

## Chapter 3

### Useful things – Part 1

*Copyright © Billy Hollis 2016-2019. All rights reserved. Do not copy, republish, reproduce, or distribute. Posting of this content to the web by anyone except the author is strictly forbidden by the copyright.*

Chapter 2 on composition and layout explored how you put pieces together to get the results you want in XAML user interfaces. In this chapter, we'll expand the toolbox of pieces you can use, and cover some additional syntax details you will need to locate and use those pieces.

We have more foundational concepts to explore later in the book, including data binding and templating. This chapter focuses on more practical aspects, including:

- Important aspects of XAML syntax: namespaces for finding things and element syntax for setting properties
- Declaring reusable resources in XAML
- Useful elements, such as Border, Pivot, Image, Flyout, and ToolTip
- What I call “option” controls, which include ToggleSwitch, CheckBox, and RadioButton
- Using text in XAML interfaces, including font properties and formatting text in a TextBlock
- Discussion of the Margin property, and its cousin, the Padding property

This chapter won't make your head hurt like the last one did. It doesn't have deep concepts; it covers details that you need to know to become fluent in XAML.

There are a lot of XAML elements. So far, you have only learned a few. This chapter includes some more that I think you will use often.

To keep from overloading you on details, I'm deferring additional useful elements to later chapters. Chapter xx, entitled “Graphical Elements”, will discuss elements, properties, and concepts that give you powerful graphical capabilities. Chapter xx, entitled “Useful Things – Part Two” contains even more elements that are not used as often, but which you might well need at some point.

### XML namespaces in XAML

In Chapter 1, we briefly mentioned that XAML locates the classes for visual elements and other objects using XML namespaces.<sup>1</sup> The visual designer places several namespace declarations at the top of a page of XAML when it creates a new one. Here is the XAML for a blank Page in an Uno Platform project as created by Visual Studio:

```
<Page
  x:Class="Chapter3.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

---

<sup>1</sup> XML namespaces are not specific to XAML. They've been around a long time. If you are interested in the nuances of XML namespaces, you should consult a reference on XML. I'm only going to tell you the basics of using them in XAML.

```

xmlns:local="using:Chapter3"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d">

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock Text="Hello, world !" Margin="20" FontSize="30" />
</Grid>
</Page>

```

All the lines that begin with “xmlns” are declaring XML namespaces. All of them except for the first one have a colon and a string of characters, including “xmlns:x” and “xmlns:local”.

The first one, which has no colon, is called the *default namespace*. It is used to locate classes for things declared as XML elements without a prefix, such as <Button> and <TextBox>.<sup>2</sup>

All other namespaces need a *namespace prefix*. That’s the string of characters after the colon. An element or object from one of those namespaces must be declared with its prefix. For example, I find it useful in some pages to add a namespace declaration for the shared project in an Uno solution, like this:

```
xmlns:shared="using:Chapter3.Shared"
```

We’ve saw an example of using a prefix in Chapter 1. In the data binding XAML example in Chapter 1, the DataContext of a Grid was set to a Book object with these lines:

```

...
xmlns:local="using:Chapter1.Shared"
...
<Grid>
    <Grid.DataContext>
        <local:Book Title="Ender's Game" Genre="Science Fiction" />
    </Grid.DataContext>

```

The declaration of the Book object is <local:Book . . .>. XAML pages and user controls created by Visual Studio have a declaration for the “local” prefix, which is the project that the page is in. In this case, for the example to work, the Book object would have to be in that project.<sup>3</sup>

Other prefixes can point to different namespaces, which reside in different projects and DLLs. If you drag over something from the toolbox that is in a different DLL, such as a third party control, then the visual designer will automatically insert the namespace declaration for that DLL.<sup>4</sup>

If you copy XAML from one page and paste it into another that does not have the appropriate xmlns declarations, you’ll see a compile error. In those cases, you need to do two things:

- Ensure that you have a project reference to the DLLs for those namespaces

<sup>2</sup> You will probably never need to change anything about the default namespace. The line declaring the default namespace is just a background aspect of practically every XAML page since the technology was in beta. In fact, the “winfx” part of the declaration comes from the beta name for a collection of technologies that included an earlier XAML platform, WPF.

<sup>3</sup> Sometimes the “local” namespace is set to “using:ProjectName”, while sometimes it is set to “using:ProjectName.Shared”. This may be an artifact that will be made consistent later.

<sup>4</sup> This is one of the few really useful things about the visual designer.

- Copy the necessary namespace declarations from the top of the original page into the top of target page

### Element vs. Attribute syntax

Back in Chapter One, you learned that you could set property values on elements with XML attributes, like this:

```
<Button Background="Green">  
    I'm a button  
</Button>
```

This XAML declares a Button visual element and sets the Background property of the Button using an XML attribute. This “attribute syntax” is the most common way to set a property of an object declared in XAML.

It’s not the only way, though. Here are five lines of XAML that produce exactly the same compiled results, with bolded lines showing the new way to set the Background property:

```
<Button>  
    <Button.Background>  
        Green  
    </Button.Background>  
    I'm a button  
</Button>
```

This way of declaring property values in XAML is called “element syntax”. The property value is enclosed by XML elements that declare a class and a property on the class, such as “<Button.Background> ... </Button.Background>”. The previous example setting the DataContext property of a Grid used element syntax, too:

```
<Grid>  
    <Grid.DataContext>  
        <local:Book Title="Ender's Game" Genre="Science Fiction" />  
    </Grid.DataContext>
```

Why do we need two ways to set a property value, especially when one of them is so verbose? It’s because not all property values can be declared with a simple string. Some need a more complex definition. The DataContext example shows this. Creating a Book object requires a XAML declaration, and it goes between the start and end tags for <Grid.DataContext>.

Even properties such as Background need this capability in some cases. For example, we might want the background of the Button to be a gradient fill. In that case, the Background property of the Button needs to be set to a gradient brush object, and a gradient brush can’t be defined with a simple string.

The gradient brush requires its own XAML definition, and the space between the elements furnishes a place to put it. Here is the XAML that sets the Button’s background to a simple gradient brush, with the start and end tags for the property setting bolded:

```
<Button Content="I'm a button">  
    <Button.Background>  
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
```

```

        <GradientStop Color="Green" Offset="0"/>
        <GradientStop Color="LightGreen" Offset="0.5"/>
        <GradientStop Color="DarkGreen" Offset="1"/>
    </LinearGradientBrush>
</Button.Background>
</Button>

```

This XAML is exactly the same as the previous XAML setting the Background property to green, except that the named color “Green” is replaced by the gradient brush.

We’ll be looking at gradient brushes in detail in Chapter 5, but for now you just need to realize that it takes several lines of XAML to declare one. Element syntax is the way to set a property to such a value.

When I teach classes in XAML, one of the more common hurdles is understanding how XML element tags are used for both object declaration, as in <Button>, and property setting, as in <Button.Background>. It’s easy to get confused about this. Figure xx may help you make the distinction by making it clear what each line of XAML is doing in using element syntax for a property.

A <b>visual element</b> – an instance of Button – is declared as part of the page	<Button Content="I'm a button">
A <b>property</b> of the Button, the Background property, is being set	<Button.Background>
The <b>value</b> of the Background property is an object, a LinearGradientBrush	<LinearGradientBrush> . . . </LinearGradientBrush>
The <b>closing tag</b> for the Button’s Background <b>property</b> declaration	</Button.Background>
The <b>closing tag</b> for the Button <b>element</b>	</Button>

Figure xx – Element syntax for setting properties uses XML elements for two purposes. The <Button> element tag is for declaring an object, in this case a visual element. The <Button.Background> element tag indicates a property setting for the Background property of the Button.

You must become comfortable with element syntax because there are many properties that require it. Examples include properties that take templates of various kinds, because those templates often require many lines of XAML to define.

### Default properties

Many visual elements have a property called the *default property*. That property can be set using element syntax, but the elements for setting the property are optional.

You’ve already seen such a property – the Content property of a ContentControl. Recall the XAML above for a Button:

```

<Button Background="Green">
    I'm a button
</Button>

```

This is a shortcut for setting the property with element syntax:

```
<Button Background="Green">
  <Button.Content>
    I'm a button
  </Button.Content>
</Button>
```

For a default property, the element tags declaring the property setting are optional. Here are the default properties for some commonly-used elements:

ContentControl (Button, ScrollViewer, UserControl, etc.) – Content property

Panel (Grid, StackPanel, etc.) – Children collection

Border – Child property

TextBlock – Text property

You are not required to use default property syntax to set these properties. Here are a couple of perfectly fine lines of XAML that use an attribute to set a default property to a string.

