Chapter 23

Triggers and behaviors

The introduction of a markup language such as XAML into a graphical programming environment might seem at first to be merely an alternative way to construct an assemblage of user-interface elements. But we've seen that the markup language tends to have much more profound consequences. The markup language induces us to divide the program more decisively between the interactive visuals and the underlying business logic. This further suggests that we might benefit from formalizing such a separation in an application architecture such as MVVM, and that turns out to be quite valuable.

At the same time, markup languages like XAML tend to have some intrinsic deficiencies in comparison with code. While code generally defines a dynamic process, markup languages are usually restricted to describing a fixed state. Several features have been added to Xamarin.Forms to help compensate for these deficiencies. These features include markup extensions, the resource dictionary, styles, and data binding.

In this chapter, you'll see two more of these features, called *triggers* and *behaviors*. Triggers cause changes to the user interface in response to events or property changes, while behaviors are more open-ended, allowing entire chunks of functionality to be added to existing visual elements. Both triggers and behaviors can be part of a `Style` definition. Often triggers and behaviors are supported by code that can contain animations.

It is unlikely that triggers and behaviors would have even been conceived or invented in a code-only programming environment. However, like the resource dictionary, styles, and data binding, these features help developers structure their applications more productively by suggesting additional ways to conceptualize the various pieces and components of these programs—and additional ways to reuse and share code.

Triggers and behaviors are implemented with several classes that will be introduced in the course of this chapter. You'll make use of these triggers and behaviors with two collection properties that are defined by both `VisualElement` and `Style`:

- `Triggers` property of type `IList<TriggerBase>`
- `Behaviors` property of type `IList<Behavior>`

Let's begin with triggers.

Triggers

In the most general (and vaguest) sense, a trigger is a condition that results in a response. More specifically, a trigger responds to a property change or the firing of an event by setting another property or running some code. Almost always, the properties that are set, or the code that is run, involve the user interface and are represented in XAML.

Both `VisualElement` and `Style` define a `Triggers` property of type `IList<TriggerBase>`. The abstract `TriggerBase` class derives from `BindableObject`. Four sealed classes derive from `TriggerBase`:

- `Trigger` for setting properties (or running code) in response to a property change.
- `EventTrigger` for running code in response to an event.
- `DataTrigger` for setting properties (or running code) in response to a property change referenced in a data binding.
- `MultiTrigger` for setting properties (or running code) when multiple triggers occur.

The differences between these will become much clearer in practice.

The simplest trigger

In its usual form, the `Trigger` class checks for a property change of an element and responds by setting another property of the same element.

For example, suppose you've designed a page that contains several `Entry` views. You've decided that when a particular `Entry` gets the input focus, you want the `Entry` to become larger. You want to make the `Entry` stand out, including the text that the user types.

Much more specifically, when the `IsFocused` property of the `Entry` becomes `True`, you want the `Scale` property of the `Entry` to be set to 1.5. When the `IsFocused` property reverts back to `False`, you want the `Scale` property to also revert to its previous value.

To accommodate this concept, `Trigger` defines three properties:

- `Property` of type `BindableProperty`.
- `Value` of type `Object`.
- `Setters` of type `IList<Setter>`. This is the content property of `Trigger`.

All these properties must be set for the `Trigger` to work. From `TriggerBase`, `Trigger` inherits another essential property:

- `TargetType` of type `Type`.

This is the type of the element on which the `Trigger` is attached.

The `Property` and `Value` properties of `Trigger` are sometimes said to constitute a *condition*. When the value of the property referenced by `Property` equals `Value`, the condition is true, and the collection of `Setter` objects are applied to the element.

As you'll recall from Chapter 12, "Styles," `Setter` defines two properties that happen to be the same as the first two `Trigger` properties:

- `Property` of type `BindableProperty`.
- `Value` of type `Object`.

With triggers we're only dealing with bindable properties. The `Trigger` condition property must be backed by a `BindableProperty` as well as the property set by the `Setter`.

When the condition becomes false, the `Setter` objects are "un-applied," meaning that the property referenced by the `Setter` reverts to what its value would be without the `Setter`, which might be the default value of the property, a value set directly on the element, or a value applied through a `Style`.

Here's the XAML file for the **EntryPop** program. Each of the three `Entry` views on the page has a single `Trigger` object added to its `Triggers` collection using the `Entry.Triggers` property-element tag. Each of the `Trigger` objects has a single `Setter` added to its `Setters` collection. Because `Setters` is the content property of `Trigger`, the `Trigger.Setters` property-element tags are not required:

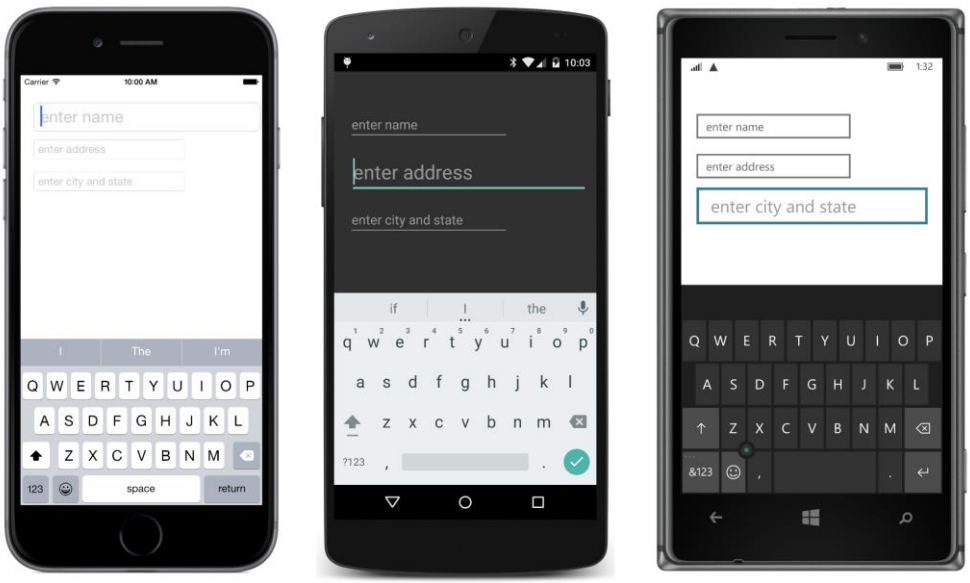
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="EntryPop.EntryPopPage"
             Padding="20, 50, 120, 0">

    <StackLayout Spacing="20">
        <Entry Placeholder="enter name"
               AnchorX="0">
            <Entry.Triggers>
                <Trigger TargetType="Entry" Property="IsFocused" Value="True">
                    <Setter Property="Scale" Value="1.5" />
                </Trigger>
            </Entry.Triggers>
        </Entry>

        <Entry Placeholder="enter address"
               AnchorX="0">
            <Entry.Triggers>
                <Trigger TargetType="Entry" Property="IsFocused" Value="True">
                    <Setter Property="Scale" Value="1.5" />
                </Trigger>
            </Entry.Triggers>
        </Entry>

        <Entry Placeholder="enter city and state"
```

```
                AnchorX="0">
            <Entry.Triggers>
                <Trigger TargetType="Entry" Property="IsFocused" Value="True">
                    <Setter Property="Scale" Value="1.5" />
                </Trigger>
            </Entry.Triggers>
        </Entry>
    </StackLayout>
</ContentPage>
```

Each `Trigger` object must have its `TargetType` set, and in this case that's an `Entry`. Internally, the `Trigger` uses a `PropertyChanged` handler to monitor the value of the `IsFocused` property. When that property equals `True`, then the single `Setter` object sets the `Scale` property to 1.5. The `AnchorX` setting of zero directs the scaling to occur from the left side of the `Entry`. (The nondefault value of `AnchorX` means that the `Entry` views won't be positioned correctly if you change the orientation of the iOS screen.)

When the `Entry` loses input focus and the `IsFocused` property becomes `False` again, the `Trigger` automatically removes the application of the `Setter`, in which case the `Scale` property reverts to its pre-`Trigger` value, which isn't necessarily its default value.

Here are the enlarged `Entry` views with input focus:



Each `Entry` view in this example has only one `Trigger`, and each `Trigger` has only one `Setter`, but in the general case, a visual element can have multiple `Trigger` objects in its `Triggers` collection, and each `Trigger` can have multiple `Setter` objects in its `Setters` collection.

If you were to do something like this in code, you'd attach a `PropertyChanged` event handler to each `Entry` and respond to changes in the `IsFocused` property by setting the `Scale` property. The

advantage of the `Trigger` is that you can do the entire job in markup right where the element is defined, leaving code for jobs presumably more important than increasing the size of an `Entry` element!

For this reason, it's unlikely that you will have the need to create `Trigger` objects in code. Nevertheless, the **EntryPopCode** program demonstrates how you'd do it. The code has been fashioned to resemble the XAML as much as possible:

```
public class EntryPopCodePage : ContentPage
{
    public EntryPopCodePage()
    {
        Padding = new Thickness(20, 50, 120, 0);
        Content = new StackLayout
        {
            Spacing = 20,
            Children =
            {
                new Entry
                {
                    Placeholder = "enter name",
                    AnchorX = 0,
                    Triggers =
                    {
                        new Trigger(typeof(Entry))
                        {
                            Property = Entry.IsFocusedProperty,
                            Value = true,
                            Setters =
                            {
                                new Setter
                                {
                                    Property = Entry.ScaleProperty,
                                    Value = 1.5
                                }
                            }
                        }
                    }
                },
                new Entry
                {
                    Placeholder = "enter addresss",
                    AnchorX = 0,
                    Triggers =
                    {
                        new Trigger(typeof(Entry))
                        {
                            Property = Entry.IsFocusedProperty,
                            Value = true,
                            Setters =
                            {
                                new Setter
                                {
                                    Property = Entry.ScaleProperty,
                                    Value = 1.5
```

```
                            }
                        }
                    }
                }
            },
            new Entry
            {
                Placeholder = "enter city and state",
                AnchorX = 0,
                Triggers =
                {
                    new Trigger(typeof(Entry))
                    {
                        Property = Entry.IsFocusedProperty,
                        Value = true,
                        Setters =
                        {
                            new Setter
                            {
                                Property = Entry.ScaleProperty,
                                Value = 1.5
                            }
                        }
                    }
                }
            }
        }
    };
}
}
```

The only real difference between the XAML and the code is the treatment of the `TargetType` property. In XAML, the `TargetType` property is set to "Entry" in each of the three `Trigger` definitions. In code, however, `typeof(Entry)` must be passed as an argument to the `Trigger` constructor. If you check the documentation of the `Trigger` class, you'll find that the `TargetType` property is get-only. The XAML parser uses the `TargetType` attribute setting to instantiate the `Trigger` object.

The `Style` class also defines a `Triggers` property of type `IList<TriggerBase>`, which means that you can use a `Style` to share `Trigger` objects among multiple elements. The **StyledTriggers** program shows how. Notice that both the `Style` and the `Trigger` tags contain a `TargetType` property setting. The `Style` contains one `Setter` object and uses `Style.Triggers` property-element tags for the single `Trigger` object, which also contains a `Setter` object:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="StyledTriggers.StyledTriggersPage"
             Padding="20, 50, 120, 0">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Entry">
                <Setter Property="AnchorX" Value="0" />
```

```
                <Style.Triggers>
                    <Trigger TargetType="Entry" Property="IsFocused" Value="True">
                        <Setter Property="Scale" Value="1.5" />
                    </Trigger>
                </Style.Triggers>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Spacing="20">

        <Entry Placeholder="enter name" />

        <Entry Placeholder="enter address" />

        <Entry Placeholder="enter city and state" />

    </StackLayout>
</ContentPage>
```

Because the `Style` has no dictionary key, it is an implicit style that is automatically applied to all the elements of type `Entry`. The individual `Entry` elements need only contain what is unique to each element.

Perhaps after experimenting with the **EntryPop** program (or the two variations), you decide that you don't want the `Scale` property to simply "pop" to a value of 1.5. You want an animation. You want it to "swell" in size when it gains input focus, and to be animated back down to normal when it loses input focus.

That, too, is possible.

Trigger actions and animations

Although some triggers can be realized entirely in XAML, others require a little support from code. As you know, Xamarin.Forms has no direct support for implementing animations in XAML, so if you want to use a trigger to animate an element, you'll need some code.

There are a couple of ways to invoke an animation from XAML. The most obvious way is to use `EventTrigger`, which defines two properties:

- `Event` of type `string`.
- `Actions` of type `IList<TriggerAction>`.

When the element on which you've attached the trigger fires that particular event, the `EventTrigger` invokes all the `TriggerAction` objects in the `Actions` collection.

For example, `VisualElement` defines two events related to input focus: `Focused` and `Unfocused`. You can set those event names to the `Event` property of two different `EventTrigger` objects. When

the element fires the event, the objects of type `TriggerAction` are invoked. Your job is to supply a class that derives from `TriggerAction`. This derived class overrides a method named `Invoke` to respond to the event.

Xamarin.Forms defines both a `TriggerAction` class and a `TriggerAction<T>` class, but both classes are abstract. Generally you'll derive from `TriggerAction<T>` and set the type parameter to the most generalized class the trigger action can support.

For example, suppose you want to derive from `TriggerAction<T>` for a class that calls `ScaleTo` to animate the `Scale` property. Set the type parameter to `VisualElement` because that's the class that is referenced by the `ScaleTo` extension method. An object of that type is also passed to `Invoke`.

By convention, a class that derives from `TriggerAction` has an `Action` suffix in its name. Such a class can be as simple as this:

```
public class ScaleAction : TriggerAction<VisualElement>
{
    protected override void Invoke(VisualElement visual)
    {
        visual.ScaleTo(1.5);
    }
}
```

When you include this class in an `EventTrigger` that is attached to an `Entry` view, the particular `Entry` object is passed as an argument to the `Invoke` method, which animates that `Entry` object using `ScaleTo`. The `Entry` expands to 150 percent of its original size in a default duration of a quarter second.

Of course, you probably don't want to make the class that specific. That simple `ScaleAction` class would work fine for the `Focused` event, but you would need a different one for the `Unfocused` event to animate the `Scale` property back down to 1.

Your `Action<T>` derivative can include properties to make the class very generalized. You can even make the `ScaleAction` class so generalized that it essentially becomes a wrapper for the `ScaleTo` method. Here's the version of `ScaleAction` in the **Xamarin.FormsBook.Toolkit** library:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class ScaleAction : TriggerAction<VisualElement>
    {
        public ScaleAction()
        {
            // Set defaults.
            Anchor = new Point (0.5, 0.5);
            Scale = 1;
            Length = 250;
            Easing = Easing.Linear;
        }

        public Point Anchor { set; get; }
```

```
        public double Scale { set; get; }

        public int Length { set; get; }

        [TypeConverter(typeof(EasingConverter))]
        public Easing Easing { set; get; }

        protected override void Invoke(VisualElement visual)
        {
            visual.AnchorX = Anchor.X;
            visual.AnchorY = Anchor.Y;
            visual.ScaleTo(Scale, (uint)Length, Easing);
        }
    }
}
```

You might wonder whether you should back these properties with bindable properties so that they can be targets of data bindings. You can't do that, however, because `TriggerAction` derives from `Object` rather than `BindableObject`. Keep the properties simple.

Notice the `TypeConverter` attribute on the `Easing` property. This `Easing` property will probably be set in XAML, but the XAML parser doesn't know how to convert text strings like "SpringIn" and "SinOut" to objects of type `Easing`. The following custom type converter (also in **Xamarin.Forms-Book.Toolkit**) assists the XAML parser in converting text strings into `Easing` objects:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class EasingConverter : TypeConverter
    {
        public override bool CanConvertFrom(Type sourceType)
        {
            if (sourceType == null)
                throw new ArgumentNullException("EasingConverter.CanConvertFrom: sourceType");

            return (sourceType == typeof(string));
        }

        public override object ConvertFrom(CultureInfo culture, object value)
        {
            if (value == null || !(value is string))
                return null;

            string name = ((string)value).Trim();

            if (name.StartsWith("Easing"))
            {
                name = name.Substring(7);
            }

            FieldInfo field = typeof(Easing).GetRuntimeField(name);

            if (field != null && field.IsStatic)
```

```
            {
                return (Easing)field.GetValue(null);
            }

            throw new InvalidOperationException(
                String.Format("Cannot convert \"{0}\" into Xamarin.Forms.Easing", value));
        }
    }
}
```

The **EntrySwell** program defines an implicit `Style` for `Entry` in its `Resources` dictionary. That `Style` has two `EventTrigger` objects in its `Triggers` collection, one for `Focused` and the other for `Unfocused`. Both invoke a `ScaleAction` but with different property settings:

```
<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="EntrySwell.EntrySwellPage"
             Padding="20, 50, 120, 0">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Entry">
                <Style.Triggers>
                    <EventTrigger Event="Focused">
                        <toolkit:ScaleAction Anchor="0, 0.5"
                                             Scale="1.5"
                                             Easing="SpringOut" />
                    </EventTrigger>

                    <EventTrigger Event="Unfocused">
                        <toolkit:ScaleAction Anchor="0, 0.5"
                                             Scale="1" />
                    </EventTrigger>
                </Style.Triggers>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Spacing="20">
        <Entry Placeholder="enter name" />

        <Entry Placeholder="enter address" />

        <Entry Placeholder="enter city and state" />
    </StackLayout>
</ContentPage>
```

Notice that `EventTrigger` does not require the `TargetType` property. The only constructor that `EventTrigger` defines has no parameters.

As each `Entry` gets input focus, you'll see it grow larger and then briefly overshoot the 1.5 `Scale`

value. That's the effect of the `SpringOut` easing function.

What if you wanted to use a custom easing function? You would need to define such an easing function in code, of course, and you can do that in the code-behind file. But how would you reference that easing function in XAML? Here's how:

First, remove the `ResourceDictionary` tags from the XAML file. Those tags instantiate the `ResourceDictionary` and set it to the `Resources` property.

Second, in the constructor of the code-behind file, instantiate the `ResourceDictionary` and set it to the `Resources` property. Do this before `InitializeComponent` so that it exists when the XAML file is parsed:

```
Resources = new ResourceDictionary();
InitializeComponent();
```

Third, between those two statements, add an `Easing` object with a custom easing function to the `Resources` dictionary:

```
Resources = new ResourceDictionary();
Resources.Add("customEase", new Easing(t => -6 * t * t + 7 * t));
InitializeComponent();
```

This quadratic formula maps 0 to 0 and 1 to 1, but 0.5 to 2, so it will be obvious if this easing function is correctly used by the animation.

Finally, reference that dictionary entry using `StaticResource` in the `EventTrigger` definition:

```
<EventTrigger Event="Focused">
    <toolkit:ScaleAction Anchor="0, 0.5"
                         Scale="1.5"
                         Easing="{StaticResource customEase}" />
</EventTrigger>
```

Because the object in the `Resources` dictionary is of type `Easing`, the XAML parser will assign it directly to the `Easing` property of `ScaleAction` and bypass the `TypeConverter`.

Among the code samples for this chapter is a solution named **CustomEasingSwell** that demonstrates this technique.

Do not use `DynamicResource` to set the custom `Easing` object to the `Easing` property, perhaps in hopes of defining the easing function in code at a later time. `DynamicResource` requires the target property to be backed by a bindable property; `StaticResource` does not.

You've seen how you can use `Trigger` to set a property in response to a property change, and `EventTrigger` to invoke a `TriggerAction` object in response to an event firing.

But what if you wanted to invoke a `TriggerAction` in response to a property change? Perhaps you want to invoke an animation from XAML but there is no appropriate event for an `EventTrigger`.

There is a second way to invoke a `TriggerAction` derivative that involves the regular `Trigger`

class rather than `EventTrigger`. If you look at the documentation of `TriggerBase` (the class from which all the other trigger classes derive), you'll see the following two properties:

- `EnterActions` of type `IList<TriggerAction>`
- `ExitActions` of type `IList<TriggerAction>`

When used with `Trigger`, all the `TriggerAction` objects in the `EnterActions` collection are invoked when the `Trigger` condition becomes true, and all the objects in the `ExitActions` collection are invoked when the condition becomes false again.

The **EnterExitSwell** program demonstrates this technique. It uses `Trigger` to monitor the `IsFocused` property and invokes two instances of `ScaleAction` to increase the size of the `Entry` when `IsFocused` becomes `True` and to decrease the size of the `Entry` when `IsFocused` stops being `True`:

```
<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="EnterExitSwell.EnterExitSwellPage"
             Padding="20, 50, 120, 0">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Entry">
                <Style.Triggers>
                    <Trigger TargetType="Entry" Property="IsFocused" Value="True">
                        <Trigger.EnterActions>
                            <toolkit:ScaleAction Anchor="0, 0.5"
                                                 Scale="1.5"
                                                 Easing="SpringOut" />
                        </Trigger.EnterActions>

                        <Trigger.ExitActions>
                            <toolkit:ScaleAction Anchor="0, 0.5"
                                                 Scale="1" />
                        </Trigger.ExitActions>
                    </Trigger>
                </Style.Triggers>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Spacing="20">
        <Entry Placeholder="enter name" />

        <Entry Placeholder="enter address" />

        <Entry Placeholder="enter city and state" />
    </StackLayout>
</ContentPage>
```

In summary, you can invoke a class derived from `TriggerAction<T>` either with a change in a property by using `Trigger` or with an event firing by using `EventTrigger`.

