Chapter Two

Putting Pieces Together – Composition and Layout in XAML

In the previous chapter, you saw several examples of how pieces fit together in XAML. Child elements fit into panels, which arrange and size those child elements. Content fits into a Button, determining how the Button's visual appearance. Panels can nest inside other panels.

These are all examples of XAML's *composition model*. To learn to think in XAML, this concept of building a user interface out of small, fine-grained pieces must become second nature to you. The process is akin to learning a foreign language that has a different syntax and structure from the one you first learned. It takes time, but eventually you learn to think the way the language expects you to. At that point, you are able to use it much more effectively.

A companion concept is *layout*, which determines how elements are sized and positioned. If you don't understand layout in detail you will struggle to get the results you want. Do not be discouraged by this. It is a natural phase of learning to think in XAML, and with experience, layout will become mostly second nature to you.

In this chapter, we'll explore the ideas of composition and layout more deeply, and discuss related concepts such as how pieces are connected or related in XAML. Topics include:

- A simple example of XAML composition and the resulting visual rendering
- The different categories of visual elements, and how those categories are typically used
- More on panels and how they arrange and size their children
- How pieces can connect and communicate via attached properties
- Details on the Grid, RelativePanel, and Canvas panels
- Deeper exploration of the concept of content in a Control, and examples of using content
- How data binding can tie together visual elements
- Ways you can leverage the composition model to create user interfaces that are simultaneously more compelling and faster to create than those you have created in the past

A detailed example of composition

In the last chapter, we saw an example of how XAML is often used for data entry views. The following example changes that one a bit so that I can show you graphically how the composition of the view works.

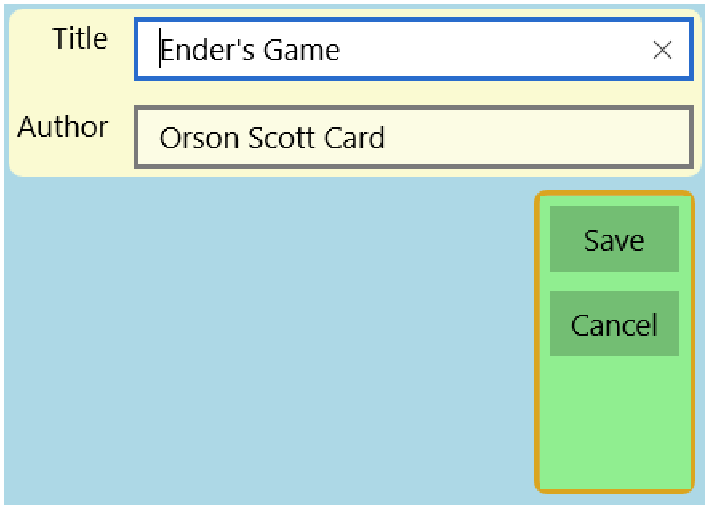
Here is the altered XAML we are going to analyze:

```
<Grid Background="LightBlue">
    <Grid.ColumnDefinitions>
```

