Part 2. Essential XAML Syntax

Working with Property Elements and Attached Properties

Property Elements

In XAML, properties of classes are normally set as XML attributes:

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large"
       TextColor="Aqua" />
```

However, there is an alternative way to set a property in XAML. Let's try this alternative with `TextColor`. First delete the existing `TextColor` setting:

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large" />
```

Open up the empty-element `Label` tag by separating it into start and end tags:

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large">


</Label>
```

Within these tags, add start and end tags that consist of the class name and a property name separated by a period:

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
```

```
        FontSize="Large">
  <Label.TextColor>

  </Label.TextColor>
</Label>
```

Set the property value as content of these new tags, like this:

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large">
  <Label.TextColor>
    Aqua
  </Label.TextColor>
</Label>
```

These two ways to specify the `TextColor` property are functionally equivalent, but you can't use both ways for the same property because that would effectively be setting the property twice.

With this new syntax, some handy terminology can be introduced:

- `Label` is an *object element* . It is a Xamarin.Forms object expressed as an XML element.
- `Text`, `VerticalOptions`, `FontAttributes` and `FontSize` are *property attributes* . They are Xamarin.Forms properties expressed as XML attributes.
- In that final snippet, `TextColor` has become a *property element* . It is a Xamarin.Forms property but it is now an XML element.

The definition of property elements might at first seem to be a violation of XML syntax, but it's not. The period has no special meaning in XML. As far as XML goes, `Label.TextColor` is simply a normal child element.

In XAML, however, this syntax is very special. One of the rules for property elements is that nothing else can appear in the `Label.TextColor` tag. The value of the property is always defined as content between the property-element start and end tags.

You can use property-element syntax on more than one property:

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center">
  <Label.FontAttributes>
```

```
    Bold
  </Label.FontAttributes>
  <Label.FontSize>
    Large
  </Label.FontSize>
  <Label.TextColor>
    Aqua
  </Label.TextColor>
</Label>
```

Or you can use property-element syntax for all the properties:

```
<Label>
  <Label.Text>
    Hello, XAML!
  </Label.Text>
  <Label.FontAttributes>
    Bold
  </Label.FontAttributes>
  <Label.FontSize>
    Large
  </Label.FontSize>
  <Label.TextColor>
    Aqua
  </Label.TextColor>
  <Label.VerticalOptions>
    Center
  </Label.VerticalOptions>
</Label>
```

At first, property-element syntax might seem like a ridiculously long-winded replacement for something comparatively quite simple, and in these examples that is certainly the case.

However, property-element syntax becomes essential when the value of a property is too complex to be expressed as a simple string. Within the property-element tags you can instantiate another object and set its properties. For example, you can explicitly set a property such as `VerticalOptions` to a `LayoutOptions` value with property settings:

```
<Label>
```

```
  <Label.Text>
    Hello, XAML!
  </Label.Text>
  <Label.FontAttributes>
    Bold
  </Label.FontAttributes>
  <Label.FontSize>
    Large
  </Label.FontSize>
  <Label.TextColor>
    Aqua
  </Label.TextColor>
  <Label.VerticalOptions>
    <LayoutOptions Alignment="Center" />
  </Label.VerticalOptions>
</Label>
```

Another example: The `Grid` has two properties named `RowDefinitions` and `ColumnDefinitions`. These two properties are of type `RowDefinitionCollection` and `ColumnDefinitionCollection`, which are collections of `RowDefinition` and `ColumnDefinition` objects. Unless there is a syntax to define all the rows and column dimensions in the collection as a single string—which is surely possible but hasn't been done—you'll need property element syntax.

Here's the beginning of the XAML file for a `GridDemoPage` class, showing the property element tags for the `RowDefinitions` and `ColumnDefinitions` collections:

```
<?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.GridDemoPage"
             Title="Grid Demo Page"
             Padding="0, 20, 0, 0">

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

```
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="100" />
    </Grid.ColumnDefinitions>

    ...