```
<Button Background="Green" Content="I'm a button" />
<TextBlock Text="Some text here" />
```

Usually the Text property of a TextBlock is set to a string, so attribute syntax works fine and default property syntax is less common.<sup>5</sup> If a Button needs only text inside, setting Content with attribute syntax is fine. The visual designer will set the Content property that way if you drag a Button onto the design surface.

However, thinking in XAML means you are often putting something more visual in a Button or other ContentControl, so you should be comfortable shifting gears into element-based property syntax whenever you need it.

To wrap up this section on element vs. attribute syntax, let's look at a summary. Figure xx shows three different ways of setting the Content property of a Button, and all of them compile to the same result. The top Button uses attribute syntax, the second Button uses default property syntax, and the bottom Button uses element syntax.

---

<sup>5</sup> At the end of this chapter, we're going to see some other, more flexible ways of setting the text displayed in a TextBlock.

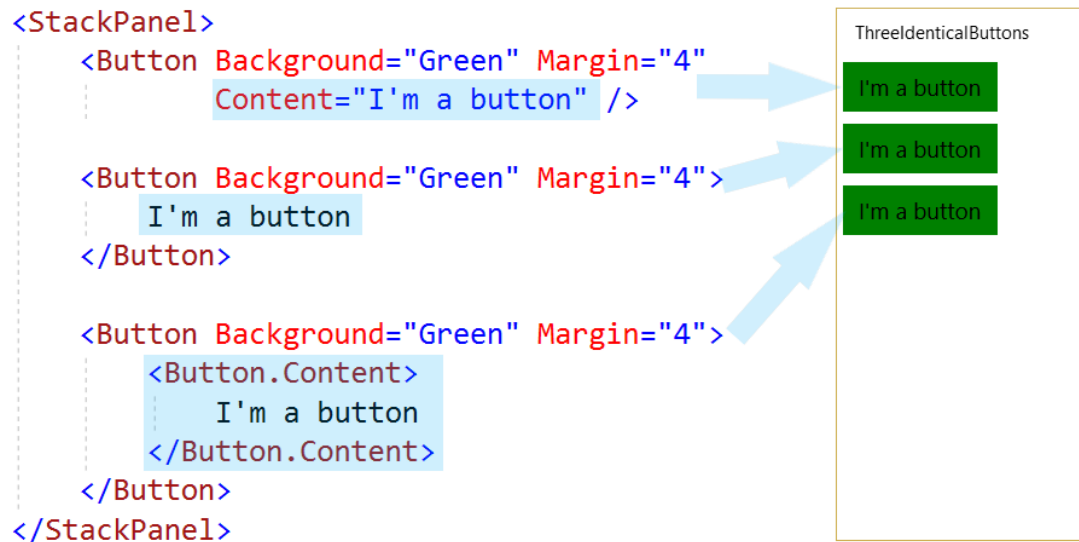


Figure xx – Three ways to set the Content property of a Button. All three compile to the same end result.

## Resources and resource collections

If you are a lazy developer like me, you like to reuse stuff. XAML provides a place to put re-usable assets, and a means to refer to them when you need them.

Re-usable assets are called *resources*,<sup>6</sup> and are stored in a collection. Every element can have such a collection, and the collection is stored in the element's Resources property.

Any kind of object can become a resource. We will see many uses of resources, but for the example syntax, I will focus on brushes and data objects as resources.

A collection of resources is a special kind of dictionary called a ResourceDictionary. Because it's a dictionary, each resource in the collection must have a key. Here is the XAML syntax to declare such a collection for a Page:

```

<Page.Resources>
  <SolidColorBrush Color="PapayaWhip" x:Key="MySweetBrush" />
  <local:Book Title="Ready Player One"
    Genre="Science Fiction" x:Key="SampleBook" />
</Page.Resources>

```

The first resource is a SolidColorBrush with a color of PapayaWhip. The dictionary key for the resource is declared with the "x:Key" attribute. In this case, the key is "MySweetBrush". We will be seeing that key later, because it will be used to refer back to this brush.

The second resource is a data object. As in Chapter 1, we assume that the current project has a class of type Book, with property Title and Genre. Then we can declare a Book object using the local namespace.

<sup>6</sup> Resources in XAML have nothing to do with resources in .NET .resx files.

The declaration for the Book object is exactly the same as in Chapter 1, except that we must also include a dictionary key using “x:Key”. That’s the case with any resource. The XAML definition for the resource looks exactly the same as it would if the object were declared elsewhere in XAML, except that a key is added.

By convention, a Resources collection is usually declared at the top of an element’s XAML.

### Referring to a resource

You can assign a resource to a property setting using the StaticResource markup extension. For example, assigning MySweetBrush (defined in the example above) to the Background of two Buttons in a StackPanel would look like this:

```
<StackPanel>
  <Button Background="{StaticResource MySweetBrush}" Margin="5">
    I love whipped papayas
  </Button>
  <Button Background="{StaticResource MySweetBrush}" Margin="5">
    Well, as long as they're ripe
  </Button>
</StackPanel>
```

The resulting buttons would look like Figure xx.

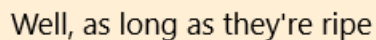
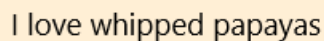


Figure xx – Two Button elements with their Background property set using a resource. The resource was declared as a SolidColorBrush with the color PapayaWhip.

Both Button elements are using the same brush, defined as a resource. If the resource definition of the brush changes, say by changing to a different color, then both Button elements will get the altered brush.

That has obvious advantages. If lots of elements are using the same brush, then the brush can be changed in the resource collection, and all the elements will change visually.

In fact, this is the basis for themes in XAML. We will go into styling and theming in Chapter 7, but I will mention that XAML applications for Uno Platform have many resources already available because of built-in themes.

A related concept to StaticResource is ThemeResource. You might have already noticed that a new Page in a Visual Studio UWP project has the following declaration for a Grid already included:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
```

I’ve been leaving that automatically-generated setting out of my examples so far to keep down the clutter. I will still leave it out for most examples until we discuss styling and theming in Chapter 7, where I will go into detail about ThemeResource.

Referring to the Book object as a resource has the same syntax. Recall that we set the DataContext of a Grid with XAML like this:

```
<Grid>
  <Grid.DataContext>
    <local:Book Title="Ender's Game" Genre="Science Fiction" />
  </Grid.DataContext>
```

If the Book object is declared as a resource name "SampleBook" (as we showed above), then setting the DataContext property for the Grid can be changed to this:

```
<Grid DataContext="{StaticResource SampleBook}">
```

### Resources at different levels in the XAML tree

I mentioned earlier that every element can have its own dictionary of resources. The dictionary is stored in the Resources property for the element.

Resources defined for a container are available to all elements inside the container. That's why the example above with the Button elements works. The resource MySweetBrush is defined in the Resources collection for the Page, so that resource is available to every element on the Page.

If a resource is in the Resources collection for a StackPanel inside the Page, then those resources are only accessible for elements inside the StackPanel. That includes children, children of children, and so on.

In the chapter "Useful Things, Part 2", we'll be looking at the Application object defined by App.xaml, which is automatically included when Visual Studio creates a UWP project. That object can have a Resources collection too. We could move those resources in the example above to App.xaml, and the result in XAML would look like this:

```
<Application
  x:Class="ShowResources.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:ShowResources"
  RequestedTheme="Light">
  <Application.Resources>
    <SolidColorBrush Color="PapayaWhip" x:Key="MySweetBrush" />
    <local:Book Title="Ready Player One"
      Genre="Science Fiction" x:Key="SampleBook" />
  </Application.Resources>
</Application>
```

Resources declared in App.xaml are available to every element in the application. This has major consequences that we'll delve into in Chapter 7. We'll also talk there about what happens when



resources of the same name are declared at different levels in the application.<sup>7</sup> For the next few chapters, we'll just declare our resources at some convenient level and not worry about the nuances.

### Finding resources via Intellisense and the Properties Window

Intellisense will try to help you locate a resource when you are typing a StaticResource markup extension. Once you have typed the beginning of the markup extension, a drop down will show you resources that match the property type you are setting.

So if you are setting the Background property, the drop down will only include resources that are Brush objects. All resources further up in the application are listed, including some defined automatically for you at the system level. As you type letters in XAML, the list will filter down to resources that match. Figure xx shows the Intellisense window for setting the Background property of a Button, and the progression of the window as letters are typed in XAML.