But don't use `EnterActions` and `ExitActions` with `EventTrigger`. `EventTrigger` invokes only the `TriggerAction` objects in its `Actions` collection.

More event triggers

The previous chapter on animation showed several examples of a `Button` that rotated or scaled itself when it was clicked. While most of those animation examples were taken to extremes for purposes of making amusing demonstrations, it's not unreasonable for a `Button` to respond to a click with a little animation. This is a perfect job for `EventTrigger`.

Here's another `TriggerAction` derivative. It's similar to `ScaleAction` but includes two calls to `ScaleTo` rather than just one and hence is named `ScaleUpAndDownAction`:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class ScaleUpAndDownAction : TriggerAction<VisualElement>
    {
        public ScaleUpAndDownAction()
        {
            Anchor = new Point(0.5, 0.5);
            Scale = 2;
            Length = 500;
        }

        public Point Anchor { set; get; }

        public double Scale { set; get; }

        public int Length { set; get; }

        protected override async void Invoke(VisualElement visual)
        {
            visual.AnchorX = Anchor.X;
            visual.AnchorY = Anchor.Y;
            await visual.ScaleTo(Scale, (uint)Length / 2, Easing.SinOut);
            await visual.ScaleTo(1, (uint)Length / 2, Easing.SinIn);
        }
    }
}
```

This class hard-codes the `Easing` functions to keep the code simple.

The **ButtonGrowth** program defines an intrinsic `Style` that sets three `Button` properties and includes an `EventTrigger` that invokes `ScaleUpAndDownAction` with default parameters in response to the `Clicked` event:

```
<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="ButtonGrowth.ButtonGrowthPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Button">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions"   Value="CenterAndExpand" />
                <Setter Property="FontSize"          Value="Large" />

                <Style.Triggers>
                    <EventTrigger Event="Clicked">
                        <toolkit:ScaleUpAndDownAction />
                    </EventTrigger>
                </Style.Triggers>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Button Text="Button #1" />
        <Button Text="Button #2" />
        <Button Text="Button #3" />
    </StackLayout>
</ContentPage>
```

Here are three buttons as they've grown in size in response to clicks:

Would it have been possible to use two instances of `ScaleAction` here instead of `ScaleUpAndDownAction`—one instance that scaled the `Button` up in size and the other that scaled it down? No.

We're only dealing with one event—the `Clicked` event—and everything has to be invoked when that event is fired. An `EventTrigger` can certainly invoke multiple actions, but these actions occur simultaneously. Two `ScaleAction` instances running simultaneously would battle each other.

However, there is a solution. Here's a `DelayedScaleAction` class that derives from `ScaleAction` but includes a `Task.Delay` call prior to the `ScaleTo` call:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class DelayedScaleAction : ScaleAction
    {
        public DelayedScaleAction() : base()
        {
            // Set defaults.
            Delay = 0;
        }

        public int Delay { set; get; }

        async protected override void Invoke(VisualElement visual)
        {
            visual.AnchorX = Anchor.X;
            visual.AnchorY = Anchor.Y;
            await Task.Delay(Delay);
            await visual.ScaleTo(Scale, (uint)Length, Easing);
        }
    }
}
```

You can now modify the **ButtonGrowth** XAML file to include two `DelayedScaleAction` objects triggered by the `Clicked` event. These are both invoked simultaneously, but the second has its `Delay` property set to the same value as the `Length` property of the first, so the first `ScaleTo` ends as the second `ScaleTo` begins:

```
<Style TargetType="Button">
    …
    <Style.Triggers>
        <EventTrigger Event="Clicked">
            <toolkit:DelayedScaleAction Scale="2"
                                        Length="250"
                                        Easing="SinOut" />

            <toolkit:DelayedScaleAction Delay="250"
                                        Scale="1"
                                        Length="250"
                                        Easing="SinIn" />
        </EventTrigger>
    </Style.Triggers>
</Style>
```

`DelayedScaleAction` is a little more difficult to use than `ScaleUpAndDownAction`, but it's more

flexible, and you can also define classes named `DelayedTranslateAction` and `DelayedRotateAction` to add to the mix.

In the previous chapter you saw a `Button` derivative named `JiggleButton` that runs a brief animation when the `Button` is clicked. This is a type of animation that you can alternatively implement using a `TriggerAction`. The advantage is that you can use it with the normal `Button` class, and potentially separate the effect from a particular type of view and a particular event so it could be used with other views and other events.

Here's a `TriggerAction` derivative that implements the same type of animation as `JiggleButton` but with three properties to make it more flexible. To more clearly distinguish it from the earlier code, the name of this class is `ShiverAction`:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class ShiverAction : TriggerAction<VisualElement>
    {
        public ShiverAction()
        {
            Length = 1000;
            Angle = 15;
            Vibrations = 10;
        }

        public int Length { set; get; }

        public double Angle { set; get; }

        public int Vibrations { set; get; }

        protected override void Invoke(VisualElement visual)
        {
            visual.Rotation = 0;
            visual.AnchorX = 0.5;
            visual.AnchorY = 0.5;
            visual.RotateTo(Angle, (uint)Length,
                new Easing(t => Math.Sin(Math.PI * t) *
                                Math.Sin(Math.PI * 2 * Vibrations * t)));
        }
    }
}
```

Notice that `Invoke` initializes the `Rotation` property of the target visual element to zero. This is to avoid problems when the `Button` is pressed twice in succession and `Invoke` is called while the previous animation is still running.

The XAML file of the **ShiverButtonDemo** program defines an implicit `Style` that includes the `ShiverAction` with rather extreme values set to its three properties:

```
<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="ShiverButtonDemo.ShiverButtonDemoPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Button">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions"   Value="CenterAndExpand" />
                <Setter Property="FontSize"          Value="Large" />

                <Style.Triggers>
                    <EventTrigger Event="Clicked">
                        <toolkit:ShiverAction Length="3000"
                                              Angle="45"
                                              Vibrations="25" />
                    </EventTrigger>
                </Style.Triggers>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Button Text="Button #1" />
        <Button Text="Button #2" />
        <Button Text="Button #3" />
    </StackLayout>
</ContentPage>
```

The three `Button` elements share the same instance of `ShiverAction`, but each call to the `Invoke` method is for a specific `Button` object. Each button's shivering is independent of the others.

But what if you want to use `ShiverAction` to respond to `Tapped` events on an element rather than `Clicked` events on a `Button`—for example, to vibrate a `Frame` with some content, or an `Image`? The `Tapped` event is only defined by `TapGestureRecognizer`, but you can't attach an `EventTrigger` to a `TapGestureRecognizer` because `TapGestureRecognizer` does not have a `Triggers` collection. Nor can you attach an `EventTrigger` to a `View` object and specify the `Tapped` event. That `Tapped` event won't be found on the `View` object.

The solution is to use a behavior, as will be demonstrated later in this chapter.

It's also possible to use `EventTrigger` objects for entry validation. Here's a `TriggerAction` derivative named `NumericValidationAction` with a generic argument of `Entry`, so it applies only to `Entry` views. When `Invoke` is called, the argument is an `Entry` object, so it can access properties specific to `Entry`, in this case `Text` and `TextColor`. The method checks whether the `Text` property of the `Entry` can be parsed into a valid `double`. If not, the text is colored red to alert the user:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class NumericValidationAction : TriggerAction<Entry>
    {
        protected override void Invoke(Entry entry)
        {
```

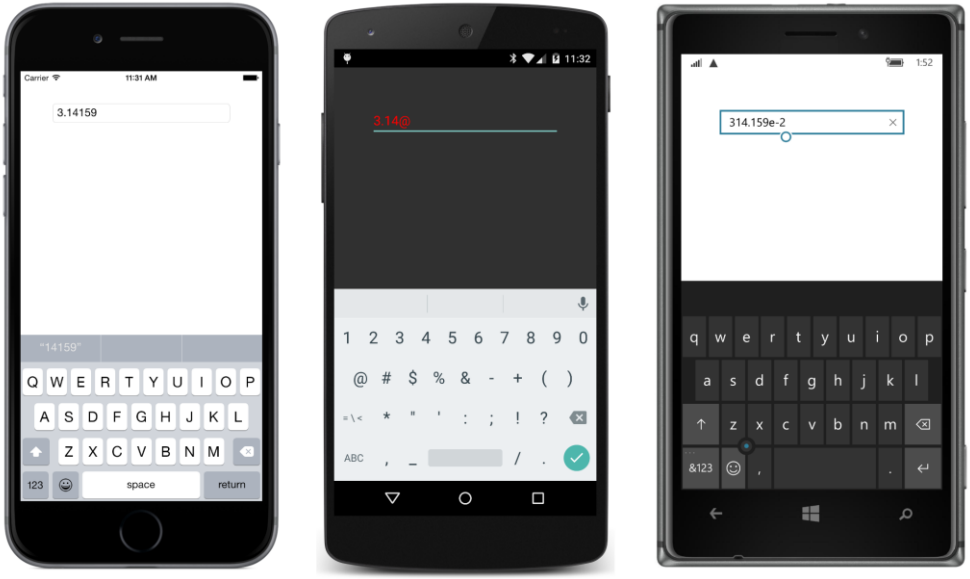
```
            double result;
            bool isValid = Double.TryParse(entry.Text, out result);
            entry.TextColor = isValid ? Color.Default : Color.Red;
        }
    }
}
```

You can attach this code to an `Entry` with an `EventTrigger` for the `TextChanged` event, as demonstrated in the **TriggerEntryValidation** program:

```
<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="TriggerEntryValidation.TriggerEntryValidationPage"
             Padding="50">

    <StackLayout>
        <Entry Placeholder="Enter a System.Double">
            <Entry.Triggers>
                <EventTrigger Event="TextChanged">
                    <toolkit:NumericValidationAction />
                </EventTrigger>
            </Entry.Triggers>
        </Entry>
    </StackLayout>
</ContentPage>
```

Whenever the text changes, the `Invoke` method of `NumericValidationAction` is called.

The screenshot shows valid numeric entries for the iOS and Windows 10 Mobile devices, but an invalid number in the Android device:



Unfortunately, this doesn't work quite right on the Universal Windows Platform: If an invalid number is typed in the `Entry`, the text turns red only when the `Entry` loses input focus. However, it works fine on the other Windows Runtime platforms (Windows 8.1 and Windows Phone 8.1).

Data triggers

So far, you've only seen triggers that operate within the context of a particular object. A `Trigger` responds to a change in a property of an object by changing another property of that same object, or by invoking an `Action` that affects that object. The `EventTrigger` similarly responds to an event fired by an object to invoke an `Action` on that same object.

The `DataTrigger` is different. Like the other `TriggerBase` derivatives, the `DataTrigger` is attached to a visual element or defined in a `Style`. However, the `DataTrigger` can detect a property change in *another* object through a data binding, and either change a property in the object that it's attached to or (by using the `EnterActions` and `ExitActions` collection inherited from `TriggerBase`) invoke a `TriggerAction` on that object.

`DataTrigger` defines the following three properties.

- `Binding` of type `BindingBase`.
- `Value` of type `Object`.
- `Setters` of type `IList<Setter>`. This is the content property of `DataTrigger`.

From the perspective of an application program, the `DataTrigger` is very similar to `Trigger` except that the property of `Trigger` named `Property` is replaced with the `Binding` property. Both `Trigger`

and `DataTrigger` require the `TargetType` property to be set.

What is the other object that the `Binding` property of `DataTrigger` references? It can be part of a ViewModel in an MVVM scenario or another element on the page.

You might recall the **SchoolOfFineArt** library from Chapter 19, "Collection views." The `Student` class in that library defines a property named `Sex` of type `string` that is set to either "Male" or "Female". The **GenderColors** program presented below uses that property in conjunction with a `DataTrigger` to set a blue or pink color for the student (regardless how hopelessly old-fashioned that color scheme might seem).

A `ListView` displays all the students of the high school, and a `ViewCell` formats each student to display the student's photo, full name, and the current grade-point average:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:school="clr-namespace:SchoolOfFineArt;assembly=SchoolOfFineArt"
             x:Class="GenderColors.GenderColorsPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <ContentPage.BindingContext>
        <school:SchoolViewModel />
    </ContentPage.BindingContext>

    <StackLayout BindingContext="{Binding StudentBody}">
        <Label Text="{Binding School}"
               FontSize="Large"
               FontAttributes="Bold"
               HorizontalTextAlignment="Center" />

        <ListView ItemsSource="{Binding Students}"
                  VerticalOptions="FillAndExpand">
            <ListView.RowHeight>
                <OnPlatform x:TypeArguments="x:Int32"
                            iOS="70"
                            Android="70"
                            WinPhone="100" />
            </ListView.RowHeight>

            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <Grid Padding="0, 5">
                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="80" />
                                <ColumnDefinition Width="*" />
                            </Grid.ColumnDefinitions>

                            <Image Grid.Column="0"
                                   Source="{Binding PhotoFilename}"
```

```
                                VerticalOptions="Center" />

                        <StackLayout Grid.Column="1"
                                     VerticalOptions="Center">
                            <Label Text="{Binding FullName}"
                                   FontSize="22"
                                   TextColor="Pink">
                                <Label.Triggers>
                                    <DataTrigger TargetType="Label"
                                                 Binding="{Binding Sex}"
                                                 Value="Male">
                                        <Setter Property="TextColor" Value="#8080FF" />
                                    </DataTrigger>
                                </Label.Triggers>
                            </Label>

                            <Label Text="{Binding GradePointAverage,
                                                  StringFormat='G.P.A. = {0:F2}'}"
                                   FontSize="16" />
                        </StackLayout>
                    </Grid>
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</StackLayout>
</ContentPage>
```

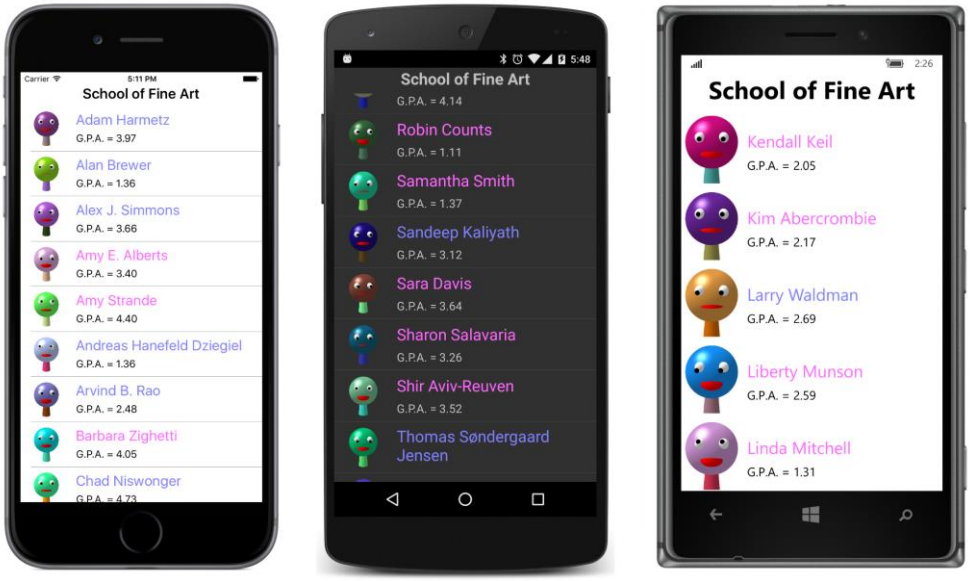
The program uses `ViewCell` rather than `ImageCell`, so it has access to a `Label` onto which it can attach a `DataTrigger`. A trigger cannot be attached directly to a `Cell` or `Cell` derivative because there's no `Triggers` collection defined for these classes.

The `Label` displays the `FullName` property of the `Student` object and the `TextColor` is set to `Pink`. But a `DataTrigger` checks whether the `Sex` property of the `Student` object equals "Male", and if so it uses a `Setter` to set the `TextColor` to a light blue. Here is that `Label` isolated from the rest of the cell:

```
<Label Text="{Binding FullName}"
       FontSize="Large"
       TextColor="Pink">
    <Label.Triggers>
        <DataTrigger TargetType="Label"
                     Binding="{Binding Sex}"
                     Value="Male">
            <Setter Property="TextColor" Value="#8080FF" />
        </DataTrigger>
    </Label.Triggers>
</Label>
```

The `BindingContext` of the `DataTrigger` is the same as the `BindingContext` of the `Label` to which it is attached. That `BindingContext` is a particular `Student` object, so the `Binding` on the `DataTrigger` only needs to specify the `Sex` property.

Here it is in action:



Something quite similar can be done with a data binding directly from the `Sex` property of the `Student` object to the `TextColor` property of the `Label` (or the `ImageCell`), but it would require a binding converter. The `DataTrigger` does the job without any additional code.

However, by itself the `DataTrigger` cannot mimic the **ColorCodedStudents** program in Chapter 19. That program displays a student in red if that student's grade-point average falls dangerously below a 2.0 criterion. The less-than numeric comparison requires some code. This too is a job for a behavior, and once you learn about behaviors later in this chapter, you should be able to code something like this yourself.

It's also possible for `DataTrigger` to reference another element on the page to monitor a property of that element.

For example, one of the classic tasks in graphical environments is to disable a button if nothing has been typed into a text-entry field. Perhaps the text-entry field is a filename, and the button executes some code to load or save that file. It doesn't make any sense for the button to be enabled if the filename is blank.

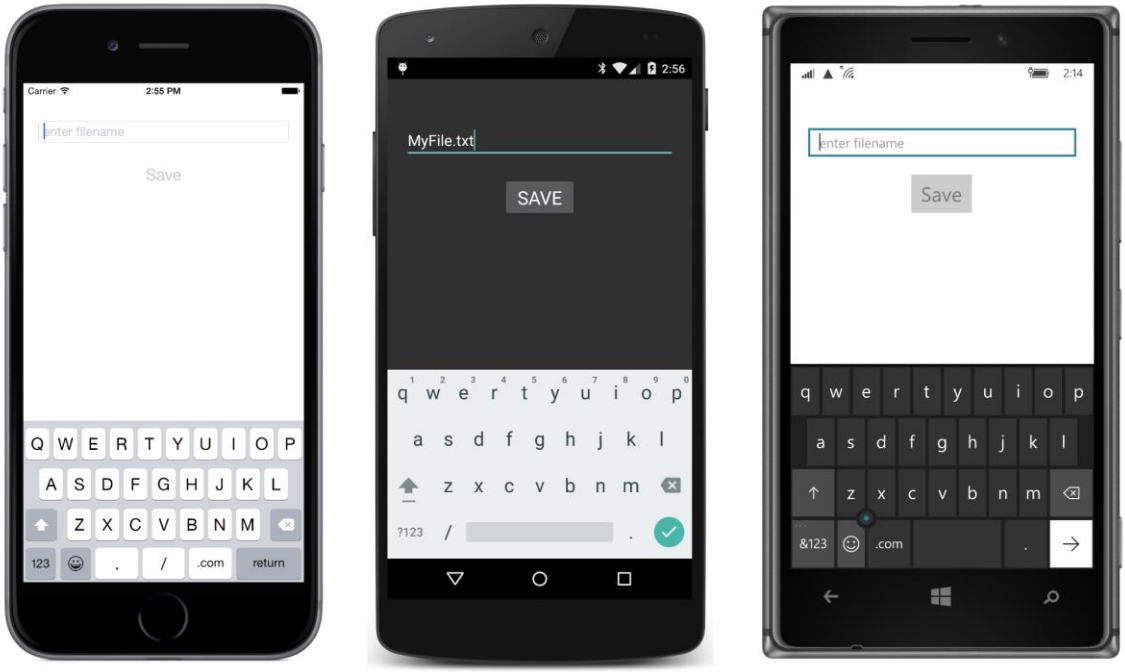
You can do that job entirely in XAML with a `DataTrigger`. Here's the markup in the **ButtonEnabler** project:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ButtonEnabler.ButtonEnablerPage"
             Padding="20, 50">
```

```
    <StackLayout Spacing="20">
        <Entry x:Name="entry"
               Text=""
               Keyboard="Url"
               Placeholder="enter filename" />

        <Button Text="Save"
                FontSize="Large"
                HorizontalOptions="Center">
            <Button.Triggers>
                <DataTrigger TargetType="Button"
                             Binding="{Binding Source={x:Reference entry},
                                               Path=Text.Length}"
                             Value="0">
                    <Setter Property="IsEnabled" Value="False" />
                </DataTrigger>
            </Button.Triggers>
        </Button>
    </StackLayout>
</ContentPage>
```

The `DataTrigger` on the `Button` sets its `Binding` property with a `Source` referencing the `Entry` element. The `Path` is set to `Text.Length`. The `Text` property of the `Entry` element is of type `string`, and `Length` is a property of `string`, so this binding refers to the number of characters entered in the `Entry` element. The `Value` property of `DataTrigger` is set to zero, so when there are zero characters entered into the `Entry`, the `Setter` property is invoked, which sets the `IsEnabled` property of the `Button` to `False`.