  </Grid>
</ContentPage>
```

Notice the abbreviated syntax for defining auto-sized cells, cells of pixel widths and heights, and star settings.

This XAML fragment has another potential use for property elements. The `Padding` on the `ContentPage` tag is set to 20 units on the top, but that's only to avoid overlapping the status bar on the iPhone. That padding isn't required on Android and Windows Phone. (Nor is it required when the page is navigated to through a `NavigationPage` as the pages in XamlSamples are, but let's continue as if this were a standalone page in a single-page application.)

Fortunately there is a way to embed some platform-specific markup in a XAML file using a class named `OnPlatform<T>`. This is a generic class that has three properties named `iOS`, `Android`, and `WinPhone` of type `T`. The `OnPlatform<T>` class also defines an implicit cast of itself to type `T` that returns the appropriate object depending on which platform it's running on.

It sounds complicated but the XAML syntax is actually quite straightforward. First, separate out the `Padding` as a property element of the `ContentPage` class:

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

  <ContentPage.Padding>
    0, 20, 0, 0
  </ContentPage.Padding>
```

```
  ...

</ContentPage>
```

No change. Now replace the content of the `ContentPage.Padding` tags with `OnPlatform tags`. Keep in mind that `OnPlatform` is a generic class. You need to specify the generic type argument, in this case, `Thickness`, which is the type of `Padding` property. Fortunately, there's a XAML attribute specifically for use with generic arguments called `x:TypeArguments`:

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

  <ContentPage.Padding>
    <OnPlatform x:TypeArguments="Thickness">

    ...

    </OnPlatform>
  </ContentPage.Padding>

  …

</ContentPage>
```

Between the `OnPlatform` tags put three property-element tags for the `iOS`, `Android`, and `WinPhones` properties containing the `Padding` value for each platform:

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

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

```
      <OnPlatform.Android>
        0, 0, 0, 0
      </OnPlatform.Android>
      <OnPlatform.WinPhone>
        0, 0, 0, 0
      </OnPlatform.WinPhone>
    </OnPlatform>
  </ContentPage.Padding>

  ...

</ContentPage>
```

If the content of the individual property elements can be represented by simple strings (as these can) you can instead define them as property attributes:

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

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

  ...

</ContentPage>
```

However, since the Android and Windows Phone values are the defaults for `Padding`, those two attribute settings can be removed:

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

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

  ...

</ContentPage>
```

And that is the standard markup you can use in a `ContentPage` to avoid overlapping the iPhone status bar.

(As mentioned earlier, in the downloadable XamlSamples solution, this padding is not required because these individual pages are navigated to via a `NavigationPage`, and in that cases, the iPhone status bar is automatically accounted for.)

Attached Properties

As you've seen, the `Grid` requires property elements for the `RowDefinitions` and `ColumnDefinitions` collections to define the rows and columns. However, there must also be some way for the programmer to indicate the row and column where each child of the `Grid` resides.

Within the tag for each child of the `Grid` you specify the row and column of that child using the following attributes:

- `Grid.Row`
- `Grid.Column`

The default values of these attributes are 0. You can also indicate if a child spans more than one row or column with these attributes:

- `Grid.RowSpan`
- `Grid.ColumnSpan`

These two attributes have default values of 1.

Here's the complete GridDemoPage.xaml file:

```
<?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.GridDemoPage"
             Title="Grid Demo Page">

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

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

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="100" />
    </Grid.ColumnDefinitions>

    <Label Text="Autosized cell"
           Grid.Row="0" Grid.Column="0"
           TextColor="White"
           BackgroundColor="Blue" />

    <BoxView Color="Silver"
             HeightRequest="0"
             Grid.Row="0" Grid.Column="1" />

    <BoxView Color="Teal"
             Grid.Row="1" Grid.Column="0" />

    <Label Text="Leftover space"
           Grid.Row="1" Grid.Column="1"
```

```
            TextColor="Purple"
            BackgroundColor="Aqua"
            HorizontalTextAlignment="Center"
            VerticalTextAlignment="Center" />

    <Label Text="Span two rows (or more if you want)"
            Grid.Row="0" Grid.Column="2" Grid.RowSpan="2"
            TextColor="Yellow"
            BackgroundColor="Blue"
            HorizontalTextAlignment="Center"
            VerticalTextAlignment="Center" />