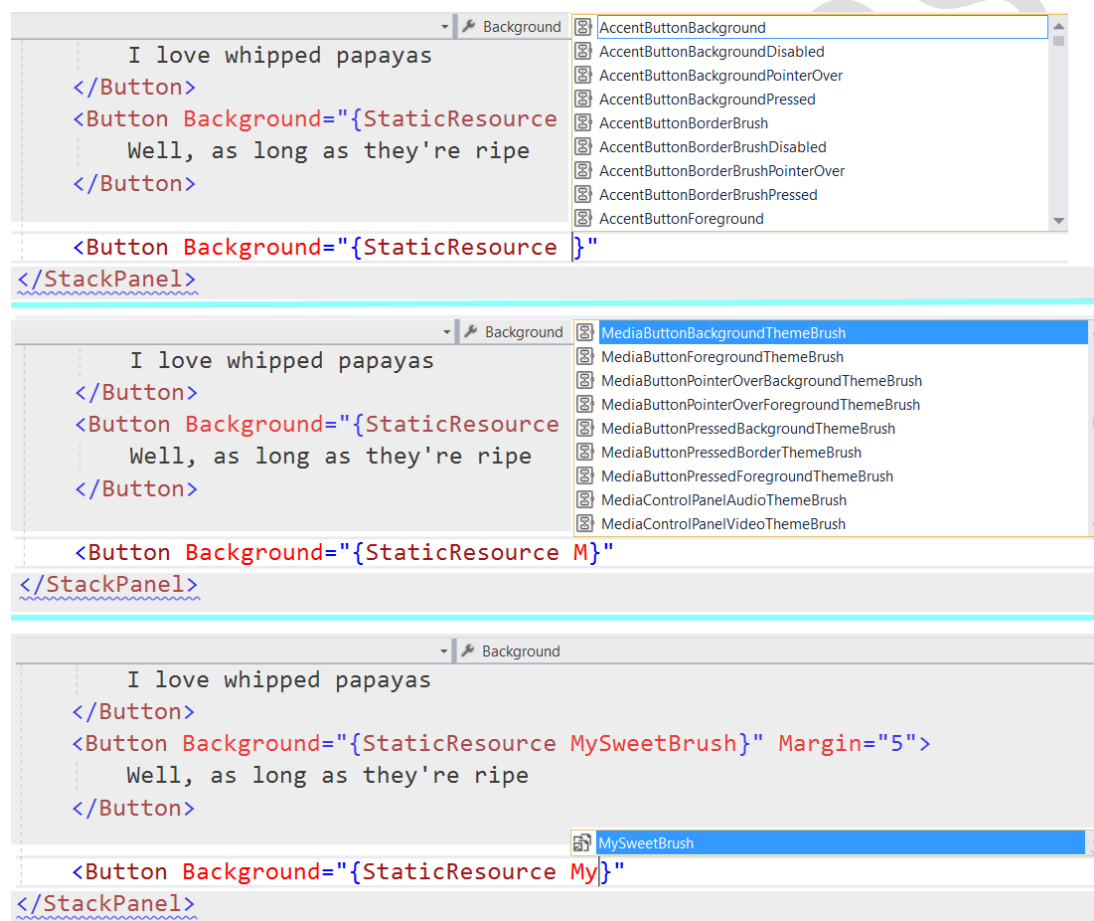


Figure xx – When setting the Background property of an element to a resource, Intellisense will offer a list of available resource objects that are of type Brush, and will filter the list as letters are typed in. Only Brush objects are included in the list because the Background property is of type Brush. (Incidental XAML is grayed out to help you see the line where the work is occurring.)

<sup>7</sup> If you just can't wait, here's a quick summary: A resource with the same name at a lower level in the app overrides the resource with that name further up.

You can also get to lists of resources for a property in the Properties Window. To the right of each property value in the Properties Window is a square glyph. A green arrow and circle indicates that glyph for the Background property of a Button in Figure xx

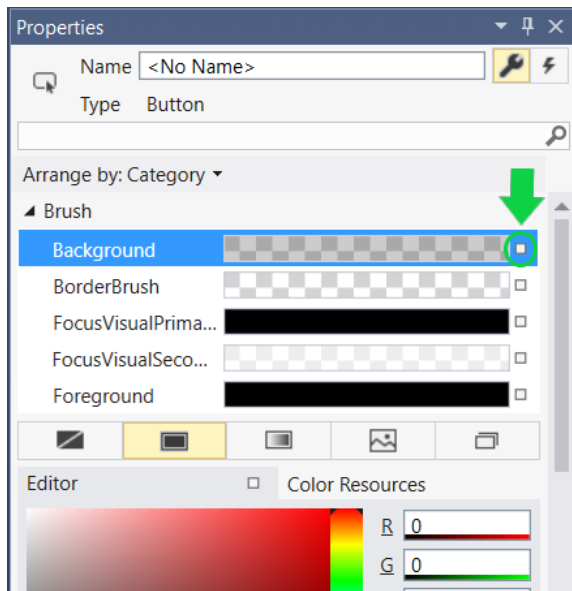


Figure xx – The square glyph next to a property value such as the Background property brings up a contextual menu with several options. The glyph has a green circle around it and a green arrow pointing to it. Those options are shown in Figure xx.

When the glyph is pressed, a menu with several options affecting the property is shown. You can see that menu in Figure xx. Two of the options allow you to set the value of the property using a resource. Those options are highlighted in purple, with an arrow to help you locate them.

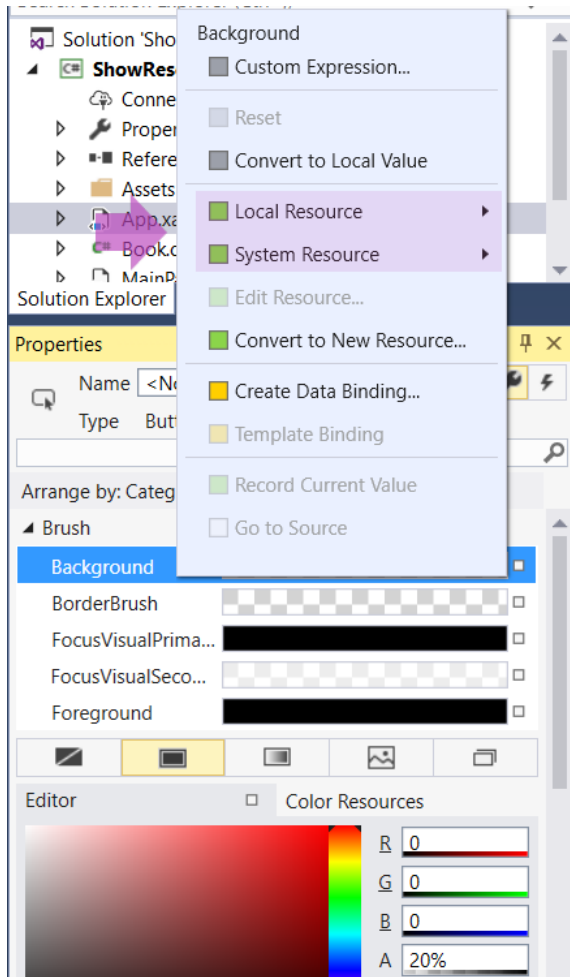


Figure xx – When the square glyph next to a property is pressed, a menu appears with options concerning the property. Two of them, highlighted in purple in the figure to help you see them, allow setting the property value to a resource.

The “Local Resource” option will display a drop down of the matching resources that are defined in your application. The “System Resource” option will give you a drop down of the resources defined for you as system resources. This includes all the resources in the current theme, as well resources based on Windows system colors.

When you choose a resource from either the “Local Resource” or the “System Resource” list, the property is set to that resource in XAML, using the same markup extension syntax we discussed earlier.

You will learn more about resources in Chapter X. What we’ve covered will be sufficient for using resources in the next few chapters – and we will need to use them a lot, as you’ll see.

### Useful elements

The XAML toolbox in Visual Studio contains about 70 different visual elements. Even then, there are various elements you will use in XAML, such as `Polygon`, that are not in the toolbox.

They're all part of the XAML API for a reason, and part of learning to think in XAML is to know what most of them do. If you proceed into sophisticated XAML development, you'll probably use around 90% of them at one time or another.<sup>8</sup>

We've seen several visual elements already, and you will probably use all of these regularly:

- Grid
- StackPanel
- Button
- ScrollViewer
- TextBlock
- TextBox
- Ellipse and Rectangle shapes
- SymbolIcon

Next, let's look at some of the other elements you will likely use on a routine basis.

## Border

So far, we've seen two types of containers in XAML. Panels contain a collection of children and do layout. ContentControls hold a single item of content, although that item can be a Panel which then holds multiple child items.

The Border element is also a container. We saw it briefly in some earlier examples, but I didn't talk much about how it worked.

Like a ContentControl, Border holds just one child item. However, that item is set with the Child property rather than the Content property. Child is the default property of Border, so whatever you put between the Border's start and end tags becomes the Border's child element.

The purpose of the Border element is, naturally enough, to draw a border around its child element. Panels such as StackPanel and Grid do not have any properties for including a border. Instead, as befits the compositional philosophy of XAML, those panels get a border by becoming the child element of a Border element.

Border has several properties that affect the appearance of the border drawn around the child:

Property	Purpose	Example
BorderBrush	Specify the color or texture of the border	BorderBrush="Blue"
BorderThickness	Specify the thickness of the border, either uniformly or for each side	BorderThickness="3" BorderThickness="4,2,8,2"
CornerRadius	Specify the amount of round on corner, either uniformly or for each corner	CornerRadius="5" CornerRadius="10,5,8,10"

---

<sup>8</sup> You can compare the process of learning XAML to learning a foreign language. This book focuses mostly on the syntax and structure of the language, with just enough "vocabulary" to get you started. As you work in XAML, you'll learn more "vocabulary", that is, more elements and objects you can define and use in XAML.

Figure xx has an example that uses all three of those properties, with both the XAML and the rendered appearance.

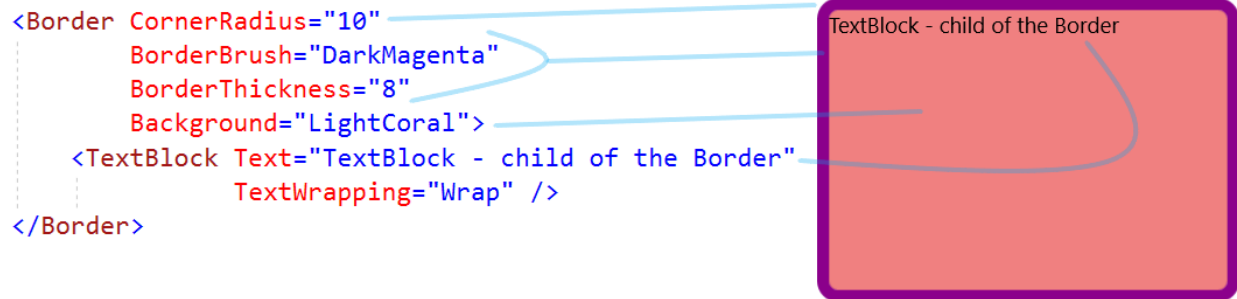


Figure xx – a Border element, showing typical settings for the three specific properties of Border, namely BorderBrush, BorderThickness, and CornerRadius. The standard Background property, available for most elements, is also set to show how it fits in with the other properties.

Like other elements, Border also has a Background property. Background and BorderBrush can be the same color, but that’s redundant; just use Background with BorderThickness at its default of zero if you don’t need the border to be a different color from the background.