Based on the input in the `Entry` element, the `Button` is disabled on the iPhone and Windows 10 Mobile screens shown here but enabled on the Android screen:



Although this represents a tiny enhancement to the user interface, if you didn't have a `DataTrigger` you'd need to implement this enhancement in code in a `TextChanged` handler of the `Entry`, or you'd need to write a binding converter for a `Binding` between the `IsEnabled` property of the `Button` and the `Text.Length` property of the `Entry`.

The XAML file in **ButtonEnabler** contains a crucial property setting that you might not have noticed:

```
<Entry … Text="" … />
```

When an `Entry` is first created, the `Text` property is not an empty string but `null`, which means that the data binding in the `DataTrigger` is trying to reference the `Length` property of a `null` string object, and it will fail. Because the binding fails, the `Button` will be enabled when the program first starts up. It only becomes disabled after the user types a character and backspaces.

Initializing the `Text` property to an empty string has no other effect but to allow the `DataTrigger` to work when the program starts up.

Combining conditions in the MultiTrigger

Both the `Trigger` and the `DataTrigger` effectively monitor a property to determine if it's equal to a particular value. That's called the trigger's *condition*, and if the condition is true, then a collection of `Setter` objects are invoked.

As a programmer, you might begin wondering whether you can have multiple conditions in a trigger. But once you start talking about multiple conditions, you need to determine whether you want to combine conditions with a logical OR operation or an AND operation—whether the trigger is invoked if *any* of the conditions are true, or if it requires that *all* the conditions be true.

If you want a trigger invoked when multiple conditions are all true—the logical AND case—that's the last of the four classes that derive from `TriggerBase`. The `MultiTrigger` defines two collection properties:

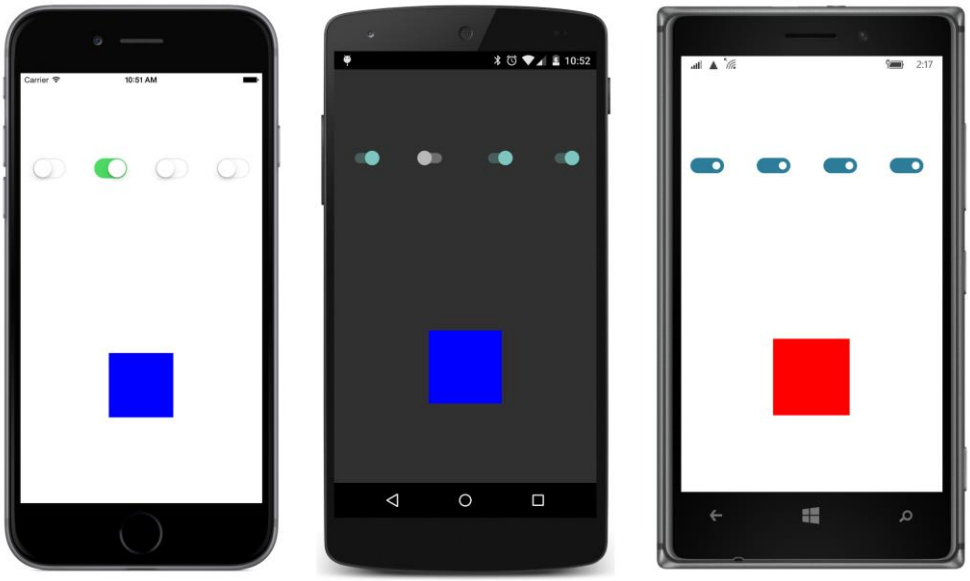
- `Conditions` of type `IList<Condition>`
- `Setters` of type `IList<Setter>`

`Condition` is an abstract class and has two descendent classes:

- `PropertyCondition`, which has `Property` and `Value` properties like `Trigger`
- `BindingCondition`, which has `Binding` and `Value` properties like `DataTrigger`

You can mix multiple `PropertyCondition` and `BindingCondition` objects in the same `Conditions` collection of the `MultiTrigger`. When all the conditions are true, all the `Setter` objects in the `Setters` collection are applied.

Let's look at a simple example: In the **AndConditions** program, four `Switch` elements share the page with a blue `BoxView`. When all the `Switch` elements are turned on, the `BoxView` turns red:



The XAML file shows how this is done. The `Triggers` collection of the `BoxView` contains a `Multi-`

`Trigger`. The `TargetType` property is required. The `Conditions` collection contains four `BindingCondition` objects, each of which references the `IsToggled` property of one of the four `Switch` elements and checks for a `True` value. If all the conditions are true, the `MultiTrigger` sets the `Color` property of the `BoxView` to `Red`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AndConditions.AndConditionsPage">
    <StackLayout>
        <Grid VerticalOptions="CenterAndExpand">
            <Switch x:Name="switch1" Grid.Column="0"
                    HorizontalOptions="Center" />

            <Switch x:Name="switch2" Grid.Column="1"
                    HorizontalOptions="Center" />

            <Switch x:Name="switch3" Grid.Column="2"
                    HorizontalOptions="Center" />

            <Switch x:Name="switch4" Grid.Column="3"
                    HorizontalOptions="Center" />
        </Grid>

        <BoxView WidthRequest="100"
                 HeightRequest="100"
                 VerticalOptions="CenterAndExpand"
                 HorizontalOptions="Center"
                 Color="Blue">
            <BoxView.Triggers>
                <MultiTrigger TargetType="BoxView">
                    <MultiTrigger.Conditions>
                        <BindingCondition Binding="{Binding Source={x:Reference switch1},
                                                            Path=IsToggled}"
                                          Value="True" />

                        <BindingCondition Binding="{Binding Source={x:Reference switch2},
                                                            Path=IsToggled}"
                                          Value="True" />

                        <BindingCondition Binding="{Binding Source={x:Reference switch3},
                                                            Path=IsToggled}"
                                          Value="True" />

                        <BindingCondition Binding="{Binding Source={x:Reference switch4},
                                                            Path=IsToggled}"
                                          Value="True" />
                    </MultiTrigger.Conditions>

                    <Setter Property="Color" Value="Red" />
                </MultiTrigger>
            </BoxView.Triggers>
        </BoxView>
    </StackLayout>
</ContentPage>
```

That's the AND combination. What about an OR combination?

Because the `Triggers` collection can accommodate multiple `DataTrigger` objects, you might think that this would work:

```
<BoxView WidthRequest="100"
            HeightRequest="100"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center"
            Color="Blue">
    <BoxView.Triggers>
        <DataTrigger TargetType="BoxView"
                     Binding="{Binding Source={x:Reference switch1}, Path=IsToggled}"
                     Value="True">
            <Setter Property="Color" Value="Red" />
        </DataTrigger>

        <DataTrigger TargetType="BoxView"
                     Binding="{Binding Source={x:Reference switch2}, Path=IsToggled}"
                     Value="True">
            <Setter Property="Color" Value="Red" />
        </DataTrigger>

        <DataTrigger TargetType="BoxView"
                     Binding="{Binding Source={x:Reference switch3}, Path=IsToggled}"
                     Value="True">
            <Setter Property="Color" Value="Red" />
        </DataTrigger>

        <DataTrigger TargetType="BoxView"
                     Binding="{Binding Source={x:Reference switch4}, Path=IsToggled}"
                     Value="True">
            <Setter Property="Color" Value="Red" />
        </DataTrigger>
    </BoxView.Triggers>
</BoxView>
```

And if you try it, you might find that it does seem to work at first. But as you further experiment with turning various `Switch` elements on and off, you'll find that it really does *not* work.

Whether it should work or shouldn't work is open to debate. The four `DataTrigger` objects all target the same `Color` property, and if each `DataTrigger` works independently to determine whether that `Setter` should be applied or not, then this really shouldn't work as a logical OR.

However, keep in mind Victorian mathematician Augustus De Morgan's laws of logic, which state (using C# syntax for AND, OR, logical negation, and equivalence):

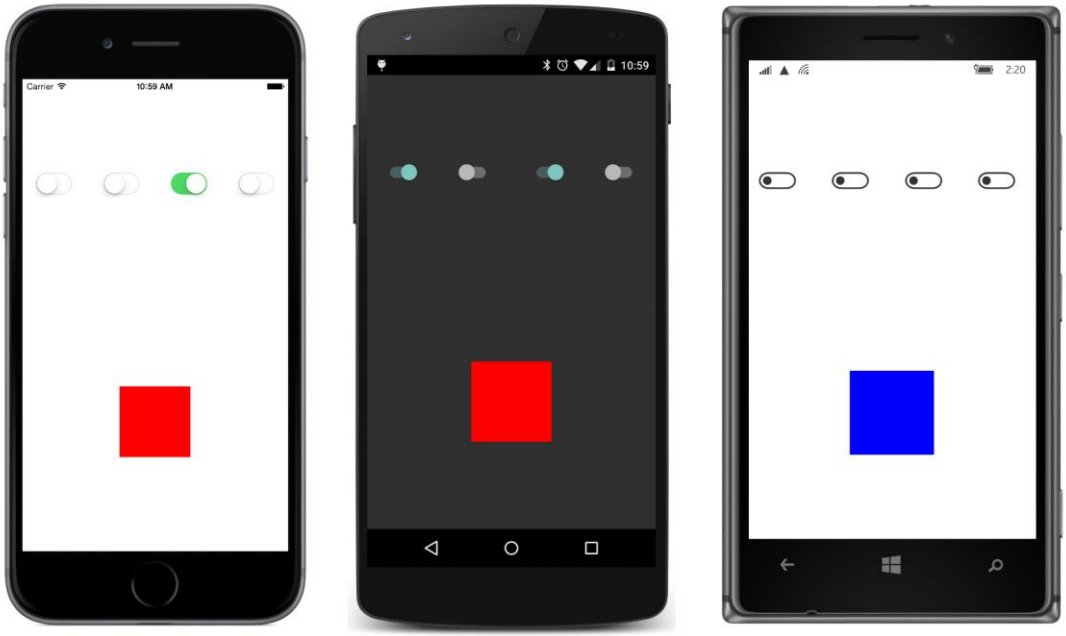
$$A \mid B \;==\; !(!A \;\&\; !B)$$

$$A \;\&\; B \;==\; !(!A \mid !B)$$

This means you can use `MultiTrigger` to perform a logical OR as the **OrConditions** program demonstrates:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="OrConditions.OrConditionsPage">
    <StackLayout>
        <Grid VerticalOptions="CenterAndExpand">
            <Switch x:Name="switch1" Grid.Column="0"
                    HorizontalOptions="Center" />

            <Switch x:Name="switch2" Grid.Column="1"
                    HorizontalOptions="Center" />

            <Switch x:Name="switch3" Grid.Column="2"
                    HorizontalOptions="Center" />

            <Switch x:Name="switch4" Grid.Column="3"
                    HorizontalOptions="Center" />
        </Grid>

        <BoxView WidthRequest="100"
                 HeightRequest="100"
                 VerticalOptions="CenterAndExpand"
                 HorizontalOptions="Center"
                 Color="Red">
            <BoxView.Triggers>
                <MultiTrigger TargetType="BoxView">
                    <MultiTrigger.Conditions>
                        <BindingCondition Binding="{Binding Source={x:Reference switch1},
                                                            Path=IsToggled}"
                                          Value="False" />

                        <BindingCondition Binding="{Binding Source={x:Reference switch2},
                                                            Path=IsToggled}"
                                          Value="False" />

                        <BindingCondition Binding="{Binding Source={x:Reference switch3},
                                                            Path=IsToggled}"
                                          Value="False" />

                        <BindingCondition Binding="{Binding Source={x:Reference switch4},
                                                            Path=IsToggled}"
                                          Value="False" />
                    </MultiTrigger.Conditions>

                    <Setter Property="Color" Value="Blue" />
                </MultiTrigger>
            </BoxView.Triggers>
        </BoxView>
    </StackLayout>
</ContentPage>
```

It's the same as **AndConditions** except all the logic is flipped around. All the `BindingCondition` objects check for a `False` value of the `IsToggled` property, and if all the conditions are satisfied, the normally red `BoxView` is colored blue:



Here's another way you can think of these two programs: In **AndConditions**, the `BoxView` is always blue unless all the `Switch` elements are toggled on. In **OrConditions**, the `BoxView` is always red unless all the `Switch` elements are off.

Suppose you have a scenario involving two `Entry` fields and a `Button`. You want to enable the `Button` if either `Entry` field contains some text.

Flip the logic upside down: You really want to disable the `Button` if both `Entry` fields contain no text. That's fairly easy:

```
<StackLayout>
    <Entry x:Name="entry1"
           Text="" />

    <Entry x:Name="entry2"
           Text="" />

    <Button Text="Send">
        <Button.Triggers>
            <MultiTrigger TargetType="Button">
                <MultiTrigger.Conditions>
                    <BindingCondition Binding="{Binding Source={x:Reference entry1},
                                                        Path=Text.Length}"
                                      Value="0" />
                    <BindingCondition Binding="{Binding Source={x:Reference entry2},
                                                        Path=Text.Length}"
                                      Value="0" />
                </MultiTrigger.Conditions>
```

```
            <Setter Property="IsEnabled" Value="False" />
        </MultiTrigger>
    </Button.Triggers>
</Button>
</StackLayout>
```

Notice that the two `Entry` fields initialize the `Text` property to an empty string so that the property isn't equal to `null`. If both `Text` properties have a length of zero, then the two `BindingConditions` are satisfied and the `IsEnabled` property of the `Button` is set to `False`.

However, it's not so easy to adapt this to enable the `Button` only if *both* `Entry` views have some text. If you try to flip the logic around, you must change the `BindingCondition` objects so that they check for a `Text` property with a length *not* equal to zero, and that's not an option.

To help realize the logic, you can use some intermediary invisible `Switch` elements:

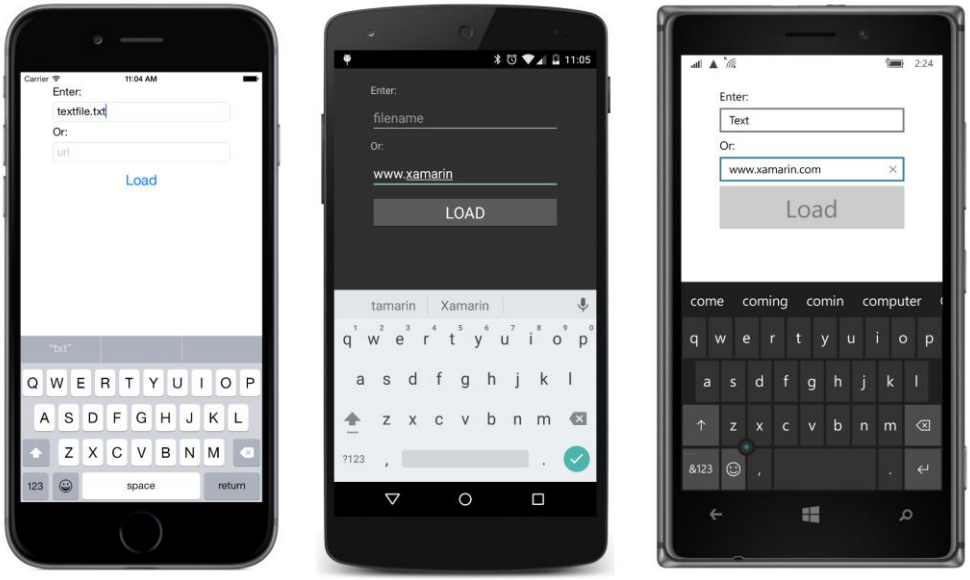
```
<StackLayout>
    <Entry x:Name="entry1"
           Text="" />

    <Switch x:Name="switch1"
            IsVisible="False">
        <Switch.Triggers>
            <DataTrigger TargetType="Switch"
                         Binding="{Binding Source={x:Reference entry1},
                                           Path=Text.Length}"
                         Value="0">
                <Setter Property="IsToggled" Value="True" />
            </DataTrigger>
        </Switch.Triggers>
    </Switch>

    <Entry x:Name="entry2"
            Text="" />

    <Switch x:Name="switch2"
            IsVisible="False">
        <Switch.Triggers>
            <DataTrigger TargetType="Switch"
                         Binding="{Binding Source={x:Reference entry2},
                                           Path=Text.Length}"
                         Value="0">
                <Setter Property="IsToggled" Value="True" />
            </DataTrigger>
        </Switch.Triggers>
    </Switch>

    <Button Text="Send"
            IsEnabled="False">
        <Button.Triggers>
            <MultiTrigger TargetType="Button">
                <MultiTrigger.Conditions>
```

```
                <BindingCondition Binding="{Binding Source={x:Reference switch1},
                                                    Path=IsToggled}"
                                  Value="False" />
                <BindingCondition Binding="{Binding Source={x:Reference switch2},
                                                    Path=IsToggled}"
                                  Value="False" />
            </MultiTrigger.Conditions>

            <Setter Property="IsEnabled" Value="True" />
        </MultiTrigger>
    </Button.Triggers>
</Button>
</StackLayout>
```

Each `Entry` now has a companion `Switch` that uses a `DataTrigger` to set its `IsToggled` property to `True` if the length of the `Text` property of the `Entry` is zero. The two `Switch` elements can then be used in the `MultiTrigger`. If both `Switch` elements have their `IsToggled` properties set to `True`, then both `Entry` fields contain some text, and the `IsEnabled` property of the `Button` can be set to `True`.

If you want to actually combine AND and OR operations, you'll need to engage in some deeper levels of logic.

For example, suppose you have a scenario with two `Entry` views and a `Button`, and the `Button` should be enabled only if either of the two `Entry` views contains some text, but not if both `Entry` views contain some text:



Perhaps (as this screenshot suggests) one of the `Entry` views is for a filename and the other is for a URL, and the program needs one and only one of these two text strings.

