

## Chapter 8

# Code and XAML in Harmony

A code file and a XAML file always exist as a pair. The two files complement each other. Despite being referred to as the “code-behind” file to the XAML, very often the code is prominent in taking on the more active and interactive parts of the application. This implies that the code-behind file must be able to refer to elements defined in XAML with as much ease as objects instantiated in code. Likewise, elements in XAML must be able to fire events that are handled in code-based event handlers. That’s what this chapter is all about.

But first let’s explore a couple of unusual techniques for instantiating objects in a XAML file.

## Passing Arguments

---

As you’ve seen, the XAML parser instantiates elements by calling the parameterless constructor of the corresponding class or structure, and then initializes the resultant object by setting properties from attribute values. This seems reasonable. However, developers using XAML sometimes have a need to instantiate objects with constructors that require arguments, or by calling a static creation method. These needs usually didn’t involve the API itself, but instead involve external data classes referenced by the XAML file and which interact with the API.

The 2009 XAML specification introduced an `x:Arguments` element and an `x:FactoryMethod` attribute for these cases, and `Xamarin.Forms` supports them. These techniques are not often used in ordinary circumstances, but you should see how they work in case the need arises.

## Constructors with Arguments

To pass arguments to a constructor of an element in XAML, the element must be separated into start and end tags. Follow the start tag of the element with `x:Arguments` start and end tags. Within those `x:Arguments` tags, include one or more constructor arguments.

The **ParameteredConstructorDemo** sample demonstrates this technique using three different constructors of the `Color` structure. The constructor with three parameters requires red, green, and blue values ranging from 0 to 1. The constructor with four parameters adds an alpha channel as the fourth parameter (which is set here to 0.5), and the constructor with a single parameter indicates a gray shade from 0 (black) to 1 (white):

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

```

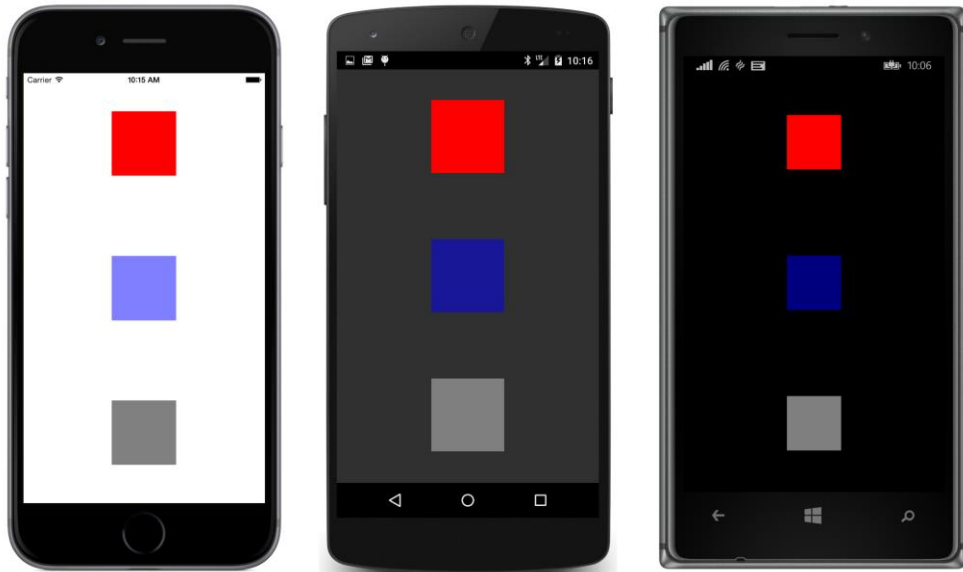
<StackLayout>
  <BoxView WidthRequest="100"
    HeightRequest="100"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand">
    <BoxView.Color>
      <Color>
        <x:Arguments>
          <x:Double>1</x:Double>
          <x:Double>0</x:Double>
          <x:Double>0</x:Double>
        </x:Arguments>
      </Color>
    </BoxView.Color>
  </BoxView>

  <BoxView WidthRequest="100"
    HeightRequest="100"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand">
    <BoxView.Color>
      <Color>
        <x:Arguments>
          <x:Double>0</x:Double>
          <x:Double>0</x:Double>
          <x:Double>1</x:Double>
          <x:Double>0.5</x:Double>
        </x:Arguments>
      </Color>
    </BoxView.Color>
  </BoxView>

  <BoxView WidthRequest="100"
    HeightRequest="100"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand">
    <BoxView.Color>
      <Color>
        <x:Arguments>
          <x:Double>0.5</x:Double>
        </x:Arguments>
      </Color>
    </BoxView.Color>
  </BoxView>
</StackLayout>
</ContentPage>

```

The number of elements within the `x:Arguments` tags, and the types of these elements, must match one of the constructors of the class or structure. You can't just separate arguments with commas or something similar, so the datatype tags defined in the XAML 2009 specification really help. Here's the result:



The blue BoxView is light against the light background and dark against the dark background because it's 50% transparent and lets the background show through.

## Can I Call Methods from XAML?

At one time, the answer to this question was "Don't be ridiculous," but now it's a qualified "Yes." Don't get too excited, though. The only methods you can call in XAML are those that return objects (or values) of the same type as the class (or structure) that defines the method. These methods must be public (obviously) and static. They are sometimes called *creation methods* or *factory methods*. You can instantiate an element in XAML through a call to one of these methods by specifying the method's name using the `x:FactoryMethod` attribute, and its arguments using the `x:Arguments` element.