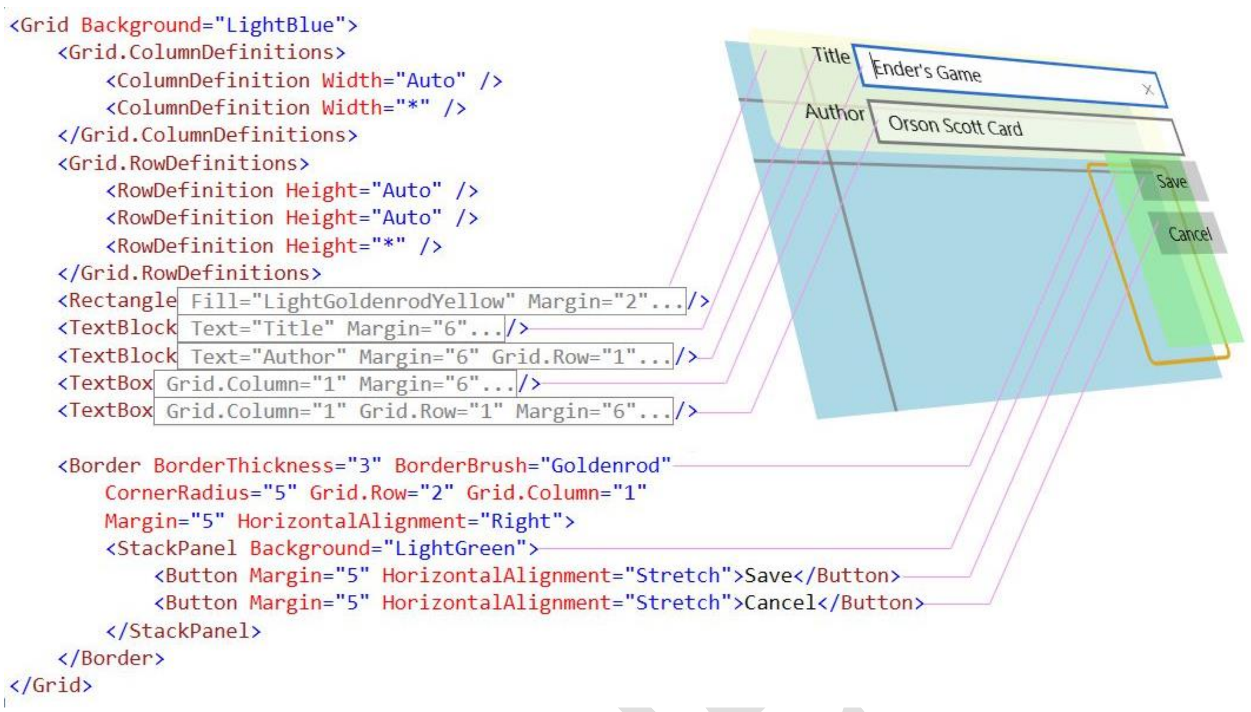
```
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Rectangle Fill="LightGoldenrodYellow" Margin="2"
               Grid.RowSpan="2" Grid.ColumnSpan="2"
               RadiusX="7" RadiusY="7" />
        <TextBlock Text="Title" Margin="6"
               HorizontalAlignment="Right" />
        <TextBlock Text="Author" Margin="6" Grid.Row="1"
               HorizontalAlignment="Right" />
        <TextBox Grid.Column="1" Margin="6"
             Text="Ender's Game" />
        <TextBox Grid.Column="1" Grid.Row="1" Margin="6"
             Text="Orson Scott Card" />

        <Border BorderThickness="3" BorderBrush="Goldenrod"
            CornerRadius="5" Grid.Row="2" Grid.Column="1"
            Margin="5" HorizontalAlignment="Right">
            <StackPanel Background="LightGreen">
                <Button Margin="5" HorizontalAlignment="Stretch">Save</Button>
                <Button Margin="5" HorizontalAlignment="Stretch">Cancel</Button>
            </StackPanel>
        </Border>
    </Grid>
```

That XAML will yield a rendering that looks much like this:



To understand how the pieces defined in the XAML correspond to the visual elements in the final rendering, here is an exploded view of the visual elements, with each XAML tag connected to the element it defines. I have added some cosmetic elements to show the rows and columns of the Grid so that you can better see how other elements fit into the Grid.



```
<Grid Background="LightBlue">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Rectangle Fill="LightGoldenrodYellow" Margin="2".../>
    <TextBlock Text="Title" Margin="6".../>
    <TextBlock Text="Author" Margin="6" Grid.Row="1".../>
    <TextBox Grid.Column="1" Margin="6".../>
    <TextBox Grid.Column="1" Grid.Row="1" Margin="6".../>

    <Border BorderThickness="3" BorderBrush="Goldenrod"
        CornerRadius="5" Grid.Row="2" Grid.Column="1"
        Margin="5" HorizontalAlignment="Right">
        <StackPanel Background="LightGreen">
            <Button Margin="5" HorizontalAlignment="Stretch">Save</Button>
            <Button Margin="5" HorizontalAlignment="Stretch">Cancel</Button>
        </StackPanel>
    </Border>
</Grid>
```

You can think of such a view as a stack of layers. The XAML elements higher in the rendering stack are laid down first[1], and then the later elements are rendered on top of them. Elements declared in containers are rendered inside their containers, with the container at a lower level than the elements it contains.