Like the Background property, BorderBrush is of type Brush. That means it can also be set to draw the border with textures, gradient fills, and other fancy effects. I’ll tell you more about the Brush class in Chapter 6. I’ll also include details on the color system in XAML. For now, I’ll stick to named colors for all Brush settings, just as I’ve been doing for the last two chapters.<sup>9</sup>

The CornerRadius property allows the Border to have rounded corners. Each corner can have a different rounding, but you’ll often want to make them all the same to get the modern UI “rounded rectangle” look. The CornerRadius property is set by either entering a single number for all four corners, or by entering four comma-delimited numbers. Figure xx shows such a rounded border.

If plain rounded rectangles are too prosaic for your taste, you can get some interesting visual effects from CornerRadius quite easily. For example, a Border with CornerRadius="10, 10, 0, 0" only rounds the top corners, giving the border the visual outline of a piece of toast. Figure xx show such a Border, using appropriate colors to complement the toast metaphor.

---

<sup>9</sup> Named colors are those the XAML parser recognizes and knows the associated color channel values. When you are setting a property of type Brush, such as BorderBrush, in XAML, you’ll get a dropdown that shows the list of named colors.

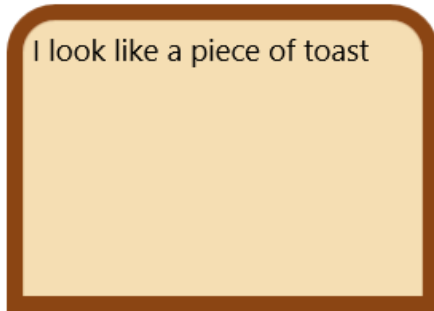


Figure xx – a Border with rounded corners on top, but square corners on bottom.

The XAML for this Border is quite similar to the earlier example, with just some property changes:

```
<Border CornerRadius="20,20,0,0"
        BorderBrush="SaddleBrown"
        BorderThickness="8"
        Background="Wheat" >
    <TextBlock Text="I look like a piece of toast"
              TextWrapping="Wrap" Margin="5" />
</Border>
```

Another effect we've used for applications is to use opposite rounded corners. This gets away from the harsh, square rectangles that are too often used in modern Windows apps without making everything look like Apple-ish rounded rectangles. Figure xx shows several borders with the `CornerRadius` set to round opposite corners, and with various Background colors.

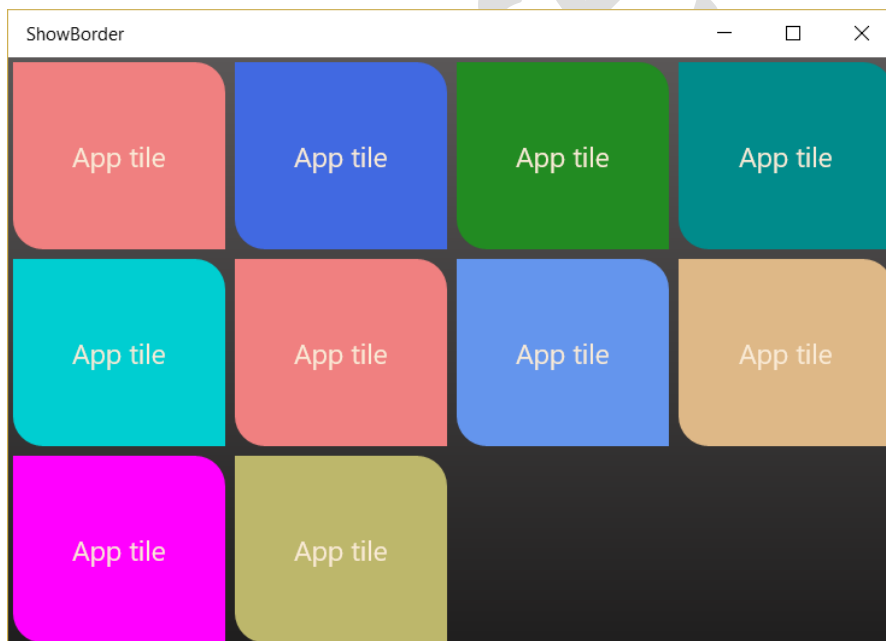


Figure xx – A set of Border elements, all with `CornerRadius` set to "0,20,0,20" to get rounded opposite corners. The child `TextBlocks` are aligned to the center of each Border. The complete XAML for this view is in the code downloads for this book, but it's a useful exercise to see if you can replicate this view.

BorderBrush and BorderThickness are not set in that example. CornerRadius still works even when those other Border properties have not been set.

It's common for the child of a Border to be a Grid or other Panel, just as it's common for Content of a Button to be a panel. That allows the Border to be drawn around a group of related elements.

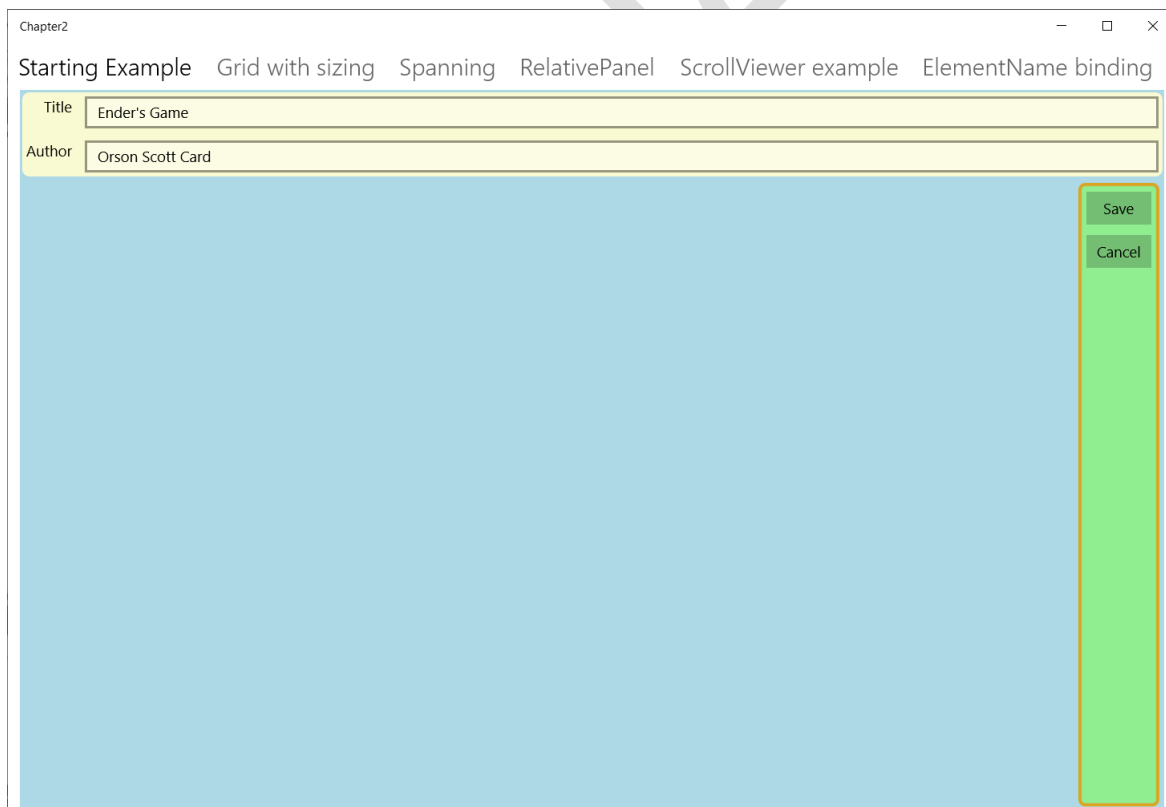
Most controls, such as the Button and TextBox classes, have their own built-in border, so it is unnecessary to place them in a Border decorator.<sup>10</sup> Borders are most useful as a container for panels.

## Pivot control

Most UI technologies have something called a "Tab" or "TabControl". This visual element allows a screen to show different sub-screens when the user chooses an individual Tab.

XAML for the Uno Platform doesn't have such a control.<sup>11</sup> The closest equivalent is the Pivot control.

Pivot isn't as flexible as Tab, but it looks nicer and has better touch support. Figure xx shows a typical screen that uses a Pivot. It's from the application that holds most of the XAML referenced in Chapter 2, and the individual screens are for examples in the chapter.



<sup>10</sup> To be precise, most controls have a Border element as part of their control template. That template contains the visual sub-parts that make up the rendering of the control. You'll get a brief introduction in chapter xx.

<sup>11</sup> That is, UWP does not have such a control in the current version. The element set is likely to be expanded in future versions, so we may see something at some point that resembles a more traditional tab-style control.

Figure xx – In the program you download that has examples from this book, a Pivot is used to separate the examples in a chapter.

