Part 3. XAML Markup Extensions

Exploring the Generalized Object-Reference Syntax

Markup Extensions are also covered in our Creating Mobile Apps with Xamarin.Forms book. Preview copies of **Chapter 10. XAML Markup Extensions** (and other chapters) are available for free download.

XAML Markup Extensions

In general, properties of an object are set to explicit values, such as a string, a number, an enumeration member, or a string that is converted to a value behind the scenes.

Sometimes, however, properties must instead reference values defined somewhere else, or which might require a little processing by code at runtime. For these purposes, XAML *markup extensions* are available.

These XAML markup extensions are not extensions of XML. XAML is entirely legal XML. They're called "extensions" because they are backed by code in classes that implement `IMarkupExtension`. You can write your own custom markup extensions.

In many cases, XAML markup extensions are instantly recognizable in XAML files as attribute settings delimited by curly braces: { and }, but sometimes markup extensions appear in markup as conventional elements.

Shared Resources

Some pages contain several views that have properties all set to the same values. For example:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">

  <StackLayout>
    <Button Text="Do this!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            Rotation="-15"
            TextColor="Red"
            Font="Large" />

    <Button Text="Do that!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            Rotation="-15"
            TextColor="Red"
            Font="Large" />

    <Button Text="Do the other thing!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            Rotation="-15"
            TextColor="Red"
            Font="Large" />

  </StackLayout>
</ContentPage>
```

If one of these properties needs to be changed, you might prefer to make the change just once rather than three times. If this were code, you'd likely be using constants and static read-only objects to help keep such values consistent and easy to modify.

In XAML, one popular solution is to store such values or objects in a *resource dictionary*. The `VisualElement` class defines a property named `Resources` of type `ResourceDictionary`, which is a dictionary with keys of type `string` and values of type `object`. You can put objects into this dictionary and then reference them from markup, all in XAML.

To use a resource dictionary on a page, include a pair of `Resources` property-element tags. It's most convenient to put these right at the top of the page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">

  <ContentPage.Resources>

  </ContentPage.Resources>

  …

</ContentPage>
```

It's also necessary to explicitly include `ResourceDictionary` tags:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">

  <ContentPage.Resources>
    <ResourceDictionary>

    </ResourceDictionary>
  </ContentPage.Resources>
  …

</ContentPage>
```

Now objects and values of various types can be added to the resource dictionary. These types must be instantiable—they can't be abstract classes, for example—and generally have a public parameterless constructor. Each item requires a dictionary key specified with the `x:Key` attribute. For example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">

  <ContentPage.Resources>
    <ResourceDictionary>
      <LayoutOptions x:Key="horzOptions"
                     Alignment="Center" />

      <LayoutOptions x:Key="vertOptions"
                     Alignment="Center"
                     Expands="True" />
    </ResourceDictionary>
  </ContentPage.Resources>

  …

</ContentPage>
```

These two items are values of the structure type `LayoutOptions`, and each has a unique key and one or two properties set. In code and markup, it's much more common to use the static fields of `LayoutOptions`, but here it's more convenient to set the properties.

Now it's necessary to set the `HorizontalOptions` and `VerticalOptions` properties of these buttons to these resources, and that's done with the `StaticResource` XAML markup extension:

```
<Button Text="Do this!"
        HorizontalOptions="{StaticResource horzOptions}"
        VerticalOptions="{StaticResource vertOptions}"
        BorderWidth="3"
        Rotation="-15"
        TextColor="Red"
        Font="Large" />
```

The `StaticResource` markup extension is always delimited with curly braces, and includes the dictionary key.

The name `StaticResource` distinguishes it from `DynamicResource`, which Xamarin.Forms also supports. `DynamicResource` is for dictionary keys associated with values that might change during runtime, while `StaticResource` accesses elements from the dictionary just once when the elements on the page are constructed.