In the lower right of the rendered view is a good example. The Grid, with its light blue background, is rendered first. Then the Border with its Goldenrod outline is rendered in the appropriate cell of the Grid, in this case aligned to the right of the cell. The StackPanel with a LightGreen background is contained in the Border, so it renders inside the border. Finally, the two Button elements reside inside the StackPanel.

Thinking about a XAML view in layers can help you see new ways of using elements to visualize data for your users. After we have discussed data binding in more depth, we'll see examples of shapes in layers used for visualization.

Categories of visual elements

In the last chapter, you saw four categories of visual elements used for various purposes:

Type of element	Usage	Example(s)
Panel	Size and arrange child elements	StackPanel, Grid
Control	Provide user interaction	TextBox, Button
Shape	Provide graphical shape	Ellipse, Rectangle
Decorator	Influence visual presentation of other elements	Border

[1] People in the printing industry would use the word "compositing" for this concept.

As we said in the previous chapter, the generic word for a XAML visual object is an *element*. We'll see why as we examine the base classes in the XAML visual element hierarchy.

The Family Tree for XAML Elements

I'm not generally fond of showing class diagrams because you can get plenty of those out of the help files. Sometimes, though, a class diagram is the cleanest way to describe a set of related classes. Figure 2-1 shows an annotated class diagram of some of the major visual elements in XAML.