The simplest use of Pivot is to give it a collection of PivotItem controls, one for each “tab” you need. Each PivotItem is a HeaderedContentControl. That means it has a Content property, which holds the part of the screen that gets switched out for each tab, and a Header property, which holds content for the header area of the PivotItem. Here is an example with three PivotItems, each just holding a differently-colored Ellipse as content.

```
<Pivot>
  <PivotItem Header="Forest Green Ellipse">
    <Ellipse Margin="10" Fill="ForestGreen" />
  </PivotItem>
  <PivotItem Header="Steel Blue Ellipse" Foreground="DarkGreen">
    <Ellipse Margin="10" Fill="SteelBlue" />
  </PivotItem>
  <PivotItem>
    <PivotItem.Header>
      <StackPanel Orientation="Horizontal">
        <Rectangle Fill="PaleTurquoise" Width="20" />
        <TextBlock Text="Pale Turquoise Ellipse" />
        <Rectangle Fill="PaleTurquoise" Width="20" />
      </StackPanel>
    </PivotItem.Header>
    <Ellipse Margin="10" Fill="PaleTurquoise" />
  </PivotItem>
</Pivot>
```

The Header for the last PivotItem is defined with element syntax to demonstrate yet again that content areas can be filled with anything you like. That allows you, for example, to put visual indicators on PivotItem headers for sub-screens that have errors or warnings. Figure xx shows this Pivot control, once with the first PivotItem selected and one with the last PivotItem selected.

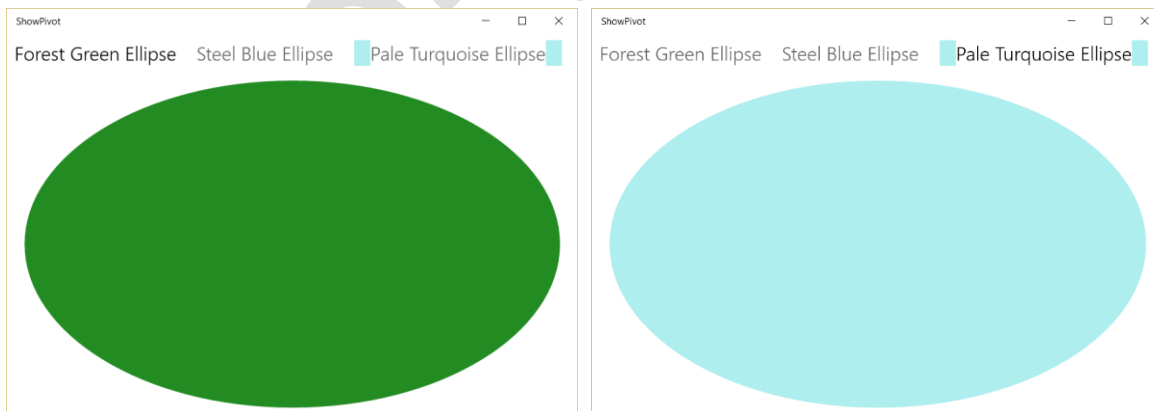


Figure xx – A simple Pivot control with three PivotItems. To the left, the first PivotItem is selected. On the right, the third PivotItem is selected. The third PivotItem has its Header property defined to hold more than just a string.

The Pivot has one significant difference in behavior from TabControl in WPF. If the view is too narrow to hold all of the headers for PivotItems, then hovering a mouse over the header area will yield navigation



buttons on each side of the headers. Also, in this dynamic mode, the currently selected `PivotItem` will automatically have its header positioned as the first header, and the other headers will behave as a carousel. You can see this behavior in Figure xx, in which the second `PivotItem` is selected, so its header is the first header displayed in the header area.

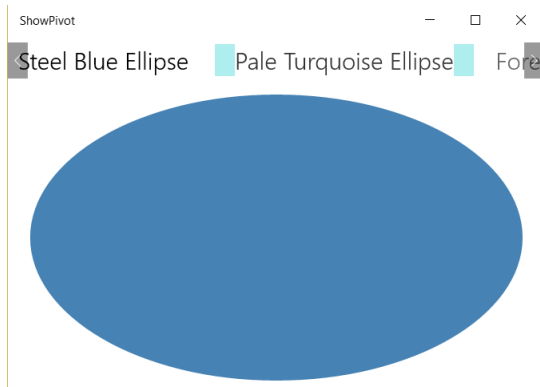


Figure xx – When headers are too big for the view, the Pivot puts the selected item’s header first, and the other headers become a wrapped carousel controlled by buttons on each side of the headers. The carousel control buttons only appear when the mouse is over the header area.

The selected `PivotItem` can also be changed with keystroke navigation. When headers are in focus, right and left arrow buttons will navigate among `PivotItems`. When the content of a `PivotItem` is in focus, `Ctrl-Left` and `Ctrl-Right` will navigate among `PivotItems`.

These keyboard behaviors are not obvious or easily discoverable, so you may want to have a `ToolTip` or other means for your users to discover them.

The Windows 10 SDK contains a sample that shows more of the behavior of the Pivot control. The solution is named simply “Pivot”, and if you find yourself using Pivot a lot, you ought to invest some time exploring that sample program. However, we have covered the basics that you need to use Pivot in routine ways.<sup>12</sup>

Because Pivot was designed for simplicity and is most at home in touch-based interactions, if you have a sophisticated desktop application, you may find Pivot limiting. I have, on occasion. Fortunately, it is straight-forward to combine two other controls to get pivot-like functionality with basically unlimited flexibility. We will do a short example in Chapter 5 to illustrate that.

The Pivot control is descended from `ItemsControl`, so there are scenarios where the `PivotItems` in the Pivot are driven by data from a collection. Chapter 5 has an example. However, Pivot is more commonly used by defining the `PivotItems` in it individually, as in the example above.

## Image control

In the modern software world, images and photos are everywhere. Many applications need to display those images, and that’s done in XAML with the `Image` control.

---

<sup>12</sup> The Windows 10 SDK samples are a rich area for exploration. But you’ll find that they are a lot easier to understand once you have gained some fluency with XAML. They are constructed to show off functionality, but not necessarily to make it easy for a XAML noob to find where the functionality exists within the sample.

The main property you need to set for an Image control is the Source property, which points to the location of the image. It can take a file name on a local or network disk, an image that's in your project, or a URL for an image on the web.

You don't usually want to place images inside your Visual Studio project, because that inflates the size of the EXE or DLL for the application. You might want to do that for small images used as icons, but as we'll see in Chapter 6, you have better alternatives for icons than bitmapped images. You will see such embedded images in some of my demonstration programs, but that's just for convenience of distribution.

By default, an Image control will maintain the aspect ratio<sup>13</sup> of the image or photo it contains, and will grow or shrink the image to fit within the space given to the Image control. For other possibilities on how the image is handled, you should check the Stretch property.

It's rare in typical applications for the Source property to be hard-coded to a particular image location. More commonly, Source is set with a data binding to some dynamic data. Because of that, I'll defer examples for the Image control to the next chapter, in which data binding takes center stage.

## Flyout

You're no doubt familiar with the idea of a ToolTip, which displays additional information on a hover. ToolTip is available in UWP XAML, and we'll talk about it below. However, UWP includes another element for popup style functionality. It's a more modern alternative to the ToolTip control called Flyout. It's a ContentControl, so it can show anything XAML can display.

I use Flyout more than ToolTip in Uno and UWP apps because it's just as flexible in displaying contents, but it can also have interaction with the user. I also find Flyout to be a better fit for apps with touch interaction.

The simplest use of Flyout is to display some temporary information, much like a tooltip. However, unlike a traditional tooltip, the information stays around until the user dismisses it by clicking or tapping somewhere on the view outside the Flyout. Also, Flyout does not appear on a hover – it only appears in response to a tap, click or other interaction event.

There are several ways to use Flyout. In this chapter, I'm only going to show a couple of common ones:

- Using a Flyout like a tooltip
- Using a Flyout for more information and confirmation of user action

In the chapter "Useful Things, Part 2" I will show additional uses of Flyout, and discuss related interaction needs such as dialog boxes.

## Flyout as substitute for tooltip

In its simplest form, Flyout can just display some read-only content. Controls such as Button have a Flyout property, and setting that property will display the Flyout content on a click or tap. Here is some typical XAML that displays read-only content:

---

<sup>13</sup> Aspect ratio is the proportion of the width and height of the image. An image that maintains its aspect ratio gets bigger or smaller, but is not distorted.

```

<Button Click="Button_Click" Grid.Row="1">
  <Button.Flyout>
    <Flyout>
      <StackPanel>
        <TextBlock Text="You have used 7 of 10 credits" />
        <StackPanel Orientation="Horizontal">
          <Rectangle Height="20" Width="70" Fill="SteelBlue" />
          <Rectangle Height="20" Width="30"
            Stroke="SteelBlue" StrokeThickness="2" />
        </StackPanel>
      </StackPanel>
    </Flyout>
  </Button.Flyout>
  Use credit
</Button>

```

Figure xx shows the result when the Button is clicked. The Flyout appears, and the logic in the attached click event for the button also runs. When the user clicks or taps anywhere else, the Flyout disappears.<sup>14</sup>

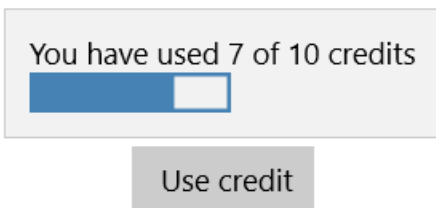


Figure xx – A Flyout with read-only content. It appears when the Button is clicked or tapped, and is dismissed when the user clicks or taps somewhere else.

#### Flyout with user interaction

Unlike traditional tooltips, Flyout can also receive user interaction. This is useful for things such as confirmation of an action.

Here is a typical interactive usage for flyout. Figure xx shows a Button for processing a batch of something. The Button also has another Button inside it with content of a SymbolIcon showing a symbol for video. That internal Button has an attached Flyout to tell the user what the action is doing and get a confirmation that they want to do it.

<sup>14</sup> Naturally, in a real app, the “progress” on using credits would be data driven. However, we need to discuss some more data binding concepts first, so we will see a similar data-driven example in the next chapter.

