XAML Properties

By Pete Brown, author of *Windows Store App Development*

You need to know about the XAML property system in order to understand dependency and attached properties and how they enable binding and animation. The types of properties you'll use in XAML aren't the same as you may be using; they have a different underlying implementation. In this article, based on chapter 4 of Windows Store App Development, the author shows you a few different ways to specify properties in markup.

Most classes we develop expose properties that can be manipulated by consuming code. For example, a `Person` class might expose a `LastName` property. Similarly, the `TextBlock` class exposes, among many others, a `Text` property. These properties provide a wrapped way to manipulate the data associated with a class.

In WinRT XAML, properties are even more important because they can be the source or target of data binding, or the target of an animation, or more. For those reasons, a special property system had to be developed.

In this section I'll cover how to use properties in XAML. First, I'll explain the different ways of specifying properties using both property element syntax and XML attributes. From there, I'll go into the dependency property system and how it works with the other WinRT XAML subsystems. At the same time, we'll look at a specialized type of dependency property called an attached property.

Finally, we'll take a brief look at property paths—something that will be essential when you look at control templates or animation.

Let's start by figuring out how to specify property values in markup.

Property syntax

There are two ways to reference properties in XAML: inline with the element as you would any XML attribute, or as a nested subelement. Which you should choose depends on what you need to represent. Simple values are typically represented with inline properties, whereas complex values are typically represented with element properties.

The use of an inline property requires a type converter that will convert the string representation—for example, the `"Black"` in `Background="Black"`—into a correct underlying type (in this case, a `SolidColorBrush`). The example in the following listing shows a built-in type converter being used to convert the string `"Black"` for the inline property `Background`. The type converter is built into the system and invoked automatically.

Listing 1 Specifying a property value inline using an XML attribute

```
<Page
  x:Class="PropertyExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:PropertyExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
```

```
    <Grid Background="Black" #A
          Width="600">
    </Grid>
  </Page>
```

#A Inline properties

This listing also shows a numeric property, `Width`, which gets converted to a double from the string `"600"`. Built-in converters can be extremely simple as in the numeric case or more complex as in the case of the brush converter or, as you'll see in later chapters, the mini-language that specifies points in geometry.

Another way to specify properties is to use the expanded property element syntax. Although this can generally be used for any property, it's typically required only when you need to specify something more complex than the inline syntax will easily allow. The syntax for element properties is `<Type.PropertyName>value</Type.Property-Name>`, as shown here.

Listing 2 Specifying a property value using property element syntax

```
  <Page
    x:Class="PropertyExample.MainPage"
    IsTabStop="false"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:PropertyExample"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Width="600">
      <Grid.Background> #A
        Black
      </Grid.Background>
    </Grid>
  </Page>
```

#A Element property

The use of the string to invoke the type converter is, in its end result, identical to using `<SolidColorBrush Color="Black" />` in place of `"Black"`. Though these examples are rarely seen in practice, the more complex example of setting the `Background` property to a `LinearGradientBrush` is common, so we'll cover that next.

Rather than have the brush represented as a simple string such as `"Black"` as shown in the previous listing, the value can be an element containing a complex set of elements and properties such as the `<LinearGradientBrush>` shown in the next listing.

Listing 3 A more complex example of the property element syntax

```
  <Page
    x:Class="PropertyExample.MainPage"
    IsTabStop="false"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:PropertyExample"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Grid Width="600">
      <Grid.Background> #A
        <LinearGradientBrush>
          <LinearGradientBrush.GradientStops>
            <GradientStop Offset="0.0" Color="Black" />
            <GradientStop Offset="0.5" Color="Blue" />
            <GradientStop Offset="0.5" Color="Orange" />
            <GradientStop Offset="1.0" Color="DarkGray" />
          </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
      </Grid.Background>
    </Grid>
  </Page>
```

#A Background property

In this example, you expand the `Background` property using property element syntax. You then have the ability to nest a complex type, such as the `LinearGradientBrush`, within it.

Now that you know how to specify properties in markup, let's dive deeper into how those properties work.

Dependency properties

Dependency properties are part of the property system introduced with WPF and Silverlight and used in WinRT XAML. In markup and in consuming code, they're indistinguishable from .NET properties, except that they can be data bound, serve as the target of an animation, or be set by a style.

> **TIP** A property can't be the target of an animation or obtain its value through binding unless it's a dependency property.

To have dependency properties in a class, the class must derive from `Dependency-Object` or one of its subclasses. Typically, you'll do this only for visuals and other elements that you'll use within XAML and not in classes defined outside of the user interface.

In regular code, when you create a property, you typically back it by a private field in the containing class. Storing a dependency property differs in that the location of its backing value depends on its current state, and the CLR property wrapper is just a convenience, as shown in figure 1. The way that location is determined is called value precedence.