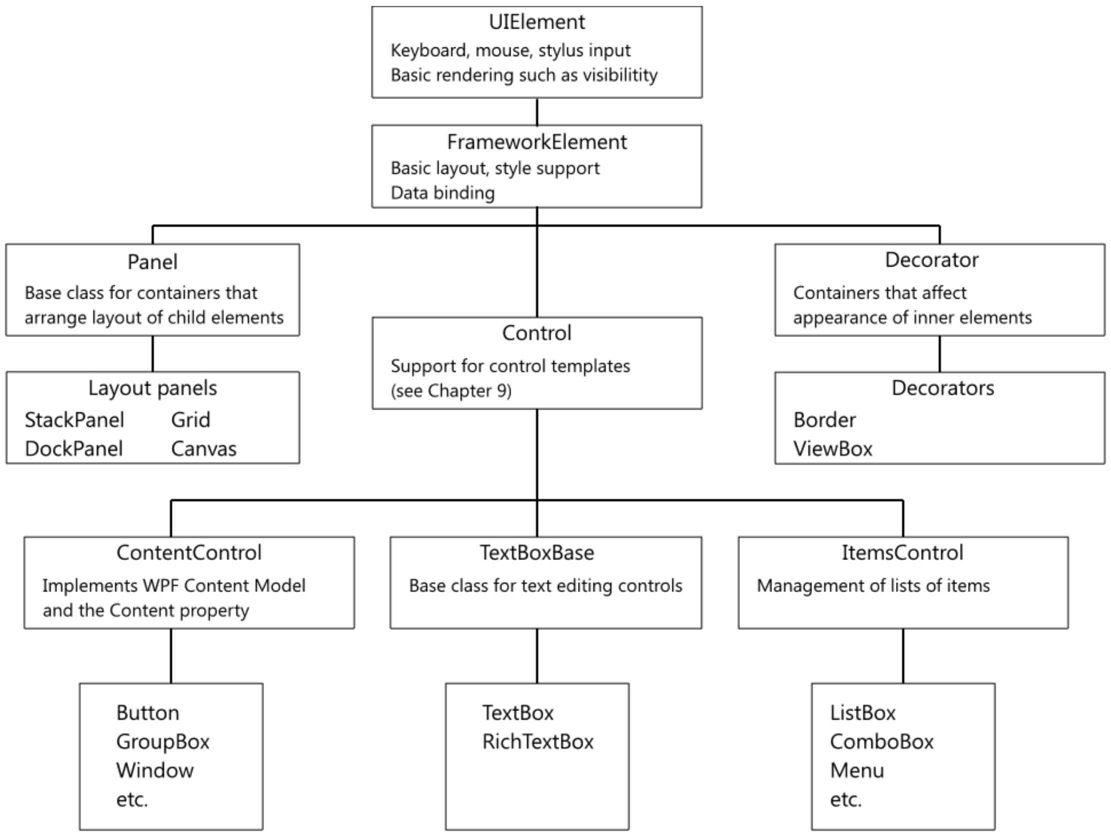

UIElement
Keyboard, mouse, stylus input
Basic rendering such as visibilitity
FrameworkElement
Basic layout, style support
Data binding
Panel
Base class for containers that arrange layout of child elements
Layout panels
StackPanel
Grid
DockPanel
Canvas
Control
Support for control templates (see Chapter 9)
Decorator
Containers that affect appearance of inner elements
Decorators
Border
ViewBox
ContentControl
Implements WPF Content Model and the Content property
TextBoxBase
Base class for text editing controls
ItemsControl
Management of lists of items
Button
GroupBox
Window
etc.
TextBox
RichTextBox
ListBox
ComboBox
Menu
etc.

Figure 2-1

This diagram is not intended to be complete; there are several classes in the hierarchy that are not included in Figure 2-1. {Fix this diagram with correct list controls for UWP}

The classes above `UIElement` (not shown) implement functionality that I won't be discussing in this book. And I'll extend the hierarchy with details on `ContentControls` and `ItemsControls` when discussing those types of controls in depth.

The elements covered in this chapter are panels and a few controls of the sub-category called content controls. Those elements are best for illustrating the composition model, and panels supply layout functionality. Later chapters contain coverage of additional elements.

The UIElement and FrameworkElement Classes

The base classes for visual XAML elements implement functionality needed by all elements. Input via the keyboard, mouse, touch, and stylus, for example, are implemented on those base classes. So are data binding and other capabilities we'll see later, such as styling.

You will use the `FrameworkElement` type on occasion. The children of a panel are a collection of `FrameworkElement` instances. To loop through such a collection, you'll need an index variable of type `FrameworkElement`. Here is an example that loops through the children of a Grid, and disables a particular event type for each child that is a Button.[2]

```
foreach (FrameworkElement ChildElement in MainGrid.Children)
{
    if (ChildElement.GetType() == typeof(Button))
    {
        ChildElement.IsTapEnabled = false;
    }
}
```

Otherwise, these types won't be of much use to for routine application programming.

Panels and Layout

If you've done HTML development, you're probably comfortable with the idea that you place your visual elements in a container that does the final arrangement. XAML takes that concept and adds many more options for controlling the final layout result.

Many desktop technologies, however, don't do much dynamic layout. Most screen elements are placed in a static position and don't move or change size. If your background focuses on these technologies, you *must* learn to do more dynamic layout, and this chapter contains a section to help you understand how it works.

Different Panels for Different Purposes

In XAML, you can choose from several different panels to contain your visual elements, each performing a different type of layout. Combining and nesting these panels gives even more flexibility. The last chapter featured a simple example of nested panels.

That means you can't just pick one type of panel and stick with it. Instead, you need to learn what each panel can do, so that you can choose the combination of panels that will give you the precise user interface layout that you want.

First, I'll cover some characteristics shared by all panels. Then I'll briefly explain each of the commonly-used panels and how it works. I'll finish our discussion of panels by suggesting a few ways you might want to use them in combination.