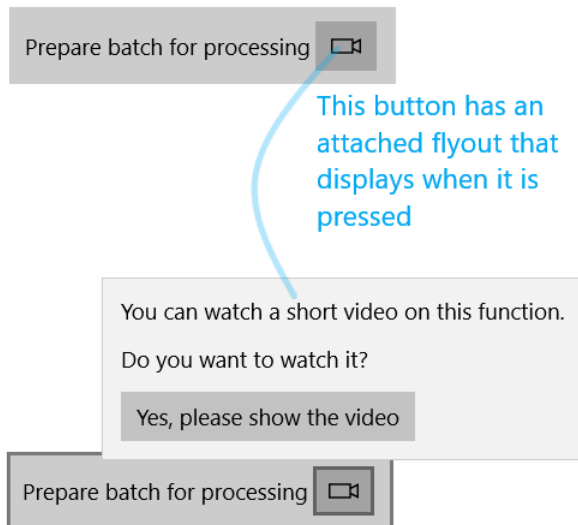


Figure xx – a Button with another Button inside showing a video symbol. The internal button has a Flyout that shows when it is pressed, which is shown at the bottom of the figure.

Here is the XAML for Figure xx. The syntax for the interactive Flyout on the embedded Button is shown in bold.

```
<Button Margin="10">
  <StackPanel Orientation="Horizontal">
    <TextBlock Text="Prepare batch for processing"
      VerticalAlignment="Center" />
    <Button Margin="3">
      <SymbolIcon Symbol="Video" />

      <Button.Flyout>
        <Flyout>
          <StackPanel>
            <TextBlock Text="You can watch a short video on this function."
              Margin="0,0,0,12" />
            <TextBlock Text="Do you want to watch it?" Margin="0,0,0,12" />
            <Button Content="Yes, please show the video" />
          </StackPanel>
        </Flyout>
      </Button.Flyout>
    </Button>
  </StackPanel>
</Button>
```

The example does not include attachment to the events or commands that would actually do something.<sup>15</sup> It's just to show you how to compose several pieces to get a useful interaction pattern.

If you are not used to thinking about compositional interfaces, you probably would not think it helpful to have a Button inside another Button. But with the Flyout capability, that internal Button can be part of an intuitive UI design.

<sup>15</sup> You folks are developers, so I don't worry much about you figuring out anything that involves code. We've already talked in Chapter 1 about code behind, and we'll be talking about Commands in chapter x.

## ToolTip control

Tooltips have been a mainstay of mouse-centric interfaces for over twenty years. They don't work as well with touch-centric interfaces, because touch interaction has no hover capability<sup>16</sup> to activate the tooltip. However, as Windows 10 becomes the default operating system on corporate desktops, UWP becomes more commonly used to create applications that are mouse and keyboard driven. For those cases, tooltips are still helpful.

In fact, such interfaces in XAML have the capacity to include much more helpful tooltips. That's because the ToolTip control, which displays the tooltip, is a ContentControl. The content of the ToolTip can be anything that's helpful to the user, and the content can use data bindings to become data driven. Figure xx has an example tooltip from a XAML application<sup>17</sup>:



Figure xx – A tooltip showing data driven contextual information. In this case, a potential temporary employee has a photo, skill set, and available schedule which appear when the cursor hovers over the potential employee's name.

In an application that expects mouse interaction, the ToolTip control can be one of the ways you give information to users in layers of detail. The ToolTip can contain information not quite as important as what you display all the time, but still helpful if a user needs to drill down a bit to make a decision or check an option. However, ToolTip is not capable of any interaction by the user – you can't put a Button or other interactive element in a ToolTip for a user to access, as you can with a Flyout.

A tooltip for an element is set with an attached property, using the ToolTipService class. To set a plain string tooltip, just set ToolTipService.ToolTip to a string, like this:

```
<Button ToolTipService.ToolTip="This is the tooltip for the button"
```

<sup>16</sup> Some stylus based interfaces do detect hover, and can use a tooltip. But tooltips are rarely used for anything but mouse-centric interfaces. Some applications use a touch "press and hold" interaction to pop up a tooltip, but it's more common for press-and-hold to bring up a context menu.

<sup>17</sup> This application was written in 2008 on the earliest XAML platform, Windows Presentation Foundation. However, XAML in UWP has the same capabilities for producing useful, data-driven tooltips.

```

        FontSize="20" Margin="10">
    I'm a button with a tooltip when you hover over me
</Button>

```

But that's no fun. To really get a helpful tooltip, we need to drop into element-based syntax, and put some content in the tooltip. Here's an example:

```

<Button FontSize="20" Margin="10">
    <ToolTipService.ToolTip>
        <ToolTip>
            <Grid MaxWidth="200">
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="*" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>
                <SymbolIcon Foreground="CadetBlue" Symbol="AddFriend" />
                <TextBlock Grid.Row="1" TextWrapping="Wrap"
                    Text="We can have symbols, images, or data. Isn't that useful?" />
                <SymbolIcon Grid.Row="2" Foreground="ForestGreen"
                    Symbol="BrowsePhotos" />
            </Grid>
        </ToolTip>
    </ToolTipService.ToolTip>
    I'm a button with a tooltip when you hover over me
</Button>

```

When activated with a mouse hover, that tooltip will look like Figure xx.



Figure xx – A ToolTip control that uses a three row Grid to hold two symbols and a TextBlock. The layout inside the ToolTip can be as complex as desired, because ToolTip is a ContentControl.

ToolTip can display any content you place in it, but as mentioned earlier, the user can't interact with it. If you put a Button in a ToolTip, for example, the user can't press it.

But the ToolTip works well for simple, read-only information. We'll see examples of data driven tooltips in the next two chapters.

## Option Controls

Most applications need to allow users to make a choice among various options. A flexible way to do that is to present the user with a drop-down list, typically in a ComboBox control. ComboBox is a list-oriented control, and we're going to look at those<sup>18</sup> in chapter 5.

<sup>18</sup> List-oriented controls are descended from ItemsControl, which was briefly mentioned in the previous chapter. In Chapter 4, we'll look at ListView, GridView, FlipView, ListBox, and ComboBox, which are the most commonly used list-oriented controls. We'll also see the relationship of Pivot to those controls.

For simpler choices, the user might need to turn something on or off, or select from a small list of options that does not require a drop-down control. For the simple on/off switch capability, XAML offers two controls, `ToggleSwitch` and `CheckBox`. For a small list of options, XAML has a `RadioButton` control that behaves much as `RadioButton` controls do in older UI technologies.

## ToggleSwitch and CheckBox

For situations where the user needs to select a true or false option, or turn something on and off, you have two choices. `ToggleSwitch` is more modern in appearance while `CheckBox` is more compact and has a more traditional look. Figure xx shows both of them side by side, with the possible states of each.

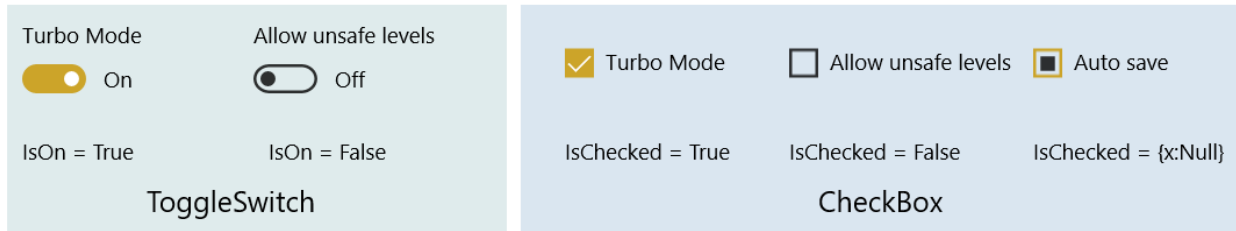


Figure xx – Side by side visual comparison of `ToggleSwitch` and `CheckBox` controls, showing possible states for each. `ToggleSwitch` does not have a null state.

## CheckBox

The controls differ in more than visual appearance. Let’s look at `CheckBox` first, since you are likely to be familiar with a similar control in older UI stacks. Then we’ll see how `ToggleSwitch` varies from it.

Most older UI technologies have a control named “`CheckBox`”, and the one in XAML behaves superficially much like those older controls. It has a Boolean property named `IsChecked` to indicate a true or false condition, and the application can then map that true/false to some application option. It also has `Checked` and `Unchecked` events to detect when the user changes the state of the `CheckBox`.

`IsChecked` is a nullable Boolean, which means it can also have an indeterminate state. Some older `CheckBox` controls in other UI stacks have this capability too. The indeterminate state means “the user hasn’t made a choice on this option”.

The `CheckBox` in XAML has a different visual appearance for each of the states. Figure xx shows a `CheckBox` for each of the values of `IsChecked`, true, false, and null (indeterminate). Here is the XAML for those three `CheckBox` elements, with the `IsChecked` property settings bolded:

```
<CheckBox Grid.Column="2" Content="Turbo Mode" Margin="30,10,5,10"
  IsChecked="True"/>>
<CheckBox Grid.Column="3" Content="Allow unsafe levels" Margin="10,10,5,10"
  IsChecked="False"/>>
<CheckBox Grid.Column="4" Content="Auto save" Margin="10,10,5,10"
  IsChecked="{x:Null}"/>>
```

`CheckBox` is a `ContentControl`, so the information to the right of the checked/unchecked box can be anything you want to show. For example, you could use a `SymbolIcon` instead of text as content.<sup>19</sup>

---

<sup>19</sup> You would need to use default property syntax or element syntax for that, of course. We’ve discussed this, so I’m sure you already knew that.