    <Label Text="Span two columns"
            Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2"
            TextColor="Blue"
            BackgroundColor="Yellow"
            HorizontalTextAlignment="Center"
            VerticalTextAlignment="Center" />

    <Label Text="Fixed 100x100"
            Grid.Row="2" Grid.Column="2"
            TextColor="Aqua"
            BackgroundColor="Red"
            HorizontalTextAlignment="Center"
            VerticalTextAlignment="Center" />

  </Grid>
</ContentPage>
```

The `Grid.Row` and `Grid.Column` settings of 0 are not required but are generally included for purposes of clarity.

Here's what it looks like on all three platforms:

Judging solely from the syntax, these `Grid.Row`, `Grid.Column`, `Grid.RowSpan`, and `Grid.ColumnSpan` attributes appear to be static fields or properties of `Grid`, but interestingly enough, `Grid` does not define anything named `Row`, `Column`, `RowSpan`, or `ColumnSpan`.

Instead, `Grid` defines four bindable properties named `RowProperty`, `ColumnProperty`, `RowSpanProperty`, and `ColumnSpanProperty`. These are special types of bindable properties known as *attached properties*. They are defined by the `Grid` class but set on children of the `Grid`.

When you wish to use these attached properties in code, the `Grid` class provides static methods named `SetRow`, `GetColumn`, and so forth. But in XAML, these attached properties are set as attributes in the children of the `Grid` using simple properties names.

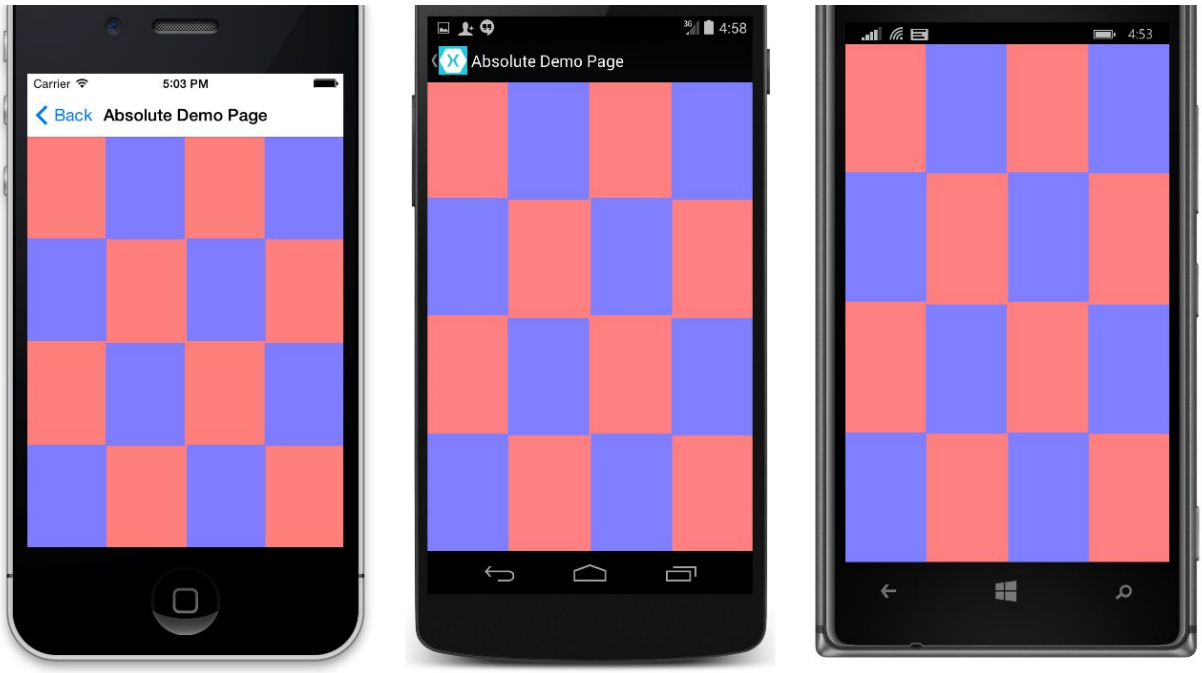
Attached properties are always recognizable in XAML files as attributes containing both a class and a property name separated by a period. They are called *attached properties* because they are defined by one class (in this case, `Grid`) but attached to other objects (in this case, children of the `Grid`). During layout, the `Grid` can interrogate the values of these attached properties to know where to place each child.

The `AbsoluteLayout` class defines two attached properties named `LayoutBounds` and `LayoutFlags`. Here's something of a checkerboard pattern realized using the proportional positioning and sizing features of `AbsoluteLayout`:

```
<?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.AbsoluteDemoPage"
             Title="Absolute Demo Page">

  <AbsoluteLayout BackgroundColor="#FF8080">
```

```
    <BoxView Color="#8080FF"
             AbsoluteLayout.LayoutBounds="0.33, 0, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />

    <BoxView Color="#8080FF"
             AbsoluteLayout.LayoutBounds="1, 0, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />

    <BoxView Color="#8080FF"
             AbsoluteLayout.LayoutBounds="0, 0.33, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />

    <BoxView Color="#8080FF"
             AbsoluteLayout.LayoutBounds="0.67, 0.33, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />

    <BoxView Color="#8080FF"
             AbsoluteLayout.LayoutBounds="0.33, 0.67, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />

    <BoxView Color="#8080FF"
             AbsoluteLayout.LayoutBounds="1, 0.67, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />

    <BoxView Color="#8080FF"
             AbsoluteLayout.LayoutBounds="0, 1, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />

    <BoxView Color="#8080FF"
             AbsoluteLayout.LayoutBounds="0.67, 1, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />

  </AbsoluteLayout>
</ContentPage>
```

And here it is:



For something like this, you might question the wisdom of using XAML. Certainly the repetition and regularity of the `LayoutBounds` rectangle suggests it might be better realized in code.

That's certainly a legitimate concern, and there's no problem with balancing the use of code and markup when defining your user interfaces. It's easy to define some of the visuals in XAML and then use the constructor of the code-behind file to add some more visuals that might be better generated in loops.

Content Properties

In the previous examples, the `StackLayout`, `Grid`, and `AbsoluteLayout` objects are set to the `Content` property of the `ContentPage`, and the children of these layouts are actually items in the `Children` collection. Yet these `Content` and `Children` properties are nowhere in the XAML file.

They certainly could be. The `Content` and `Children` properties could be included as property elements:

```
<?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.XamlPlusCodePage"
             Title="XAML + Code Page">
  <ContentPage.Content>
    <StackLayout>
```

```
      <StackLayout.Children>
        <Slider VerticalOptions="CenterAndExpand"
                ValueChanged="OnSliderValueChanged" />

        <Label x:Name="valueLabel"
               Text="A simple Label"
               FontSize="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Button Text="Click Me!"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                Clicked="OnButtonClicked" />
      </StackLayout.Children>
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

The real question is: Why are they *not* required in the XAML file?

Elements defined in Xamarin.Forms for use in XAML are allowed to have one property flagged in the `ContentProperty` attribute on the class. If you look up the `ContentPage` class in the online Xamarin.Forms documentation, you'll see this attribute.

```
[Xamarin.Forms.ContentProperty("Content")]
public class ContentPage : Page
```

This means that the `Content` property-element tags are not required. Any XML content that appears between the start and end `ContentPage` tags is assumed to be assigned to the `Content` property.

`StackLayout`, `Grid`, `AbsoluteLayout`, and `RelativeLayout` all derive from `Layout<View>`, and if you look up `Layout<T>` in the Xamarin.Forms documentation, you'll see another `ContentProperty` attribute:

```
[Xamarin.Forms.ContentProperty("Children")]
public abstract class Layout<T> : Layout ...
```

That allows content of the layout to be automatically added to the `Children` collection without explicit `Children` property-element tags.

Here are all the `ContentProperty` attribute definitions in Xamarin.Forms:

Element	Content Property
ContentPage	Content
ContentView	Content
Frame	Content
Label	Text
Layout<T>	Children
ScrollView	Content
ViewCell	View

Summary

With property elements and attached properties, much of the basic XAML syntax has been established. However, there still sometimes exist needs to set properties to objects that must be referenced in an indirect manner. This solution is covered in the next part, Part 3. XAML Markup Extensions.