[2] The Tapped event is a variation on the classic Click or Clicked event for visual elements in other technologies. Click is still available in UWP XAML, and in fact it's what you should use normally to capture interaction with Buttons and other similar visual elements. Tapped is more restrictive. It can't be generated through the keyboard. Only point-capable interactions, including mouse, touch, and stylus, can generate a Tapped event.

The Panel Class

All panels descend from a XAML class named `Panel`. Since `Panel` is an abstract class, it can't be directly instantiated. It just serves as the base class for all panels, encompassing those included in XAML and those developed by third parties.[3]

Children of a Panel

A panel can contain multiple child elements. As we saw above, those elements are placed in the panel's `Children` collection. The elements in the `Children` collection must descend from the XAML `FrameworkElement` class.

In code, you can add elements to a panel using the `Add` method of the `Children` collection, the same way that you would add elements to any other collection. Early in your development, you won't need to do that very much.[4]

Typically, a panel's XAML definition contains a set of visual elements between the panel's start and end tags. Those visual elements are added to the panel's `Children` collection automatically. We saw that in our first XAML example from Chapter 1, which I'll reproduce here.

```
<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>
```

The `Button` and `TextBox` element definitions are inside the definition for `StackPanel`. When the XAML is compiled into a running program, the `Button` and `TextBox` become items in the StackPanel's Children collection.

You can see a representation of the Children collection of a panel in Visual Studio using the Document Outline. This is a tree-based diagram of the elements in a page of XAML. Figure xx shows the Document Outline for the XAML example just above. If the Document Outline is not showing, you can make it

[3] Creating your own layout panel requires a fairly deep knowledge of XAML and underlying APIs. Since the set of panels included in XAML is sufficient for the vast majority of routine application programs, this book does not explain how to build your own layout panels. The last chapter of this book includes a number of references for your investigation into more advanced topics, such as building layout panels.

[4] Advanced XAML applications with dynamic user interfaces often need visual elements added on the fly. An example would be a questionnaire which is constructed from data-driven values. In such a case, the Children collection of a panel can be created or modified in code.

visible by selecting View | Other Windows | Document Outline in Visual Studio. We'll be looking in detail at the Visual Studio designer for XAML towards the end of this chapter.

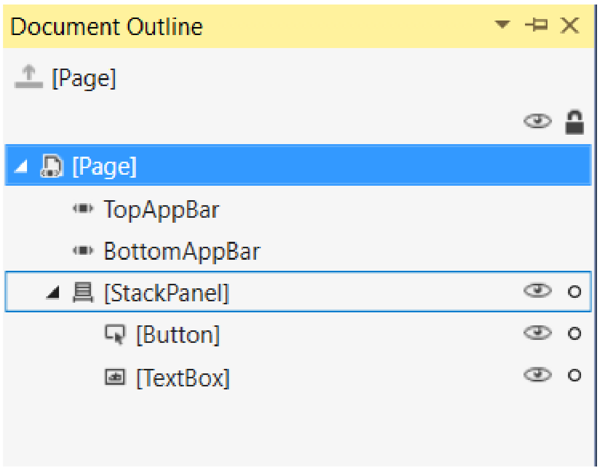


Figure xx – The Document Outline shows the children of panels, such as the StackPanel from the previous example. The order of the elements in the `Children` collection is important. The example above illustrates this. The final rendered Page showed the elements stacked in the order in which they were added to the panel. If elements are stacked in layers as seen earlier in the chapter, later elements are rendered on top of earlier elements in the stack.

StackPanel

I covered the basics of `StackPanel` in the previous chapter. It stacks elements vertically or horizontally, depending on the Orientation property.

StackPanel is the simplest panel to use. It doesn't need any information except the Orientation property to arrange its children.

However, other Panels have more flexibility, and need more information on where to put each child element. Next, I'll discuss where they get that information.