CheckBox is intended for applications that primarily use mouse and keyboard. For touch applications, a CheckBox has to be so large it starts to look like something on a toddler's toy.

## ToggleSwitch

For turning options on and off, modern applications have begun to use a different visual behavior that is more suggestive of a physical switch, as you saw in the side-by-side comparison in Figure xx. In XAML, this touch-optimized alternative to CheckBox is called ToggleSwitch.<sup>20</sup>

Besides the difference in visual appearance, ToggleSwitch uses the IsOn property instead of IsChecked, and IsOn doesn't have an indeterminate state. ToggleSwitch also uses a single event named Toggled to detect changed state instead of different events for different state changes.

Figure xx shows ToggleSwitch in its two states. Here is the XAML for the two ToggleSwitch elements in that figure:

```
<ToggleSwitch Header="Turbo Mode" Margin="10,10,5,10"
              IsOn="True"/>
<ToggleSwitch Header="Allow unsafe levels" Grid.Column="1" Margin="0,10,30,10"
              IsOn="False"/>
```

While CheckBox is a ContentControl, ToggleSwitch is not. The property you set to get the label above the switch is Header, as shown in the XAML above.

Even though the name is different, you can use the Header property the same way you use a Content property in a ContentControl. For example, you could set the Header of a ToggleSwitch to a StackPanel holding a TextBlock and a SymbolIcon, using XAML like this:

```
<ToggleSwitch Grid.Row="4" Margin="10,10,5,10"
              IsOn="True">
  <ToggleSwitch.Header>
    <StackPanel Orientation="Horizontal">
      <SymbolIcon Symbol="Flag" />
      <TextBlock Text="Turbo mode" Margin="3" />
    </StackPanel>
  </ToggleSwitch.Header>
</ToggleSwitch>
```

Figure xx shows this ToggleSwitch rendered.

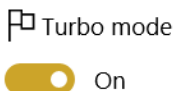


Figure xx – A ToggleSwitch with a Header made up of a symbol and text, stacked horizontally in a StackPanel.

The ToggleSwitch control has an additional capability that CheckBox does not for displaying information to the user about the option it controls. In addition to the Header property, which as we saw in Figure xx

---

<sup>20</sup> Don't confuse ToggleSwitch with ToggleButton, which is an ancestor of CheckBox and is only preferred in rare scenarios. It's likely you'll never need ToggleButton, so you may choose to take it out of the Visual Studio toolbox to prevent confusion with ToggleSwitch.



displays above the switch visual, ToggleSwitch also has OnContent and OffContent properties. When IsOn is True, the OnContent shows, and when it's False, the OffContent shows. By default, they both show to the right of the switch visual.

OnContent and OffContent are set in XAML just like Content in ContentControls, so you can put any visual you like in these properties. SymbolIcon comes in handy for OnContent and OffContent. Here is the XAML for a ToggleSwitch using SymbolIcon to supply on and off visuals. Figure xx shows the ToggleSwitch in on and off states.

```
<ToggleSwitch Grid.Row="4" Grid.Column="3"
    Header="Video or Audio" >
    <ToggleSwitch.OnContent>
        <SymbolIcon Symbol="Camera" />
    </ToggleSwitch.OnContent>
    <ToggleSwitch.OffContent>
        <SymbolIcon Symbol="Microphone" />
    </ToggleSwitch.OffContent>
</ToggleSwitch>
```

Video or Audio



Video or Audio



Figure xx – A ToggleSwitch with SymbolIcon elements for OnContent and OffContent. Only one of the SymbolIcon elements is shown at a time, depending on the value of the IsOn property.

You can also create your own visuals to go in OnContent and OffContent. For example, here is some XAML that uses shapes and TextBlocks to put up simple visual indicators for a switch to turn on or pause recording. The visual result is shown in Figure xx.

```
<ToggleSwitch Grid.Row="5" Grid.Column="4" Header="Video is">
    <ToggleSwitch.OffContent>
        <StackPanel Orientation="Horizontal">
            <StackPanel>
                <Rectangle Fill="Black" Margin="3"
                    Height="6" Width="25" />
                <Rectangle Fill="Black" Margin="3"
                    Height="6" Width="25" />
            </StackPanel>
            <TextBlock Text="Paused" Margin="3" />
        </StackPanel>
    </ToggleSwitch.OffContent>
    <ToggleSwitch.OnContent>
        <StackPanel Orientation="Horizontal">
            <Ellipse Height="25" Width="25" Fill="Firebrick" />
            <TextBlock Text="Recording" />
        </StackPanel>
    </ToggleSwitch.OnContent>
</ToggleSwitch>
```



Figure xx - A ToggleSwitch with shapes and TextBlocks to furnish different visual appearance for on and off. The user can toggle the state by clicking the visual switch, or by clicking the OnContent or OffContent visual.

This example isn't very interesting visually, but we'll see lots more possibilities for putting graphics in content in Chapter 6. Figure xx shows a preview, in which the ToggleSwitch indicates whether a cow is dead or alive by having a normal graphic for the cow for IsOn=True and the same graphic flipped upside down for IsOn=False. We'll cover the complete process for creating that example in Chapter 6.



Figure xx – A ToggleSwitch with a simple graphic of a cow used for OnContent and OffContent. The OffContent version of the cow is flipped upside down. This gives an intuitive idea to the user that the animal is either alive or dead for different states of IsOn. Chapter 6 will go into detail on building the graphics for this example.

I prefer ToggleSwitch for most of my visual designs in Universal Windows applications that require the user to select a True/False or On/Off option. It works well for both touch and mouse interaction, and is modern looking. CheckBox looks a bit old-fashioned, but if you are only using a mouse, are tight on space, or need an indeterminate state, it will serve your needs.

## RadioButton

Like CheckBox, RadioButton has been around a long time in other UI technologies, and RadioButton works in XAML very similarly to its older cousins. Also like CheckBox, each RadioButton has an IsChecked property that can be True, False, or null.

The most common use it to place a group of them in a container, such as a StackPanel. Then only one of them can have IsChecked as True; clicking on a different one in the group causes the new one to have IsChecked as True, and switches IsChecked on the previously selected RadioButton to False.

A group of RadioButton controls works well for choosing one of four or five options. For a longer list of options, a drop-down generally works better. In fact, if you're working on a small screen, you may want to avoid using RadioButton altogether.

If for some reason you can't have your group of RadioButton controls all in the same container, you can group them together by giving them all the same GroupName property. GroupName is just a string, so you normally set it to something suggestive of the group's function.

Here is the XAML for a typical set of RadioButton controls in a StackPanel. Figure xx shows the visual result.

```
<StackPanel>
  <TextBlock Text="Are you addicted to writing code?" Margin="4" />
  <RadioButton Margin="4" Content="No. Well not much." />
  <RadioButton Margin="4" Content="Only a little bit." />
  <RadioButton Margin="4" Content="Quite a bit." />
  <RadioButton Margin="4" Content="Have to write 1000 lines a day." />
</StackPanel>
```

Are you addicted to writing code?

- No. Well not much.
- Only a little bit.
- Quite a bit.
- Have to write 1000 lines a day.

Figure xx – A typical set of RadioButton controls. They reside in the same StackPanel, so only one can be selected at any time.

## Working with Text

In Chapter One, we covered the basics of the TextBox for editing text, and the TextBlock for display of text. To briefly recap, both of them have properties to control the typeface, size and so forth, including:

- FontSize
- FontWeight (e.g. Bold)
- FontFamily
- FontStyle (e.g. Italic)

### TextBox

TextBox has lots of properties for controlling the text inside, such as SelectionStart and SelectionLength. I'll leave it to you to explore those properties; if you have used a TextBox control in other technologies, you won't have any trouble figuring out the one in XAML.

There are a couple of minor points I'll mention that might not be obvious.

1. A TextBox does not wrap text by default. Instead, it will scroll text back and forth inside itself if the text gets too wide. To get it to wrap, set the TextWrapping property. Besides the default of NoWrap, it can be set to Wrap or WrapWholeWords.
2. If you do set a TextBox to wrap, by default it will stretch vertically to show as many lines as it needs to. If you put one in a StackPanel or auto-sized Grid row, it will get as tall as it needs to show all the text. To constrain it, use MaxHeight on the TextBox or its container.

### TextBlock

The TextBlock element was introduced in Chapter One, and it's been used in lots of examples since then. Typical XAML for it, as we've seen several times, looks like this:

```
<TextBlock Text="Some text to display" FontSize="20" />
```

Normally, the Text property is set. Optionally, various font properties can be set, and the text color can be set with the Foreground property.<sup>21</sup>

TextBlock is a lot more flexible than our simple usage indicates, though. It can display sections of text with different formatting.

To do that, the syntax changes to allow the individual sections to be declared. Each section is declared in an element called a Run. Each Run has a Text property value, and optionally varying property values for font and foreground.

When using Run elements, the Text property for the TextBlock itself is left blank, but the TextBlock may declare other properties such as a default Foreground and FontSize for all Runs. Each Run may override those properties if needed. Here is the XAML for some formatted text in a TextBlock:

```
<TextBlock Grid.Column="2" FontSize="20"
    Foreground="SteelBlue" TextWrapping="Wrap">
    <Run Text="The TextBlock element can have " />
    <Run FontSize="28" Foreground="DarkBlue"
        Text="different sections with " />
    <Run FontSize="16" FontStyle="Italic"
        Text="different sizes, styles, and colors " />
    <Run Text="and it will combine them and wrap as needed." />
</TextBlock>
```

Figure xx shows the resulting TextBlock. The visual result combines all the sections of text, and wraps the text based on the room given the TextBlock.<sup>22</sup>

The TextBlock element can  
have different  
sections with *different*  
*sizes, styles, and colors* and it will  
combine them and wrap as  
needed.