The `Color` structure defines seven static methods that return `Color` values, so these qualify. This XAML file makes use of three of them:

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

    <StackLayout>
        <BoxView WidthRequest="100"
                HeightRequest="100"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand">
            <BoxView.Color>
                <Color x:FactoryMethod="FromRgb">
                    <x:Arguments>
                        <x:Int32>255</x:Int32>
                        <x:Int32>0</x:Int32>
                    </x:Arguments>
                </Color>
            </BoxView.Color>
        </BoxView>
    </StackLayout>
</ContentPage>
```

```

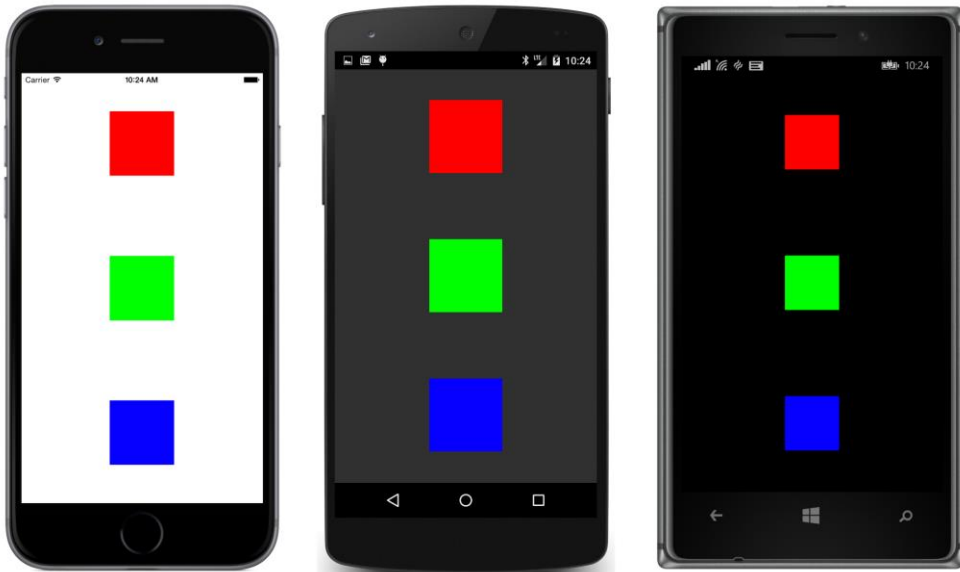
        <x:Int32>0</x:Int32>
    </x:Arguments>
</Color>
</BoxView.Color>
</BoxView>

<BoxView WidthRequest="100"
    HeightRequest="100"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand">
    <BoxView.Color>
        <Color x:FactoryMethod="FromRgb">
            <x:Arguments>
                <x:Double>0</x:Double>
                <x:Double>1.0</x:Double>
                <x:Double>0</x:Double>
            </x:Arguments>
        </Color>
    </BoxView.Color>
</BoxView>

<BoxView WidthRequest="100"
    HeightRequest="100"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand">
    <BoxView.Color>
        <Color x:FactoryMethod="FromHsla">
            <x:Arguments>
                <x:Double>0.67</x:Double>
                <x:Double>1.0</x:Double>
                <x:Double>0.5</x:Double>
                <x:Double>1.0</x:Double>
            </x:Arguments>
        </Color>
    </BoxView.Color>
</BoxView>
</StackLayout>
</ContentPage>

```

The first two static methods invoked here are both named `Color.FromRgb`, but the types of elements within the `x:Arguments` tags distinguish between int arguments that range from 0 to 255 and double argument that range from 0 to 1. The third one is the `Color.FromHsla` method that creates a `Color` value from hue, saturation, luminosity, and alpha components. Interestingly, this is the only way to define a `Color` value from HSL values in a XAML file using the `Xamarin.Forms` API. Here's the result:



## The x:Name Attribute

---

In most real applications, the code-behind file needs to reference elements defined in the XAML file. You saw one way to do it in the **CodePlusXaml** program in the previous chapter: If the code-behind file has knowledge of the layout of the visual tree defined in the XAML file, it can start from the root element (the page itself) and locate specific elements within the tree. This process is called “walking the tree” and can be useful for locating particular elements on a page.

Generally a better approach is to give elements in the XAML file a name similar to a variable name. To do this you use an attribute that is intrinsic to XAML called `Name`. Because the prefix `x` is almost universally used for attributes intrinsic to XAML, this `Name` attribute is commonly referred to as `x:Name`.

The **XamlClock** project demonstrates the use of `x:Name`. Here is the `XamlClockPage.xaml` file containing two `Label` controls named `timeLabel` and `dateLabel`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlClock.XamlClockPage">
    <StackLayout>
        <Label x:Name="timeLabel"
              FontSize="Large"
              HorizontalOptions="Center"
              VerticalOptions="EndAndExpand" />

        <Label x:Name="dateLabel"
              HorizontalOptions="Center"
              VerticalOptions="StartAndExpand" />
    </StackLayout>
</ContentPage>
```