What you need is an exclusive-OR (XOR) operation, and it's a combination of AND, OR, and negation operators:

$$A \text{ ^ } B \;==\; (A \mid B) \;\&\; !(A \;\&\; B)$$

This can be done with three `MultiTrigger` objects, two of which are on intermediary invisible `Switch` elements and the final one is on the `Button` itself. Here's the **XorConditions** XAML file with comments describing the various pieces of the logic:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XorConditions.XorConditionsPage"
             Padding="50, 20">

    <StackLayout>
        <Label Text="Enter:" />

        <Entry x:Name="entry1"
               Text=""
               Placeholder="filename" />

        <!-- IsToggled is true if entry1 has no text -->
        <Switch x:Name="switch1"
                IsVisible="False">
            <Switch.Triggers>
                <DataTrigger TargetType="Switch"
                             Binding="{Binding Source={x:Reference entry1},
                                               Path=Text.Length}"
                             Value="0">
                    <Setter Property="IsToggled" Value="True" />
                </DataTrigger>
            </Switch.Triggers>
        </Switch>

        <Label Text="Or:" />

        <Entry x:Name="entry2"
               Text=""
               Placeholder="url" />

        <!-- IsToggled is true if entry2 has no text -->
        <Switch x:Name="switch2"
                IsVisible="False">
            <Switch.Triggers>
                <DataTrigger TargetType="Switch"
                             Binding="{Binding Source={x:Reference entry2},
                                               Path=Text.Length}"
                             Value="0">
                    <Setter Property="IsToggled" Value="True" />
                </DataTrigger>
            </Switch.Triggers>
        </Switch>

        <!-- IsToggled is true if either Entry has some text (OR operation) -->
```

```
<Switch x:Name="switch3"
        IsToggled="True"
        IsVisible="False">
    <Switch.Triggers>
        <MultiTrigger TargetType="Switch">
            <MultiTrigger.Conditions>
                <BindingCondition Binding="{Binding Source={x:Reference switch1},
                                                    Path=IsToggled}"
                                  Value="True" />
                <BindingCondition Binding="{Binding Source={x:Reference switch2},
                                                    Path=IsToggled}"
                                  Value="True" />
            </MultiTrigger.Conditions>

            <Setter Property="IsToggled" Value="False" />
        </MultiTrigger>
    </Switch.Triggers>
</Switch>

<!-- IsToggled is true if both Entry's have some text (AND operation) -->
<Switch x:Name="switch4"
        IsVisible="False">
    <Switch.Triggers>
        <MultiTrigger TargetType="Switch">
            <MultiTrigger.Conditions>
                <BindingCondition Binding="{Binding Source={x:Reference switch1},
                                                    Path=IsToggled}"
                                  Value="False" />
                <BindingCondition Binding="{Binding Source={x:Reference switch2},
                                                    Path=IsToggled}"
                                  Value="False" />
            </MultiTrigger.Conditions>

            <Setter Property="IsToggled" Value="True" />
        </MultiTrigger>
    </Switch.Triggers>
</Switch>

<!-- Button is enabled if either Entry has some text but not both (XOR operation) -->
<Button Text="Load"
        IsEnabled="False"
        FontSize="Large">
    <Button.Triggers>
        <MultiTrigger TargetType="Button">
            <MultiTrigger.Conditions>
                <BindingCondition Binding="{Binding Source={x:Reference switch3},
                                                    Path=IsToggled}"
                                  Value="True" />
                <BindingCondition Binding="{Binding Source={x:Reference switch4},
                                                    Path=IsToggled}"
                                  Value="False" />
            </MultiTrigger.Conditions>

            <Setter Property="IsEnabled" Value="True" />
```

```
                </MultiTrigger>
            </Button.Triggers>
        </Button>
    </StackLayout>
</ContentPage>
```

Of course, once the XAML gets this extravagant, nobody will fault you if you simply decide to enable or disable the `Button` in code!

Behaviors

Triggers and behaviors are generally discussed in tandem because they have some applicational overlap. Sometimes you'll be puzzled whether to use a trigger or behavior because either seems to do the job.

Anything you can do with a trigger you can also do with a behavior. However, a behavior always involves some code, which is a class that derives from `Behavior<T>`. Triggers only involve code if you're writing an `Action<T>` derivative for an `EventTrigger` or for `EnterActions` or `ExitActions` collections of the other triggers.

Obviously, if you can do what you need using one of the triggers without writing any code, then don't use a behavior. But sometimes it's not so clear.

Let's compare a trigger and behavior that do the same job.

The **TriggerEntryValidation** program shown earlier in this chapter uses a class named `NumericEntryAction` that checks whether a number typed into an `Entry` view qualifies as a valid `double` value and colors the text red if it doesn't:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class NumericValidationAction : TriggerAction<Entry>
    {
        protected override void Invoke(Entry entry)
        {
            double result;
            bool isValid = Double.TryParse(entry.Text, out result);
            entry.TextColor = isValid ? Color.Default : Color.Red;
        }
    }
}
```

This is referenced in an `EventTrigger` attached to an `Entry`:

```
<Entry Placeholder="Enter a System.Double">
    <Entry.Triggers>
        <EventTrigger Event="TextChanged">
            <toolkit:NumericValidationAction />
        </EventTrigger>
    </Entry.Triggers>
```

```
</Entry>
```

You can use a behavior for this same job. The first step is to derive a class from `Behavior<T>`. The generic argument is the most generalized base class that the behavior can handle. In this example, that's an `Entry` view. Then, override two virtual methods, named `OnAttachedTo` and `OnDetachingFrom`. The `OnAttachedTo` method is called when the behavior is attached to a particular visual object, and it gives your behavior a chance to initialize itself. Often this involves attaching some event handlers to the object. The `OnDetachingFrom` method is called when the behavior is removed from the visual object. Even if this occurs only when the program is terminating, you should undo anything the `OnAttachedTo` method does.

Here's the `NumericValidationBehavior` class:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class NumericValidationBehavior : Behavior<Entry>
    {
        protected override void OnAttachedTo(Entry entry)
        {
            base.OnAttachedTo(entry);
            entry.TextChanged += OnEntryTextChanged;
        }

        protected override void OnDetachingFrom(Entry entry)
        {
            base.OnDetachingFrom(entry);
            entry.TextChanged -= OnEntryTextChanged;
        }

        void OnEntryTextChanged(object sender, TextChangedEventArgs args)
        {
            double result;
            bool isValid = Double.TryParse(args.NewTextValue, out result);
            ((Entry)sender).TextColor = isValid ? Color.Default : Color.Red;
        }
    }
}
```

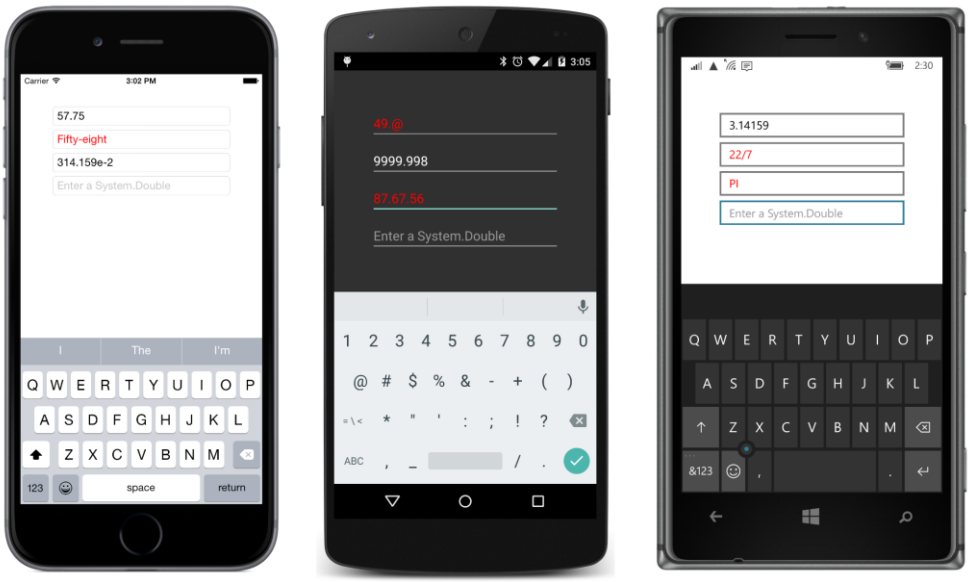
The `OnAttachedTo` method attaches a handler for the `TextChanged` event of the `Entry`, and the `OnDetachingFrom` method detaches that handler. The handler itself does the same job as the `Invoke` method in `NumericValidationAction`.

Because the `NumericValidationBehavior` class installs the handler for the `TextChanged` event, the behavior can be used without specifying anything beyond the class name. Here's the XAML file for the **BehaviorEntryValidation** program, which differs from the earlier program that used an `EventTrigger` by specifying the behavior in an implicit style that is applied to four `Entry` views:

```
<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="BehaviorEntryValidation.BehaviorEntryValidationPage"
             Padding="50">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Entry">
                <Style.Behaviors>
                    <toolkit:NumericValidationBehavior />
                </Style.Behaviors>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Entry Placeholder="Enter a System.Double" />

        <Entry Placeholder="Enter a System.Double" />

        <Entry Placeholder="Enter a System.Double" />

        <Entry Placeholder="Enter a System.Double" />
    </StackLayout>
</ContentPage>
```

This `Style` object is shared among the four `Entry` views, so only a single `NumericValidationBehavior` object is instantiated. As this single object is attached to each of the four `Entry` views, it attaches a `TextChanged` handler on each one so that the single `NumericValidationBehavior` object operates independently on the four views:



In this particular example, the `TriggerAction` would be preferred over the `Behavior` because it's

less code and the code doesn't refer to a particular event, so it's more generalized.

But a behavior can be as generalized or as specific as you want, and behaviors also have the ability to participate more fully within the XAML file through data bindings.

Behaviors with properties

The `Behavior<T>` class derives from the `Behavior` class, which derives from `BindableObject`. This suggests that your `Behavior<T>` derivative can define its own bindable properties.

Earlier you saw some `Action<T>` derivatives such as `ScaleAction` and `ShiverAction` that defined some properties to give them more flexibility. But a `Behavior<T>` derivative can define bindable properties that can serve as source properties for data bindings. This means that you don't have to hard-code the behavior to modify a particular property, such as setting the `TextColor` property of an `Entry` to `Red`. You can instead decide later how you want the behavior to affect the user interface, and implement that right in the XAML file. This gives the behavior a greater amount of flexibility and allows the XAML to play a greater role in the aspect of the behavior that pertains to the user interface.

Here is a class in the **Xamarin.FormsBook.Toolkit** library called `ValidEmailBehavior`, which is similar to `NumericValidationBehavior` except that it uses a regular expression to determine whether the `Text` property of an `Entry` is a valid email address:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class ValidEmailBehavior : Behavior<Entry>
    {
        static readonly BindablePropertyKey IsValidPropertyKey =
            BindableProperty.CreateReadOnly("IsValid",
                                            typeof(bool),
                                            typeof(ValidEmailBehavior),
                                            false);

        public static readonly BindableProperty IsValidProperty =
            IsValidPropertyKey.BindableProperty;

        public bool IsValid
        {
            private set { SetValue(IsValidPropertyKey, value); }
            get { return (bool)GetValue(IsValidProperty); }
        }

        protected override void OnAttachedTo(Entry entry)
        {
            entry.TextChanged += OnEntryTextChanged;
            base.OnAttachedTo(entry);
        }

        protected override void OnDetachingFrom(Entry entry)
        {
            entry.TextChanged -= OnEntryTextChanged;
            base.OnDetachingFrom(entry);
```

```
        }

        void OnEntryTextChanged(object sender, TextChangedEventArgs args)
        {
            Entry entry = (Entry)sender;
            IsValid = IsValidEmail(entry.Text);
        }

        bool IsValidEmail(string strIn)
        {
            if (String.IsNullOrEmpty(strIn))
                return false;

            try
            {
                // from https://msdn.microsoft.com/en-us/library/01escwtf(v=vs.110).aspx
                return Regex.IsMatch(strIn,
                      @"^(?(""")(""".+?(?<!\\)""@)|(([0-9a-z]((\.(?!\.))|" +
                      @"[-!#\$%&'\*\+/=\?\^`\{\}\|~\w])*)" +
                      @"(?<=[0-9a-z])@))(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|" +
                      @"(([0-9a-z][-\w]*[0-9a-z]*\.)+[a-z0-9][\-a-z0-9]{0,22}[a-z0-9]))$",
                      RegexOptions.IgnoreCase, TimeSpan.FromMilliseconds(250));
            }
            catch (RegexMatchTimeoutException)
            {
                return false;
            }
        }
    }
}
```

Instead of setting the `Text` property of the `Entry` to `Red`, `ValidEmailBehavior` defines an `IsValid` property that is backed by a bindable property. Because it makes no sense for code external to this class to set the `IsValid` property, it is a read-only bindable property. The `Bindable.CreateReadOnly` call creates a private bindable-property key that is used by the `SetValue` call in the private `set` accessor of `IsValid`. The public `IsValidProperty` bindable property is referenced by the `GetValue` call as usual.

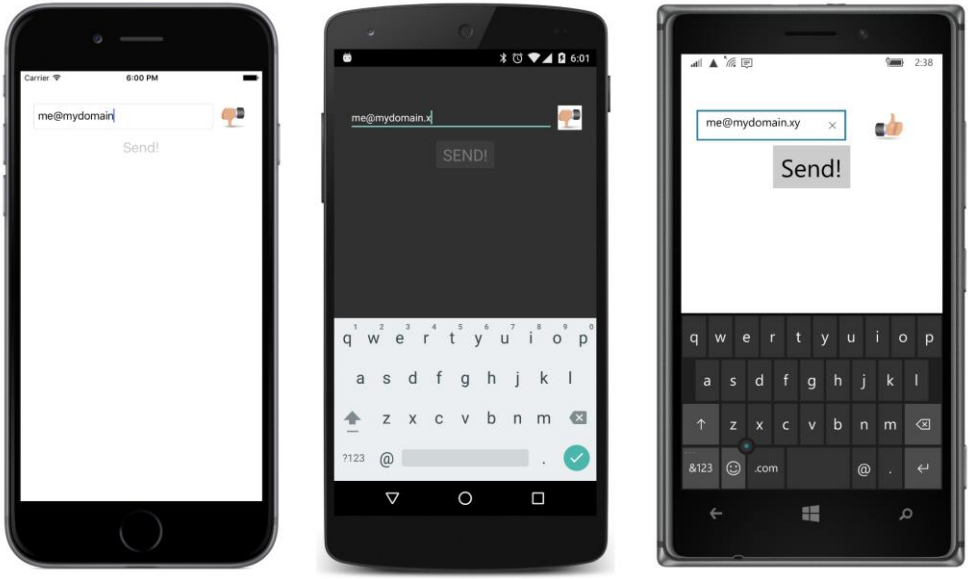
How you use that `IsValid` property is entirely up to you.

For example, the **EmailValidationDemo** program binds that `IsValid` property to the `IsVisible` property of an `Image` displaying a "thumb up" picture. That "thumb up" bitmap sits on top of another `Image` element with a "thumb down" to indicate when a valid email address has been typed. That `IsValid` property is also bound to the `IsEnabled` property of a **Send** button. Notice that the `Source` of both data bindings is the `ValidEmailBehavior` object:

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

```
            Padding="20, 50">

    <StackLayout>
        <StackLayout Orientation="Horizontal">
            <Entry Placeholder="Enter email address"
                   Keyboard="Email"
                   HorizontalOptions="FillAndExpand">
                <Entry.Behaviors>
                    <toolkit:ValidEmailBehavior x:Name="validEmail" />
                </Entry.Behaviors>
            </Entry>

            <Grid HeightRequest="40">
                <Image Source=
                        "{local:ImageResource EmailValidationDemo.Images.ThumbsDown.png}" />

                <Image Source="{local:ImageResource EmailValidationDemo.Images.ThumbsUp.png}"
                       IsVisible="{Binding Source={x:Reference validEmail},
                                           Path=IsValid}"/>
            </Grid>
        </StackLayout>

        <Button Text="Send!"
                FontSize="Large"
                HorizontalOptions="Center"
                IsEnabled="{Binding Source={x:Reference validEmail},
                                    Path=IsValid}" />
    </StackLayout>
</ContentPage>
```

As you're typing an email address, it's not considered valid until it has at least a two-character top-level domain:



The two bitmaps are part of the common **EmailValidationDemo** project. The `ImageResource` markup extension class used to reference the bitmaps was discussed in Chapter 13, "Bitmaps," and it must be part of the same assembly that contains the bitmaps:

```
namespace EmailValidationDemo
{
    [ContentProperty ("Source")]
    public class ImageResourceExtension : IMarkupExtension
    {
        public string Source { get; set; }

        public object ProvideValue (IServiceProvider serviceProvider)
        {
            if (Source == null)
                return null;

            return ImageSource.FromResource(Source);
        }
    }
}
```

What if you have multiple `Entry` views on the same page that need to check for valid email addresses. Could you include the `ValidEmailBehavior` class in a `Behaviors` collection of a `Style`?

No you cannot. The `ValidEmailBehavior` class defines a property named `IsValid`. This means that a particular instance of `ValidEmailBehavior` always has a particular state, which is the value of this property. This has a significant implication:

A behavior that maintains state—such as a field or a property—cannot be shared, which means it shouldn't be included in a `Style`.

If you need to use `ValidEmailBehavior` for multiple `Entry` views on the same page, don't put it in a `Style`. Add a separate instance to the `Behaviors` collections of each of the `Entry` views.

The advantage of this `IsValid` property outweighs the disadvantages, however, because you can use the property in a variety of ways. Here's a program called **EmailValidationConverter** that uses the `IsValid` property with a binding converter already in the **Xamarin.FormsBook.Toolkit** library to choose between two text strings:

```
<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="EmailValidationConverter.EmailValidationConverterPage"
             Padding="50">

    <StackLayout>
        <StackLayout Orientation="Horizontal">
            <Entry Placeholder="Enter email address"
                   HorizontalOptions="FillAndExpand">
                <Entry.Behaviors>
                    <toolkit:ValidEmailBehavior x:Name="validEmail" />
                </Entry.Behaviors>
            </Entry>

            <Label HorizontalTextAlignment="Center"
                   VerticalTextAlignment="Center">
                <Label.Text>
                    <Binding Source="{x:Reference validEmail}"
                             Path="IsValid">
                        <Binding.Converter>
                            <toolkit:BoolToObjectConverter x:TypeArguments="x:String"
                                                           FalseObject="Not yet!"
                                                           TrueObject="OK!" />
                        </Binding.Converter>
                    </Binding>
                </Label.Text>
            </Label>
        </StackLayout>

        <Button Text="Send!"
                FontSize="Large"
                HorizontalOptions="Center"
                IsEnabled="{Binding Source={x:Reference validEmail},
                                    Path=IsValid}" />
    </StackLayout>
</ContentPage>
```

The `BoolToObjectConverter` chooses between the two text strings "Not yet!" and "OK!".

However, you can do this same thing with a little more straightforward logic and no binding converter by using a `DataTrigger`, as the **EmailValidationTrigger** program demonstrates. The "Not yet!" text is assigned to the `Text` property of the `Label`, while a `DataTrigger` on the `Label` contains a

binding to the `IsValid` property to set the "OK!" text:

```
<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="EmailValidationTrigger.EmailValidationTriggerPage"
             Padding="50">

    <StackLayout>
        <StackLayout Orientation="Horizontal">
            <Entry Placeholder="Enter email address"
                   HorizontalOptions="FillAndExpand">
                <Entry.Behaviors>
                    <toolkit:ValidEmailBehavior x:Name="validEmail" />
                </Entry.Behaviors>
            </Entry>

            <Label Text="Not yet!"
                   HorizontalTextAlignment="Center"
                   VerticalTextAlignment="Center">
                <Label.Triggers>
                    <DataTrigger TargetType="Label"
                                 Binding="{Binding Source={x:Reference validEmail},
                                                   Path=IsValid}"
                                 Value="True">
                        <Setter Property="Text" Value="OK!" />
                    </DataTrigger>
                </Label.Triggers>
            </Label>
        </StackLayout>

        <Button Text="Send!"
                FontSize="Large"
                HorizontalOptions="Center"
                IsEnabled="{Binding Source={x:Reference validEmail},
                                    Path=IsValid}" />
    </StackLayout>
</ContentPage>
```

Referencing a behavior from a data binding in a `DataTrigger` is a powerful technique.

Toggles and check boxes

In Chapter 15, "The interactive interface," and Chapter 16, "Data binding," you saw how to construct traditional `CheckBox` views. However, another approach to custom views is to incorporate the interactive logic of the view in a behavior and then realize the visuals entirely in XAML. This approach gives you the flexibility of customizing the visuals with markup rather than code. Because the visual appearance is not part of the underlying logic, you can create ad hoc visuals whenever you use the behavior.