For the `BorderWidth` property it's necessary to store a double in the dictionary. XAML conveniently defines tags for common data types like `x:Double` and `x:Int32`:

```
<ContentPage.Resources>
    <ResourceDictionary>
      <LayoutOptions x:Key="horzOptions"
                     Alignment="Center" />

      <LayoutOptions x:Key="vertOptions"
                     Alignment="Center"
                     Expands="True" />

      <x:Double x:Key="borderWidth">
        3
      </x:Double>
    </ResourceDictionary>
```

```
  </ContentPage.Resources>
```

You don’t need to put it on three lines. This dictionary entry for a rotation angle only takes up one line:

```
<ContentPage.Resources>
    <ResourceDictionary>
      <LayoutOptions x:Key="horzOptions"
                     Alignment="Center" />

      <LayoutOptions x:Key="vertOptions"
                     Alignment="Center"
                     Expands="True" />

      <x:Double x:Key="borderWidth">
        3
      </x:Double>

      <x:Double x:Key="rotationAngle">-15</x:Double>
    </ResourceDictionary>
  </ContentPage.Resources>
```

Those two resources can be referenced in the same way as the `LayoutOptions` values:

```
<Button Text="Do this!"
        HorizontalOptions="{StaticResource horzOptions}"
        VerticalOptions="{StaticResource vertOptions}"
        BorderWidth="{StaticResource borderWidth}"
        Rotation="{StaticResource rotationAngle}"
        TextColor="Red"
        Font="Large" />
```

For resources of type `Color` and `Font` you can use the same string representations you use when directly assigning attributes of these types. The type converters are invoked when the resource is created. Here's a resource of type `Color`:

```
<Color x:Key="textColor">Red</Color>
```

A `Font` resource is defined similarly (although `Font` is now deprecated by `FontFamily`, `FontSize`, and `FontAttributes`):

```
<Font x:Key="font">Large</Font>
```

Now all the properties except `Text` are defined by resource settings:

```
<Button Text="Do this!"
        HorizontalOptions="{StaticResource horzOptions}"
        VerticalOptions="{StaticResource vertOptions}"
        BorderWidth="{StaticResource borderWidth}"
        Rotation="{StaticResource rotationAngle}"
        TextColor="{StaticResource textColor}"
        Font="{StaticResource font}" />
```

Although red button text might look good on the iPhone, it's a little dark for the black backgrounds of Android and Windows Phone. Might the `textColor` resource be made platform dependent?

Here's how an `OnPlatform` object can be part of the resource dictionary:

```
<OnPlatform x:Key="textColor"
            x:TypeArguments="Color"
            iOS="Red"
            Android="Aqua"
            WinPhone="#80FF80" />
```

Notice that `OnPlatform` gets both an `x:Key` attribute because it's an object in the dictionary and an `x:TypeArguments` attribute because it's a generic class. The `iOS`, `Android`, and `WinPhone` attributes are converted to `Color` values when the object is initialized.

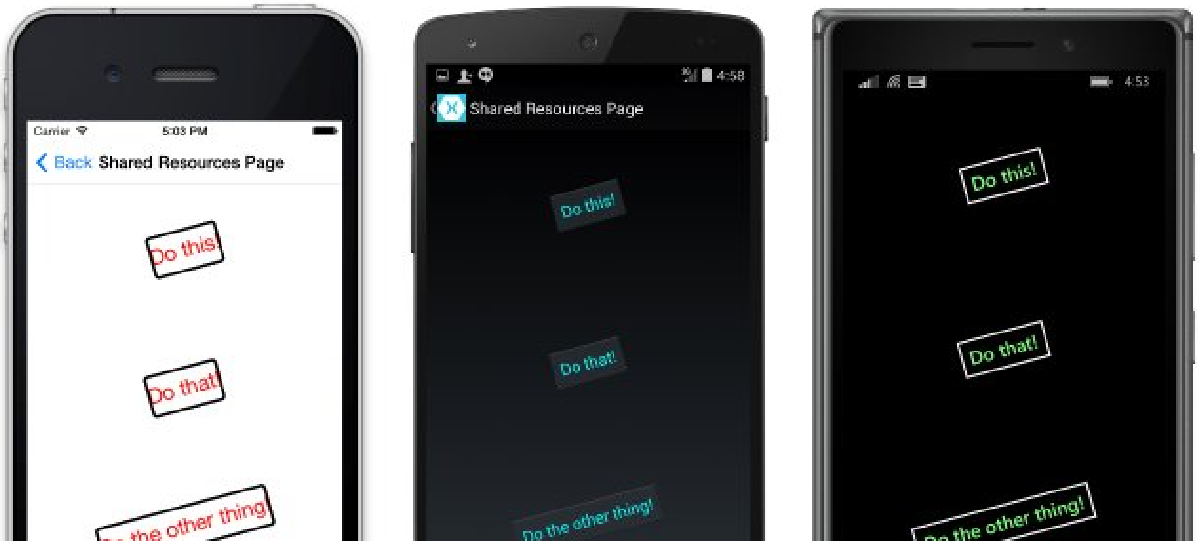
Here's the final complete XAML file with three buttons accessing six shared values:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
```