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

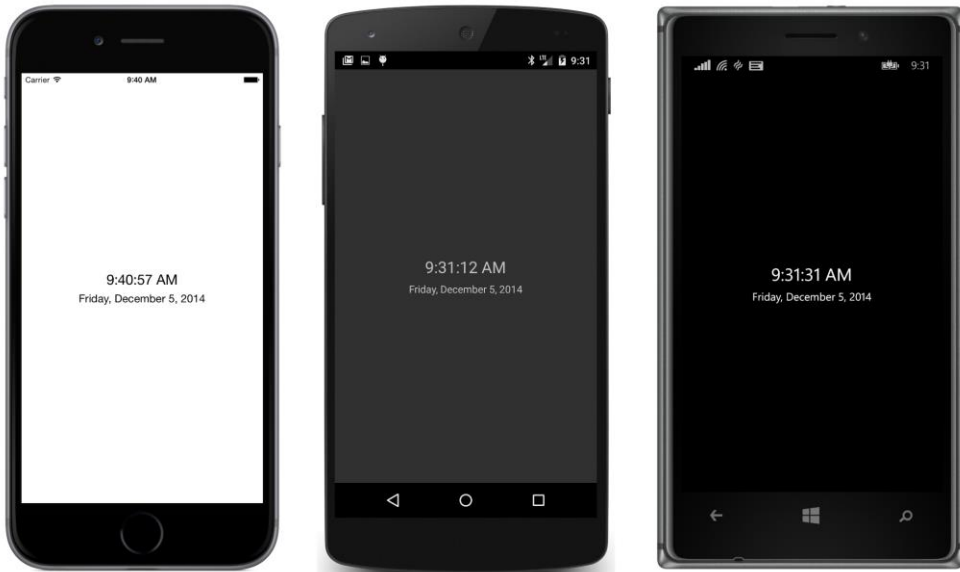
The rules for `x:Name` are the same as for C# variable names. (You'll see why shortly). The name must begin with a letter or underscore, and must contain only letters, underscores, and numbers.

Like the clock program in Chapter 5, **XamlClock** uses `Device.StartTimer` to fire a periodic event for updating the time and date. Here's the `XamlClockPage` code-behind file:

```
namespace XamlClock  
{  
    public partial class XamlClockPage  
    {  
        public XamlClockPage()  
        {  
            InitializeComponent();  
  
            Device.StartTimer(TimeSpan.FromSeconds(1), OnTimerTick);  
        }  
  
        bool OnTimerTick()  
        {  
            DateTime dt = DateTime.Now;  
            timeLabel.Text = dt.ToString("T");  
            dateLabel.Text = dt.ToString("D");  
            return true;  
        }  
    }  
}
```

This timer callback method is called once per second. The method must return `true` to continue the timer. If it returns `false`, the timer stops and must be restarted with another call to `Device.StartTimer`.

The callback method references `timeLabel` and `dateLabel` as if they were normal variables, and sets the `Text` properties of each:



This is not a visually impressive clock, but it's definitely functional.

How is it that the code-behind file can reference the elements identified with `x:Name`? Is it magic? Of course not. The mechanism is very evident when you examine the `XamlClockPage.xaml.g.cs` file that the XAML parser generates from the XAML file as the project is being built:

```
//-----
// <auto-generated>
//   This code was generated by a tool.
//   Runtime Version:4.0.30319.35317
//
//   Changes to this file may cause incorrect behavior and will be lost if
//   the code is regenerated.
// </auto-generated>
//-----

namespace XamlClock {
    using System;
    using Xamarin.Forms;
    using Xamarin.Forms.Xaml;

    public partial class XamlClockPage : ContentPage {

        private Label timeLabel;

        private Label dateLabel;

        private void InitializeComponent() {
            this.LoadFromXaml(typeof(XamlClockPage));
            timeLabel = this.FindByName<Label>("timeLabel");
            dateLabel = this.FindByName<Label>("dateLabel");
        }
    }
}
```

```

    }
}
}

```

As the build-time XAML parser chews through the XAML file, every `x:Name` attribute becomes a private field in this generated code file. This allows code in the code-behind file to reference these names as if they were normal fields—which they definitely are. However, the fields are initially null. Only when `InitializeComponent` is called at runtime are the two fields set via the `FindByName` method, which is defined in the `NameScopeExtensions` class. If the constructor of your code-behind file tries to reference these two fields prior to the `InitializeComponent` call, they will have null values.

This generated code file also implies another rule for `x:Name` values that is now very obvious but rarely stated explicitly: The names cannot duplicate names of fields or properties defined in the code-behind file.

Because these are private fields, they can only be accessed from the code-behind file and not from other classes. If a `ContentPage` derivative must expose public fields or properties to other classes, you must define those yourself.

Obviously `x:Name` values must be unique within a XAML page. This can sometimes be a problem if you're using `OnPlatform` for platform-specific elements in the XAML file. For example, here's a XAML file that expresses the iOS, Android, and WinPhone properties of `OnPlatform` as property elements to select one of three `Label` views:

```

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

    <OnPlatform x:TypeArguments="View">
        <OnPlatform.iOS>
            <Label Text="This is an iOS device"
                  HorizontalOptions="Center"
                  VerticalOptions="Center" />
        </OnPlatform.iOS>

        <OnPlatform.Android>
            <Label Text="This is an Android device"
                  HorizontalOptions="Center"
                  VerticalOptions="Center" />
        </OnPlatform.Android>

        <OnPlatform.WinPhone>
            <Label Text="This is an Windows Phone device"
                  HorizontalOptions="Center"
                  VerticalOptions="Center" />
        </OnPlatform.WinPhone>
    </OnPlatform>
</ContentPage>

```

The `x:TypeArguments` attribute of `OnPlatform` must match the type of the target property exactly. This `OnPlatform` element is implicitly being set to the `Content` property of `ContentPage`, and this `Content`



property is of type `View`, so the `x:TypeArguments` attribute of `OnPlatform` must specify `View`. However, the properties of `OnPlatform` can be set to any class that derives from that type. The objects set to the `iOS`, `Android`, and `WinPhone` properties can in fact be different types just as long as they derive from `View`.