Here is a class in the **Xamarin.FormsBook.Toolkit** library named `ToggleBehavior`. Like the Xamarin.Forms `Switch` element, it defines a property named `IsToggled` that is backed by a bindable

property. `ToggleBehavior` simply installs a `TapGestureRecognizer` to the visual that it's attached to and toggles the state of the `IsToggled` property whenever a tap is detected:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class ToggleBehavior : Behavior<View>
    {
        TapGestureRecognizer tapRecognizer;

        public static readonly BindableProperty IsToggledProperty =
            BindableProperty.Create<ToggleBehavior, bool>(tb => tb.IsToggled, false);

        public bool IsToggled
        {
            set { SetValue(IsToggledProperty, value); }
            get { return (bool)GetValue(IsToggledProperty); }
        }

        protected override void OnAttachedTo(View view)
        {
            base.OnAttachedTo(view);

            tapRecognizer = new TapGestureRecognizer ();
            tapRecognizer.Tapped += OnTapped;
            view.GestureRecognizers.Add(tapRecognizer);
        }

        protected override void OnDetachingFrom(View view)
        {
            base.OnDetachingFrom(view);

            view.GestureRecognizers.Remove(tapRecognizer);
            tapRecognizer.Tapped -= OnTapped;
        }

        void OnTapped(object sender, EventArgs args)
        {
            IsToggled = !IsToggled;
        }
    }
}
```

The `ToggleBehavior` class defines a property, which means that you cannot share a `ToggleBehavior` in a `Style`.

Here's a simple application. The **ToggleLabel** program attaches `ToggleBehavior` to a `Label` and uses the `IsToggled` property with a `DataTrigger` to switch the text of the `Label` between "Paused" and "Playing," perhaps for a music application:

```
<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="ToggleLabel.ToggleLabelPage">

    <Label Text="Paused"
           FontSize="Large"
           HorizontalOptions="Center"
           VerticalOptions="Center">
        <Label.Behaviors>
            <toolkit:ToggleBehavior x:Name="toggleBehavior" />
        </Label.Behaviors>

        <Label.Triggers>
            <DataTrigger TargetType="Label"
                         Binding="{Binding Source={x:Reference toggleBehavior},
                                           Path=IsToggled}"
                         Value="True">
                <Setter Property="Text" Value="Playing" />
            </DataTrigger>
        </Label.Triggers>
    </Label>
</ContentPage>
```

Of course, such a program would probably need to run some code when the `Label` is toggled. Keep in mind that `Behavior` derives from `BindableObject`, which means that any `BindableProperty` that you define in a behavior automatically generates a `PropertyChanged` event when the property changes.

This means that you can attach a handler to the `PropertyChanged` event of `ToggleBehavior` and check for changes in the `IsToggled` property. This is demonstrated in the **FormattedTextToggle** program, which expands the **ToggleLabel** program to include a `Frame` and some formatted text that more clearly indicates the two options that the tap switches between:

```
<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="FormattedTextToggle.FormattedTextTogglePage">

    <StackLayout>
        <Frame HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               OutlineColor="Accent"
               BackgroundColor="Transparent">

            <Frame.Behaviors>
                <toolkit:ToggleBehavior x:Name="toggleBehavior"
                                        PropertyChanged="OnBehaviorPropertyChanged" />
            </Frame.Behaviors>

            <Label>
                <Label.FormattedText>
                    <FormattedString>
                        <FormattedString.Spans>
                            <Span Text="Paused / "
```

```
                                FontSize="Large"
                                FontAttributes="Bold" />

                        <Span Text="Playing"
                                FontSize="Small" />
                    </FormattedString.Spans>
                </FormattedString>
            </Label.FormattedText>

            <Label.Triggers>
                <DataTrigger TargetType="Label"
                             Binding="{Binding Source={x:Reference toggleBehavior},
                                               Path=IsToggled}"
                             Value="True">
                    <Setter Property="FormattedText">
                        <Setter.Value>
                            <FormattedString>
                                <FormattedString.Spans>
                                    <Span Text="Paused"
                                          FontSize="Small" />

                                    <Span Text=" / Playing"
                                          FontSize="Large"
                                          FontAttributes="Bold" />
                                </FormattedString.Spans>
                            </FormattedString>
                        </Setter.Value>
                    </Setter>
                </DataTrigger>
            </Label.Triggers>
        </Label>
    </Frame>

    <Label x:Name="eventLabel"
           Text=""
           FontSize="Large"
           Opacity="0"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />
  </StackLayout>
</ContentPage>
```

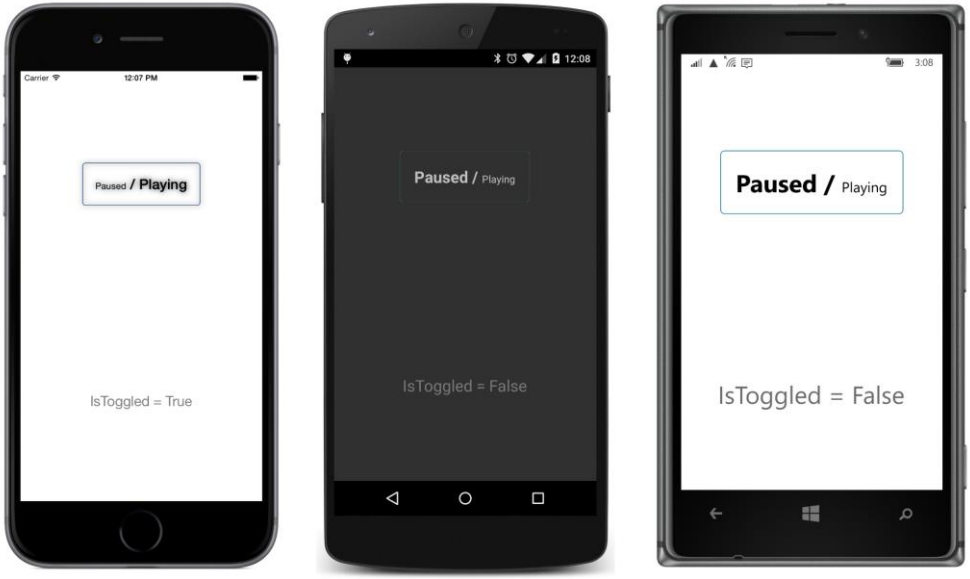
The `ToggleBehavior` is attached to the `Frame`, and the `Frame` contains a `Label`. (Notice that the `BackgroundColor` of the `Frame` is set to `Transparent` rather than the default value of `null`. This is necessary to trap the tap events on the Windows Runtime platforms.)

This program demonstrates one way to solve a common problem with toggle buttons: Does the text (or icon) refer to a state or an action? The `Label` here makes it clear by displaying the text "Paused / Playing" but with the word "Paused" larger than the word "Playing". When the `IsToggled` property is `True`, the `DataTrigger` changes that display so that the word "Playing" is larger than the word "Paused".

The `PropertyChanged` event on the `ToggleBehavior` is handled in the code-behind file:

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

    void OnBehaviorPropertyChanged(object sender, PropertyChangedEventArgs args)
    {
        if (args.PropertyName == "IsToggled")
        {
            eventLabel.Text = "IsToggled = " + ((ToggleBehavior)sender).IsToggled;
            eventLabel.Opacity = 1;
            eventLabel.FadeTo(0, 1000);
        }
    }
}
```

The `OnBehaviorPropertyChanged` handler checks for a change in the property named "IsToggled". Keep in mind that the `sender` argument to the event handler is not the visual element whose taps are being detected (which is the `Frame`) but the `ToggleBehavior` itself. The code sets the `Text` property of the `Label` at the bottom of the page and sets the `Opacity` to 1, but then fades it out over the course of a second to give a sense of an event firing:



If you like the idea of defining the visuals of a toggle view in XAML but prefer a little more structure, the **Xamarin.FormsBook.Toolkit** library has a class named `ToggleBase` that derives from `ContentView` and incorporates `ToggleBehavior`. The constructor adds the `ToggleBehavior` to the `Behaviors` collection of the class and then attaches an event handler to it. The class also defines a `Toggled` event and its own `IsToggled` property that fires that event:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class ToggleBase : ContentView
    {
        public event EventHandler<ToggledEventArgs> Toggled;

        public static readonly BindableProperty IsToggledProperty =
            BindableProperty.Create("IsToggled", typeof(bool), typeof(ToggleBase), false,
                                    BindingMode.TwoWay,
                propertyChanged: (bindable, oldValue, newValue) =>
                {
                    EventHandler<ToggledEventArgs> handler = ((ToggleBase)bindable).Toggled;
                    if (handler != null)
                        handler(bindable, new ToggledEventArgs((bool)newValue));
                });

        public ToggleBase()
        {
            ToggleBehavior toggleBehavior = new ToggleBehavior();
            toggleBehavior.PropertyChanged += OnToggleBehaviorPropertyChanged;
            Behaviors.Add(toggleBehavior);
        }

        public bool IsToggled
        {
            set { SetValue(IsToggledProperty, value); }
            get { return (bool)GetValue(IsToggledProperty); }
        }

        protected void OnToggleBehaviorPropertyChanged(object sender,
                                                       PropertyChangedEventArgs args)
        {
            if (args.PropertyName == "IsToggled")
            {
                IsToggled = ((ToggleBehavior)sender).IsToggled;
            }
        }
    }
}
```

The `ToggleBase` class defines all the logic of a toggle view without the visuals. In truth, it doesn't require the `ToggleBehaviors` class. It could install its own `TapGestureRecognizer`, but the result would be basically the same.

You can instantiate the `ToggleBase` class in a XAML file and supply the visuals as content of the `ToggleBase`. Here's a program called **TraditionalCheckBox** that uses two Unicode characters for an unchecked box and a checked box, similar to the `CheckBox` views in Chapters 15 and 16:

```
<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="TraditionalCheckBox.TraditionalCheckBoxPage">
```

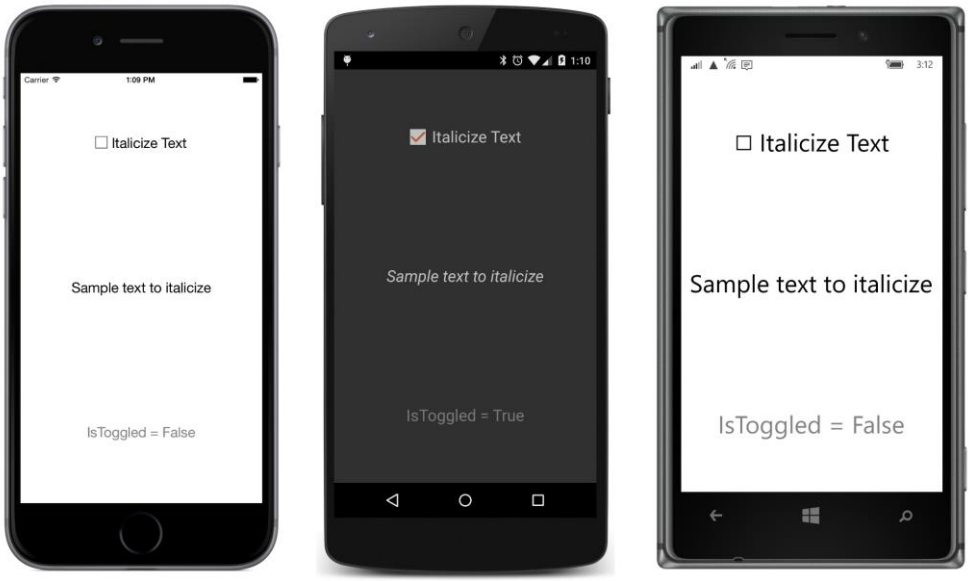
```
    <StackLayout>
        <toolkit:ToggleBase x:Name="checkbox"
                            HorizontalOptions="Center"
                            VerticalOptions="CenterAndExpand"
                            Toggled="OnToggleBaseToggled">
            <StackLayout Orientation="Horizontal">
                <Label Text="&#x2610;"
                       FontSize="Large">
                    <Label.Triggers>
                        <DataTrigger TargetType="Label"
                                     Binding="{Binding Source={x:Reference checkbox},
                                                       Path=IsToggled}"
                                     Value="True">
                            <Setter Property="Text" Value="&#x2611;" />
                        </DataTrigger>
                    </Label.Triggers>
                </Label>

                <Label Text="Italicize Text"
                       FontSize="Large" />
            </StackLayout>
        </toolkit:ToggleBase>

        <Label Text="Sample text to italicize"
               FontSize="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand">
            <Label.Triggers>
                <DataTrigger TargetType="Label"
                             Binding="{Binding Source={x:Reference checkbox},
                                               Path=IsToggled}"
                             Value="True">
                    <Setter Property="FontAttributes" Value="Italic" />
                </DataTrigger>
            </Label.Triggers>
        </Label>

        <Label x:Name="eventLabel"
               Text=""
               FontSize="Large"
               Opacity="0"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

    </StackLayout>
</ContentPage>
```

The XAML file uses the `IsToggled` property as the source of two very similar data bindings, each within a `DataTrigger`. In both cases the `Source` property is set to the `ToggleBase` instance, and the `Path` property is set to the `IsToggled` property of `ToggleBase`. The first `DataTrigger` switches between the empty box and the checked box to indicate the state of the toggle, and the second `DataTrigger` italicizes some text when the `CheckBox` is toggled on.

In addition, the `Toggled` event of `ToggleBase` is handled in the code-behind file with a fade-out `Label`:

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

    void OnToggleBaseToggled(object sender, ToggledEventArgs args)
    {
        eventLabel.Text = "IsToggled = " + args.Value;
        eventLabel.Opacity = 1;
        eventLabel.FadeTo(0, 1000);
    }
}
```

Here's the result:



If you need multiple instances of a particular type of toggle view, you can encapsulate the visuals in a class that derives from `ToggleBase`.

The next example derives from `ToggleBase` to make a view that is very much like the Xamarin.Forms `Switch`, except with visuals created entirely in XAML. This "switch clone" is realized with a little `BoxView` that moves back and forth in a `Frame`. For implementing the animation, the **Xamarin.FormsBook.Toolkit** library includes a `TranslateAction` class with properties that provide arguments for a call to `TranslateTo`:

```
namespace Xamarin.FormsBook.Toolkit
```

```
{
    public class TranslateAction : TriggerAction<VisualElement>
    {
        public TranslateAction()
        {
            // Set defaults.
            Length = 250;
            Easing = Easing.Linear;
        }

        public double X { set; get; }

        public double Y { set; get; }

        public int Length { set; get; }

        [TypeConverter(typeof(EasingConverter))]
        public Easing Easing { set; get; }

        protected override void Invoke(VisualElement visual)
        {
            visual.TranslateXYTo(X, Y, (uint)Length, Easing);
        }
    }
}
```

The `SwitchClone` class that mimics the `Switch` is part of the **SwitchCloneDemo** project. It's entirely done in XAML. The root element is the base class of `ToggleBase`, and the `x:Class` attribute indicates the derived class of `SwitchClone`. The `Resources` dictionary defines several constants that allow for visuals that are not too large, but still big enough to be a proper touch target:

```
<toolkit:ToggleBase
        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="SwitchCloneDemo.SwitchClone"
        x:Name="toggle">

    <toolkit:ToggleBase.Resources>
        <ResourceDictionary>
            <x:Double x:Key="height">20</x:Double>
            <x:Double x:Key="width">50</x:Double>
            <x:Double x:Key="halfWidth">25</x:Double>
        </ResourceDictionary>
    </toolkit:ToggleBase.Resources>

    <Frame Padding="2"
           OutlineColor="Accent"
           BackgroundColor="Transparent">
        <AbsoluteLayout WidthRequest="{StaticResource width}">
            <BoxView Color="Accent"
                     WidthRequest="{StaticResource halfWidth}"
                     HeightRequest="{StaticResource height}">
```

```
                <BoxView.Triggers>
                    <DataTrigger TargetType="BoxView"
                                 Binding="{Binding Source={x:Reference toggle},
                                                   Path=IsToggled}"
                                 Value="True">
                        <DataTrigger.EnterActions>
                            <toolkit:TranslateAction X="{StaticResource halfWidth}"
                                                     Length="100" />
                        </DataTrigger.EnterActions>

                        <DataTrigger.ExitActions>
                            <toolkit:TranslateAction Length="100" />
                        </DataTrigger.ExitActions>
                    </DataTrigger>
                </BoxView.Triggers>
            </BoxView>
        </AbsoluteLayout>
    </Frame>
</toolkit:ToggleBase>
```

Notice that the root element has a name of "toggle." This allows the data binding in the `DataTrigger` on the `BoxView` to reference the `IsToggled` property defined by the `ToggleBase` class. The `DataTrigger` does not include a `Setter` but instead uses `EnterActions` and `ExitActions` to invoke the `TranslateAction` for shifting the `BoxView` back and forth.

The code-behind file for `SwitchClone` has nothing but an `InitializeComponent` call, but if you need other properties (for example, for color or some accompanying text) you can define them there.

At least that's the way it was originally coded. Later on, the program refused to build on the Windows Runtime platforms. Perhaps the problem had something to do with the root element in the XAML file referencing a class in a library. Regardless, a code-only version of the class did work, and this is the one included with the sample code for this chapter:

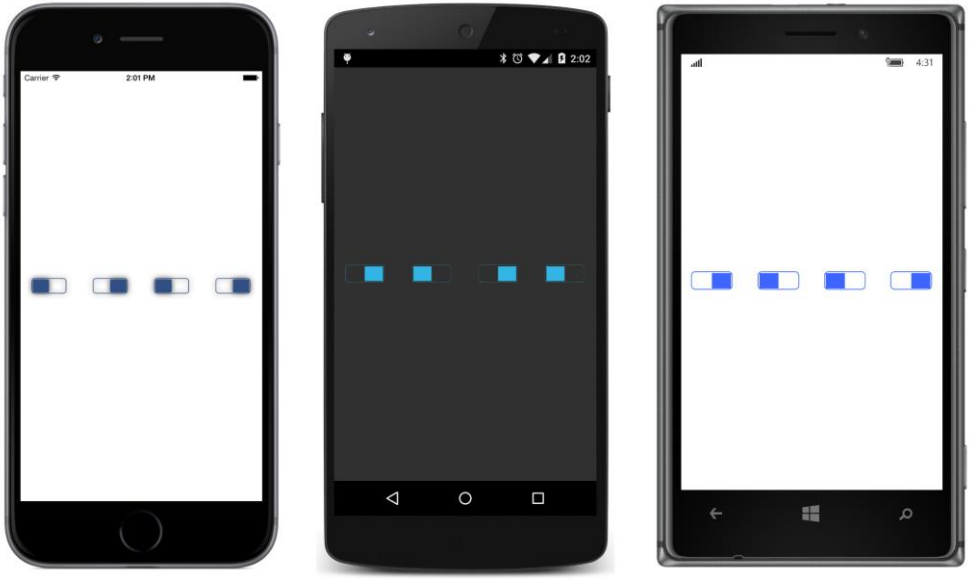
```
class SwitchClone : ToggleBase
{
    const double height = 20;
    const double width = 50;
    const double halfWidth = 25;

    public SwitchClone()
    {
        BoxView boxView = new BoxView
        {
            Color = Color.Accent,
            WidthRequest = halfWidth,
            HeightRequest = height
        };

        DataTrigger dataTrigger = new DataTrigger(typeof(BoxView))
        {
            Binding = new Binding("IsToggled", source: this),
            Value = true,
        };
```

```
        dataTrigger.EnterActions.Add(new TranslateAction
        {
            X = halfWidth,
            Length = 100
        });

        dataTrigger.ExitActions.Add(new TranslateAction
        {
            Length = 100
        });

        boxView.Triggers.Add(dataTrigger);

        Content = new Frame
        {
            Padding = 2,
            OutlineColor = Color.Accent,
            BackgroundColor = Color.Transparent,
            Content = new AbsoluteLayout
            {
                WidthRequest = width,
                Children =
                {
                    boxView
                }
            }
        };
    }
}
```

The `SwitchCloneDemoPage` class displays four of these switch clones in a row:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:SwitchCloneDemo"
             x:Class="SwitchCloneDemo.SwitchCloneDemoPage">

    <Grid VerticalOptions="Center">
        <local:SwitchClone Grid.Column="0"
                           HorizontalOptions="Center" />

        <local:SwitchClone Grid.Column="1"
                           HorizontalOptions="Center" />

        <local:SwitchClone Grid.Column="2"
                           HorizontalOptions="Center" />

        <local:SwitchClone Grid.Column="3"
                           HorizontalOptions="Center" />
    </Grid>
</ContentPage>
```

And here they are:



Of course, once you start thinking about using animations, you might start getting some interesting (or perhaps downright odd) ideas of what a toggle view might look like. To give you a few more options, here's a `RotateAction` class:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class RotateAction : TriggerAction<VisualElement>
    {
        public RotateAction()
        {
            // Set defaults.
            Anchor = new Point (0.5, 0.5);
            Rotation = 0;
            Length = 250;
            Easing = Easing.Linear;
        }

        public Point Anchor { set; get; }

        public double Rotation { set; get; }

        public int Length { set; get; }

        [TypeConverter(typeof(EasingConverter))]
        public Easing Easing { set; get; }

        protected override void Invoke(VisualElement visual)
        {
            visual.AnchorX = Anchor.X;
            visual.AnchorY = Anchor.Y;
            visual.RotateTo(Rotation, (uint)Length, Easing);
```

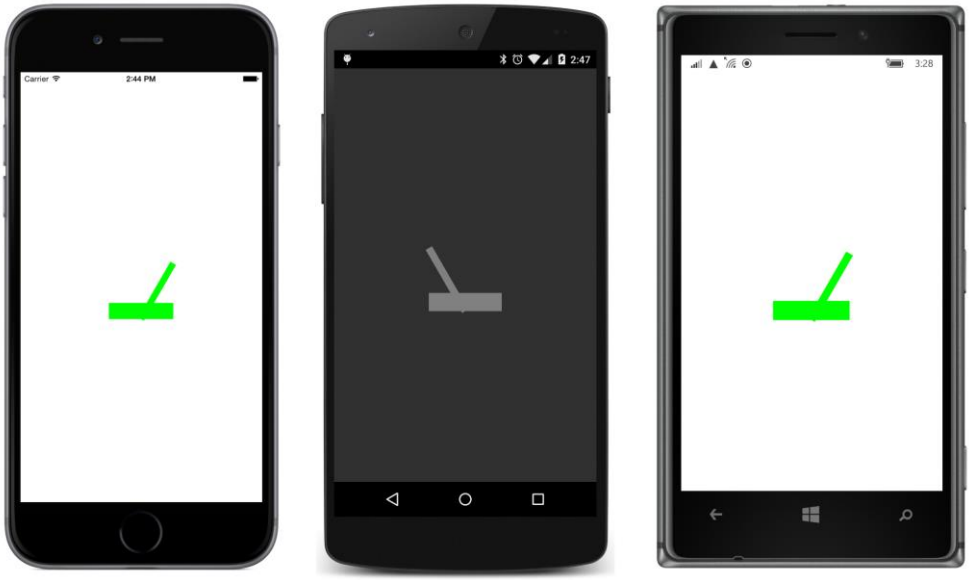
```
        }
    }
}
```

The **LeverToggle** program has a XAML file that is devoted to a single toggle switch constructed from two `BoxView` elements. The first `BoxView` resembles a base for the second, which functions like a lever. Notice that the `DataTrigger` on the second `BoxView` contains a `Setter` to change the color of the `BoxView` as well as `EnterActions` and `ExitActions` to invoke animations that move the lever back and forth:

```
<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="LeverToggle.LeverTogglePage">

    <toolkit:ToggleBase x:Name="toggle"
                        HorizontalOptions="Center"
                        VerticalOptions="Center">
        <AbsoluteLayout>
            <BoxView Color="Gray"
                     AbsoluteLayout.LayoutBounds="0, 75, 100, 25">
                <BoxView.Triggers>
                    <DataTrigger TargetType="BoxView"
                                 Binding="{Binding Source={x:Reference toggle},
                                                   Path=IsToggled}"
                                 Value="True">
                        <Setter Property="Color" Value="Lime" />
                    </DataTrigger>
                </BoxView.Triggers>
            </BoxView>

            <BoxView Color="Gray"
                     AbsoluteLayout.LayoutBounds="45, 0, 10, 100"
                     AnchorX="0.5"
                     AnchorY="1"
                     Rotation="-30">
                <BoxView.Triggers>
                    <DataTrigger TargetType="BoxView"
                                 Binding="{Binding Source={x:Reference toggle},
                                                   Path=IsToggled}"
                                 Value="True">

                        <Setter Property="Color" Value="Lime" />

                        <DataTrigger.EnterActions>
                            <toolkit:RotateAction Anchor="0.5, 1" Rotation="30" />
                        </DataTrigger.EnterActions>

                        <DataTrigger.ExitActions>
                            <toolkit:RotateAction Anchor="0.5, 1" Rotation="-30" />
                        </DataTrigger.ExitActions>
                    </DataTrigger>
```

```
                </BoxView.Triggers>
            </BoxView>
        </AbsoluteLayout>
    </toolkit:ToggleBase>
</ContentPage>
```

The untoggled state is shown on the Android screen, while the iOS and Windows 10 Mobile screens show the toggled state:



Responding to taps

The various manifestations of toggle views demonstrate one way to respond to taps within a XAML file. If tap events were integrated into the `VisualElement` class, you could get at them more directly and with greater ease using `EventTrigger`. But you can't attach an `EventTrigger` to a `TapGestureRecognizer`.