```
           xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
           x:Class="XamlSamples.SharedResourcesPage"
           Title="Shared Resources Page">

<ContentPage.Resources>
  <ResourceDictionary>
    <LayoutOptions x:Key="horzOptions"
                   Alignment="Center" />

    <LayoutOptions x:Key="vertOptions"
                   Alignment="Center"
                   Expands="True" />

    <x:Double x:Key="borderWidth">3</x:Double>

    <x:Double x:Key="rotationAngle">-15</x:Double>

    <OnPlatform x:Key="textColor"
                x:TypeArguments="Color"
                iOS="Red"
                Android="Aqua"
                WinPhone="#80FF80" />

    <Font x:Key="font">Large</Font>
  </ResourceDictionary>
</ContentPage.Resources>

<StackLayout>
  <Button Text="Do this!"
          HorizontalOptions="{StaticResource horzOptions}"
          VerticalOptions="{StaticResource vertOptions}"
          BorderWidth="{StaticResource borderWidth}"
          Rotation="{StaticResource rotationAngle}"
```

```
                TextColor="{StaticResource textColor}"
                Font="{StaticResource font}" />

        <Button Text="Do that!"
                HorizontalOptions="{StaticResource horzOptions}"
                VerticalOptions="{StaticResource vertOptions}"
                BorderWidth="{StaticResource borderWidth}"
                Rotation="{StaticResource rotationAngle}"
                TextColor="{StaticResource textColor}"
                Font="{StaticResource font}" />

        <Button Text="Do the other thing!"
                HorizontalOptions="{StaticResource horzOptions}"
                VerticalOptions="{StaticResource vertOptions}"
                BorderWidth="{StaticResource borderWidth}"
                Rotation="{StaticResource rotationAngle}"
                TextColor="{StaticResource textColor}"
                Font="{StaticResource font}" />

    </StackLayout>
</ContentPage>
```

The screenshots verify the consistent styling, and the platform-dependent styling:



Although it is most common to define the `Resources` collection at the top of the page, keep in mind that the `Resources` property is defined by `VisualElement`, and you can have `Resources` collections on other elements on the page. For example, try adding one to the `StackLayout` in this example:

```
<StackLayout>
  <StackLayout.Resources>
    <ResourceDictionary>
      <Color x:Key="textColor">Blue</Color>
    </ResourceDictionary>
  </StackLayout.Resources>

    …

</StackLayout>
```

You'll discover that the text color of the buttons is now blue. Basically, whenever the XAML parser encounters a `StaticResource` markup extension, it searches up the visual tree and uses the first `ResourceDictionary` it encounters containing that key.

Sometimes developers new to XAML wonder if they can put a visual element such as `Label` or `Button` in a `ResourceDictionary`. While it's surely possible, it doesn't make much sense. The purpose of the `ResourceDictionary` is to share objects. A visual element cannot be shared. The same instance cannot appear twice on a single page.

The x:Static Markup Extension

Despite the similarities of their names, `x:Static` and `StaticResource` are very different. `StaticResource` returns an object from a resource dictionary while `x:Static` accesses a public static field, static property, or constant defined by a class or structure, or an enumeration member. The `StaticResource` markup extension is supported by XAML implementations that define a resource dictionary, while `x:Static` is an intrinsic part of XAML, as the `x` prefix reveals.

Here are a few examples that demonstrates how `x:Static` can explicitly reference static fields and enumeration members:

```
<Label Text="Hello, XAML!"
       VerticalOptions="{x:Static LayoutOptions.Start}"
       HorizontalTextAlignment="{x:Static TextAlignment.Center}"
       TextColor="{x:Static Color.Aqua}" />
```

So far, this is not all that impressive. But the `x:Static` markup extension can also reference static fields or properties from your own code. For example, here's an `AppConstants` class that contains some constants you might want to use on multiple pages throughout an application:

```
using System;
using Xamarin.Forms;

namespace XamlSamples
{
    static class AppConstants
    {
        public static readonly Thickness PagePadding =
            new Thickness(5, Device.OnPlatform(20, 0, 0), 5, 0);

        public static readonly Font TitleFont =
            Font.SystemFontOfSize(Device.OnPlatform(35, 40, 50),
FontAttributes.Bold);

        public static readonly Color BackgroundColor =
            Device.OnPlatform(Color.White, Color.Black, Color.Black);

        public static readonly Color ForegroundColor =
            Device.OnPlatform(Color.Black, Color.White, Color.White);
    }
}
```

To reference static fields of this class in the XAML file, you'll need some way to indicate within the XAML file where this file is located. You do this with an XML namespace declaration.

Recall that the XAML files created as part of the standard Xamarin.Forms XAML template contain two XML namespace declarations: one for accessing Xamarin.Forms classes and another for referencing tags and attributes intrinsic to XAML:

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

You'll need additional XML namespace declarations to access other classes. Each additional XML namespace declaration defines a new prefix. To access classes local to the shared application PCL, such as `AppConstants`, XAML programmers often use the prefix `local`. The namespace declaration must indicate the CLR (Common Language Runtime) namespace name (also known as the .NET namespace name, which is the name that appears in a C# `namespace` definition or in a `using` directive) and the assembly containing the code. Often these are the same. Here's the syntax:

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

Notice that the keyword `clr-namespace` is followed by a colon and then the .NET namespace name, followed by a semicolon, the keyword `assembly`, an equal sign, and the assembly name.

Yes, a colon follows `clr-namespace` and an equal sign follows `assembly`. This difference is deliberate: Often standard XML namespace declarations reference a URI that begins a URI scheme name such as `http`, which is always followed by a colon. The `clr-namespace` part of this string is intended to mimic that convention.

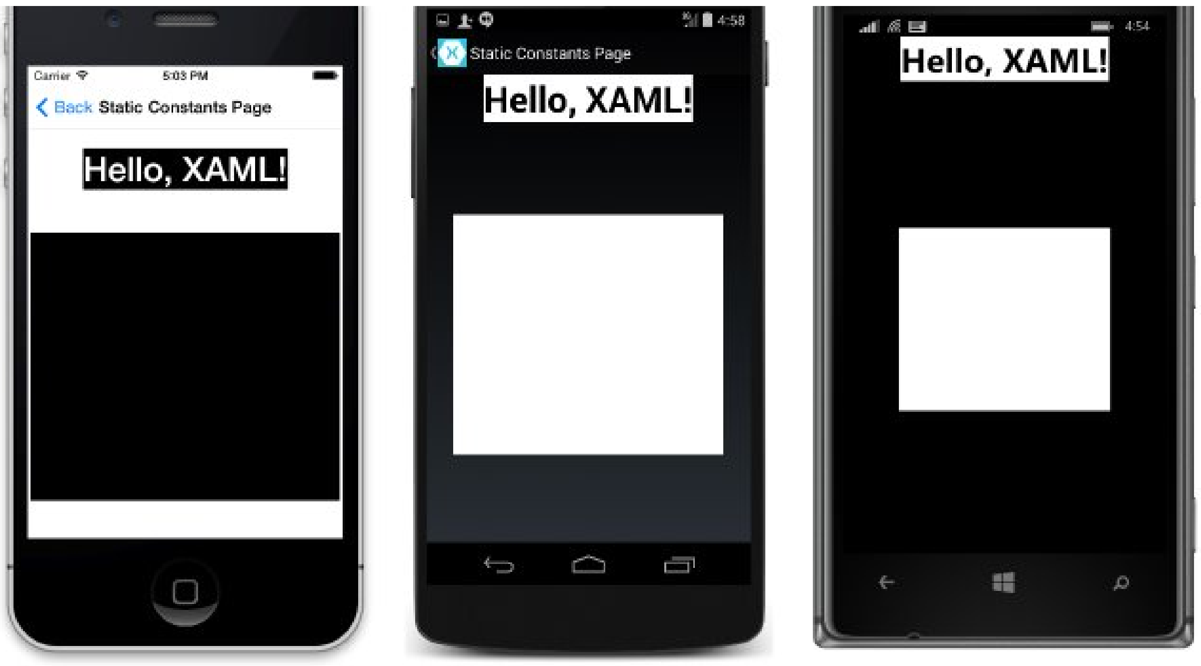
You can also define XML namespace declarations for .NET namespaces in any assembly that the PCL references. For example, here's a `sys` prefix for the standard .NET `System` namespace, which is in the mscorlib assembly (which once stood for "Microsoft Common Object Runtime Library", but now means "Multilanguage Standard Common Object Runtime Library");

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

Both these namespace declarations are included in the StaticConstantsPage sample. Notice that the `BoxView` dimensions are set to `Math.PI` and `Math.E`, but scaled by a factor of 100:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-
namespace:XamlSamples;assembly=XamlSamples"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             x:Class="XamlSamples.StaticConstantsPage"
             Title="Static Constants Page"
             Padding="{x:Static local:AppConstants.PagePadding}">

  <StackLayout>
     <Label Text="Hello, XAML!"
            TextColor="{x:Static local:AppConstants.BackgroundColor}"
            BackgroundColor="{x:Static
local:AppConstants.ForegroundColor}"
            Font="{x:Static local:AppConstants.TitleFont}"
            HorizontalOptions="Center" />

    <BoxView WidthRequest="{x:Static sys:Math.PI}"
             HeightRequest="{x:Static sys:Math.E}"
             Color="{x:Static local:AppConstants.ForegroundColor}"
             HorizontalOptions="Center"
             VerticalOptions="CenterAndExpand"
             Scale="100" />
  </StackLayout>
</ContentPage>
```

