

Chapter 1

A High-Altitude Overview of XAML

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.

Learning XAML is faster and smoother if you start by understanding some key concepts and how they fit together. This chapter is your starting point.

Completeness is not the goal. There will be plenty of opportunity to fill in details later in the book.

In this chapter, we'll look at the most important ideas you need to know in their most basic form. The XAML topics I'll introduce in this chapter include:

- The raw basics of XAML as a markup language
- The Page class
- Automatic sizing and positioning of XAML elements, and discussion of the device independent units used for size and position
- Code-behind
- Two commonly used shapes, Ellipse and Rectangle, and an element for displaying common icons, SymbolIcon
- The TextBox and Button controls, and the TextBlock element
- The StackPanel and Grid containers
- Basics of data binding
- Using Visual Studio to create XAML applications for the Uno Platform

A step-by-step example is available in Appendix X if you would like an exercise that puts all these concepts together to produce a minimal, but functional, XAML program.

XAML – The Markup Language for the the Uno Platform and the Universal Windows Platform

The technology for XAML is built-in to the Universal Windows Platform. The Uno Platform has extended the use of XAML to other platforms, including iOS, Android, and WebAssembly for use in browsers.

All of these platforms have both a rendering engine for XAML applications and libraries that contain the classes you will use in XAML markup, such as Button and TextBox.

Some of these classes implement visual elements, such as controls and shapes. Others implement non-visual functionality, such as data binding.

You could write a complete XAML application using only code. Well, you could if you were a masochist. I doubt that, so in this book we'll talk about using XAML as a markup language to describe the user interface for an app on the Uno Platform.

XAML – A Markup Language for User Interfaces

XAML stands for eXtensible Application Markup Language. As mentioned in the Introduction, it's pronounced "xammel", rhyming with "camel". XAML for the Uno Platform can be written by hand, or it can be written by visual designers available in Visual Studio 2017 and higher.¹ Older versions of Visual Studio support a different flavor of XAML for other platforms, such as Windows Presentation Foundation (WPF).

Those visual designers are useful for some parts of user interface construction, but they won't do everything you'll need. Serious XAML developers will sometimes find it necessary to edit XAML directly. I'll introduce XAML to you briefly before looking at the visual designers.

XAML is XML with a certain schema. You can edit it with any text editor, as long as you produce valid XAML. However, the XAML editor in Visual Studio has Intellisense, and has a visual preview of the results, so normally that is where your XAML editing will take place.

There's nothing magical about XAML. Anything that's possible in XAML is also possible in code, though the XAML is often more compact. XAML is simply a convenient way to describe an interface of the platforms that support it.

A Sample Page of XAML

A page of XAML describes a set of instances of classes. Some are instances of visual elements that contribute to the visual appearance of a user interface. Others are helper classes for the visual ones.

You can set properties for those classes in XAML by using an XAML attribute. For example, the following line

```
<TextBox Text="Text for editing" />
```

declares an instance of the TextBox class, and sets the Text property of the class to a string.

¹ Two visual designers for XAML are installed with Visual Studio 2015 and 2017. One is available directly from Visual Studio, and when I speak of a visual designer, this is the one I'm talking about. Another visual designer is in a separate program called Blend for Visual Studio. It can do some things that the regular visual designer cannot. We won't need it until we look at some deeper concepts in Chapters 5 and 6.

XAML also determines how the class instances are grouped together. The grouping takes the form of a logical tree. There is a root element, which contains other elements. Those elements may then contain sub-elements, and so on.

Here's an example to illustrate the idea. This is real, valid XAML, though for a very simple and pointless user interface.

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

  <StackPanel Background="LightBlue">
    <Button Margin="5">
      I'm a button
    </Button>
    <TextBox Text="Text for editing" />
  </StackPanel>

</Page>
```

This small example includes some lines of XAML that are normally written for you by the Visual Studio XAML editor. If you want to try out the example above in a new XAML project in Visual Studio, you should not paste the entire example above. You should only paste the section bounded by the StackPanel tags.

The first line tells us that the root element we are using is a Page. We'll be discussing the Page class a bit later in this chapter. Right now, you can just think of it as representing a rectangular portion of the screen. Its nearest equivalent in other .NET user interface technologies is a Form in Windows Forms.

The declaration of the Page element includes several XML namespaces. These tell the XAML parser where to get the classes used in the XAML. Don't worry about these now. Chapter 3 discusses the nuances of XAML namespaces. For now, we'll focus on the XAML that includes the start and end tags for the StackPanel visual element and the lines in between.

The declaration for the StackPanel, Button, and TextBox classes contain XML attributes that set properties for those classes. You can set or change these properties in the Properties Window, which will write the appropriate XAML for you, or you can directly edit the attributes in the XAML.

Some of the attributes are self-explanatory. You could guess that the Background attribute of the

StackPanel specifies the background color for the StackPanel.²

I'm assuming that you have a minimal understanding of XML, at least to the point of understanding the difference between an XML element and an XML attribute. If that's not true for you, then you should gain some exposure to XML basics before working with XAML. Appendix A contains a short summary of basic XML terminology and concepts.

After the Page element, the next element is a StackPanel. As we'll see later, the StackPanel is a container for other elements such as buttons and textboxes. Because the Page is the root element, it contains the StackPanel as a sub-element or child element.

Why do we need the StackPanel? Why not just put the button and the textbox directly on the Page? Unlike forms in Windows Forms or pages in ASP.NET, *a set of buttons and textboxes can't be placed directly on a Page in XAML*. A Page does not know how to do layout of multiple child elements. It only knows how to hold a single child element.

To get anything more than a trivial Page with one element, the root element of the Page must be a layout container, that is, a container that knows how to arrange multiple elements. These layout-capable containers are called *panels*. StackPanel is one such panel.

The child elements of the StackPanel are the visual elements that will be inside the StackPanel. In our example, there are two. The first is a Button, which will contain the text "I'm a button". The second is a TextBox, which will contain the text "Text for editing".

The Button and TextBox have different XAML forms. The Button has a start tag, <Button>, and an end tag, </Button>. Between those tags is the string that will show in the Button. However, the TextBox does not have an end tag. Its declaration is simpler, and the slash before the closing angle bracket means that the definition of the TextBox is complete and no end tag is needed.

If we place the XAML above in an application, we will need to choose a target platform. If we choose the Universal Windows Platform, at runtime we will get a window on the Windows 10 desktop displaying the page. The window would look something like Figure 2-1.

² The full story on the Background property is quite a bit more complex. We'll see more about that later in the chapter, with even more detail in later chapters.

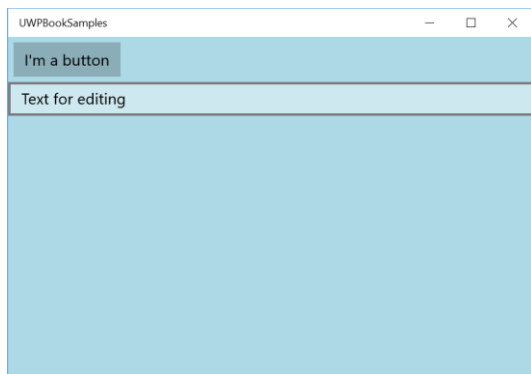


Figure 2-1 The first XAML example would produce a screen at runtime that looks similar to this one

There's a lot more to know about XAML and its syntax, but this brief introduction is enough for now.

Automatic Sizing and Positioning

Did you notice that the XAML didn't set the height or width of the `StackPanel`, the `Button`, or the `TextBox`? Even the `Page` doesn't have a declared size – it gets a default size when it runs. Everything on the `Page` *automatically sized* itself. The `StackPanel`, for example, gets its height and width from the `Page` because its default behavior is to stretch to the size of its container.

You can create XAML interfaces in which all sizes and positions are explicitly declared, just like you can in older technologies such as Windows Forms. **For most applications, that is not recommended.**

That's a polite way of saying “Don't do that – ever.” I've never seen an effective XAML interface done that way.³ You'll take maximum advantage of XAML by using its built-in features for sizing and positioning visual elements. Used properly, these features can allow your application to run at varying resolutions and sizes.

Once upon a time, that was a convenience. Now it's a necessity. With the exploding variations of devices with different form factors, aspect ratios, and pixel densities, an interface on the Uno Platform needs to be able to size itself at runtime. If you create an interface with lots of hard-coded sizes, it's almost certain that it won't respond properly on various devices on which it might run.

You will need a few hard-coded sizes for most of your interfaces, but using a hard-coded size should be a last resort, not a first one. When it is necessary to hard code the size of a visual element, use the

³ There might be some odd edge cases that use all hardcoded sizes effectively, but I've never done or seen such an application.

Height and Width properties. You can set either one, or both:

```
<StackPanel Background="LightBlue" Height="300" Width="200">
  <Button Height="50" Margin="5">
    I'm a button
  </Button>
  <TextBox Width="150" Text="Text for editing" />
</StackPanel>
```

Again, for emphasis, don't routinely set Height and Width for your elements. Most of your visual elements should be automatically sized.

While you are first learning how to write XAML, you will often get results you will not expect. Don't get frustrated. It just means you are not fluent with the rules of layout yet. With more experience, configuring elements to get the size and layout you want will become easy, and most experienced XAML developers are able to create sophisticated layouts in XAML much faster than in other technologies.⁴

Scalable Units for Measurement

When an explicit size or position is declared for a XAML element, the units look as if they might be screen pixels, and are even called "pixels" in some dialogs in Visual Studio. However, these settings are not in true screen pixels, as they are in Windows Forms. Instead, they can be considered "virtual pixels". They have two key differences from physical pixels:

1. The pixels are scalable. For example, Windows 10 automatically scales interfaces depending on the size and pixel density of the current device. If a display uses a high pixel density, such as those implementing the 4K specification, one "pixel" unit is almost certainly more than a physical pixel.⁵
2. Unlike measurements done in physical pixels, XAML's units can be fractional. In fact, the properties that affect size and position in XAML are all double precision numbers.⁶ While specifying a precise decimal number might not make a visible difference at a normal size, a user interface element that is dramatically magnified might need the precision.

Visual Elements available in XAML

Just like other user interface technologies, XAML has visual elements that go on an interface. The

⁴ You're going to have to take that assertion on faith for a while. Most XAML newbies are pretty slow at first. But hang in there. You'll be amazed how fast you create compelling modern interfaces in XAML if you stick with it and develop some deep expertise.

⁵ You can consult the Windows 10 documentation if you want to know more about how and when it does automatic scaling. The resources section at the end of the book contains a link you can use.

⁶ That is, the data type of those properties is System.Double.

XAML visual element set is quite rich, but in this chapter, we'll only look at a few of the elements you will use most often. Some of these will feel pretty familiar to you, such as buttons and textboxes. In future chapters, I'll tell you more about the element set and how it differs from what you've used in the past.

It may sound strange to you that I keep using the word “element” to describe a visual thing that goes on an interface. In most user interface technologies, the term “control” is used for things placed on a visual design surface. In XAML, the word “control” has a more restrictive meaning. A control is an element that descends from the Control class, and is usually designed for user interaction.⁷ A Button is a control, and so is a TextBox.

However, the StackPanel we saw earlier is not a control. It's not intended for user interaction. It's intended as a container to lay out the child elements placed within it. As I mentioned earlier, containers that are capable of holding multiple child controls and arranging them relative to one another are called *panels* in XAML, and they descend from the Panel class. StackPanel is an example.

In XAML, the generic term for something visual that goes on a user interface is “element”. A control is one type of element. A panel is another. Yet another is a shape, such as a rectangle or ellipse. Let's take a look at those two shapes.

Ellipse and Rectangle

User interfaces in XAML typically use graphical elements more than interfaces in older technologies. That's partly because they are easy to create and place where you need them, and they have all the automatic sizing capabilities of most elements.

Two of the most commonly used graphical elements are Ellipse and Rectangle. They both descend from the Shape element, and they are quite similar. Here is the XAML for an Ellipse:

```
<Ellipse Fill="LightCoral" Height="50" Width="100" />
```

The Fill property sets the interior of the Ellipse. In the line above, it looks like the Fill is set to a color named LightCoral. However, the Fill property actually takes a *brush*, which is a more flexible concept than just a color. One simple type of brush is a SolidColorBrush. The XAML above sets the Fill property to a SolidColorBrush with color LightCoral. There is a long list of named colors that can be used in XAML. As we'll see in chapter x, you can also set colors of brushes with RGB values.

The Ellipse has a Stroke property for the brush used to draw an outline around the Ellipse. The thickness of the outline is set with a property named StrokeThickness. Both of these properties must

⁷ Controls also have an important capability not included with other elements. The rendering of a control can be radically changed using a feature called control templates. They will be discussed in Chapter 6.

be set in XAML to see an outline.⁸ Here is XAML for an Ellipse with an outline:

```
<Ellipse Fill="LightCoral"  
        Stroke="DarkOrchid" StrokeThickness="3"  
        Height="50" Width="100" />
```

The above Ellipse would look like the figure below, assuming it was on a white background:



The Rectangle element has the same properties discussed above for Ellipse. In fact, if you just change the Ellipse tag to Rectangle in the XAML above, the result will then look like this:



The examples above use hard-coded sizing of the Ellipse and Rectangle. I've been nagging you not to do that. But it was needed to show the Ellipse and Rectangle in isolation. Those shapes are normally in a panel that does some or all of their sizing, and we'll see that in action after covering the Grid panel below.

SymbolIcon

In chapter 5, we'll cover some additional options for graphical capabilities beyond Ellipse and Rectangle. To start you down the path of thinking graphically in XAML, I'd like to show you one easy to use element right now. It's called SymbolIcon, and it lets you get some basic shapes into your interface by merely selecting the shape you want from a drop-down for the Symbol property. Here's a basic example:

```
<SymbolIcon Symbol="TwoBars" />
```

This SymbolIcon will display as a typical mobile-phone signal indicator with two bars, shown in Figure xx.

⁸ That is, the default of the Stroke property is a transparent brush, and the default StrokeThickness is zero. If either of these is left at the default, no outline is drawn.



Figure xx – A SymbolIcon with the Symbol property set to “TwoBars”.

There are several dozen symbols to choose from. The usual auto-complete drop-down will show in XAML when you type “Symbol=” as a property setting, as shown on the left in Figure xx. However, it only shows the text property settings for the various symbols. To see the symbols along with the settings in a drop-down, use the Properties Window to set the Symbol property, as shown on the right in Figure xx.

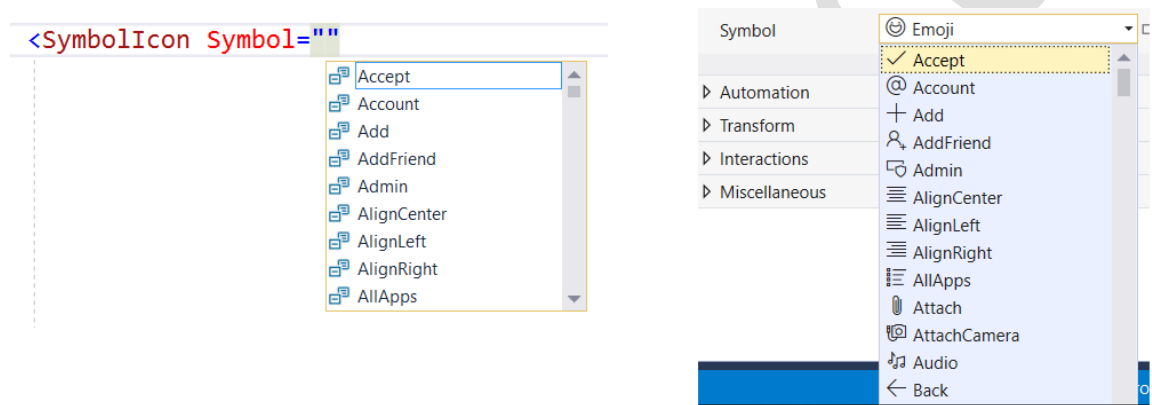


Figure xx – Setting the Symbol property for SymbolIcon. On the left is the dropdown that appears when directly editing XAML. On the right is the dropdown in the Property Window.

As we will see below, SymbolIcon combined with Button allows you to quickly create a Button with a symbol instead of text inside.

TextBox control

Superficially, a TextBox in XAML works much as you would expect. It has a Text property to get or set text. It allows the user to edit the text, including support for cut, copy, and paste operations.

A XAML TextBox can look radically different from its equivalents in other UI technologies, but we won't worry about that for several more chapters. For now, we'll just use the TextBox control with its default appearance.

Earlier, we saw one significant difference in the XAML TextBox. By default, the size of a TextBox is

influenced by the panel that contains it.⁹ In our first example, the width of the TextBox was determined by the StackPanel that contained it.

We'll talk a lot more about XAML layout in chapter 2. That includes discussion of how elements such as TextBox are sized and positioned by various panels.

TextBox has a Foreground property to set the brush used to paint the text in the control. All controls have a Foreground property for any text they display. Like the Fill and Stroke properties of the shapes above, the easiest way to set the Foreground brush is to use a named color.¹⁰

Button Control and the Concept of Content

As with TextBox, the Button control in XAML looks much like it does in other user interface stacks. You can set the text in the button, and handle an event when the user clicks the button.

There is one major difference from earlier UI technologies. You can put a lot more than just text into a Button.

In other UI technologies, the Text property is typically used to set a string you want to show in a Button. However, in XAML, you are not limited to just showing text. You can show anything in a button that XAML can display.

Instead of a Text property, XAML controls such as Button have a property named Content. It can be any object, either visual or non-visual. For example, you can set the content to a shape.

Like other properties, the Content property can be set with an attribute, like this:

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

The Content property can also be set by placing something between the start tag and end tag for a Button.¹¹ We did that in the earlier example of a XAML page:

```
<Button Margin="5">
    I'm a button
</Button>
```

Controls that take content are called, naturally enough, *content controls*, and they all descend from a base type of ContentControl.

To set the Content property to something other than a string, just replace the text between the tags.

⁹ There's one panel for which that's not true – the Canvas. We'll discuss the Canvas in chapter 3.

¹⁰ In most production applications, the brushes used for text in the application are set via the styling system. We'll get to styles in Chapter 6.

¹¹ You can define Content between the start and end tags of the Button because the default property of Button is Content. We'll talk more about default properties in the next chapter.

Here's an example, setting the content to a `SymbolIcon` with a symbol showing a globe:

```
<Button>  
    <SymbolIcon Symbol="World" />  
</Button>
```

Such a `Button` would look much like Figure xx



Figure xx – A `Button` with its `Content` set to a `SymbolIcon` showing a globe.

Just like `TextBox` controls, the sizes of `Button` controls are partially determined by their container, and the size of a `Button` may need to vary based on its content. It is not a good idea to hard code the sizes of your buttons for consistency, the way you might have done in a previous technology such as Windows Forms.¹²

A Page is a `ContentControl`

Earlier in the chapter, I mentioned that you cannot put multiple controls directly in a `Page`. That's because `Page` also descends from `ContentControl`. It can only take one item of content.

Normally, that one item is a panel, or other container that itself contains a panel. Then, the panel inside the page can contain multiple controls.

In Chapter 2, we'll cover exactly how content is rendered in a content control. That is, I'll describe how XAML decides what to do with a control's content

Displaying text with `TextBlock`

So far, we have seen examples of visual elements that interact with the user (controls), those that arrange child elements (panels), and a couple of graphical shapes. Another specialized element is the `TextBlock` element. It allows you to place text in your user interface, and to format the text with various typefaces, sizes, colors, and weights.

The `TextBlock` element replaces the `Label` control in some earlier technologies. To emphasize, it's not a control, even though it lives in the same namespace as other XAML controls.¹³

¹² The way to get consistency with your elements in XAML is with styles. We won't get to styles for several chapters, so in the meantime, the examples will have some duplicated settings in the elements, such as the `Margin` settings we'll be discussing in a few paragraphs.

¹³ Just for reference, the namespace holding most XAML controls is `Windows.UI.Xaml.Controls`.

TextBlock has properties for controlling the typeface used, including `FontFamily`, `FontSize`, `FontStyle` (italic, for example), and `FontWeight` (such as `Bold`). Here is XAML for a typical TextBlock:

```
<TextBlock Text="I'm learning XAML."  
           FontSize="20"  
           FontStyle="Italic"  
           Foreground="CadetBlue"  
           />
```

That TextBlock would look like this on a white background:

I'm learning XAML.

TextBlock has other text formatting capabilities, and I'll be covering those in chapter 3.

Margins on visual elements

By default, panels such as `StackPanel` will often size child elements to go right to the edge of the panel. And, if elements are adjacent, there may be no space between the elements by default. This can make for some crowded and ugly interfaces.

To keep the automatic sizing, but allow more flexibility, all visual elements in XAML have a `Margin` property. `Margin` is a structure with four members, one each for the top, left, bottom, and right sides of the element.

If an element's `Margin` property is set, then the panel takes the `Margin` into account when sizing and positioning the element. Setting all four members of `Margin` to 4 units, for example, would ensure that the edge of the element stays at least four units away from its parent panel or any adjacent elements.

Our earlier simple XAML example had a `Margin` property setting on the `Button`. As you can see in Figure 2-1, the resulting screen has some room included around the `Button`, separating it from its neighbors.

StackPanel and Grid

XAML has several panels, and they fill various needs for laying out a user interface. You've already seen the simplest, which is the `StackPanel`.

A panel can contain any number of visual elements. The contained elements are referred to as the *children* of the panel. All panels have a `Children` property that contains a collection of the panel's children.

All the children placed in a `StackPanel` are arranged in a stack, hence the name. By default, the stack is vertical.

The `Orientation` property sets the orientation of the stack. Setting `Orientation` to a value of

Horizontal causes the stack to become horizontal.

The Grid panel is the most flexible panel for most XAML applications, so you will be using it a lot. Grid is so flexible that the Visual Studio XAML designers use it as the default layout container for a new Page or UserControl.

The Grid container arranges its children by placing them in a cell, or in a rectangular array of cells. The available cells are determined by the defined rows and columns of the Grid.

The rows and columns can be different sizes. The height of a row or the width of a column can be set in three principle ways:

- Set to a hard-coded number of virtual pixels
- Proportionally to other rows or columns
- Automatically sized to contain the largest element placed with the row or column

The rows and columns for a Grid are defined with properties named RowDefinitions and ColumnDefinitions. Here is the XAML to define a Grid with three equally sized columns and two equally sized rows:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <!--child elements of the Grid go here-->
</Grid>
```

The Grid divides its internal space into cells based on its defined rows and columns. The Grid defined above has six cells. Child elements can be placed in a single cell, or in a rectangular set of adjacent cells.

Child elements are placed in the Grid below the definitions for rows and columns. This is different from HTML, and is a common point of confusion for XAML newbies. The XAML above shows the place where child elements for the Grid would be placed.

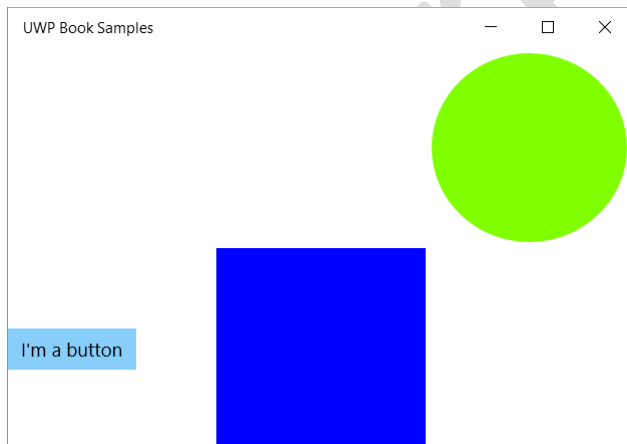
Child elements specify their location with a XAML capability called *attached properties*. Those properties are set with the attributes Grid.Row and Grid.Column.

By default, Grid.Row and Grid.Column are both zero. An element with those default settings would render in the top left cell because the first column and first row have an index of zero.¹⁴ But if Grid.Column for an element were set to 2, for example, the element would render in the third column.

Let's expand the XAML above to assign some child elements their own cells:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <!--child elements of the Grid go here-->
  <Rectangle Fill="Blue" Grid.Row="1" Grid.Column="1" />
  <Ellipse Fill="Chartreuse" Grid.Column="2" Margin="5" />
  <Button Background="LightSkyBlue" Grid.Row="1">
    I'm a button
  </Button>
</Grid>
```

When run, a Page containing this XAML would look something like this:



However, that's just the way it looks for one size of the Window. The Ellipse and Rectangle don't have a hard-coded size, so they automatically size to the cell in which they are placed. If the window containing the Page with this XAML is resized, the Grid will also resize. That will cause the cells in the

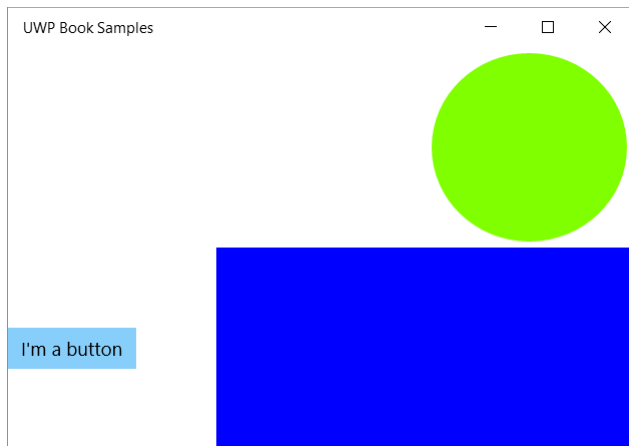
¹⁴ The collections for rows and columns are zero-based collections, meeting the standard convention for collections in C# and VB.

Grid to resize, which will resize the shapes in those cells.

The blue Rectangle has no setting for Margin, so it sizes all the way to the edge of its cell. However, the chartreuse Ellipse has a Margin setting of 5, so it is smaller than its cell.

Not all elements stretch to fill their cells. The Button, for example, by default is centered vertically, and left-aligned horizontally.¹⁵ Later in this chapter, we'll see the most common settings you can use to control alignment of elements.

To allow elements to span cells, the attached properties `Grid.RowSpan` and `Grid.ColumnSpan` are used. If we set `Grid.ColumnSpan` to 2 for the blue Rectangle, it will then span both of the lower right cells, and the window will look like this:



HorizontalAlignment and VerticalAlignment

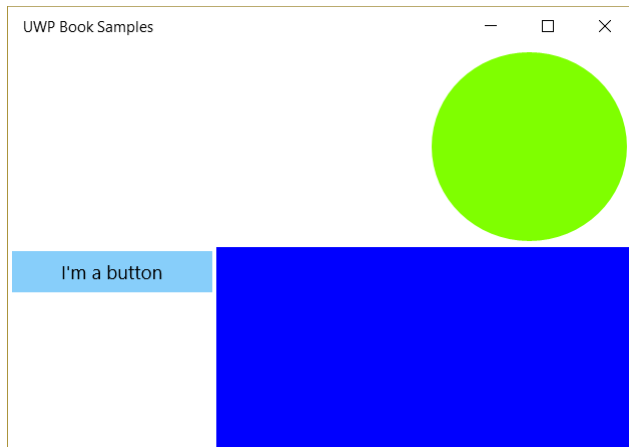
Each element has a default for its alignment when placed in a panel. It's very common for you to override that default alignment to get the arrangement you want.

For example, suppose you wanted the Button in the example above to be at the top of its cell, and stretched to the full width of the cell. Here is how the XAML would change for such a Button:

```
<Button Background="LightSkyBlue" Margin="3"
        VerticalAlignment="Top"
        HorizontalAlignment="Stretch" >
    I'm a button
</Button>
```

The window with this changed Button will look like this:

¹⁵ The default behavior of a Button in a Grid cell is different in Windows 10 / UWP XAML than in earlier XAML technologies. In WPF and Silverlight, the Button expands to fill a Grid cell, just as the Rectangle does above.



The properties that influence the alignment are `HorizontalAlignment` and `VerticalAlignment`. Here are the possible values for each.

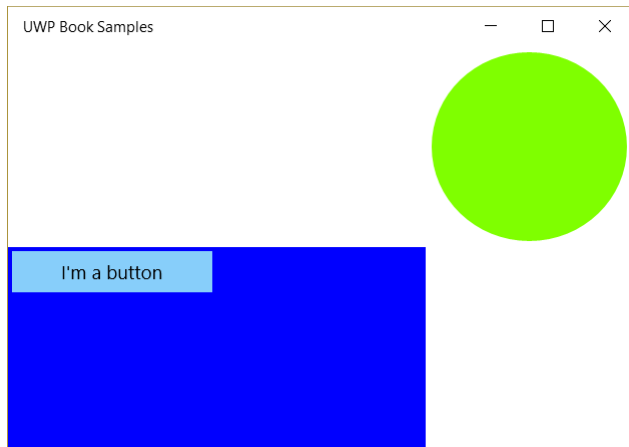
HorizontalAlignment	VerticalAlignment
Left	Top
Center	Center
Right	Bottom
Stretch	Stretch

As the example above with the Button demonstrated, an element can be stretched to fit its container in one dimension, while being aligned in the other dimension without stretching. The Button above was stretched horizontally, but positioned at the top of cell vertically. Of course, both alignment properties could be set to `Stretch`, which would cause the element to stretch to fill its container. Some elements, such as shapes, have their alignment properties set to `Stretch` by default.

Layering elements in cells

So far, no Grid cell has contained more than one child element. However, you can place as many elements in a Grid cell as you like. They will be layered on top of one another, with the first ones in the XAML rendered at the bottom of the stack, and the later ones on the top.

If the `Grid.Column` setting for the Rectangle is removed, for example, the window will change to look like this:



The Button is on top of the Rectangle because it is below the Rectangle in the Grid's Children collection.

The technique of putting a rectangle in an array of Grid cells is very useful. There is no property to set the background of a Grid cell. However, placing a Rectangle in the cell with the desired background appearance will take care of that need.

Nested Containers

Suppose I have a Grid like the example above, but instead of a single Button in the lower left cell, I would like several buttons in a vertical stack. There are some brute force ways I could do that with Margin settings and alignment properties, but the easy way is to place a StackPanel in that Grid cell, and then place the Button controls in the StackPanel. Here is the XAML for that technique:

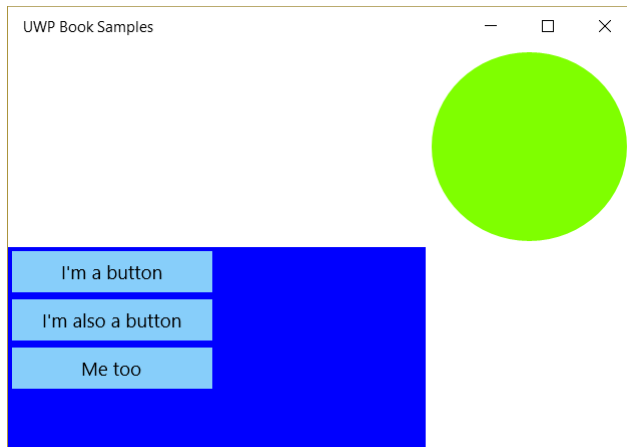
```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <!--child elements of the Grid go here-->
  <Rectangle Fill="Blue" Grid.Row="1"
    Grid.ColumnSpan="2"/>
  <Ellipse Fill="Chartreuse" Grid.Column="2" Margin="5" />
  <StackPanel Grid.Row="1">
    <Button Background="LightSkyBlue" Margin="3"
```

```

        HorizontalAlignment="Stretch" >
    I'm a button
</Button>
<Button Background="LightSkyBlue" Margin="3"
        HorizontalAlignment="Stretch" >
    I'm also a button
</Button>
<Button Background="LightSkyBlue" Margin="3"
        HorizontalAlignment="Stretch" >
    Me too
</Button>
</StackPanel>
</Grid>

```

The resulting window would look like this:



This technique is called nesting of containers or nesting of panels. It's very commonly used to get the right layout for XAML interfaces, and the nesting can go several levels deep.

We will discuss some more examples in Chapter 2.

Automatically sized rows and columns

So far, all our grid examples have used rows and columns that were the same size. In the next chapter, I'll go through all the possibilities for sizing rows and columns. However, one capability is so useful and important that I want to tell you about it now.

If you set a row or column to be automatically sized, it will then use its largest element to determine its size. Automatic sizing of a column is done by setting the Width property of its ColumnDefinition to "Auto". It will then become as wide as it needs to be to hold the widest element placed in it. Similarly, a row with Height set to "Auto" has its height automatically sized to accommodate the element in the row with the largest height.

Suppose we have a Grid defined with the following XAML, with the middle row and the first column automatically sized:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <TextBlock Grid.Row="1" Text="I set the size of my cell" />
  <Rectangle Fill="BlueViolet" />
  <Rectangle Grid.Column="1" Fill="Chartreuse" />
  <Rectangle Grid.Column="1" Grid.Row="1" Fill="LightCoral" />
  <Rectangle Grid.Row="2" Fill="CadetBlue" />
</Grid>
```

The TextBlock is in the middle row, so the height of that row is taken from the height of the TextBlock. By default, the TextBlock sizes itself with the height needed to hold the font size.

The TextBlock also sizes the width of the first column. The TextBlock becomes wide enough to hold the text placed in it, and that width then becomes the width of its column, since it is the widest element in the column.

I have included several Rectangle elements so that you can see the remaining cells. As we saw above, Shape elements with no hard-coded size fill their cell because they stretch by default. However, their default, non-stretched size is zero, so they do not affect any automatically sized rows or columns.