Getting around that little restriction is the purpose of a behavior devoted solely to a tap. This is called `TapBehavior`:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class TapBehavior : Behavior<View>
    {
        TapGestureRecognizer tapGesture;

        static readonly BindablePropertyKey IsTriggeredKey =
            BindableProperty.CreateReadOnly("IsTriggered", typeof(bool),
                                            typeof(TapBehavior), false);
```

```
        public static readonly BindableProperty IsTriggeredProperty =
                                            IsTriggeredKey.BindableProperty;

        public bool IsTriggered
        {
            private set { SetValue(IsTriggeredKey, value); }
            get { return (bool)GetValue(IsTriggeredProperty); }
        }

        protected override void OnAttachedTo(View view)
        {
            base.OnAttachedTo(view);

            tapGesture = new TapGestureRecognizer();
            tapGesture.Tapped += OnTapped;
            view.GestureRecognizers.Add(tapGesture);
        }

        protected override void OnDetachingFrom(View view)
        {
            base.OnDetachingFrom(view);

            view.GestureRecognizers.Remove(tapGesture);
            tapGesture.Tapped -= OnTapped;
        }

        async void OnTapped(object sender, EventArgs args)
        {
            IsTriggered = true;
            await Task.Delay(100);
            IsTriggered = false;
        }
    }
}
```

The `TapBehavior` class defines a Boolean property named `IsTriggered`, but it doesn't function exactly like a normal property. For one thing, it's backed by a read-only bindable property. This means that the `IsTriggered` property can be set only within the `TapBehavior` class, and the only time the class sets `IsTriggered` is in the event handler for the `TapGestureRecognizer`, when the `IsTriggered` property becomes `true` for a mere one-tenth of a second.

In other words, the `Tapped` event is converted into a very brief spike of a property value—somewhat reminiscent of how events are triggered in digital hardware. But the `IsTriggered` property can then be referenced in a `DataTrigger`.

Suppose you like the idea of the `ShiverButton`, but you'd like to apply the concept to something other than a `Button`, which means you need to respond to `Tapped` events. You can't use an `EventTrigger`, but the `TapBehavior` lets you use a `DataTrigger` instead.

To demonstrate, here's **BoxViewTapShiver**, which attaches `TapBehavior` objects to three `BoxView` elements, each of which also includes a `DataTrigger` that references the behavior and invokes a

`ShiverAction` in its `EnterActions` collection:

```
<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="BoxViewTapShiver.BoxViewTapShiverPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="BoxView">
                <Setter Property="WidthRequest" Value="200" />
                <Setter Property="HeightRequest" Value="50" />
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions"   Value="CenterAndExpand" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <BoxView Color="Red">
            <BoxView.Behaviors>
                <toolkit:TapBehavior x:Name="tapBehavior1" />
            </BoxView.Behaviors>

            <BoxView.Triggers>
                <DataTrigger TargetType="BoxView"
                             Binding="{Binding Source={x:Reference tapBehavior1},
                                               Path=IsTriggered}"
                             Value="True">
                    <DataTrigger.EnterActions>
                        <toolkit:ShiverAction />
                    </DataTrigger.EnterActions>
                </DataTrigger>
            </BoxView.Triggers>
        </BoxView>

        <BoxView Color="Green">
            <BoxView.Behaviors>
                <toolkit:TapBehavior x:Name="tapBehavior2" />
            </BoxView.Behaviors>

            <BoxView.Triggers>
                <DataTrigger TargetType="BoxView"
                             Binding="{Binding Source={x:Reference tapBehavior2},
                                               Path=IsTriggered}"
                             Value="True">
                    <DataTrigger.EnterActions>
                        <toolkit:ShiverAction />
                    </DataTrigger.EnterActions>
                </DataTrigger>
            </BoxView.Triggers>
        </BoxView>
```

```
        <BoxView Color="Blue">
            <BoxView.Behaviors>
                <toolkit:TapBehavior x:Name="tapBehavior3" />
            </BoxView.Behaviors>

            <BoxView.Triggers>
                <DataTrigger TargetType="BoxView"
                             Binding="{Binding Source={x:Reference tapBehavior3},
                                               Path=IsTriggered}"
                             Value="True">
                    <DataTrigger.EnterActions>
                        <toolkit:ShiverAction />
                    </DataTrigger.EnterActions>
                </DataTrigger>
            </BoxView.Triggers>
        </BoxView>
    </StackLayout>
</ContentPage>
```

Each of the three `TapBehavior` objects has a unique name, which is referenced by the corresponding `DataTrigger`. When you tap a `BoxView`, it shivers, and they all work independently.

It is very tempting to put the `TapBehavior` and `DataTrigger` objects in a `Style` to cut down on the repetitive markup, but that won't work. That would cause a single `TapBehavior` to be shared among the three `BoxView` elements. Moreover, each `DataTrigger` refers to a corresponding `TapBehavior` by name.

If you want to cut down on the markup in this case, you'll once again need to define a new class. The **ShiverViews** program demonstrates this. It first defines a class named `ShiverView` that derives from `BoxView` and adds the `TapBehavior` and `DataTrigger`:

```
<BoxView 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="ShiverViews.ShiverView">

    <BoxView.Behaviors>
        <toolkit:TapBehavior x:Name="tapBehavior" />
    </BoxView.Behaviors>

    <BoxView.Triggers>
        <DataTrigger TargetType="BoxView"
                     Binding="{Binding Source={x:Reference tapBehavior},
                                       Path=IsTriggered}"
                     Value="True">
            <DataTrigger.EnterActions>
                <toolkit:ShiverAction />
            </DataTrigger.EnterActions>
        </DataTrigger>
    </BoxView.Triggers>
</BoxView>
```

As with the `SwitchClone` class, you could also add some properties in the code-behind file and reference them in the XAML file.

The `ShiverViewsPage` XAML file can then just instantiate three independent `ShiverView` objects with an implicit style:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:ShiverViews"
             x:Class="ShiverViews.ShiverViewsPage">

    <StackLayout>
        <StackLayout.Resources>
            <ResourceDictionary>
                <Style TargetType="local:ShiverView">
                    <Setter Property="WidthRequest" Value="200" />
                    <Setter Property="HeightRequest" Value="50" />
                    <Setter Property="HorizontalOptions" Value="Center" />
                    <Setter Property="VerticalOptions"   Value="CenterAndExpand" />
                </Style>
            </ResourceDictionary>
        </StackLayout.Resources>

        <local:ShiverView Color="Red" />
        <local:ShiverView Color="Green" />
        <local:ShiverView Color="Blue" />
    </StackLayout>
</ContentPage>
```

Radio buttons

The radios built into the dashboards of old automobiles often featured a row of half a dozen (or so) buttons that could be "programmed" for various radio stations. Pushing in one of these buttons caused the radio to jump to that preselected station, and also caused the button for the previous selection to pop out.

Those old car radios are now antiques, but mutually exclusive options on our computer screens are still represented by visual objects we call *radio buttons*.

Radio buttons are somewhat similar to toggles or check boxes. But radio buttons are always found in a group of two or more. Selecting or checking any button in that group causes the others to become unchecked.

The logic behind radio buttons is complicated because an application might feature several groups of radio buttons on the same page, and these groups should function independently. Pressing a button in one group should only affect the other buttons within that group, and not the buttons in any other group.

Traditionally, radio buttons were grouped with a common parent. In Xamarin.Forms terminology, radio buttons that are children of one `StackLayout` are considered to be in the same group, while

radio buttons that are children of another `StackLayout` are in another independent group.

However, there is a more generalized way to distinguish groups of radio buttons, and that is by giving each group a unique name, which really means that each radio button within that group references the same name.

The problem with these names is that they add some extra overhead, particularly when you need only one group of radio buttons. For that reason, there should be an allowance for a group of radio buttons that is *not* identified by a name. This is called the *default* group.

Here is a `RadioBehavior` class in the **Xamarin.FormsBook.Toolkit** library that is based on those principles. You attach this behavior to every view that you want to convert into a radio button. Like the `ToggleBehavior` class, `RadioBehavior` sets a `TapGestureRecognizer` on the visual element to which it's attached. It doesn't define an `IsToggled` property like `ToggleBehavior`, but it does define an `IsChecked` property that is quite similar and indicates whether the radio button is checked or unchecked. The `RadioBehavior` class also defines a `GroupName` property of type `string` to identify the group; a `null` value or an empty string indicates the default group.

The `RadioBehavior` class needs to store all the instantiated radio buttons by group, so it defines two static collections, one of which is a simple `List<RadioBehavior>` for all the objects in the default group, and the other is a `Dictionary` with a key corresponding to the group name that references a `List<RadioBehavior>` collection for all the objects in that named group:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class RadioBehavior : Behavior<View>
    {
        TapGestureRecognizer tapRecognizer;
        static List<RadioBehavior> defaultGroup = new List<RadioBehavior>();
        static Dictionary<string, List<RadioBehavior>> radioGroups =
                                    new Dictionary<string, List<RadioBehavior>>();

        public RadioBehavior()
        {
            defaultGroup.Add(this);
        }

        public static readonly BindableProperty IsCheckedProperty =
            BindableProperty.Create("IsChecked",
                                    typeof(bool),
                                    typeof(RadioBehavior),
                                    false,
                                    propertyChanged: OnIsCheckedChanged);

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

        static void OnIsCheckedChanged(BindableObject bindable, object oldValue,
```

```
                                object newValue)
{
    RadioBehavior behavior = (RadioBehavior)bindable;

    if ((bool)newValue)
    {
        string groupName = behavior.GroupName;
        List<RadioBehavior> behaviors = null;

        if (String.IsNullOrEmpty(groupName))
        {
            behaviors = defaultGroup;
        }
        else
        {
            behaviors = radioGroups[groupName];
        }

        foreach (RadioBehavior otherBehavior in behaviors)
        {
            if (otherBehavior != behavior)
            {
                otherBehavior.IsChecked = false;
            }
        }
    }
}

public static readonly BindableProperty GroupNameProperty =
    BindableProperty.Create("GroupName",
                            typeof(string),
                            typeof(RadioBehavior),
                            null,
                            propertyChanged: OnGroupNameChanged);

public string GroupName
{
    set { SetValue(GroupNameProperty, value); }
    get { return (string)GetValue(GroupNameProperty); }
}

static void OnGroupNameChanged(BindableObject bindable, object oldValue,
                               object newValue)
{
    RadioBehavior behavior = (RadioBehavior)bindable;
    string oldGroupName = (string)oldValue;
    string newGroupName = (string)newValue;

    if (String.IsNullOrEmpty(oldGroupName))
    {
        // Remove the Behavior from the default group.
        defaultGroup.Remove(behavior);
    }
    else
```

```
        {
            // Remove the RadioBehavior from the radioGroups collection.
            List<RadioBehavior> behaviors = radioGroups[oldGroupName];
            behaviors.Remove(behavior);

            // Get rid of the collection if it's empty.
            if (behaviors.Count == 0)
            {
                radioGroups.Remove(oldGroupName);
            }
        }

        if (String.IsNullOrEmpty(newGroupName))
        {
            // Add the new Behavior to the default group.
            defaultGroup.Add(behavior);
        }
        else
        {
            List<RadioBehavior> behaviors = null;

            if (radioGroups.ContainsKey(newGroupName))
            {
                // Get the named group.
                behaviors = radioGroups[newGroupName];
            }
            else
            {
                // If that group doesn't exist, create it.
                behaviors = new List<RadioBehavior>();
                radioGroups.Add(newGroupName, behaviors);
            }

            // Add the Behavior to the group.
            behaviors.Add(behavior);
        }
    }

    protected override void OnAttachedTo(View view)
    {
        base.OnAttachedTo(view);

        tapRecognizer = new TapGestureRecognizer ();
        tapRecognizer.Tapped += OnTapRecognizerTapped;
        view.GestureRecognizers.Add(tapRecognizer);
    }

    protected override void OnDetachingFrom(View view)
    {
        base.OnDetachingFrom(view);

        view.GestureRecognizers.Remove(tapRecognizer);
        tapRecognizer.Tapped -= OnTapRecognizerTapped;
    }
```

```
        void OnTapRecognizerTapped(object sender, EventArgs args)
        {
            IsChecked = true;
        }
    }
}
```

The `TapGestureRecognizer` handler at the bottom of the listing is very simple: When the visual object is tapped, the `RadioBehavior` object attached to that visual object sets its `IsChecked` property to `true`. If the `IsChecked` property was previously `false`, that change causes a call to the `OnIsCheckedChanged` method, which sets the `IsChecked` property of all the `RadioBehavior` objects in the same group to `false`.

Here's a simple demonstration of some interactive logic for selecting the size of a T-shirt. The three radio buttons are simple `Label` elements with text properties of "Small", "Medium", and "Large", and that's why the program is called **RadioLabels**. Each `Label` has a `RadioBehavior` in its `Behaviors` collection. Each `RadioBehavior` is given an `x:Name` for data bindings, but all the `RadioBehavior` objects have a default `GroupName` property setting of `null`. Each `Label` also has a `DataTrigger` in its `Triggers` collection that is bound to the corresponding `RadioBehavior` to turn the `TextColor` of the `Label` to green when the `IsChecked` property is `true`.

Notice that the `IsChecked` property for the middle `RadioBehavior` property is initialized to `true` to select that object when the program starts up:

```
<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"
             xmlns:local="clr-namespace:RadioLabels"
             x:Class="RadioLabels.RadioLabelsPage"
             Padding="0, 50, 0, 0">

    <StackLayout>
        <Grid>
            <Grid.Resources>
                <ResourceDictionary>
                    <Style TargetType="Label">
                        <Setter Property="FontSize" Value="Medium" />
                        <Setter Property="HorizontalTextAlignment" Value="Center" />
                    </Style>
                </ResourceDictionary>
            </Grid.Resources>

            <Label Text="Small"
                   TextColor="Gray"
                   Grid.Column="0">
                <Label.Behaviors>
                    <toolkit:RadioBehavior x:Name="smallRadio" />
                </Label.Behaviors>
```

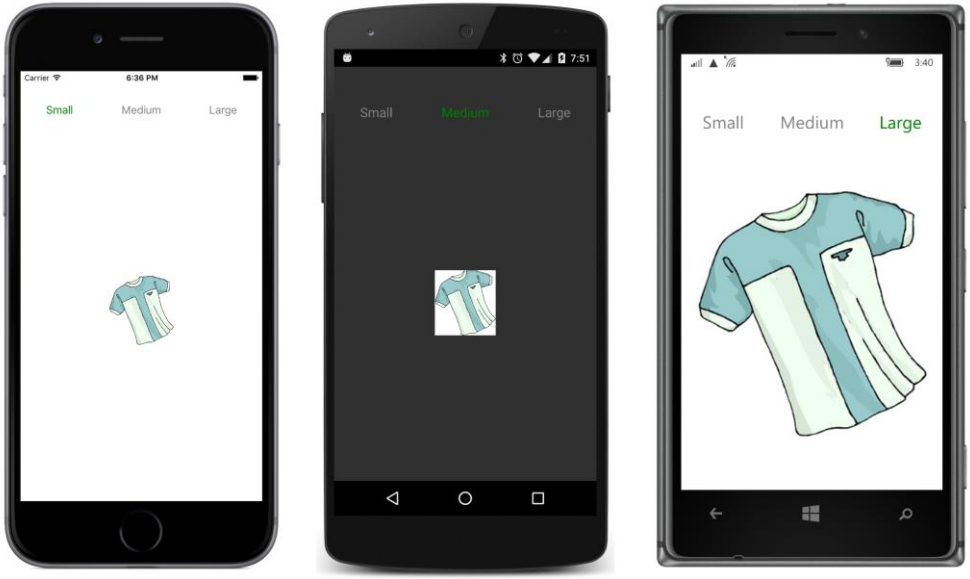
```
            <Label.Triggers>
                <DataTrigger TargetType="Label"
                             Binding="{Binding Source={x:Reference smallRadio},
                                               Path=IsChecked}"
                             Value="True">
                    <Setter Property="TextColor" Value="Green" />
                </DataTrigger>
            </Label.Triggers>
        </Label>

        <Label Text="Medium"
               TextColor="Gray"
               Grid.Column="1">
            <Label.Behaviors>
                <toolkit:RadioBehavior x:Name="mediumRadio"
                                       IsChecked="True" />
            </Label.Behaviors>

            <Label.Triggers>
                <DataTrigger TargetType="Label"
                             Binding="{Binding Source={x:Reference mediumRadio},
                                               Path=IsChecked}"
                             Value="True">
                    <Setter Property="TextColor" Value="Green" />
                </DataTrigger>
            </Label.Triggers>
        </Label>

        <Label Text="Large"
               TextColor="Gray"
               Grid.Column="2">
            <Label.Behaviors>
                <toolkit:RadioBehavior x:Name="largeRadio" />
            </Label.Behaviors>

            <Label.Triggers>
                <DataTrigger TargetType="Label"
                             Binding="{Binding Source={x:Reference largeRadio},
                                               Path=IsChecked}"
                             Value="True">
                    <Setter Property="TextColor" Value="Green" />
                </DataTrigger>
            </Label.Triggers>
        </Label>
    </Grid>

    <Grid VerticalOptions="CenterAndExpand"
          HorizontalOptions="Center">

        <Image Source="{local:ImageResource RadioLabels.Images.tee200.png}"
               IsVisible="{Binding Source={x:Reference smallRadio},
                                   Path=IsChecked}" />
        <Image Source="{local:ImageResource RadioLabels.Images.tee250.png}"
               IsVisible="{Binding Source={x:Reference mediumRadio},
```