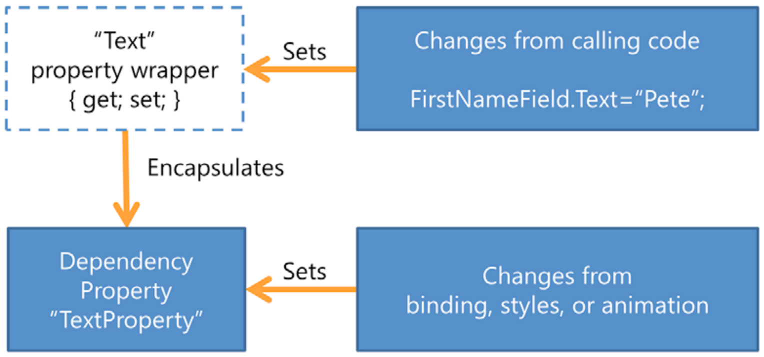

Figure 1 Dependency properties aren't backed by private fields like normal CLR properties. Instead, the CLR property wrapper is just a convenience. Inside the wrapper, it uses the SetValue and GetValue methods to access the dependency property itself.

Value precedence

Dependency properties obtain their value from a variety of inputs. What follows is the order the XAML property system uses when assigning the runtime values of dependency properties, with the highest precedence listed first, as shown in figure 2.

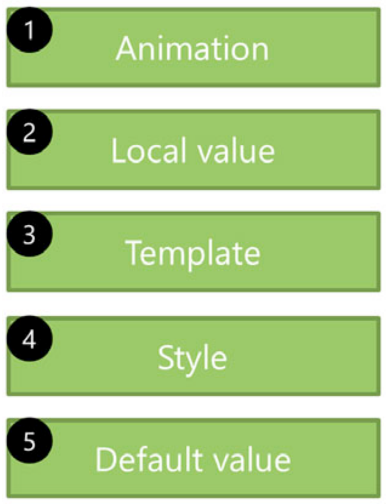

Figure 2 Value precedence for dependency properties. Value changes from animation take precedence over locally set values, which have precedence over template values, and so on. This precedence allows properties to have meaningful value changes in the app and also respond correctly to setting values in markup and code.

Here’s the precedence in more detail:

- Active or hold animations—Animations will operate on the base value for the dependency property, determined by evaluating the precedence for other inputs. In order for an animation to have any effect, it must be highest in precedence. Animations may operate on a single dependency property from multiple levels of precedence (for example, an animation defined in the control template and an animation defined locally). The value typically results from the composite of all animations, depending on the type being animated. If you think about animating the position of an element, you’ll want that animated value to take precedence over one set in code or markup. The property system helps ensure that it happens.
- Local value—Local values are specified directly in the markup and are accessed via the property wrappers for the dependency property. When you directly assign a value to a property in XAML or in code, that’s a local value. Because local values are higher in precedence than styles and templates, they’re capable of overriding values such as the font style or foreground color defined in the default style for a control.
- Templated properties—Used specifically for elements created within a control or data template, their value is taken from the template itself.
- Style setters—These are values set in a style in your application via resources defined in or merged into the `UserControl` or application resource dictionaries.
- Default value—This is the value provided or assigned when the dependency property was first created. If no default value was provided, normal runtime defaults typically apply.

There are other subtleties to this. For example, controls typically have a default implicit style, which itself sets property values and includes a control template. That control template may contain visual states, which are themselves implemented as animations in a bit of an Inception-esque nesting. If you understand the basics of value precedence, you can always figure out where a value is coming from. I still find it easier than trying to debug web pages with complicated CSS layouts and rules.

The strict precedence rules allow you to depend on behaviors within the runtime, such as being able to override elements of a style by setting them as local values from within the element itself. In the next listing, the foreground of the button will be `Red` as set in the local value and not `Black` as set in the style. The local value has a higher precedence than the applied style.

Listing 4 Dependency property precedence rules in practice

```
<Page
  x:Class="XamlExample.MainPage"
```

```
     IsTabStop="false"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     xmlns:local="using:XamlExample"
     xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
     xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
     mc:Ignorable="d">

     <Page.Resources>
       <Style x:Key="ButtonStyle" #A
              TargetType="Button">
         <Setter Property="Foreground"
                 Value="Black" />
         <Setter Property="FontSize"
                 Value="24" />
       </Style>
     </Page.Resources>
     <Grid>
       <Button Content="Local Values at Work"
               Style="{StaticResource ButtonStyle}" #B
               Foreground="Red" /> #C
     </Grid>
   </Page>
```

#A Style definition
#B Style in use
#C Local Value

The `Style` tag in `Page.Resources` is a reusable asset that sets some key properties for our button. Precedence in this case works very much like precedence in CSS: A locally declared style property overrides that defined at a higher level. The end result, in this case, is the precedence shown in figure 3.

