimum()
        {
            Control.Minimum = Element.Minimum;
        }

        void SetMaximum()
        {
            Control.Maximum = Element.Maximum;
        }

        void SetSteps()
        {
            Control.StepFrequency = (Element.Maximum - Element.Minimum) / Element.Steps;
        }

        void SetValue()
        {
            Control.Value = Element.Value;
        }
        …
    }
}
```

Finally, here's the `ValueChanged` handler for the Windows `Slider`. This has the responsibility of setting the `Value` property in the `StepSlider`, which then fires its own `ValueChanged` event. However, a special method exists for setting a value from a renderer. This method, called `SetValueFromRenderer`, is defined by the `IElementController` interface and implemented by the Xamarin.Forms `Element` class:

```
namespace Xamarin.FormsBook.Platform.WinRT
{
    public class StepSliderRenderer : ViewRenderer<StepSlider, Win.Slider>
    {
        …
        void OnControlValueChanged(object sender, RangeBaseValueChangedEventArgs args)
        {
            ((IElementController)Element).SetValueFromRenderer(StepSlider.ValueProperty,
                                                               args.NewValue);
        }
    }
}
```

The iOS `UISlider` has `MinValue`, `MaxValue`, and `Value` properties and defines a `ValueChanged` event, but it doesn't have anything like a `Steps` or `StepFrequency` property. Instead, the iOS `StepSliderRenderer` class in **Xamarin.FormsBook.Platform.iOS** makes a manual adjustment to the `Value` property before calling `SetValueFromRenderer` from the `ValueChanged` event handler:

```
using System;
using System.ComponentModel;

using UIKit;

using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer(typeof(Xamarin.FormsBook.Platform.StepSlider),
                          typeof(Xamarin.FormsBook.Platform.iOS.StepSliderRenderer))]

namespace Xamarin.FormsBook.Platform.iOS
{
    public class StepSliderRenderer : ViewRenderer<StepSlider, UISlider>
    {
        int steps;

        protected override void OnElementChanged(ElementChangedEventArgs<StepSlider> args)
        {
            base.OnElementChanged(args);

            if (Control == null)
            {
                SetNativeControl(new UISlider());
            }

            if (args.NewElement != null)
            {
                SetMinimum();
                SetMaximum();
                SetSteps();
                SetValue();

                Control.ValueChanged += OnUISliderValueChanged;
            }
            else
            {
                Control.ValueChanged -= OnUISliderValueChanged;
            }
        }

        protected override void OnElementPropertyChanged(object sender,
                                                         PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == StepSlider.MinimumProperty.PropertyName)
            {
                SetMinimum();
            }
            else if (args.PropertyName == StepSlider.MaximumProperty.PropertyName)
            {
                SetMaximum();
            }
```

```
            else if (args.PropertyName == StepSlider.StepsProperty.PropertyName)
            {
                SetSteps();
            }
            else if (args.PropertyName == StepSlider.ValueProperty.PropertyName)
            {
                SetValue();
            }
        }

        void SetMinimum()
        {
            Control.MinValue = (float)Element.Minimum;
        }

        void SetMaximum()
        {
            Control.MaxValue = (float)Element.Maximum;
        }

        void SetSteps()
        {
            steps = Element.Steps;
        }

        void SetValue()
        {
            Control.Value = (float)Element.Value;
        }

        void OnUISliderValueChanged(object sender, EventArgs args)
        {
            double increment = (Element.Maximum - Element.Minimum) / Element.Steps;
            double value = increment * Math.Round(Control.Value / increment);
            ((IElementController)Element).SetValueFromRenderer(StepSlider.ValueProperty, value);
        }
    }
}
```

Interestingly enough, the Android `SeekBar` widget has an equivalent to the `Steps` property but no equivalents to the `Minimum` and `Maximum` properties! How is this possible? The `SeekBar` actually defines an integer property named `Max`, and the `Progress` property of the `SeekBar` is always an integer that ranges from 0 to `Max`. So the `Max` property really indicates the number of steps the `SeekBar` can make, and a conversion is necessary between the `Progress` property of the `SeekBar` and the `Value` property of the `StepSlider`.

This conversion occurs in two places: The `SetValue` method converts from the `Value` property of the `StepSlider` to the `Progress` property of the `SeekBar`, and the `OnProgressChanged` method converts from the `Progress` property of the `SeekBar` to the `Value` property of the `StepSlider`.

In addition, the event handler is a little different. The `SetOnSeekBarChangeListener` method accepts an argument of type `IOnSeekBarChangeListener`, which defines three methods that report

changes to the `Seekbar`, including the method `OnProgressChanged`. The renderer itself implements that interface.

Here's the complete `StepSliderRenderer` class in the **Xamarin.FormsBook.Platform.Android** library:

```
using System.ComponentModel;

using Android.Widget;

using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

[assembly: ExportRenderer(typeof(Xamarin.FormsBook.Platform.StepSlider),
                          typeof(Xamarin.FormsBook.Platform.Android.StepSliderRenderer))]

namespace Xamarin.FormsBook.Platform.Android
{
    public class StepSliderRenderer : ViewRenderer<StepSlider, SeekBar>,
                                      SeekBar.IOnSeekBarChangeListener
    {
        double minimum, maximum;

        protected override void OnElementChanged(ElementChangedEventArgs<StepSlider> args)
        {
            base.OnElementChanged(args);

            if (Control == null)
            {
                SetNativeControl(new SeekBar(Context));
            }
            if (args.NewElement != null)
            {
                SetMinimum();
                SetMaximum();
                SetSteps();
                SetValue();