```
                                    Path=IsChecked}" />

            <Image Source="{local:ImageResource RadioLabels.Images.tee300.png}"
                   IsVisible="{Binding Source={x:Reference largeRadio},
                                       Path=IsChecked}" />
        </Grid>
    </StackLayout>
</ContentPage>
```

Another complication intrinsic to radio buttons involves making use of the selected item. In some cases you want each radio button within a group to be represented by a particular enumeration member. (In this example, such an enumeration might have three members, named `Small`, `Medium`, and `Large`.) Consolidating a group of radio buttons into an enumeration value obviously involves more code.

The **RadioLabels** program avoids those issues and simply binds the `IsChecked` properties of the three `RadioBehavior` objects to the `IsVisible` properties of three `Image` elements sharing a single-cell `Grid` at the bottom of the XAML file. These display a different size bitmap depending on the selection.

The relative sizes of these bitmaps is not so obvious in these screenshots because each platform displays the bitmaps in somewhat different sizes:



The `DataTrigger` attached to each `Label` changes the `TextColor` from its styled color of `Gray` to `Green` when that item is selected.

If you want to change multiple properties of each `Label` when that item is selected, you can add more `Setter` objects to the `DataTrigger`. But a better approach is to consolidate the `Setter` objects

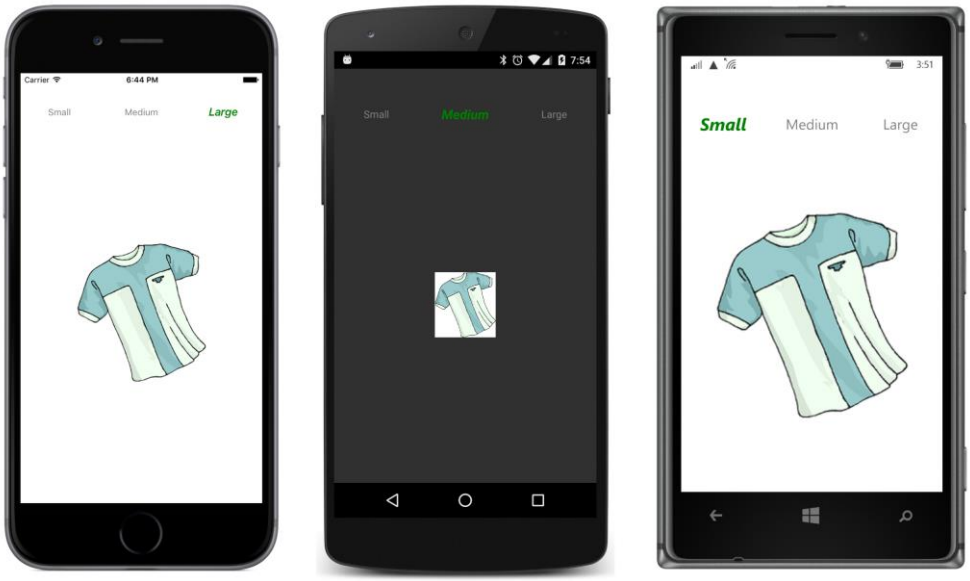
in a `Style`, and then to reference the `Style` in the `DataTrigger`.

This is demonstrated in the **RadioStyle** program. The `Resources` dictionary for the page defines a `Style` with the key of "baseStyle" that defines the appearance of an unchecked `Label`, and a `Style` with the key of "selectedStyle" that is based on "baseStyle" but defines the appearance of a checked `Label`. The `Resources` collection concludes with an implicit style for `Label` that is the same as "baseStyle":

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

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="Label">
                <Setter Property="TextColor" Value="Gray" />
                <Setter Property="FontSize" Value="Small" />
                <Setter Property="HorizontalTextAlignment" Value="Center" />
                <Setter Property="VerticalTextAlignment" Value="Center" />
            </Style>

            <Style x:Key="selectedStyle" TargetType="Label"
                   BasedOn="{StaticResource baseStyle}">
                <Setter Property="TextColor" Value="Green" />
                <Setter Property="FontSize" Value="Medium" />
                <Setter Property="FontAttributes" Value="Bold,Italic" />
            </Style>

            <!-- Implicit style -->
            <Style TargetType="Label" BasedOn="{StaticResource baseStyle}" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Grid>
            <Label Text="Small"
                   Grid.Column="0">
                <Label.Behaviors>
                    <toolkit:RadioBehavior x:Name="smallRadio" />
                </Label.Behaviors>

                <Label.Triggers>
                    <DataTrigger TargetType="Label"
                                 Binding="{Binding Source={x:Reference smallRadio},
                                                   Path=IsChecked}"
                                 Value="True">
                        <Setter Property="Style" Value="{StaticResource selectedStyle}" />
                    </DataTrigger>
                </Label.Triggers>
```

```
            </Label>

            <Label Text="Medium"
                   Grid.Column="1">
                <Label.Behaviors>
                    <toolkit:RadioBehavior x:Name="mediumRadio"
                                           IsChecked="True" />
                </Label.Behaviors>

                <Label.Triggers>
                    <DataTrigger TargetType="Label"
                                 Binding="{Binding Source={x:Reference mediumRadio},
                                                   Path=IsChecked}"
                                 Value="True">
                        <Setter Property="Style" Value="{StaticResource selectedStyle}" />
                    </DataTrigger>
                </Label.Triggers>
            </Label>

            <Label Text="Large"
                   Grid.Column="2">
                <Label.Behaviors>
                    <toolkit:RadioBehavior x:Name="largeRadio" />
                </Label.Behaviors>

                <Label.Triggers>
                    <DataTrigger TargetType="Label"
                                 Binding="{Binding Source={x:Reference largeRadio},
                                                   Path=IsChecked}"
                                 Value="True">
                        <Setter Property="Style" Value="{StaticResource selectedStyle}" />
                    </DataTrigger>
                </Label.Triggers>
            </Label>
        </Grid>

        <Grid VerticalOptions="CenterAndExpand"
              HorizontalOptions="Center">

            <Image Source="{local:ImageResource RadioStyle.Images.tee200.png}"
                   IsVisible="{Binding Source={x:Reference smallRadio},
                                       Path=IsChecked}" />

            <Image Source="{local:ImageResource RadioStyle.Images.tee250.png}"
                   IsVisible="{Binding Source={x:Reference mediumRadio},
                                       Path=IsChecked}" />

            <Image Source="{local:ImageResource RadioStyle.Images.tee300.png}"
                   IsVisible="{Binding Source={x:Reference largeRadio},
                                       Path=IsChecked}" />
        </Grid>
    </StackLayout>
</ContentPage>
```

Prior to this chapter, `Setter` objects were only found in `Style` definitions, so it might seem a little odd to see a `Setter` object in the `DataTrigger` that sets the `Style` property for the `Label`. But the screenshots demonstrate that it works fine. Now the selected item is in a larger font with bold and italic in addition to a different color:



You might also have fun creating new types of visuals to identify the selected item in a group of radio buttons. The **RadioImages** program contains four bitmaps indicating different modes of transportation. The `Image` elements that reference these bitmaps are each a child of a `ContentView` to which is attached the `RadioBehavior` and a `DataTrigger` that changes the color of the `ContentView`:

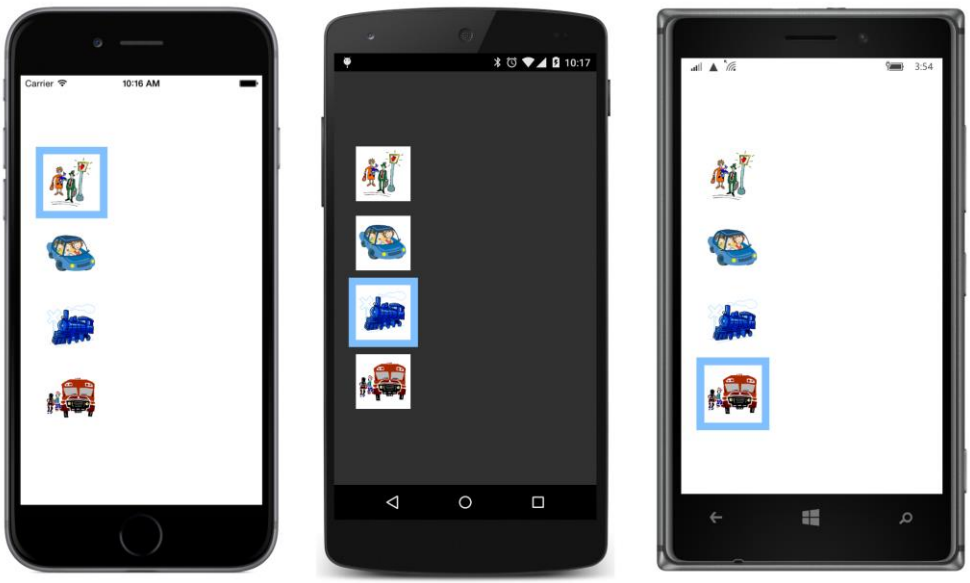
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:RadioImages"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"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

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="ContentView">
                <Setter Property="WidthRequest"    Value="75" />
                <Setter Property="HeightRequest"   Value="75" />
                <Setter Property="Padding"         Value="10" />
            </Style>

            <Color x:Key="selectedColor">#80C0FF</Color>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout HorizontalOptions="Start"
```

```
            VerticalOptions="Center"
            Padding="20, 0"
            Spacing="0">
        <ContentView>
            <ContentView.Behaviors>
                <toolkit:RadioBehavior x:Name="pedestrianRadio" />
            </ContentView.Behaviors>

            <ContentView.Triggers>
                <DataTrigger TargetType="ContentView"
                             Binding="{Binding Source={x:Reference pedestrianRadio},
                                               Path=IsChecked}"
                             Value="True">
                    <Setter Property="BackgroundColor" Value="{StaticResource selectedColor}" />
                </DataTrigger>
            </ContentView.Triggers>

            <Image Source="{local:ImageResource RadioImages.Images.pedestrian.png}" />
        </ContentView>

        <ContentView>
            <ContentView.Behaviors>
                <toolkit:RadioBehavior x:Name="carRadio" />
            </ContentView.Behaviors>

            <ContentView.Triggers>
                <DataTrigger TargetType="ContentView"
                             Binding="{Binding Source={x:Reference carRadio},
                                               Path=IsChecked}"
                             Value="True">
                    <Setter Property="BackgroundColor" Value="{StaticResource selectedColor}" />
                </DataTrigger>
            </ContentView.Triggers>

            <Image Source="{local:ImageResource RadioImages.Images.car.png}" />
        </ContentView>

        <ContentView>
            <ContentView.Behaviors>
                <toolkit:RadioBehavior x:Name="trainRadio" />
            </ContentView.Behaviors>

            <ContentView.Triggers>
                <DataTrigger TargetType="ContentView"
                             Binding="{Binding Source={x:Reference trainRadio},
                                               Path=IsChecked}"
                             Value="True">
                    <Setter Property="BackgroundColor" Value="{StaticResource selectedColor}" />
                </DataTrigger>
            </ContentView.Triggers>

            <Image Source="{local:ImageResource RadioImages.Images.train.png}" />
        </ContentView>
```

```
        <ContentView>
            <ContentView.Behaviors>
                <toolkit:RadioBehavior x:Name="busRadio" />
            </ContentView.Behaviors>

            <ContentView.Triggers>
                <DataTrigger TargetType="ContentView"
                             Binding="{Binding Source={x:Reference busRadio},
                                               Path=IsChecked}"
                             Value="True">
                    <Setter Property="BackgroundColor" Value="{StaticResource selectedColor}" />
                </DataTrigger>
            </ContentView.Triggers>

            <Image Source="{local:ImageResource RadioImages.Images.bus.png}" />
        </ContentView>
    </StackLayout>
</ContentPage>
```

Sometimes, you'll want to set an initial selected item by setting the `IsChecked` property on one of the `RadioBehavior` objects to `true`, and sometimes not. This program leaves them all unchecked at program startup, but once the user selects one of the items, there is no way to unselect them all.

The crucial factor in this scheme is that the `ContentView` is given a significant `Padding` value so it seems to surround the `Image` element when that item is selected:



Of course, even with just four items, the repetitive markup looks a bit ominous. You could derive a class from `ContentView` to consolidate the `RadioBehavior` and `DataTrigger` interaction, but you'd need to define a property on this derived class to specify the particular bitmap associated with the but-

ton, and very likely another property or an event to indicate when that item has been selected. Generally, it's easier to keep the markup for each radio button to a minimum by defining common properties using a `Style` or other resources.

If you want to create more traditional radio button visuals, that's possible as well. The Unicode characters \u25CB and \u25C9 resemble the traditional unchecked and checked radio button circles and dots.

The **TraditionalRadios** program has six radio buttons, but they are divided into two groups of three buttons each, so the `GroupName` properties need to be set for at least one of the two groups. The program chooses to set the `GroupName` for *all* the radio buttons to either "platformGroup" or "languageGroup". Each `RadioBehavior` is attached to a horizontal `StackLayout` that contains one `Label` with a `DataTrigger` that switches between the "○" and "◉" strings, and a second `Label` that displays the text to the right of that symbol:

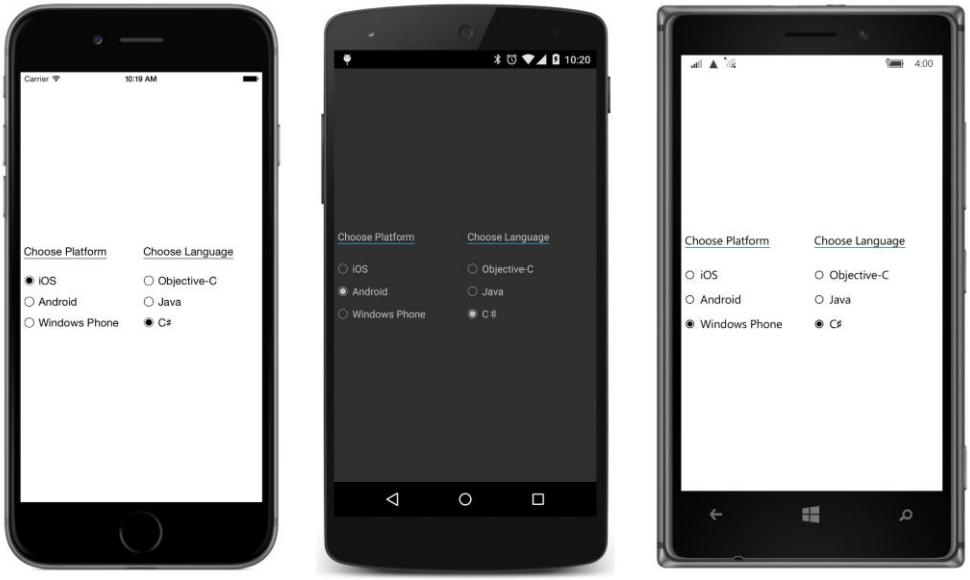
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="TraditionalRadios.TraditionalRadiosPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <x:String x:Key="uncheckedRadio">&#x25CB;</x:String>
            <x:String x:Key="checkedRadio">&#x25C9;</x:String>
        </ResourceDictionary>
    </ContentPage.Resources>

    <Grid VerticalOptions="Center" Padding="5, 0">
        <!-- Left column -->
        <StackLayout Grid.Column="0" Spacing="24">

            <!-- Header -->
            <StackLayout HorizontalOptions="Start" Spacing="0">
                <Label Text="Choose Platform" />
                <BoxView Color="Accent" HeightRequest="1" />
            </StackLayout>

            <!-- Stack of radio buttons -->
            <StackLayout Spacing="12">

                <StackLayout Orientation="Horizontal">
                    <StackLayout.Behaviors>
                        <toolkit:RadioBehavior x:Name="iosRadio"
                                               GroupName="platformGroup" />
                    </StackLayout.Behaviors>

                    <Label Text="{StaticResource uncheckedRadio}">
                        <Label.Triggers>
                            <DataTrigger TargetType="Label"
                                         Binding="{Binding Source={x:Reference iosRadio},
                                                           Path=IsChecked}"
```

```
                                Value="True">
                            <Setter Property="Text" Value="{StaticResource checkedRadio}" />
                        </DataTrigger>
                    </Label.Triggers>
                </Label>
                <Label Text="iOS" />
            </StackLayout>

            <StackLayout Orientation="Horizontal">
                <StackLayout.Behaviors>
                    <toolkit:RadioBehavior x:Name="androidRadio"
                                           GroupName="platformGroup" />
                </StackLayout.Behaviors>

                <Label Text="{StaticResource uncheckedRadio}">
                    <Label.Triggers>
                        <DataTrigger TargetType="Label"
                                     Binding="{Binding Source={x:Reference androidRadio},
                                                       Path=IsChecked}"
                                     Value="True">
                            <Setter Property="Text" Value="{StaticResource checkedRadio}" />
                        </DataTrigger>
                    </Label.Triggers>
                </Label>
                <Label Text="Android" />
            </StackLayout>

            <StackLayout Orientation="Horizontal">
                <StackLayout.Behaviors>
                    <toolkit:RadioBehavior x:Name="winPhoneRadio"
                                           GroupName="platformGroup" />
                </StackLayout.Behaviors>

                <Label Text="{StaticResource uncheckedRadio}">
                    <Label.Triggers>
                        <DataTrigger TargetType="Label"
                                     Binding="{Binding Source={x:Reference winPhoneRadio},
                                                       Path=IsChecked}"
                                     Value="True">
                            <Setter Property="Text" Value="{StaticResource checkedRadio}" />
                        </DataTrigger>
                    </Label.Triggers>
                </Label>
                <Label Text="Windows Phone" />
            </StackLayout>
        </StackLayout>
    </StackLayout>
    <!-- Left column -->
    <StackLayout Grid.Column="1" Spacing="24">

        <!-- Header -->
        <StackLayout HorizontalOptions="Start" Spacing="0">
            <Label Text="Choose Language" />
            <BoxView Color="Accent" HeightRequest="1" />
```

```
</StackLayout>

<!-- Stack of radio buttons -->
<StackLayout Spacing="12">

    <StackLayout Orientation="Horizontal">
        <StackLayout.Behaviors>
            <toolkit:RadioBehavior x:Name="objectiveCRadio"
                                   GroupName="languageGroup" />
        </StackLayout.Behaviors>

        <Label Text="{StaticResource uncheckedRadio}">
            <Label.Triggers>
                <DataTrigger TargetType="Label"
                             Binding="{Binding Source={x:Reference objectiveCRadio},
                                               Path=IsChecked}"
                             Value="True">
                    <Setter Property="Text" Value="{StaticResource checkedRadio}" />
                </DataTrigger>
            </Label.Triggers>
        </Label>
        <Label Text="Objective-C" />
    </StackLayout>

    <StackLayout Orientation="Horizontal">
        <StackLayout.Behaviors>
            <toolkit:RadioBehavior x:Name="javaRadio"
                                   GroupName="languageGroup" />
        </StackLayout.Behaviors>

        <Label Text="{StaticResource uncheckedRadio}">
            <Label.Triggers>
                <DataTrigger TargetType="Label"
                             Binding="{Binding Source={x:Reference javaRadio},
                                               Path=IsChecked}"
                             Value="True">
                    <Setter Property="Text" Value="{StaticResource checkedRadio}" />
                </DataTrigger>
            </Label.Triggers>
        </Label>
        <Label Text="Java" />
    </StackLayout>

    <StackLayout Orientation="Horizontal">
        <StackLayout.Behaviors>
            <toolkit:RadioBehavior x:Name="cSharpRadio"
                                   GroupName="languageGroup" />
        </StackLayout.Behaviors>

        <Label Text="{StaticResource uncheckedRadio}">
            <Label.Triggers>
                <DataTrigger TargetType="Label"
                             Binding="{Binding Source={x:Reference cSharpRadio},
                                               Path=IsChecked}"
```

```
                                    Value="True">
                                <Setter Property="Text" Value="{StaticResource checkedRadio}" />
                            </DataTrigger>
                        </Label.Triggers>
                    </Label>
                    <Label Text="C&#x266F;" />
                </StackLayout>
            </StackLayout>
        </StackLayout>
    </Grid>
</ContentPage>
```

In the context of modern user interfaces, these radio buttons look very quaint and old-fashioned, but at the same time quite authentic:



Fades and orientation

Already in this book, you've seen a couple of color-selection programs that let you interactively form a color by using three `Slider` elements. The final sample in this chapter is yet another color-selection program, but this one gives you options: It contains three radio buttons (actually, simple `Label` elements) labeled "RGB Hex", "RGB Float", and "HSL". These allow you to select a color in three different ways:

- As red, green, and blue hexadecimal values ranging from 00 to FF.
- As red, green, and blue floating-point values ranging from 0 to 1.
- As hue, saturation, and luminosity floating-point values ranging from 0 to 1.

It might at first seem complex to switch between these three options. You might imagine that code is required to redefine the range of the `Slider` elements and to reformat the text that is displayed to show the values. However, you can actually define the entire user interface in XAML.

The first trick is that the XAML file actually contains nine `Slider` elements with accompanying `Label` elements to display the values. Each set of three `Slider` and `Label` elements occupies a `StackLayout` with its `IsVisible` property bound to one of the `RadioBehavior` objects attached to the three radio buttons. The three `StackLayout` elements occupy a single-cell `Grid`, much like the pictures of the T-shirts in the **RadioLabels** and **RadioStyle** programs.

But let's make it more challenging: When you select one of the radio buttons, you probably expect one set of three `Slider` and `Label` elements to be replaced by another. Let's instead have the former set fade out and the new set fade in.

How can this be done?