The visual result of this XAML will look like this:

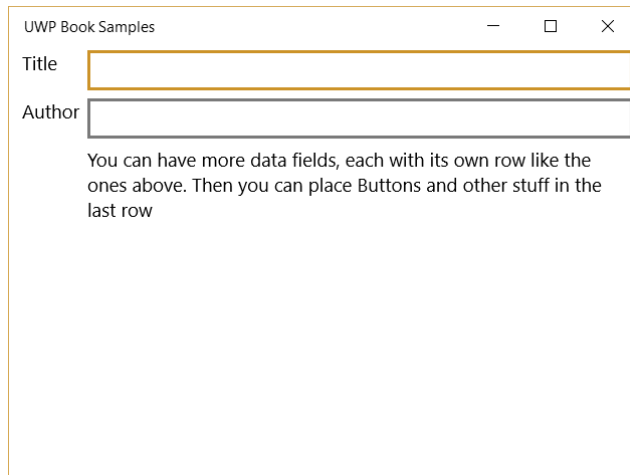


The top and bottom rows are the same size. They divide the remaining height after the middle row has sized itself to the `TextBlock`. If you place this XAML in a `Page` and run it, the top and bottom rows will resize as the `Page` resizes, but the middle row remains the same height no matter how much you resize the `Page`.

Data entry forms commonly use automatically sized rows to hold individual data fields, along with their text descriptions. Here is a simple example in XAML:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <TextBlock Text="Title" Margin="10,3,3,3" />
  <TextBox Grid.Column="1" Margin="3" />
  <TextBlock Text="Author" Margin="10,3,3,3" Grid.Row="1" />
  <TextBox Grid.Column="1" Margin="3" Grid.Row="1" />
  <TextBlock Grid.Column="2" Grid.Row="2"
    TextWrapping="Wrap" Margin="3"
    Text="You can have more data fields" />
</Grid>
```

The visual results of the XAML above would look like this, with some additional text placed in that last `TextBlock`:



The first column is sized to the widest text label placed in it, which is “Author”. If you added a data field with a text label that was wider, then the column would automatically get wider to hold it.

This automatic resizing will feel strange to you if your experience is mostly in desktop technologies such as Windows Forms or classic Visual Basic. Your mental model of screen layout expects precisely defined size and position that does not change.

While you can do that in XAML, let me emphasize one more time that you shouldn’t. Instead, you should become fluent with automatic sizing and positioning, and use it consistently.

It’s quite liberating when you make that switch. You no longer need to worry so much about precise layout. You delegate that responsibility to the layout engine, which can then change layout to suit particular circumstances such as monitors of differing size.¹⁶

Code-behind

XAML has limits on what it can express. All but the most trivial XAML applications also need to contain code. And sometimes, that code needs to interact with XAML elements defined in XAML.

A well written XAML application won’t have a lot of UI-related code.¹⁷ As we’ll see throughout the book, when you learn to think in XAML, you can do much more without code than you could in earlier technologies. However, code in a XAML application isn’t verboten, and there are times when it’s an essential part of an application. For example, if you need to add new visual elements to your interface

¹⁶ If you’ve been developing desktop software long enough, you will be familiar with the old arguments about what resolutions to support. Should it be 1280x1024? Or 1024x768? Then when wide screens came along, it got even messier. If you use dynamic layout, you can put those obsolete arguments behind you.

¹⁷ Naturally, the rest of the application might have scads of code. But if your XAML UI ends up needing a ton of code, you’re probably doing things the hard way.

on the fly, that will almost certainly be done in code.

Code that runs in concert with a XAML-defined Page or UserControl is called *code behind*. The location of the code behind is part of the XAML definition of the Page or UserControl. For example, our first XAML Page back at the beginning of the chapter started with these two lines:

```
<Page
  x:Class="Chapter1.MainPage"
```

The attribute names "x:Class" gives the namespace path of the class that holds the code behind for the Page. This line is filled in by the designer, and Visual Studio automatically creates a starter class for the code behind.

Naming a XAML Element

The code in a code-behind class must have some way to identify a XAML element before it can work with that element. That identification is commonly done with a Name property, just as it is in other designers.

Naming a XAML element in XAML is not required unless you want to reference the element from elsewhere. In our first example of XAML above, none of the elements had a name.

You can name most elements in XAML using a Name attribute, which sets the Name property on the element. All XAML controls and panels have a Name property. Here is a Button defined in XAML with a name:

```
<Button Name="TheButton">
  I'm a button
</Button>
```

You can also name a XAML element by setting the Name property in the Properties window of Visual Studio. If you declare a name for a XAML element in the designer, the Name attribute will appear in the XAML, just as other property settings do.¹⁸

Declaring Event Handlers in XAML

One common use for code behind is arranging for some code to be run when the user interacts with a visual element. As with other technologies such as Windows Forms, XAML elements generate *events* that can cause code to run. The code procedure for the event is usually called an *event handler*.

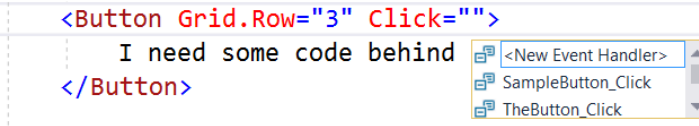
The easiest way to hook an element event to an event handler in code is to double-click the element

¹⁸ All XAML elements that descend from a base class called FrameworkElement have a Name property. However, you can declare objects in XAML that do not descend from FrameworkElement. Those classes require a different attribute to allow code to access a XAML-defined instance. That attribute is x:Name, and I'll talk about it in the next chapter.

in the visual designer. Doing this for a Button, for example, will generate an event handler for the Button's Click event, and hook the event to the handler. It's likely that you have used other visual designers that work the same way. However, the default syntax for the event attachment is different in XAML. A simple event attachment looks like this:

```
<Button Grid.Row="3" Click="Button_Click">
    I need some code behind
</Button>
```

If you begin typing "Click=" in the Button's declaration, you will see a drop down with available event handlers, and an option at the top to create a new one. Figure xx shows this drop-down.



The screenshot shows a code editor with the following XAML code: `<Button Grid.Row="3" Click="">` followed by the text "I need some code behind" and `</Button>`. A dropdown menu is open next to the empty Click attribute, listing three options: "<New Event Handler>", "SampleButton_Click", and "TheButton_Click".