Although that XAML file works, it's not exactly optimum. All three `Label` views are instantiated and initialized but only one is set to the `Content` property of the `ContentPage`. The problem with this approach arises if you need to refer to the `Label` from the code-behind file and you give each of them the same name like so:

*The following XAML file does not work!*

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

    <OnPlatform x:TypeArguments="View">
        <OnPlatform.iOS>
            <Label x:Name="deviceLabel"
                  Text="This is an iOS device"
                  HorizontalOptions="Center"
                  VerticalOptions="Center" />
        </OnPlatform.iOS>

        <OnPlatform.Android>
            <Label x:Name="deviceLabel"
                  Text="This is an Android device"
                  HorizontalOptions="Center"
                  VerticalOptions="Center" />
        </OnPlatform.Android>

        <OnPlatform.WinPhone>
            <Label x:Name="deviceLabel"
                  Text="This is a Windows Phone device"
                  HorizontalOptions="Center"
                  VerticalOptions="Center" />
        </OnPlatform.WinPhone>
    </OnPlatform>
</ContentPage>
```

This will not work because multiple elements have the same name.

You could give them different names and handle the three names in the code-behind file using `Device.OnPlatform`, but a better solution is to keep the platform specificities as small as possible. Here's the version of the **PlatformSpecificLabels** program actually included among the sample code for this chapter. It has a single `Label` and everything is platform-independent except for the `Text` property:

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

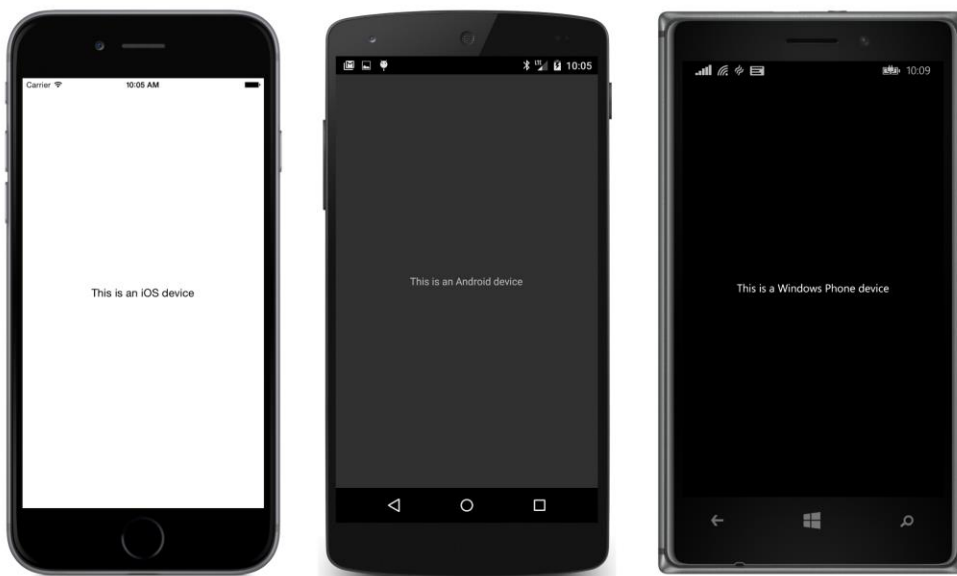
    <Label x:Name="deviceLabel"
           HorizontalOptions="Center"
           Text="This is a Windows Phone device" />
</ContentPage>
```

```

        VerticalOptions="Center">
    <Label.Text>
        <OnPlatform x:TypeArguments="x:String"
            iOS="This is an iOS device"
            Android="This is an Android device"
            WinPhone="This is a Windows Phone device" />
    </Label.Text>
</Label>
</ContentPage>

```

Here's what it looks like:



The Text property is the content property for Label, so you don't need the Label.Text tags in the previous example. This works as well:

```

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

    <Label x:Name="deviceLabel"
        HorizontalOptions="Center"
        VerticalOptions="Center">
        <OnPlatform x:TypeArguments="x:String"
            iOS="This is an iOS device"
            Android="This is an Android device"
            WinPhone="This is a Windows Phone device" />
    </Label>
</ContentPage>

```

Unlike some of the examples shown in the **TextVariations** sample in the previous chapter, there's no problem with superfluous whitespace or end-of-line characters here. All the text strings are delimited

by quotation marks within the `OnPlatform` tag.

## Custom XAML-Based Views

---

The **ScaryColorList** program in the previous chapter listed a few colors in a `StackLayout` using `Frame`, `BoxView` and `Label`. With even just three colors, the repetitive markup was starting to look very ominous. Unfortunately there is no XAML markup that duplicates the `C#` `for` and `while` loops, so your choice is to use code for generating multiple similar items, or to find a better way to do it in markup.

In this book you'll be seeing several ways to list colors in XAML, and eventually this job will become very clean and elegant. But that requires a few more steps into learning `Xamarin.Forms`. Until then, we'll be looking at some other approaches that you might find useful in similar circumstances.

One strategy is to create a custom view that has the sole purpose of displaying a color with a name and a colored box. And while we're at it, let's display the hexadecimal RGB values of the colors as well. Then use that custom view in a XAML page file for the individual colors.

What might this look like in XAML?

Or the better question is: How would you *like* it to look?

If the markup looked something like this, the repetition is not bad at all, and not so much worse than explicitly defining an array of `Color` values in code:

```
<StackLayout>
  <MyColorView Color="Red" />
  <MyColorView Color="Green" />
  <MyColorView Color="Blue" />
  ...
</StackLayout>
```

Well, actually it won't look exactly like that. `MyColorView` is obviously a custom class and not part of the `Xamarin.Forms` API. Therefore it cannot appear in the XAML file without a namespace prefix that is defined in an XML namespace declaration.

With this XML prefix applied, there won't be any confusion about this custom view being part of the `Xamarin.Forms` API, so let's give it a more dignified name of `ColorView` rather than `MyColorView`.

This hypothetical `ColorView` class is an example of a fairly easy custom view because it consists solely of existing views—specifically `Label`, `Frame`, and `BoxView`—arranged in a particular way using `StackLayout`. `Xamarin.Forms` defines a view designed specifically for the purpose of parenting such an arrangement of views, and it's called `ContentView`. Like `ContentPage`, `ContentView` has a `Content` property that you can set to a visual tree of other views. You can define the contents of the `ContentView` in code, but it's more fun to do it in XAML.