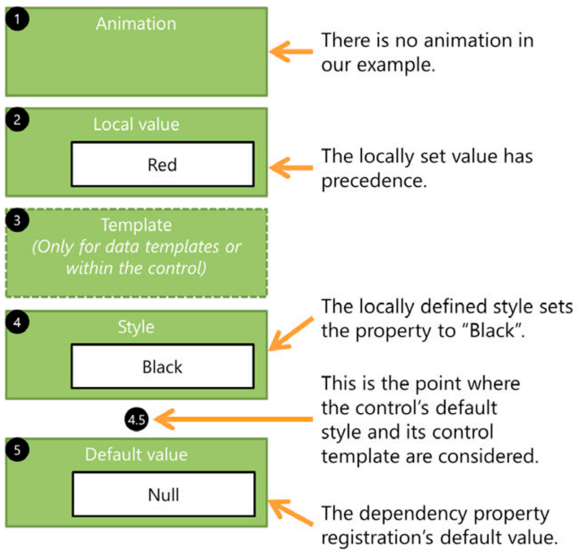
1
Animation
There is no animation in our example.
2
Local value
Red
The locally set value has precedence.
3
Template
(Only for data templates or within the control)
4
Style
Black
The locally defined style sets the property to "Black".
4.5
This is the point where the control's default style and its control template are considered.
5
Default value
Null
The dependency property registration's default value.

Figure 3 The dependency property precedence for the Foreground property of the TextBox from listing 4.6. In this example, the TextBox ends up with a Red foreground.

For more information on dependency properties, please see the MSDN page here: http://bit.ly/WinRTXamlDP. You can also implement your own dependency properties in *Windows Store App Development*, in the part that explains how you create custom controls and panels.

There's one other type of dependency property you must understand before you can truly grok the property system used in XAML. That type of property also has a slightly odd appearance and is called an attached property.

Attached properties

Attached properties are a specialized type of dependency property that's immediately recognizable in markup because of the `TypeName.AttachedPropertyName` syntax. For example, `Canvas.Left` is an attached property defined by the `Canvas` type. What makes attached properties interesting is that they're not defined by the type you use them with; instead, they're defined by another type in a potentially different class hierarchy. The class using the property doesn't need to have any knowledge of the class that defines the property.

Attached properties allow flexibility when defining classes because the classes don't need to take into account every possible scenario in which they'll be used and define properties for those scenarios. Layout is a great example of this. The flexibility of the layout system allows you to create new panels that may never have been implemented in other technologies—for example, a panel that lays out elements by degrees and levels in a circular or radial fashion versus something like the built-in `Canvas` that lays out elements by `Left` and `Top` positions.

Rather than have all elements define `Left`, `Top`, `Level`, and `Degrees` properties (as well as `GridRow` and `GridColumn` properties for grids) like you would in a technology like Windows Forms, you can use attached properties. The buttons in listing 5, for example, are contained in panels that have greatly differing layout algorithms, requiring different positioning information. In this case, we'll show a fictional `RadialPanel` in use (the panel doesn't exist, so the markup won't compile as is).

Listing 5 Attached properties in use

```
<Page
  x:Class="XamlExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:XamlExample"
  xmlns:panels="usingXamlExample.Panels"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <StackPanel>
    <Canvas Width="400" Height="200">
      <Button Canvas.Left="10" #A
              Canvas.Top="50"
              Width="200" Height=
              Content="Button in Canvas" />
    </Canvas>

    <panels:RadialPanel Width="400" Height="400">
      <Button panels:RadialPanel.Degrees="25" #B
              panels:RadialPanel.Level="3"
              Width="200" Height="100"
              Content="Button in Radial Panel" />
    </panels:RadialPanel>
  </StackPanel>
</Page>
```

#A, #B Attached properties

Attached properties aren't limited to layout. You'll find them in the animation engine for things such as `Storyboard.Target` Property as well as in other places of the framework. For more information on attached properties, please see the MSDN page at http://bit.ly/WinRTXamlAttachedProps.

Property paths

Before we wrap up our discussion of properties, there's one concept left to understand: property paths. Property paths provide a way to reference properties of objects in XAML both when you have a name for an element and when you need to indirectly refer to an element by its position in the tree.

Property paths can take several forms and may dot-down into properties of an object. They can also use parentheticals for indirect property targeting as well as for specifying attached properties. Here are some examples of property paths for the `Storyboard` target property:

```
<DoubleAnimation Storyboard.TargetName="MyButton" #A
                 Storyboard.TargetProperty="(Canvas.Left)"... /> #B
<DoubleAnimation Storyboard.TargetName="MyButton"
                 Storyboard.TargetProperty="Width"... />
...
<Button x:Name="MyButton"
        Canvas.Top="50" Canvas.Left="100" />
```

#A Path to a normal property: MyButton
#B Path to an attached property: Canvas.Left

Properties are one of the pieces that define an object’s interface. When looking at XAML, most of what you see will be objects and their properties.

Summary

XAML is simply the markup manifestation of WinRT and .NET objects. Elements and properties are absolutely essential because they’re the meat and potatoes (or tofu and broccoli) of markup.

You’ve learned how the property system used in XAML is a bit different from the one you may have used in C# in your own classes and in other presentation technologies. Dependency properties and attached properties are key to supporting the binding and animation systems and important in producing panels with flexible layout.

Last updated: 8/2/13