Attached Properties

The property system used in XAML allows a special type of property called an *attached property*. We saw these in use in the previous chapter. Grid.Row and Grid.Column are used to specify the row and column position for a child element of a Grid. Here are the key lines of XAML in that example, with the attached properties in boldface:

```
<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"
```

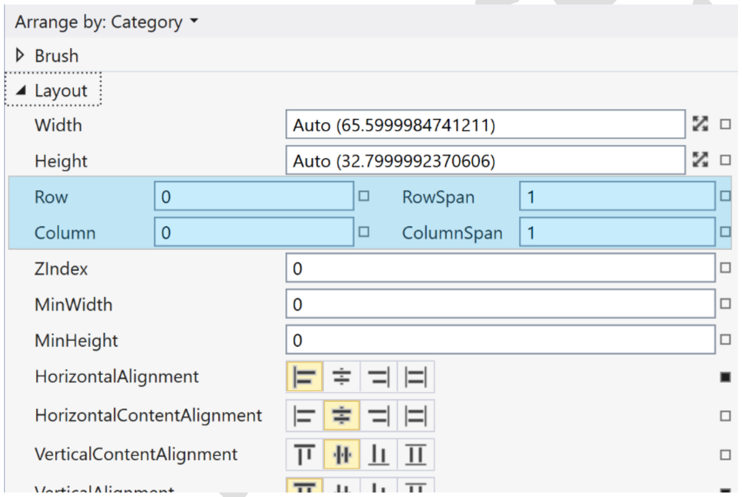
The top TextBlock does not specify values for Grid.Row or Grid.Column, so they keep the default of zero. That TextBlock will appear in the top left cell of the parent Grid, because that cell is in row zero and column zero.

The other elements are positioned by setting Grid.Row or Grid.Column or both. The parent Grid can fetch Grid.Row and Grid.Column values for each of its children, and it uses those values to place the children in appropriate grid cells.

Based on your intuitive idea of how properties work in a typical object-oriented platform, you might think that Grid.Row and Grid.Column are properties, in some sense, of the TextBlock. They're not. These attached properties are actually implemented on the Grid class. Attached properties retain not just the value being set (e.g. the row or column number), but also the associated element. The Grid can keep an attached property value for as many elements as it needs to. That's the beauty of the system; no matter how many children the Grid holds, it can store information for each one to allow it to position the children inside the Grid.

Setting Attached Properties in Visual Studio

You don't have to edit XAML directly to set attached properties. If you create a `Grid` in Visual Studio and drag a `TextBlock` into the `Grid`, the attached properties for Grid.Row, Grid.Column, Grid.RowSpan, and Grid.ColumnSpan will appear in the `TextBlock's` Properties window, in the `Layout` category. Most of the attached properties of panels that help position their children will be in the `Layout` category, as shown in Figure X.



Arrange by: Category
Brush
Layout
Width: Auto (65.5999984741211)
Height: Auto (32.7999992370606)
Row: 0 RowSpan: 1
Column: 0 ColumnSpan: 1
ZIndex: 0
MinWidth: 0
MinHeight: 0
HorizontalAlignment
HorizontalContentAlignment
VerticalContentAlignment

Figure X – Attached properties for a Grid can be modified in the Properties Window, as shown in the highlighted area above.

As with other properties, when you set an attached property in the Properties window, the designer writes the property setting into XAML.

Setting Attached Properties in Code

It is occasionally necessary to set attached properties in code. That brings up an interesting twist in the way attached properties work at the code level. If you don't expect to do this anytime soon, and don't care at this point how attached properties work underneath the XAML surface, then you can skip this section.

In code, an attached property isn't really a property at all; it's a set of static methods on the class that implements the attached property.

For every attached property, there is a Get method on the implementing class, and a Set method. The methods have a standard naming convention. The Get method for `RowSpan` must be named `GetRowSpan`, for example, and the `Set` method for `RowSpan` is similarly required to be `SetRowSpan`.