Let's put together a solution named **ColorViewList**. This solution will have two sets of XAML and code-behind files: one for a class named `ColorViewListPage` that derives from `ContentPage` (as usual), and

the second for a class named `ColorView` that derives from `ContentView`.

To create the `ColorView` class in Visual Studio, use the same procedure as when adding a new XAML page to the `ColorViewList` project. Right-click the project name in the solution explorer, and select **Add > New Item** from the context menu. In the **Add New Item** dialog, select **Visual C# > Code** at the left and **Forms Xaml Page**. Enter the name `ColorView.cs`. But right away, before you forget, go into the `ColorView.xaml` file and change `ContentPage` in the start and end tags to `ContentView`. In the `ColorView.xaml.cs` file, change the base class to `ContentView`.

The process is a little easier in Xamarin Studio. From the tool menu for the **ColorViewList** project, select **Add > New File**. In the **New File** dialog, select **Forms** at the left and **Forms ContentView Xaml** (not **Forms ContentPage Xaml**). Give it a name of `ColorView`.

You'll also need to create a XAML file and code-behind file for the `ColorViewListPage` class, as usual.

The `ColorView.xaml` file describes the layout of the individual color items but without any actual color values. Instead, the `BoxView` and two `Label` views are given names:

```
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ColorViewList.ColorView">

    <Frame OutlineColor="Accent">
        <StackLayout Orientation="Horizontal">
            <BoxView x:Name="boxView"
                    WidthRequest="70"
                    HeightRequest="70" />

            <StackLayout>
                <Label x:Name="colorNameLabel"
                       FontSize="Large"
                       VerticalOptions="CenterAndExpand" />

                <Label x:Name="colorValueLabel"
                       VerticalOptions="CenterAndExpand" />
            </StackLayout>
        </StackLayout>
    </Frame>
</ContentView>
```

In a real-life program, you'll have plenty of time later to fine-tune the visuals. Initially you'll just want to get all the named views in there.

Besides the visuals, this `ColorView` class will need a new property to set the color. This property must be defined in the code-behind file. At first, it seems reasonable to give `ColorView` a property named `Color` of type `Color` (as the earlier XAML snippet with `MyColorView` seems to suggest). But if this property were of type `Color`, how would the code get the name of the color from that `Color` value? It can't.

Instead, it makes more sense to define a property named `ColorName` of type `string`. The code-behind file can then use reflection to obtain the static field of the `Color` class corresponding to that name.

But wait: Xamarin.Forms includes a public `ColorTypeConverter` class that the XAML parser uses to convert a text color name like “Red” or “Blue” into a `Color` value. Why not take advantage of that?

Here’s the code-behind file for `ColorView`. It defines a `ColorName` property with a set accessor that sets the `Text` property of the `colorNameLabel` to the color name, and then uses `ColorTypeConverter` to convert the name to a `Color` value. This `Color` value is then used to set the `Color` property of `boxView` and the `Text` property of the `colorValueLabel` to the RGB values:

```
public partial class ColorView : ContentView
{
    string colorName;
    ColorTypeConverter colorTypeConv = new ColorTypeConverter();

    public ColorView()
    {
        InitializeComponent();
    }

    public string ColorName
    {
        set
        {
            // Set the name.
            colorName = value;
            colorNameLabel.Text = value;

            // Get the actual Color and set the other views.
            Color color = (Color)colorTypeConv.ConvertFrom(colorName);
            boxView.Color = color;
            colorValueLabel.Text = String.Format("{0:X2}-{1:X2}-{2:X2}",
                (int)(255 * color.R),
                (int)(255 * color.G),
                (int)(255 * color.B));
        }
        get
        {
            return colorName;
        }
    }
}
```

The `ColorView` class is finished. Now let’s look at `ColorViewListPage`. The `ColorViewListPage.xaml` file must list multiple `ColorView` instances, so it needs a new XML namespace declaration with a new namespace prefix to reference the `ColorView` element.

The `ColorView` class is part of the same project as `ColorViewListPage`. Generally programmers use an XML namespace prefix of `local` for such cases. The new namespace declaration appears in the root element of the XAML file (like the other two) with the following format:

```
xmlns:local="clr-namespace:ColorViewList;assembly=ColorViewList"
```

A custom XML namespace declaration for XAML must specify a Common Language Runtime (CLR)

namespace—also known as the .NET namespace—and an assembly. The keywords to specify these are `clr-namespace` and `assembly`. Often the CLR namespace is the same as the assembly, as they are in this case, but they don't need to be. The two parts are connected by a semi-colon.

Notice that a colon follows `clr-namespace` but an equal sign follows `assembly`. This apparent inconsistency is deliberate: The format of the namespace declaration is intended to mimic a URI found in conventional namespace declarations, in which a colon follows the URI scheme name.

You use the same syntax for referencing objects in external portable class libraries. The only difference in those cases is that the project also needs a reference to that external PCL. (You'll see an example in a later chapter).