The size of the resultant `BoxView` relative to the screen is platform-dependent:



Other Standard Markup Extensions

Several markup extensions are intrinsic to XAML and supported in Xamarin.Forms XAML files. Some of these have rather obscure utility but are essential when you need them:

- If a property has a non- `null` value by default but you want to set it to `null` , set it to the `{x:Null}` markup extension.
- If a property is of type `Type` , you can assign it to a `Type` object using the markup extension `{x:Type someClass}` .
- You can define arrays in XAML using the `x:Array` markup extension. This markup extension has a required attribute named `Type` that indicates the type of the elements in the array.

The ConstraintExpression Markup Extension

Markup extensions can have properties, but they are not set like XML attributes. In a markup extension, property settings are separated by commas, and no quotation marks appear within the curly braces.

This can be easily illustrated with the Xamarin.Forms markup extension named `ConstraintExpression` that's used with the `RelativeLayout` class. You can specify the location or size of a child view as a constant, or relative to a parent or other named view. The syntax of the `ConstraintExpression` lets you set the position or size of a view using a `Factor` times a property of another view, plus a `Constant`. Anything more complex than that requires code.

Here's an example:

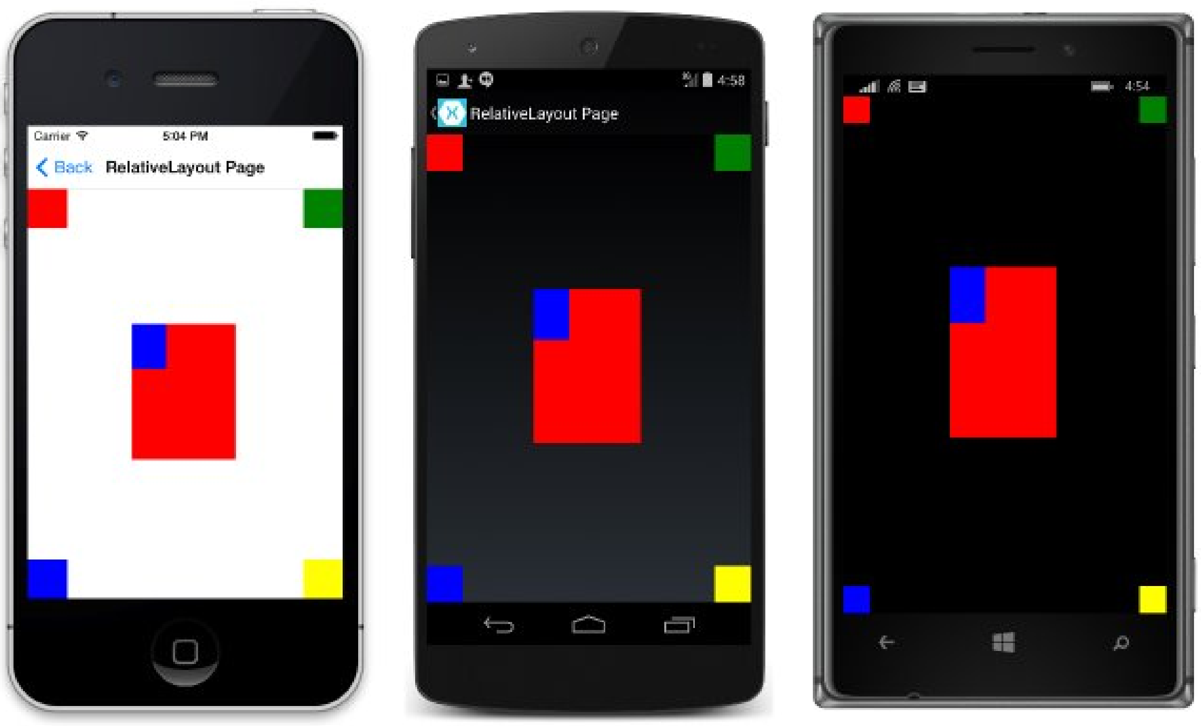
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.RelativeLayoutPage"
             Title="RelativeLayout Page">