Remember that an attached property must retain both a value and the associated element. That is, Grid.Row has to store both the row value and the element that needs to be positioned in that row. The way Get and Set methods are constructed in code satisfies this need.

The XAML compiler just takes **`Grid.Column="1"`** and knows how to work with the underlying implementation. But if you are setting an attached property in code, you must know how to use these Get and Set methods.

These `Get` and `Set` methods for an attached property require an argument for the element to which the property value is "attached".[5] For example, to get the current value of the `Grid.RowSpan` setting for an element named `Button1`, you could use a line of code like this:

```
int i = Grid.GetRowSpan(Button1);
```

The `Set` method needs two arguments. It needs the element to which the property is "attached", just as the Get method did. It also needs a second argument for the property value that is being set. To set the `Grid.RowSpan` attached property on a button named `Button1` to the value of 2, the following code would be used:

```
Grid.SetRowSpan(Button1, 2);
```

Grid

Now that we've got attached properties squared away, we can look at the other panels in depth.

We've already spent some time looking at the basics of the Grid panel. It was included in the previous "high altitude" chapter because it is the most flexible panel and most used panel in a typical XAML application. I find that almost every XAML interface I design has several `Grid` panels, sometimes contained inside one another. The `Grid` is so flexible and useful that the XAML visual designer uses it as the default layout container for a new `Page`.[6]

The `Grid` panel allocates its space to cells. Those cells are the intersections of rows and columns. Typically, the first thing you do for a `Grid` is to define the rows and columns of the `Grid`.

[5] If you have worked with Windows Forms, this coding pattern may feel familiar because extended properties in Windows Forms work much like this.

[6] For many developers, "grid" is synonymous with "data grid". That association does not apply in XAML. The Grid panel in XAML is not a data grid in any sense. In fact, the control set in UWP XAML does not include a data grid as this chapter is being written. There are various ways to simulate data grid functionality, though, and I'll tell you about some of those in Chapter 5 after explaining data templates.

The rows do not need to be the same height, and the columns do not need to be the same width. They can be, of course.

A `Grid` has a property called `ColumnDefinitions`, which contains a collection of `ColumnDefinition` elements. Each one is the definition for one column. Similarly, there is a `RowDefinitions` property for `RowDefinition` elements. Here is simple XAML showing `ColumnDefinitions` and `RowDefintions` for a `Grid`, in this case defining three columns and two rows:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
</Grid.ColumnDefinitions>

<Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
</Grid.RowDefinitions>
```

Sizing Rows and Columns

The simple example just above has no sizing information for the rows or columns, so the rows and columns default to equal sizes that add up to the entire size of the Grid. Before I can present the XAML for sizing rows and columns, I need to explain the sizing options. I'll talk about sizing columns first, and then recast the discussion for rows, because they both use the same sizing concepts.

Columns in a `Grid` can be a *fixed width*, a *proportional width*, or an *automatic size.* These settings can be mixed and matched as needed for different columns in the grid:

- **Fixed width** works just as you would expect. The width of a column can be set to a specific number of units.
- **Automatic width** means that the column is sized to the largest element contained in any cell of that column. If the column does not contain any elements, it has zero width.
- **Proportional width** relates the width of a column to the width of other columns. After fixed width columns and automatically sized columns have been sized, any remaining width is divided among proportional width columns.

The amount each proportional column gets depends on a number associated with that column. The number is sometimes called its "star" value because it is shown in XAML together with an asterisk. Here is a typical ColumnDefinition with a star value for Width:

```
<ColumnDefinition Width ="4*" />
```

All the star values for all the proportional width columns are added up to give a sum. Then each column's width is calculated by dividing that column's number by the sum, and multiplying times the available width for all the proportional columns.

Math always sounds complex when expressed in words, so let's look at an example. Suppose you have three proportional width columns, and the "star" numbers for them are 1, 4, and 5. The ColumnDefinitions collection for three such columns would look like this:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="4*" />
```

```
        <ColumnDefinition Width="5*" />
    </Grid.ColumnDefinitions>
```

The sum of the numbers for all three columns is 10. Column one gets 1/10 of the available width, column two gets 4/10 (or two fifths), and column three gets 5/10 (or a half). If the amount of width available in the grid for proportional columns were 200 units, then column one would be 20 units wide, column two would be 80 units, and column three would be 100 units.