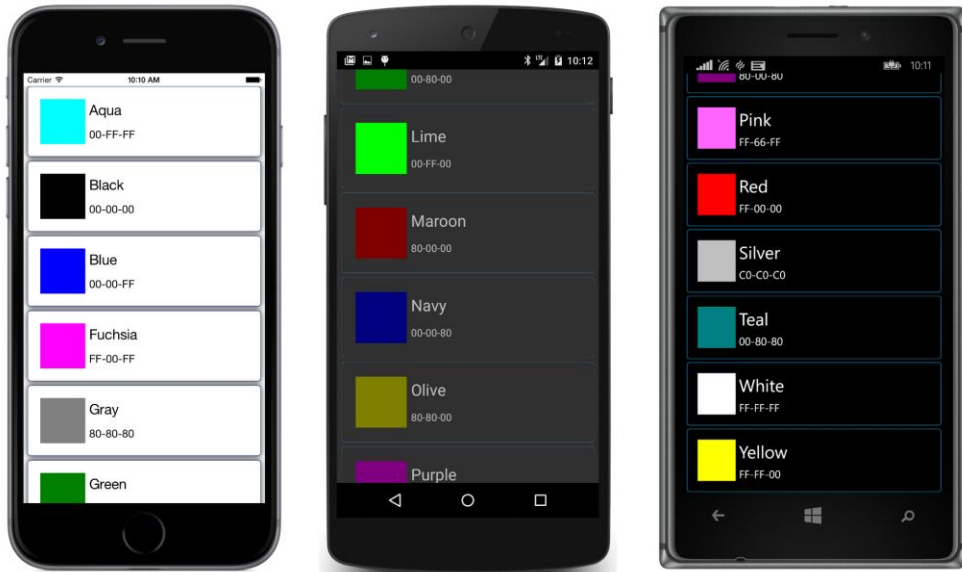
Here's the XAML for the `ColorViewListPage` class. The code-behind file contains nothing beyond the `InitializeComponent` call:

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

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

    <ScrollView>
        <StackLayout Padding="6, 0">
            <local:ColorView ColorName="Aqua" />
            <local:ColorView ColorName="Black" />
            <local:ColorView ColorName="Blue" />
            <local:ColorView ColorName="Fuchsia" />
            <local:ColorView ColorName="Gray" />
            <local:ColorView ColorName="Green" />
            <local:ColorView ColorName="Lime" />
            <local:ColorView ColorName="Maroon" />
            <local:ColorView ColorName="Navy" />
            <local:ColorView ColorName="Olive" />
            <local:ColorView ColorName="Purple" />
            <local:ColorView ColorName="Pink" />
            <local:ColorView ColorName="Red" />
            <local:ColorView ColorName="Silver" />
            <local:ColorView ColorName="Teal" />
            <local:ColorView ColorName="White" />
            <local:ColorView ColorName="Yellow" />
        </StackLayout>
    </ScrollView>
</ContentPage>
```

This is not quite as odious as the earlier example seemed to suggest, and it demonstrates how you can encapsulate visuals in their own XAML-based classes. Notice that the `StackLayout` is the child of a `ScrollView`, so the list can be scrolled:



However, there is one aspect of the **ColorViewList** project that does not qualify as a “best practice.” It is the definition of the `ColorName` property in `ColorView`. This should really be implemented as a `BindableProperty` object. Delving into bindable objects and bindable properties is a high priority for a later chapter.

## Events and Handlers

---

When you tap a `Xamarin.Forms.Button`, it fires a `Clicked` event. You can instantiate a `Button` in XAML but the `Clicked` event handler itself must reside in the code-behind file. The `Button` is only one of a bunch of views that exist primarily to generate events, so the process of handling events is crucial to coordinating XAML and code files.

Attaching an event handler to an event in XAML is as simple as setting a property, and in fact, is visually indistinguishable from a property setting. The **XamlKeypad** project is a XAML version of the **PersistentKeypad** project from Chapter 6. It illustrates setting event handlers in XAML and handling these events in the code-behind file, and also includes logic to save keypad entries when the program is terminated.

If you take look back at the constructor code of the `SimplestKeypadPage` or `PersistentKeypadPage` classes, you’ll see a couple loops to create the buttons that make up the numeric part of the keypad. Of course, this is precisely the type of thing you can’t do in XAML, but look at how much cleaner the markup in `XamlKeypadPage` is when compared to that code:

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

```

        x:Class="XamlKeypad.XamlKeypadPage">
<StackLayout VerticalOptions="Center"
    HorizontalOptions="Center">

    <Label x:Name="displayLabel"
        Font="Large"
        VerticalOptions="Center"
        XAlign="End" />

    <Button x:Name="backspaceButton"
        Text="&#x21E6;"
        Font="Large"
        IsEnabled="False"
        Clicked="OnBackspaceButtonClicked" />

    <StackLayout Orientation="Horizontal">
        <Button Text="7" StyleId="7" Font="Large"
            Clicked="OnDigitButtonClicked" />
        <Button Text="8" StyleId="8" Font="Large"
            Clicked="OnDigitButtonClicked" />
        <Button Text="9" StyleId="9" Font="Large"
            Clicked="OnDigitButtonClicked" />
    </StackLayout>

    <StackLayout Orientation="Horizontal">
        <Button Text="4" StyleId="4" Font="Large"
            Clicked="OnDigitButtonClicked" />
        <Button Text="5" StyleId="5" Font="Large"
            Clicked="OnDigitButtonClicked" />
        <Button Text="6" StyleId="6" Font="Large"
            Clicked="OnDigitButtonClicked" />
    </StackLayout>

    <StackLayout Orientation="Horizontal">
        <Button Text="1" StyleId="1" Font="Large"
            Clicked="OnDigitButtonClicked" />
        <Button Text="2" StyleId="2" Font="Large"
            Clicked="OnDigitButtonClicked" />
        <Button Text="3" StyleId="3" Font="Large"
            Clicked="OnDigitButtonClicked" />
    </StackLayout>

    <Button Text="0" StyleId="0" Font="Large"
        Clicked="OnDigitButtonClicked" />