Let's build the markup. If you just wanted to replace one `StackLayout` with another, you would bind the `IsVisible` property of the `StackLayout` to the `IsChecked` property of the corresponding `RadioBehavior`:

```
<StackLayout IsVisible="{Binding Source={x:Reference hexRadio},
                                 Path=IsChecked}">

    <!-- Trio of Slider and Label elements -->

</StackLayout>
```

To instead fade out the old and fade in the new, you would first need to initialize the `IsVisible` property of the `StackLayout` to `False` and attach a `DataTrigger` that references the `IsChecked` property of the `RadioBehavior`:

```
<StackLayout IsVisible="False">
    <StackLayout.Triggers>
        <DataTrigger TargetType="StackLayout"
                     Binding="{Binding Source={x:Reference hexRadio},
                                       Path=IsChecked}"
                     Value="True">

            …

        </DataTrigger>
    </StackLayout.Triggers>

    <!-- Trio of Slider and Label elements -->

</StackLayout>
```

Then, instead of adding a `Setter` or two to the `DataTrigger`, you need to add an `Action` derivative to the `EnterActions` and `ExitActions` collections:

```
<StackLayout IsVisible="False">
```

```
    <StackLayout.Triggers>
        <DataTrigger TargetType="StackLayout"
                     Binding="{Binding Source={x:Reference hexRadio},
                                       Path=IsChecked}"
                     Value="True">
            <DataTrigger.EnterActions>
                <toolkit:FadeEnableAction Enable="True" />
            </DataTrigger.EnterActions>

            <DataTrigger.ExitActions>
                <toolkit:FadeEnableAction Enable="False" />
            </DataTrigger.ExitActions>
        </DataTrigger>
    </StackLayout.Triggers>

    <!-- Trio of Slider and Label elements -->

</StackLayout>
```

As you'll recall, the `EnterActions` are invoked when the condition becomes true (which in this case is when the `IsChecked` property of the corresponding `RadioBehavior` is `True`), and the `ExitActions` are invoked when the condition becomes false.

This hypothetical `FadeEnableAction` class has a Boolean property named `Enable`. When the `Enable` property is `True`, we want `FadeEnableAction` to use the `FadeTo` extension method to animate the `Opacity` property from 0 (invisible) to 1 (fully visible). When `Enable` is `False`, we want `FadeTo` to animate the `Opacity` from 1 to 0. Keep in mind that as one `StackLayout` (and its children) fades out, another one simultaneously fades in.

However, the `StackLayout` won't be visible at all unless `FadeEnableAction` begins by setting `IsVisible` to `true` when `Enable` is set to `True`. Similarly, when `Enable` is set to `False`, `FadeEnableAction` must conclude by setting `IsVisible` back to `false`.

During the transition between two sets of `Slider` and `Label` elements, you probably don't want both sets responding to user input. For this reason, `FadeEnableAction` must also manipulate the `IsEnabled` property of the `StackLayout`, which enables or disables all its children. Since two animations will be going on simultaneously—as one `StackLayout` fades out and the other fades in—it makes sense to change the `IsEnabled` property halfway through the animation.

Here is a `FadeEnableAction` class in **Xamarin.FormsBook.Toolkit** that satisfies all these criteria:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class FadeEnableAction : TriggerAction<VisualElement>
    {
        public FadeEnableAction()
        {
            Length = 500;
        }

        public bool Enable { set; get; }
```

```
        public int Length { set; get; }

        async protected override void Invoke(VisualElement view)
        {
            if (Enable)
            {
                // Transition to visible and enabled.
                view.IsVisible = true;
                view.Opacity = 0;
                await view.FadeTo(0.5, (uint)Length / 2);
                view.IsEnabled = true;
                await view.FadeTo(1, (uint)Length / 2);
            }
            else
            {
                // Transition to invisible and disabled.
                view.Opacity = 1;
                await view.FadeTo(0.5, (uint)Length / 2);
                view.IsEnabled = false;
                await view.FadeTo(0, (uint)Length / 2);
                view.IsVisible = false;
            }
        }
    }
}
```

Let's give ourselves yet another challenge. In Chapter 17, "Mastering the Grid," in the section "Responding to orientation changes," you saw how to use the `Grid` to change your layout between portrait and landscape modes. Basically, all the layout on the page is divided roughly in half, and becomes two children of a `Grid`. In portrait mode, those two children go in two rows of the `Grid`, and in landscape mode, they go into two columns.

Can something like this be handled by a behavior? Accommodating a generalized response to orientation would be hard, but a simple approach might be to assume that in portrait mode, the second row should be autosized while the first row uses the rest of the available space. In landscape mode, the screen is simply divided equally in half. This is how the **GridRgbSliders** program in Chapter 17 worked, and also the **MandelbrotXF** program in Chapter 20.

The following `GridOrientationBehavior` can be attached only to a `Grid`. The `Grid` must *not* have any row definitions or column definitions defined—the behavior takes care of that—and it must contain only two children. The behavior monitors the `SizeChanged` event of the `Grid`. When that size changes, the `Behavior` sets the row and column definitions of the `Grid` and the row and column settings of the two children of the `Grid`:

```
namespace Xamarin.FormsBook.Toolkit
{
    // Assumes Grid with two children without any
    //      row or column definitions set.
    public class GridOrientationBehavior : Behavior<Grid>
    {
```

```
protected override void OnAttachedTo(Grid grid)
{
    base.OnAttachedTo(grid);

    // Add row and column definitions.
    grid.RowDefinitions.Add(new RowDefinition());
    grid.RowDefinitions.Add(new RowDefinition());
    grid.ColumnDefinitions.Add(new ColumnDefinition());
    grid.ColumnDefinitions.Add(new ColumnDefinition());

    grid.SizeChanged += OnGridSizeChanged;
}
protected override void OnDetachingFrom(Grid grid)
{
    base.OnDetachingFrom(grid);
    grid.SizeChanged -= OnGridSizeChanged;
}

private void OnGridSizeChanged(object sender, EventArgs args)
{
    Grid grid = (Grid)sender;

    if (grid.Width <= 0 || grid.Height <= 0)
        return;

    // Portrait mode
    if (grid.Height > grid.Width)
    {
        // Set row definitions.
        grid.RowDefinitions[0].Height = new GridLength(1, GridUnitType.Star);
        grid.RowDefinitions[1].Height = GridLength.Auto;

        // Set column definitions.
        grid.ColumnDefinitions[0].Width = new GridLength(1, GridUnitType.Star);
        grid.ColumnDefinitions[1].Width = new GridLength(0);

        //Position first child.
        Grid.SetRow(grid.Children[0], 0);
        Grid.SetColumn(grid.Children[0], 0);

        // Position second child.
        Grid.SetRow(grid.Children[1], 1);
        Grid.SetColumn(grid.Children[1], 0);
    }
    // Landscape mode
    else
    {
        // Set row definitions.
        grid.RowDefinitions[0].Height = new GridLength(1, GridUnitType.Star);
        grid.RowDefinitions[1].Height = new GridLength(0);

        // Set column definitions.
        grid.ColumnDefinitions[0].Width = new GridLength(1, GridUnitType.Star);
        grid.ColumnDefinitions[1].Width = new GridLength(1, GridUnitType.Star);
```

```
                //Position first child.
                Grid.SetRow(grid.Children[0], 0);
                Grid.SetColumn(grid.Children[0], 0);

                // Position second child.
                Grid.SetRow(grid.Children[1], 0);
                Grid.SetColumn(grid.Children[1], 1);
            }
        }
    }
}
```

Now let's put it all together in a program call **MultiColorSliders**. The backbone of the program is the `ColorViewModel` introduced in Chapter 18, "MVVM," and can be found in the **Xamarin.FormsBook.Toolkit** library. An instance of `ColorViewModel` is set as the `BindingContext` of the `Grid` that contains all the content of the page. The three sets of `Slider` and `Label` elements all contain bindings to the `Red`, `Green`, `Blue`, `Hue`, `Saturation`, and `Luminosity` properties of that ViewModel. For the hexadecimal option, the `DoubleToIntConverter` introduced in Chapter 17 converts from the `double` values of the `Red`, `Green`, and `Blue` properties to integers with a multiplication by 255 for display by each `Label`.

Here is the XAML file. It's rather long because it contains three sets of three `Slider` and `Label` elements, but several comments help to guide you through the various sections:

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

    <ContentPage.Resources>
        <ResourceDictionary>
            <toolkit:ColorViewModel x:Key="colorViewModel" />

            <toolkit:DoubleToIntConverter x:Key="doubleToInt" />

            <Style x:Key="baseStyle" TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>

            <Style x:Key="unselectedStyle" TargetType="Label"
                   BasedOn="{StaticResource baseStyle}">
                <Setter Property="TextColor" Value="Gray" />
            </Style>

            <Style x:Key="selectedStyle" TargetType="Label"
                   BasedOn="{StaticResource baseStyle}">
```

```
            <Setter Property="TextColor" Value="Accent" />
            <Setter Property="Scale" Value="1.5" />
        </Style>

        <!-- Implicit style for labels underneath sliders -->
        <Style TargetType="Label" BasedOn="{StaticResource baseStyle}" />
    </ResourceDictionary>
</ContentPage.Resources>

<Grid>
    <Grid.BindingContext>
        <toolkit:ColorViewModel Alpha="1" />
    </Grid.BindingContext>

    <!-- The GridOrientationBehavior takes care of the row and
            column definitions, and the row and column settings
            of the two Grid children. -->
    <Grid.Behaviors>
        <toolkit:GridOrientationBehavior />
    </Grid.Behaviors>

    <!-- First child of Grid is on top or at left. -->
    <BoxView Color="{Binding Color}" />

    <!-- Second child of Grid is on bottom or at right. -->
    <StackLayout Padding="10">

        <!-- Three-column Grid for radio labels -->
        <Grid>
            <Label Text="RGB Hex" Grid.Column="0"
                   Style="{StaticResource unselectedStyle}">
                <Label.Behaviors>
                    <toolkit:RadioBehavior x:Name="hexRadio"
                                           IsChecked="true" />
                </Label.Behaviors>

                <Label.Triggers>
                    <DataTrigger TargetType="Label"
                                 Binding="{Binding Source={x:Reference hexRadio},
                                                   Path=IsChecked}"
                                 Value="True">
                        <Setter Property="Style" Value="{StaticResource selectedStyle}" />
                    </DataTrigger>
                </Label.Triggers>
            </Label>

            <Label Text="RGB Float" Grid.Column="1"
                   Style="{StaticResource unselectedStyle}">
                <Label.Behaviors>
                    <toolkit:RadioBehavior x:Name="floatRadio" />
                </Label.Behaviors>

                <Label.Triggers>
                    <DataTrigger TargetType="Label"
```

```
                            Binding="{Binding Source={x:Reference floatRadio},
                                              Path=IsChecked}"
                            Value="True">
                        <Setter Property="Style" Value="{StaticResource selectedStyle}" />
                    </DataTrigger>
                </Label.Triggers>
            </Label>

            <Label Text="HSL" Grid.Column="2"
                   Style="{StaticResource unselectedStyle}">
                <Label.Behaviors>
                    <toolkit:RadioBehavior x:Name="hslRadio" />
                </Label.Behaviors>

                <Label.Triggers>
                    <DataTrigger TargetType="Label"
                            Binding="{Binding Source={x:Reference hslRadio},
                                              Path=IsChecked}"
                            Value="True">
                        <Setter Property="Style" Value="{StaticResource selectedStyle}" />
                    </DataTrigger>
                </Label.Triggers>
            </Label>
        </Grid>

        <!-- Single-cell Grid for three sets of sliders and labels -->
        <Grid>

            <!-- StackLayout for RGB Hex sliders and labels -->
            <StackLayout>
                <StackLayout.Triggers>
                    <DataTrigger TargetType="StackLayout"
                            Binding="{Binding Source={x:Reference hexRadio},
                                              Path=IsChecked}"
                            Value="True">
                        <DataTrigger.EnterActions>
                            <toolkit:FadeEnableAction Enable="True" />
                        </DataTrigger.EnterActions>

                        <DataTrigger.ExitActions>
                            <toolkit:FadeEnableAction Enable="False" />
                        </DataTrigger.ExitActions>
                    </DataTrigger>
                </StackLayout.Triggers>

                <Slider Value="{Binding Red, Mode=TwoWay}" />

                <Label Text="{Binding Red, StringFormat='Red = {0:X2}',
                                      Converter={StaticResource doubleToInt},
                                      ConverterParameter=255}" />

                <Slider Value="{Binding Green, Mode=TwoWay}" />

                <Label Text="{Binding Green, StringFormat='Green = {0:X2}',
```

```
                            Converter={StaticResource doubleToInt},
                            ConverterParameter=255}" />

        <Slider Value="{Binding Blue, Mode=TwoWay}" />

        <Label Text="{Binding Blue, StringFormat='Blue = {0:X2}',
                            Converter={StaticResource doubleToInt},
                            ConverterParameter=255}" />
    </StackLayout>

    <!-- StackLayout for RGB float sliders and labels -->
    <StackLayout IsVisible="False">
        <StackLayout.Triggers>
            <DataTrigger TargetType="StackLayout"
                         Binding="{Binding Source={x:Reference floatRadio},
                                           Path=IsChecked}"
                         Value="True">
            <DataTrigger.EnterActions>
                <toolkit:FadeEnableAction Enable="True" />
            </DataTrigger.EnterActions>

            <DataTrigger.ExitActions>
                <toolkit:FadeEnableAction Enable="False" />
            </DataTrigger.ExitActions>
        </DataTrigger>
        </StackLayout.Triggers>

        <Slider Value="{Binding Red, Mode=TwoWay}" />
        <Label Text="{Binding Red, StringFormat='Red = {0:F2}'}" />
        <Slider Value="{Binding Green, Mode=TwoWay}" />
        <Label Text="{Binding Green, StringFormat='Green = {0:F2}'}" />
        <Slider Value="{Binding Blue, Mode=TwoWay}" />
        <Label Text="{Binding Blue, StringFormat='Blue = {0:F2}'}" />
    </StackLayout>

    <!-- StackLayout for HSL sliders and labels -->
    <StackLayout IsVisible="False">
        <StackLayout.Triggers>
            <DataTrigger TargetType="StackLayout"
                         Binding="{Binding Source={x:Reference hslRadio},
                                           Path=IsChecked}"
                         Value="True">
                <DataTrigger.EnterActions>
                    <toolkit:FadeEnableAction Enable="True" />
                </DataTrigger.EnterActions>

                <DataTrigger.ExitActions>
                    <toolkit:FadeEnableAction Enable="False" />
                </DataTrigger.ExitActions>
            </DataTrigger>
        </StackLayout.Triggers>

        <!-- Trio of Slider and Label elements -->
```

```
                    <Slider Value="{Binding Hue, Mode=TwoWay}" />
                    <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />
                    <Slider Value="{Binding Saturation, Mode=TwoWay}" />
                    <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />
                    <Slider Value="{Binding Luminosity, Mode=TwoWay}" />
                    <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
                </StackLayout>
            </Grid>
        </StackLayout>
    </Grid>
</ContentPage>
```

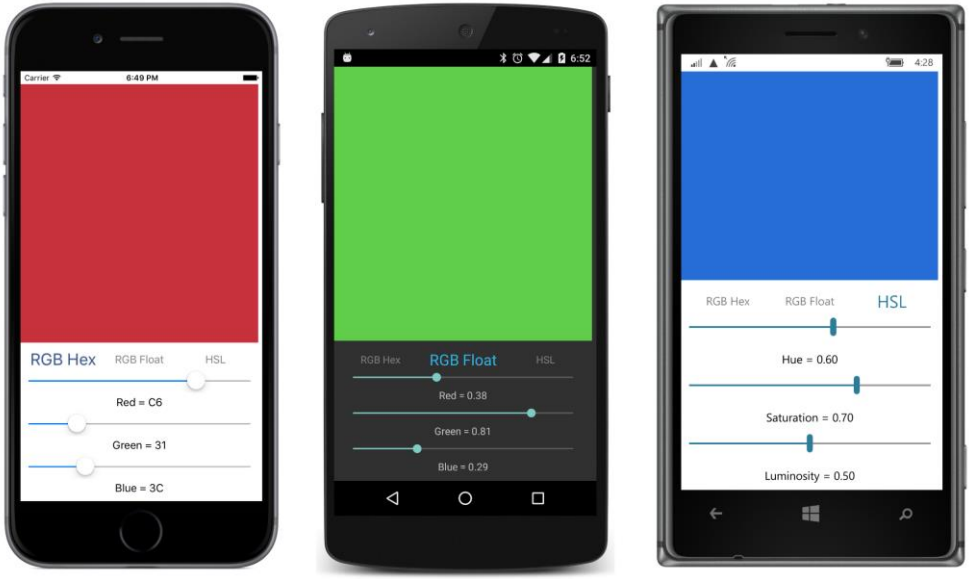
You might recall that the `ColorViewModel` class introduced in Chapter 18 rounded the color components, both coming into and going out of the ViewModel. **MultiColorSliders** happens to be the program that revealed a problem with the unrounded values. Here's the problem:

For Android, Xamarin.Forms implements the `Slider` using a `SeekBar`, and the Android `SeekBar` only has integer `Progress` values ranging from 0 to the integer `Max` property. To convert to the floating-point `Value` property of the `Slider`, Xamarin.Forms sets the `Max` property of the `SeekBar` to 1000 and then performs a calculation based on the `Minimum` and `Maximum` properties of the `Slider`. This means that when `Minimum` and `Maximum` have their default values of 0 and 1, respectively, the `Value` property only increases in increments of 0.001, and is always representable with three decimal places.

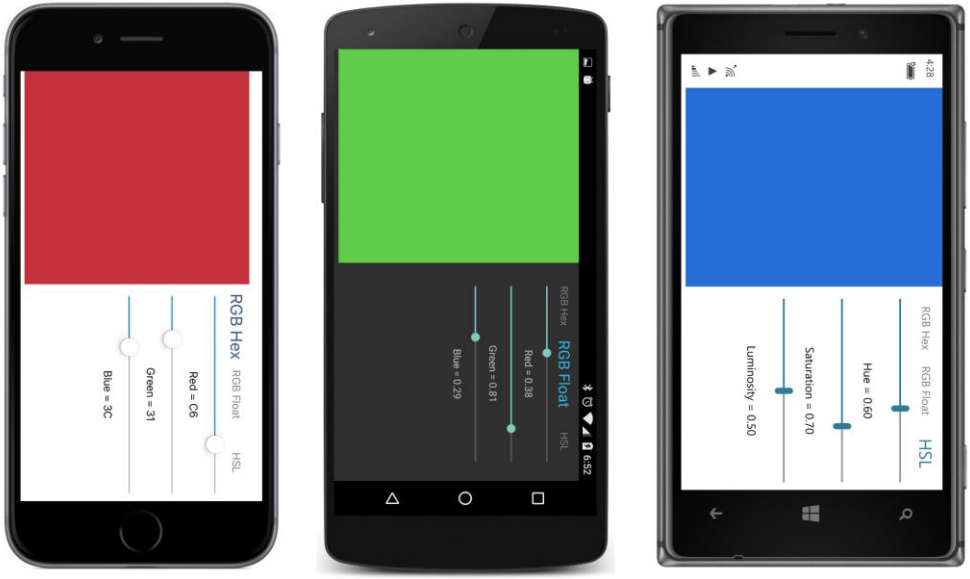
However, the `ColorViewModel` uses the `Color` structure to convert between RGB and HSL representations, and in this particular program all the properties representing RGB and HSL values are bound to `Slider` elements. Even if the values of the `Red`, `Green`, and `Blue` properties set by the `Slider` elements are rounded to the nearest 0.001, the resultant `Hue`, `Saturation`, and `Luminosity` values will have more than three decimal places. If these values are not rounded by the ViewModel, that's an issue. When the `Value` properties of the `Slider` elements are set from these values, the `Slider` effectively rounds them to three decimal places and then triggers a `PropertyChanged` event that the `ColorViewModel` responds to by creating a new `Color`, which results in new `Red`, `Green`, and `Blue` properties, and an infinite loop ensues.

The solution—as you saw in Chapter 18—was to add rounding to the `ColorViewModel`. That avoids the infinite loop.

Here's the program running in portrait mode. Each platform shows a different option selected, but you'll have to run the program yourself to see the fading animation:



Turn this book (or your computer screen or perhaps your head) sideways, and you'll see how the program responds to landscape mode:



Perhaps the best part of the **MultiColorSliders** program is the code-behind file, which contains merely a call to `InitializeComponent`:

```
namespace MultiColorSliders
{
```

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

There is, of course, a considerable amount of code support in **MultiColorSliders**, consisting of two `Behavior<T>` derivatives, an `Action<T>` derivative, an `IValueConverter` implementation, and an `INotifyPropertyChanged` implementation that functions as a ViewModel.

However, all this code is isolated in reusable components, which makes this program a model of MVVM design philosophy.