```
</Grid>
```

Figure xx – Typing an event name in a XAML element declaration generates a drop-down to specify an event for attachment.

If the element doesn't have a name, and you select "<New Event Handler>", then you'll get a generic name for the event handler. A better practice is to give the element a setting for its Name property before trying to attach the event to a handler. If the Button above has the name TheButton, a new event handler for the Click event will be named TheButton_Click¹⁹, and the XAML line you get by selecting "<New Event Handler>" will look like this:

```
<Button Name="TheButton" Click="TheButton_Click">
    I need some code behind
</Button>
```

This syntax in XAML works for both C# and Visual Basic code behind modules. In either case, a blank event handler will be present when you view the code behind.²⁰

Event Handlers in Code

Using XAML is not the only way to connect the Button to an event handler. Any of the normal ways of wiring up an event in code will work. In C#, the usual "+=" event wireup is available. In Visual Basic, you can use a Handles clause, or an AddHandler statement.

¹⁹ If you don't fill in the Name property first, the event handler will get a generic, unhelpful name. It's a good idea to fill in the Name property before trying to attach to any events.

²⁰ You can get to code behind by selecting it in the Solution Explorer, or by right-clicking in the XAML and selecting "View Code", or by pressing the hot key to go to the code (which is F7 by default).

Basics of Data Binding in XAML

Most applications work with data. The data usually comes from a database, but in the user interface layer of the application, the data is normally held in data classes. For example, a Customer class might hold data for a customer in properties such as `CompanyName` and `Phone`.

Individual controls in the user interface work with the data in these properties, allowing the user to read, enter, or change the data. In XAML, this is almost always done with *data binding*.

You are probably familiar with the idea of data binding. Many technologies include it, and the main purpose is the same for all of them. Data binding eliminates a lot of logic to move information back and forth from controls to data containers.

Data binding in XAML can certainly be used for that purpose. In fact, I consider data binding in XAML superior to data binding in any other user interface technology that I've used.

Data binding is also used for many other purposes in XAML. Creation of templates, for example depends heavily on binding. For now, we'll just consider some simple data bindings. We'll see a lot more about data binding in chapter 4.

Creating a Binding in XAML

A *binding* is used to set up data binding for one property on one element. You can think of a binding as a connector between two points. One point is always a property on an element. The other point is a property on another object, often a data object.²¹

To illustrate, suppose we have a data object which is of type `Book`. The `Book` class has a property for `Genre` and a property for `Title`.

Here is a XAML definition of a `TextBox` that includes a binding from the `TextBox`'s `Text` property to the `Title` property of such a `Book` object:

```
<TextBox Text="{Binding Title}" Name="TitleTextBox" />
```

Figure xx shows how that line of XAML creates a `Binding` object that associates the value of the `Title` property of the `Book` object with the `Text` property of the `TextBox` element.

²¹ Chapter 4 on data binding will have some diagrams to explain this further.

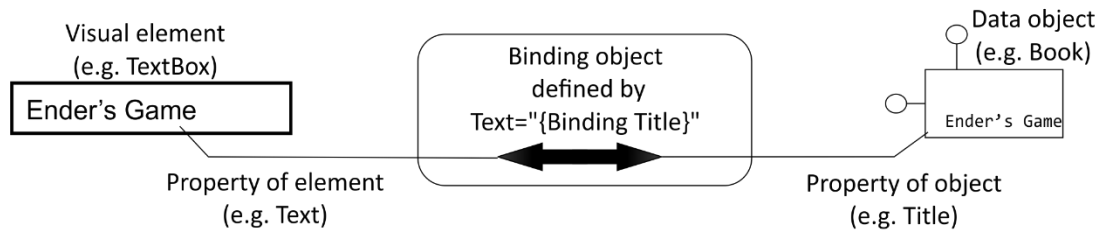


Figure xx – A binding associates a property of a data object with a property of a visual element, and handles moving the value of the property from the object to the element and back as necessary.

A data binding is defined in a special XAML construct called a *markup extension*. The markup extension for the data binding in the example above is “`{Binding Title}`”.

This is the first time we have seen such a markup extension in XAML, but we’ll see several more as we get deeper into XAML. Markup extensions are always bounded by curly braces.

A simple binding can declare just the name of a property of the data object. We will see more complex bindings in future chapters.

A simple binding does not need to say where the data is coming from. When the source of the data is not otherwise declared, it is assumed that the data will come from an object that will be found using a property known as `DataContext`.

The `DataContext` property

All elements have a `DataContext` property. It is used to point to a data object to be used in data bindings by default.

`DataContext` can be declared on the same element that declares the data binding, but you don’t usually use it that way. Instead, `DataContext` is usually declared on some container, such as a `Grid` or `Page`. `DataContext` is an *inherited property* which means once it is declared on a container, all the elements inside the container can use it. That includes not just direct children, but children of children, and so on down the tree of elements.

For real applications, you will typically set the `DataContext` of a container in code, after you fetched some data from your database. When you set the property, the data bindings that use the `DataContext` kick in automatically.

For many of the examples in this book, I will be using a different way to set `DataContext`. I will just declare a data object right in the XAML. While this isn’t practical for production applications, it can be useful for examples. It is also a way to get a data object that will make the visual preview of your `Page` or `UserControl` more like it will be in the running application.

Suppose I have a `Book` object with properties for `Title` and `Genre`. Then I could set the `DataContext` of

a Grid to a Book object directly in XAML, like this:

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

Data Bindings Have Silent Failure

What if the object in the DataContext doesn't have the property specified in the binding path? I think XAML handles that situation better than any other technology I've seen.

If a binding specifies a property that is not present on the DataContext, you will *not* get a runtime error. It will be as if the binding did not even exist. No action will be taken on the property that is part of the binding.

If you are running your project in Visual Studio, you will get an output message when a binding is unable to find the correct property on the source data object. There is also capability to halt and debug on a binding error for extreme cases, as we'll see in Chapter 3. But for routine use, allowing bindings to fail with hard errors means binding can be ubiquitous in your application without making it brittle.

A Quick Summary

Before we go on to talk about using Visual Studio for XAML applications, let's do a quick summary of the most important points covered in this chapter.

1. XAML is a variant of XML that is used to craft user interfaces.
2. Visual elements are declared in the XAML using attributes to set properties.
3. Some elements are containers called panels that arrange other elements. Examples include StackPanel and Grid.
4. StackPanel just stacks its child elements in a vertical stack.
5. Grid places its child elements in cells of the Grid, and multiple elements can be placed in a cell. Grid.Row, Grid.Column, Grid.RowSpan, and Grid.ColumnSpan are attached properties for positioning elements in cells.
6. Rows and columns of Grid panels can size themselves to accommodate the elements placed inside them.
7. Most elements should not have hard-coded sizes. That allows panels to adjust the sizes of those elements, based on variables such as window size and aspect ratio.
8. Ellipse and Rectangle are useful shapes, and the Fill, Stroke, and StrokeThickness properties control their appearance.

9. SymbolIcon is a great way to quickly get common graphical icons into your app.
10. Margin and various alignment properties are used to refine the layout in panels. Margin provides space around elements, while HorizontalAlignment and VerticalAlignment position elements within their container's area.
11. TextBox in XAML looks a lot like TextBox in other technologies, except for automatic sizing.
12. Button is also pretty familiar too, but it can hold more than text. Any shape or other XAML can be placed in the Content property of the Button.
13. TextBlock is the go-to element for read-only display of text.
14. Panels such as StackPanel and Grid can be nested to gain more control over layout.
15. Data binding ties together information in data classes with properties of elements.
16. Data bindings are defined with binding expressions, which are a special case of XAML markup extensions.
17. By default, data bindings use a property called DataContext to locate the data object for the binding.

If any of these points seem hazy, now is a good time to read over that section of the chapter again.

Using Visual Studio to Create XAML Applications

The XAML visual designer in Visual Studio is a drag and drop designer superficially similar to others such as the Windows Forms designer or ASP.NET designer. When a XAML visual design surface is active, the toolbox on the left contains a list of XAML elements that can be placed on the surface.

However, there are some significant differences from earlier designers. Some of them are clear just by looking at the designer. Figure xx shows a sample screen with the XAML visual designer active, and includes some numbered indicators of areas of the screen that show new and different features. Refer to that figure as I go through the following numbered list:

1. The visual design surface is at the top of a split-screen area. It shows a visual rendering of the XAML that is below it (see #2). This is often called the design time view.
2. The XAML for the current XAML object is in an editor just below the visual design surface. If you manually change the XAML in this editor, the changes will be reflected in the design time view above (assuming the changed XAML is valid).
3. You can change the area allocated to each of the split screens by dragging the small double bar between the screens.
4. You can click the small icon with the up and down arrows to switch the positions of the visual surface and the XAML. That is, clicking that icon with the screen as shown would cause the

XAML to go to the top of the split screen and the visual surface to go to the bottom.

5. A drop-down zoom selector allows you to zoom the visual surface in or out. That's helpful in seeing the whole surface at once, or seeing details in some small portion of the surface. Since XAML is vector-based instead of bit-mapped, zooming is a lot more useful than it would be for more traditional user interface designers.
6. The Solution Explorer contains project items relevant to a XAML application. Notice the presence of **App.xaml**, which we will discuss in the next chapter. A new project will also contain a **MainPage.xaml** for the default XAML Page for the project. Both of these XAML items also have code-behind modules available.
7. The Properties Window is different from earlier technologies such as Windows Forms. Property groups are expandable, and there's a search box to find a property. These changes were needed because XAML has a lot more properties on visual elements than other US technologies.
8. A dropdown allows you to choose your expected device form factor. This lets the visual designer show preview layout as if you were working in that size and orientation. You can change this at any time to see what the layout will look like with a different form factor.

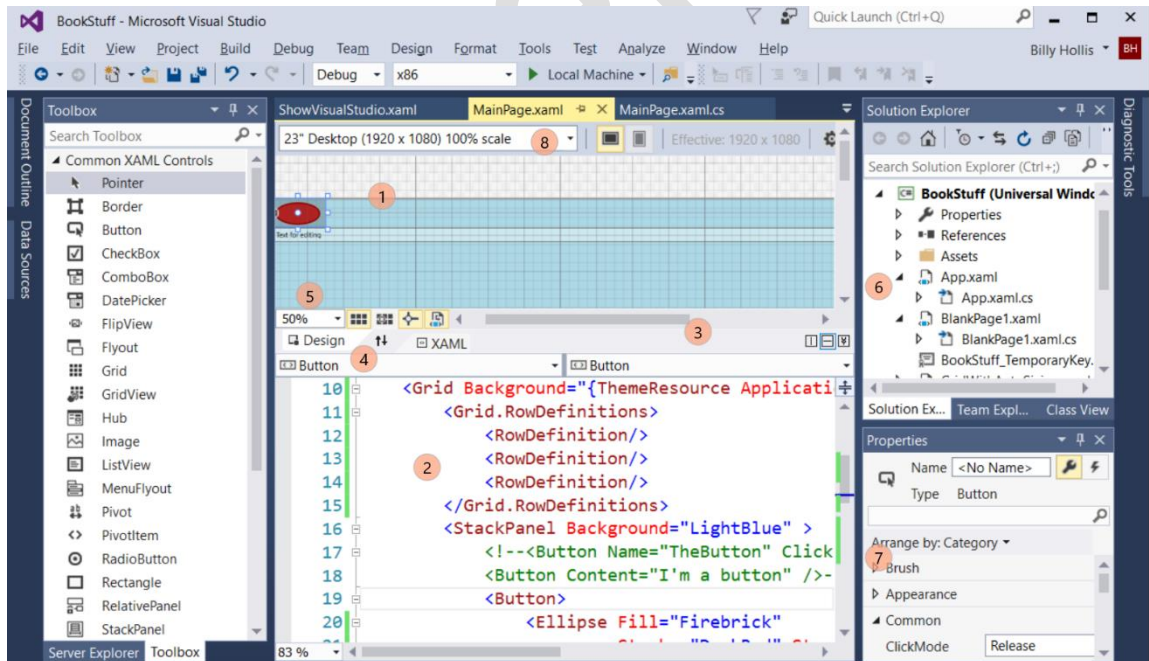


Figure 2-2 – The XAML visual designer in Visual Studio 2015. Some key differences are indicated with the numbered circles.

A new Page in the designer will always have a Grid as the default content of the Page, and all other elements in the Page will by default go inside this Grid.

The drag and drop designer isn't very helpful

The toolbox on the left contains visual elements that can be used to make up a Page. As with earlier technology platforms such as Windows Forms, you can drag those elements onto the visual design surface.

I really wish it were that easy to work with visual elements on a Page, but it's not. The dynamic nature of XAML is not conducive to drag-and-drop positioning. For example, what size should the resulting element be? What XAML capability should be used to position the element? How do you drag something into an automatically sized cell that is empty and thus has no area right now?

The conflict between the drag-and-drop model and the way XAML works becomes obvious when you drag over an element. There will be seemingly random Margin and alignment settings. You will usually end up removing much of what the designer puts in, and it's almost always faster just to write what you want in the XAML editor than to drag/drop and then fix the results.

I'm not going to discuss the drag/drop visual designer any further in this book, except to note one thing it does well with namespaces when we dive further into that topic in chapter 3. Really, don't try to use it, even for your early efforts. You'll just get frustrated, and it will delay your progress in learning to think in XAML.²²

Wrap up

At this point, you should have a rough idea of some of the most important XAML concepts. Some of those concepts include the idea of panels that contain and arrange child elements, and the concept of content in a ContentControl such as Button.

The next chapter will use those ideas as a springboard to discuss one of the biggest stumbling blocks for new XAML developers – the composition model, or in simple terms, how the pieces can be combined and fitted together.

The chapter will include more about layout panels, including additional panels named RelativePanel and Canvas. It will also look at content controls in some more detail.

Even if you have some experience with XAML, don't skip the next chapter. The composition model is critical to your work in XAML, and it is a totally different mental model for creating user interfaces than

²² You can drag elements from the toolbox directly onto the XAML editor. That will just place the tag for the element there. That works fine, but I find it's easier just to type the element I want. Autocomplete means I don't have to type it all.

other technologies. To learn to think in XAML, the composition model should become second nature to you.

Now, on to chapter 2.

Pre-release