    </StackLayout>
</ContentPage>

```

The file is a lot shorter than it would have been had the three properties on each numeric Button been formatted into three lines, but packing these all together makes the uniformity of the markup very obvious, and provides clarity rather than obscurity.

The big question is: Which would you rather maintain and modify? The code in the SimplestKey-



padPage or PersistentKeypadPage constructors or the markup in the XamlKeypadPage XAML file?

Here's the screenshot. You'll see that these keys are now arranged in calculator order rather than telephone order:



The **Backspace** button has its Clicked event set to the OnBackspaceButtonClicked handler, while the digit buttons share the OnDigitButtonClicked handler. As you'll recall, the StyleId property is often used to distinguish views sharing the same event handler, which means that the two event handlers can be implemented in the code-behind file exactly the same as the code-only program:

```
public partial class XamlKeypadPage
{
    App app = Application.Current as App;

    public XamlKeypadPage()
    {
        InitializeComponent();

        displayLabel.Text = app.DisplayLabelText;
        backspaceButton.IsEnabled = displayLabel.Text != null &&
            displayLabel.Text.Length > 0;
    }

    void OnDigitButtonClicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;
        displayLabel.Text += (string)button.StyleId;
        backspaceButton.IsEnabled = true;

        app.DisplayLabelText = displayLabel.Text;
    }
}
```

```

    }

    void OnBackspaceButtonClicked(object sender, EventArgs args)
    {
        string text = displayLabel.Text;
        displayLabel.Text = text.Substring(0, text.Length - 1);
        backspaceButton.IsEnabled = displayLabel.Text.Length > 0;

        app.DisplayLabelText = displayLabel.Text;
    }
}

```

Part of the job of the `LoadFromXaml` method called by `InitializeComponent` involves attaching these event handlers to the objects instantiated from the XAML file.

The **XamlKeypad** project includes the code that was added to the page and `App` classes in **PersistentKeypad** to save the keypad text when the program is terminated. The `App` class in **XamlKeypad** is basically the same as the one in **PersistentKeypad**.

## Tap Gestures

---

The `Xamarin.Forms.Button` responds to finger taps, but you can actually get finger taps from any class that derives from `View`, including `Label`, `BoxView`, and `Frame`. These tap events are not built into the `View` class but the `View` class also defines a property named `GestureRecognizers`. Taps are enabled by adding an object to this `GestureRecognizers` collection. An instance of any class that derives from `GestureRecognizer` can be added to this collection, but so far there's only one: `TapGestureRecognizer`.

Here's how to add a `TapGestureRecognizer` to a `BoxView` in code:

```

BoxView boxView = new BoxView
{
    Color = Color.Blue
};
TapGestureRecognizer tapGesture = new TapGestureRecognizer();
tapGesture.Tapped += OnBoxViewTapped;
boxView.GestureRecognizers.Add(tapGesture);

```

`TapGestureRecognizer` also defines a `NumberOfTapsRequired` property with a default value of 1.

To generate `Tapped` events, the `View` object must have its `IsEnabled` property set to `true`, its `IsVisible` property set to `true` (or it won't be visible at all), and its `InputTransparent` property set to `false`. These are all default conditions.

The `Tapped` handler looks just like the `Clicked` handler for the `Button`:

```

void OnBoxViewTapped(object sender, EventArgs args)
{
    ...
}

```

Normally, the sender argument of an event handler is the object that fires the event, which in this case would be the `TapGestureRecognizer` object. That would not be of much use. Instead, the sender argument to the `Tapped` handler is the view being tapped, in this case the `BoxView`. That's *much* more useful!

Like `Button`, `TapGestureRecognizer` also defines `Command` and `CommandParameter` properties; these are used when implementing the MVVM design pattern, and they are discussed in a later chapter.

`TapGestureRecognizer` also defines properties named `TappedCallback` and `TappedCallbackParameter`, and a constructor that includes a `TappedCallback` argument. These are all deprecated and should not be used.

In XAML, you can attach a `TapGestureRecognizer` to a view by expressing the `GestureRecognizers` collection as a property element:

```
<BoxView Color="Blue">
  <BoxView.GestureRecognizers>
    <TapGestureRecognizer Tapped="OnBoxViewTapped" />
  </BoxView.GestureRecognizers>
</BoxView>
```

As usual, the XAML is a little shorter than the equivalent code.

Let's make a program that's inspired by one of the first standalone computer games.

The Xamarin.Forms version of this game is called **MonkeyTap** because it's an imitation game. It contains four `BoxView` elements, colored red, blue, yellow, green. When the game begins, one of the `BoxView` elements flashes, and you must then tap that `BoxView`. That `BoxView` flashes again followed by another one, and now you must tap both in sequence. Then those two flashes are followed by a third, and so forth. (The original had sound as well, but **MonkeyTap** does not.) It's a rather cruel game because there is no way to win. The game just keeps on getting harder and harder until you lose.

The `MonkeyTapPage.xaml` file instantiates the four `BoxView` elements and a `Button` in the center labeled "Begin."

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

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

  <StackLayout>
    <BoxView x:Name="boxview0"
      VerticalOptions="FillAndExpand">
      <BoxView.GestureRecognizers>
        <TapGestureRecognizer Tapped="OnBoxViewTapped" />
      </BoxView.GestureRecognizers>
    </BoxView>
```

```

<BoxView x:Name="boxview1"
    VerticalOptions="FillAndExpand">
    <BoxView.GestureRecognizers>
        <TapGestureRecognizer Tapped="OnBoxViewTapped" />
    </BoxView.GestureRecognizers>
</BoxView>

<Button x:Name="startGameButton"
    Text="Begin"
    Font="Large"
    HorizontalOptions="Center"
    Clicked="OnStartGameButtonClicked" />

<BoxView x:Name="boxview2"
    VerticalOptions="FillAndExpand">
    <BoxView.GestureRecognizers>
        <TapGestureRecognizer Tapped="OnBoxViewTapped" />
    </BoxView.GestureRecognizers>
</BoxView>

<BoxView x:Name="boxview3"
    VerticalOptions="FillAndExpand">
    <BoxView.GestureRecognizers>
        <TapGestureRecognizer Tapped="OnBoxViewTapped" />
    </BoxView.GestureRecognizers>
</BoxView>
</StackLayout>
</ContentPage>

```

All three `BoxView` elements here have a `TapGestureRecognizer` attached but they aren't yet assigned colors. That's handled in the code-behind file because the colors won't stay constant. The colors need to be changed for the flashing effect.