Figure xx – A TextBlock with four Run elements, showing various ways sections of the text can be formatted. The text is put together from the sections, and wrapped as necessary if TextWrapping is set to Wrap.

While this isn't as flexible as formatting with HTML, formatted text in a TextBlock is often quite sufficient for business applications. As we will see in the next chapter, using separate Run elements also allows part of the text in a TextBlock to vary with data. That's phenomenally useful and it will show up in many of our examples in the next two chapters on working with data in XAML.

---

<sup>21</sup> TextBlock has no Background property. It shows only the text displayed, with its background determined by the container or other element it is on top of.

<sup>22</sup> Beginning XAML developers sometimes try to get some of this formatting capability by stacking several TextBlock elements in a StackPanel. There are times that works OK, but it lacks control over wrapping the text. For text with any significant formatting, using Run elements is easier and more flexible.

Besides including Run elements, the sections in a TextBlock can also have a LineBreak element to start a new line. Typical XAML looks like this:

```
<TextBlock FontSize="18" TextWrapping="Wrap">
  <Run Text="If you suffer from code addiction, remind yourself each day:" />
  <LineBreak />
  <LineBreak />
  <Run FontWeight="Bold" Text="I am a code addict" />
  <LineBreak />
  <LineBreak />
  <Run Text="This may help you face your problem." />
</TextBlock>
```

Visual results from this XAML are in Figure xx.

If you suffer from code addiction, remind yourself each day:

**I am a code addict**

This may help you face your problem.

Figure xx – A TextBlock formatted with LineBreak elements.

## Margin and Padding

Thinking in XAML means your elements should have a lot of dynamic layout and sizing. This doesn't mean you lose all control over how things arrange and size. It just means you use different means to define how you want things to look.

I've mentioned a few times<sup>23</sup> that you should not be hard-coding sizes of most elements. Elements should be dynamically sized based on their containers and their contents. Chapter 2 talked a lot about that.

But if elements are being dynamically sized and laid out, it's necessary to keep things from bumping up against one another. We saw the main way that's done in Chapter 2 – the Margin property. In this section, we'll look at Margin in some more depth, and also contrast it with a cousin, the Padding property.

### Margin

To recap what we say in Chapter 2, all XAML visual elements have a Margin property. It sets the amount of space that will surround the element to keep it from bumping into things.

A lot of our examples have a constant Margin setting for all four sides of an element. A Rectangle that needs 20 units of space on all sides can have Margin="20" as a property setting.

Briefly mentioned in Chapter 1 was that Margin can be set separately for top and bottom, or for all four sides. Here are three Rectangle elements in XAML with Margin settings for each of those cases:

---

<sup>23</sup> Or maybe too many times, but it's important, so I keep saying it.

```

<!--Same Margin on all four sides-->
<Rectangle Margin="10" Fill="SteelBlue" />

<!--Same Margin on top/bottom and right/left-->
<Rectangle Margin="10, 20" Fill="CadetBlue" />

<!--Different Margin for all four sides-->
<Rectangle Margin="10, 20, 5, 30" Fill="CornflowerBlue" />

```

If you place a Rectangle with a Margin setting in a Grid cell, assuming the Rectangle has its default alignment properties of Stretch, then the Margin will offset the Rectangle from the sides of the cell. If you put such a Rectangle in a vertical StackPanel, the side Margin settings will set the Rectangle away from the sides of the StackPanel, and the top and bottom Margin settings will set the Rectangle away from other elements in the stack.<sup>24</sup>

But there's one twist I haven't told you about yet, and a very useful one as it turns out. I had been using XAML about a year before I discovered that Margin settings don't have to be positive!

If an element has a negative Margin setting in a Grid cell, the edge of the element can go outside the cell. If an element in a StackPanel has a negative Margin on top, it can overlap with the element above it in the stack.

Remember the Border that looked like a piece of toast? Let's make a loaf of them using Margin settings, including some negative ones. Here's the XAML:

```

<StackPanel Width="200">
  <Border CornerRadius="20,20,0,0" Height="120"
    BorderBrush="SaddleBrown"
    BorderThickness="8"
    Background="Wheat"
    Margin="0,0,60,0">
    <TextBlock Text="I look like a piece of toast"
      TextWrapping="Wrap" Margin="5" />
  </Border>
  <Border CornerRadius="20,20,0,0" Height="120"
    BorderBrush="SaddleBrown"
    BorderThickness="8"
    Background="Wheat"
    Margin="20,-60,40,0">
    <TextBlock Text="I look like a piece of toast"
      TextWrapping="Wrap" Margin="5" />
  </Border>
  <Border CornerRadius="20,20,0,0" Height="120"
    BorderBrush="SaddleBrown"
    BorderThickness="8"
    Background="Wheat"
    Margin="40,-60,20,0">
    <TextBlock Text="I look like a piece of toast"
      TextWrapping="Wrap" Margin="5" />
  </Border>

```

---

<sup>24</sup> Don't forget that the Rectangle needs to have a Height setting to display in a StackPanel. Otherwise, the default Height will be zero, and the StackPanel doesn't stretch elements vertically just because the VerticalAlignment is Stretch. We covered that in Chapter 2 in that section on panels and layout that you probably skipped because it looked boring.

```
</StackPanel>
```

The visual result is shown in Figure xx.

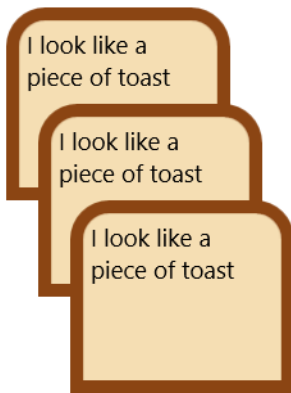


Figure xx – Three `Border` elements are stacked like a loaf using negative `Margin` settings on the top for the lower `Border` elements. Differing side `Margin` settings are also used to offset the `Border` elements horizontally.

Negative `Margin` settings are invaluable in final tweaking to get desired layouts. We will see them used in several examples of a XAML capability called data templates in the next two chapters.

## Padding

`Margin` settings are available for all elements, and set aside space around the *outside* of the element. By contrast, the `Padding` property is available for some containers, and it sets aside space *inside* the container to push child elements away from the interior edge of the container.

Figure xx shows a diagram, including some XAML and the visual results.

```
<Border Grid.Row="1" Grid.Column="1"
        BorderBrush="DarkSeaGreen"
        BorderThickness="5"
        CornerRadius="6"
        Padding="20, 30">
  <Grid >
    <Rectangle Margin="40, 50"
              Fill="SteelBlue" />
  </Grid>
</Border>
```

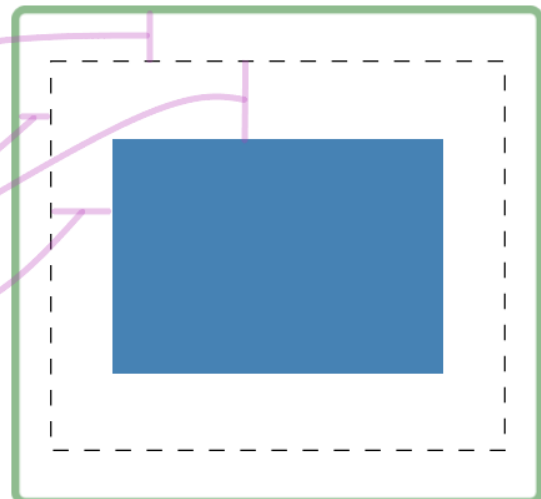


Figure xx – `Margin` and `Padding` defined in XAML and shown graphically. `Margin` sets aside space outside an element. `Padding` sets aside space inside a container, pushing child elements away from the edge.

The SteelBlue Rectangle is pushed into the interior partially by the Border's Padding and partially by its own Margin.

Unlike Margin, Padding can only be positive. Negative Margin settings can encroach on Padding settings, resulting in less Padding than the setting would suggest.

Containers that have a Padding property include Grid, StackPanel, RelativePanel, Border, and all ContentControls such as Button and ToolTip.

### Visibility of elements

Every visual element has a property named Visibility. It takes one of two values – Visible or Collapsed.

Visible is self-evident. But you need to understand a nuance about Collapsed.

If an element's Visibility property is set to Collapsed, then for rendering purposes, it doesn't exist. It's not just hidden – it doesn't take up any space at all.

So, if I have a stack of three buttons, but the middle one is set to Collapsed, then the stack will just have the first and last one shown. There will be no space between them.

This turns out to be quite useful for dynamic layout. For example, for a layout that spoofs a mailing label, we can arrange to have the visual element for the second address line to disappear if that line is empty. The city and state will then cozy up right against the first address line, leaving no space.

However, to do this in practice requires some data binding capabilities that we have not yet covered. So that example must wait until the next chapter.

### Wrap Up

Now that we've learned quite a bit about how you compose XAML to get interesting results, and learned a number of useful elements to use in that composition, it's time to get down to business. As in, looking in depth at data binding and other concepts for working with data, which is the reason most business applications exist.

The next chapter is called "Working with Data", and it will go deeper on the data binding concepts we saw in Chapter One. The concepts in this chapter changed my conception of building business applications more than anything else in XAML. Buckle up for your own ride into data applications unlike any you've ever worked with before.