  <RelativeLayout>

    <!-- Upper left -->
    <BoxView Color="Red"
             RelativeLayout.XConstraint=
                 "{ConstraintExpression Type=Constant,
                                        Constant=0}"
             RelativeLayout.YConstraint=
                 "{ConstraintExpression Type=Constant,
                                        Constant=0}" />
    <!-- Upper right -->
    <BoxView Color="Green"
             RelativeLayout.XConstraint=
                 "{ConstraintExpression Type=RelativeToParent,
                                        Property=Width,
                                        Factor=1,
                                        Constant=-40}"
```

```
         RelativeLayout.YConstraint=
             "{ConstraintExpression Type=Constant,
                                    Constant=0}" />
<!-- Lower left -->
<BoxView Color="Blue"
         RelativeLayout.XConstraint=
             "{ConstraintExpression Type=Constant,
                                    Constant=0}"
         RelativeLayout.YConstraint=
             "{ConstraintExpression Type=RelativeToParent,
                                    Property=Height,
                                    Factor=1,
                                    Constant=-40}" />
<!-- Lower right -->
<BoxView Color="Yellow"
         RelativeLayout.XConstraint=
             "{ConstraintExpression Type=RelativeToParent,
                                    Property=Width,
                                    Factor=1,
                                    Constant=-40}"
         RelativeLayout.YConstraint=
             "{ConstraintExpression Type=RelativeToParent,
                                    Property=Height,
                                    Factor=1,
                                    Constant=-40}" />

<!-- Centered and 1/3 width and height of parent -->
<BoxView x:Name="oneThird"
         Color="Red"
         RelativeLayout.XConstraint=
             "{ConstraintExpression Type=RelativeToParent,
                                    Property=Width,
                                    Factor=0.33}"
         RelativeLayout.YConstraint=
             "{ConstraintExpression Type=RelativeToParent,
                                    Property=Height,
                                    Factor=0.33}"
         RelativeLayout.WidthConstraint=
             "{ConstraintExpression Type=RelativeToParent,
                                    Property=Width,
                                    Factor=0.33}"
         RelativeLayout.HeightConstraint=
             "{ConstraintExpression Type=RelativeToParent,
                                    Property=Height,
                                    Factor=0.33}"  />

<!-- 1/3 width and height of previous -->
<BoxView Color="Blue"
         RelativeLayout.XConstraint=
             "{ConstraintExpression Type=RelativeToView,
                                    ElementName=oneThird,
                                    Property=X}"
         RelativeLayout.YConstraint=
             "{ConstraintExpression Type=RelativeToView,
                                    ElementName=oneThird,
                                    Property=Y}"
         RelativeLayout.WidthConstraint=
             "{ConstraintExpression Type=RelativeToView,
                                    ElementName=oneThird,
                                    Property=Width,
                                    Factor=0.33}"
         RelativeLayout.HeightConstraint=
             "{ConstraintExpression Type=RelativeToView,
                                    ElementName=oneThird,
                                    Property=Height,
                                    Factor=0.33}"  />
```

```
  </RelativeLayout>
</ContentPage>
```

Perhaps the most important lesson of this sample is to emphasize that no quotation marks appear within the curly braces of a markup extension. When typing the markup extension in a XAML file, it is natural to want to enclose the values of the properties in quotation marks. Resist the temptation!

Here's the program running:



Summary

The XAML markup extensions shown here provide important support for XAML files. But perhaps the most valuable XAML markup extension is `Binding`, which is covered in the next part of this series, Part 4. Data Binding Basics.