The code-behind file begins with some constants and variable fields. (You'll notice that one of them is flagged as protected; in the next chapter a class will derive from this one and need access to this field. Some methods are defined as protected as well.)

```

public partial class MonkeyTapPage
{
    const int sequenceTime = 750; // in msec
    protected const int flashDuration = 250;

    const double offLuminosity = 0.4; // somewhat dimmer
    const double onLuminosity = 0.75; // much brighter

    BoxView[] boxViews;
    Color[] colors = { Color.Red, Color.Blue, Color.Yellow, Color.Green };
    List<int> sequence = new List<int>();
    int sequenceIndex;
    bool awaitingTaps;
    bool gameEnded;
    Random random = new Random();
}

```

```

public MonkeyTapPage()
{
    InitializeComponent();
    boxViews = new BoxView[] { boxview0, boxview1, boxview2, boxview3 };
    InitializeBoxViewColors();
}

void InitializeBoxViewColors()
{
    for (int index = 0; index < 4; index++)
        boxViews[index].Color = colors[index].WithLuminosity(offLuminosity);
}
...
}

```

The constructor puts all four `BoxView` elements in an array; this allows them to be referenced by a simple index that has values of 0, 1, 2, and 3. The `InitializeBoxViewColors` method sets all the `BoxView` elements to their slightly dimmed non-flashed state.

The program is now waiting for the user to press the **Begin** button to start the first game. The same Button handles replays, so it includes a redundant initialization of the `BoxView` colors. The Button handler also prepares for building the sequence of flashed `BoxView` elements by clearing the sequence list and calling `StartSequence`:

```

public partial class MonkeyTapPage
{
    ...
    protected void OnStartGameButtonClicked(object sender, EventArgs args)
    {
        gameEnded = false;
        startGameButton.IsVisible = false;
        InitializeBoxViewColors();
        sequence.Clear();
        StartSequence();
    }

    void StartSequence()
    {
        sequence.Add(random.Next(4));
        sequenceIndex = 0;
        Device.StartTimer(TimeSpan.FromMilliseconds(sequenceTime), OnTimerTick);
    }
    ...
}

```

`StartSequence` adds a new random integer to the sequence list, initializes `sequenceIndex` to 0, and starts the timer going.

In the normal case, the timer tick handler is called for each index in the sequence list and causes the corresponding `BoxView` to flash with a call to `FlashBoxView`. The timer handler returns false when the sequence is at an end, also indicating by setting `awaitingTaps` that it's time for the user to imitate the sequence:

```

public partial class MonkeyTapPage
{
    ...
    bool OnTimerTick()
    {
        if (gameEnded)
            return false;

        FlashBoxView(sequence[sequenceIndex]);
        sequenceIndex++;
        awaitingTaps = sequenceIndex == sequence.Count;
        sequenceIndex = awaitingTaps ? 0 : sequenceIndex;
        return !awaitingTaps;
    }

    protected virtual void FlashBoxView(int index)
    {
        boxViews[index].Color = colors[index].WithLuminosity(onLuminosity);
        Device.StartTimer(TimeSpan.FromMilliseconds(flashDuration), () =>
        {
            if (gameEnded)
                return false;

            boxViews[index].Color = colors[index].WithLuminosity(offLuminosity);
            return false;
        });
    }
    ...
}

```

The flash is just a quarter second in duration. The `FlashBoxView` method first sets the luminosity for a bright color and creates a “one-shot” timer, so called because the timer callback method (here expressed as a lambda function) returns false and shuts off the timer after restoring the color’s luminosity.

The Tapped handler for the `BoxView` elements ignores the tap if the game is already ended (which only happens with a mistake by the user), and ends the game if the user taps prematurely without waiting for the program to go through the sequence. Otherwise, it just compares the tapped `BoxView` with the next one in the sequence, flashes that `BoxView` if correct, or ends the game if not:

```

public partial class MonkeyTapPage
{
    ...
    protected void OnBoxViewTapped(object sender, EventArgs args)
    {
        if (gameEnded)
            return;

        if (!awaitingTaps)
        {
            EndGame();
            return;
        }
    }
}

```

```
BoxView tappedBoxView = (BoxView)sender;
int index = Array.IndexOf(boxViews, tappedBoxView);

if (index != sequence[sequenceIndex])
{
    EndGame();
    return;
}

FlashBoxView(index);

sequenceIndex++;
awaitingTaps = sequenceIndex < sequence.Count;

if (!awaitingTaps)
    StartSequence();
}

protected virtual void EndGame()
{
    gameEnded = true;

    for (int index = 0; index < 4; index++)
        boxViews[index].Color = Color.Gray;

    startGameButton.Text = "Try again?";
    startGameButton.IsVisible = true;
}
}
```

If the user manages to “ape” the sequence all the way through, another call to `StartSequence` adds a new index to the sequence list and starts playing that new one. Eventually, though, there will be a call to `EndGame`, which colors all the boxes gray to emphasize the end, and re-enables the Button for a chance to try it again.

Here’s the program after the Button has been clicked and hidden:



I know, I know. The game is a real drag without sound.  
Let's take the opportunity in the next chapter to fix that.