                Control.SetOnSeekBarChangeListener(this);
            }
            else
            {
                Control.SetOnSeekBarChangeListener(null);
            }
        }

        protected override void OnElementPropertyChanged(object sender,
                                                         PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == StepSlider.MinimumProperty.PropertyName)
            {
                SetMinimum();
```

```
            }
            else if (args.PropertyName == StepSlider.MaximumProperty.PropertyName)
            {
                SetMaximum();
            }
            else if (args.PropertyName == StepSlider.StepsProperty.PropertyName)
            {
                SetSteps();
            }
            else if (args.PropertyName == StepSlider.ValueProperty.PropertyName)
            {
                SetValue();
            }
        }

        void SetMinimum()
        {
            minimum = Element.Minimum;
        }

        void SetMaximum()
        {
            maximum = Element.Maximum;
        }

        void SetSteps()
        {
            Control.Max = Element.Steps;
        }

        void SetValue()
        {
            double value = Element.Value;
            Control.Progress = (int)((value - minimum) / (maximum - minimum) * Element.Steps);
        }

        // Implementation of SeekBar.IOnSeekBarChangeListener
        public void OnProgressChanged(SeekBar seekBar, int progress, bool fromUser)
        {
            double value = minimum + (maximum - minimum) * Control.Progress / Control.Max;
            ((IElementController)Element).SetValueFromRenderer(StepSlider.ValueProperty, value);
        }

        public void OnStartTrackingTouch(SeekBar seekBar)
        {
        }

        public void OnStopTrackingTouch(SeekBar seekBar)
        {
        }
    }
}
```

The **StepSliderDemo** solution contains links to the **Xamarin.FormsBook.Platform** libraries and

corresponding references to those libraries. The StepSliderDemo.xaml file instantiates five `StepSlider` elements, with data bindings on three of them and an explicit event handler on the other two:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:platform=
                 "clr-namespace:Xamarin.FormsBook.Platform;assembly=Xamarin.FormsBook.Platform"
             x:Class="StepSliderDemo.StepSliderDemoPage">
    <StackLayout Padding="10, 0">
        <StackLayout.Resources>
            <ResourceDictionary>
                <Style TargetType="ContentView">
                    <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                </Style>

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

        <ContentView>
            <StackLayout>
                <platform:StepSlider x:Name="stepSlider1" />
                <Label Text="{Binding Source={x:Reference stepSlider1},
                                      Path=Value}" />
            </StackLayout>
        </ContentView>

        <ContentView>
            <StackLayout>
                <platform:StepSlider x:Name="stepSlider2"
                                     Minimum="10"
                                     Maximum="15"
                                     Steps="20"
                                     ValueChanged="OnSliderValueChanged" />
                <Label x:Name="label2" />
            </StackLayout>
        </ContentView>

        <ContentView>
            <StackLayout>
                <platform:StepSlider x:Name="stepSlider3"
                                     Steps="10" />
                <Label Text="{Binding Source={x:Reference stepSlider3},
                                      Path=Value}" />
            </StackLayout>
        </ContentView>

        <ContentView>
            <StackLayout>
                <platform:StepSlider x:Name="stepSlider4"
                                     Minimum="0"
```

```
                                    Maximum="1"
                                    Steps="100"
                                    ValueChanged="OnSliderValueChanged" />
                <Label x:Name="label4" />
            </StackLayout>
        </ContentView>

        <ContentView>
            <StackLayout>
                <platform:StepSlider x:Name="stepSlider5"
                                     Minimum="10"
                                     Maximum="20"
                                     Steps="2" />
                <Label Text="{Binding Source={x:Reference stepSlider5},
                                      Path=Value}" />
            </StackLayout>
        </ContentView>
    </StackLayout>
</ContentPage>
```

The code-behind file has the `ValueChanged` event handler:

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

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        StepSlider stepSlider = (StepSlider)sender;

        if (stepSlider == stepSlider2)
        {
            label2.Text = stepSlider2.Value.ToString();
        }
        else if (stepSlider == stepSlider4)
        {
            label4.Text = stepSlider4.Value.ToString();
        }
    }
}
```

You'll find that the `StepSlider` functions like a normal Xamarin.Forms `Slider` except that the possible values from the `StepSlider` are now under programmatic control:


The first `StepSlider` has `Value` properties in increments of 1, the second in increments of 0.25, the third in increments of 10, the fourth in increments of 0.01, and the fifth in increments of 5 with just three possible settings.

And now you can see how Xamarin.Forms provides the tools that let you take it beyond what it at first seems to be. Anything you can define in three platforms can become something usable in just one universal platform. With the C# programming language, and the power of Xamarin.Forms and renderers, you can step not only into iOS programming, or Android programming, or Windows programming, but all three at once with a single step, and continue to step into the future of mobile development.

# About the author

Charles Petzold works for Xamarin on the documentation team. His earlier years as a freelancer were spent largely writing books for Microsoft Press about Windows, Windows Phone, and .NET programming, including six editions of the legendary *Programming Windows,* from 1988 to 2012.

Petzold is also the author of two unique books on the mathematical and philosophical foundations of digital computing, *Code: The Hidden Language of Computer Hardware and Software* (Microsoft Press, 1999) and *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine* (Wiley, 2008). He lives in New York City with his wife, the writer Deirdre Sinnott.