The XAML for all three sizing options is straightforward. Each option has its own way of setting the Width property of the ColumnDefinition. We saw the "star" setting option in the XAML above.

For a fixed size, `Width` is set to number of units desired. For automatic size, `Width` is set to **Auto**.

Listing 2-1 contains a Grid with a set of column definitions showing each sizing option. The first column has a fixed width of 40. The second column has an automatic width. The third and fourth columns receive 40% and 60%, respectively, of the remaining width of the `Grid`. The remaining XAML places a Rectangle in each column.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="40" />
        <ColumnDefinition Width ="Auto"/>
        <ColumnDefinition Width ="4*" />
        <ColumnDefinition Width ="6*" />
    </Grid.ColumnDefinitions>
    <Rectangle Stroke="DarkRed" StrokeThickness="3"
               Fill="LightCoral" Margin="2" />
    <Rectangle Stroke="DarkGreen" StrokeThickness="3"
               Fill="LightGreen" Grid.Column="1" Width="30" Margin="2" />
    <Rectangle Stroke="DarkBlue" StrokeThickness="3"
               Fill="LightSteelBlue" Grid.Column="2" Margin="2" />
    <Rectangle Stroke="DarkGoldenrod" StrokeThickness="3"
               Fill="LightYellow" Grid.Column="3" Margin="2" />
</Grid>
```

Listing 2-1. A set of illustrative column definitions for a Grid, with an element in each column

The XAML in Listing 2-1 will produce a result much like Figure 2-4. The width of the first column is 40 unconditionally, while the width of the second column is based on the width of the Rectangle placed in it. The third and fourth columns share what's left, in a ratio of 2 to 3.

Notice that only the Rectangle in the second column (Grid.Column="1") has a declared Width. If it's Width were the default of zero, then that column would not show at all in the rendered result.

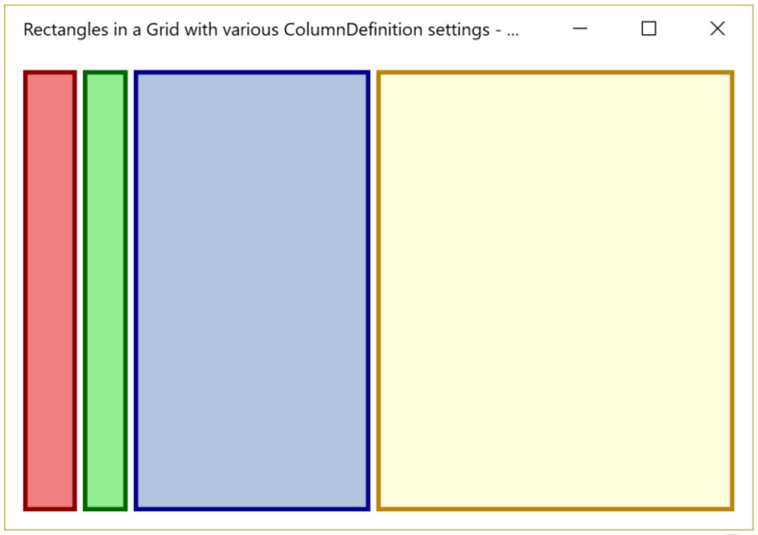

Rectangles in a Grid with various ColumnDefinition settings - ...

Figure 2-4 A `Grid` with four columns defined by the XAML example in Listing 2-1, and with four rectangles included to show the size and location of the columns.

The numbers on the last two columns could have been “6*” and “9*”, or “100*” and “150*” or any other numbers with a ratio of 2 to 3, and the result on the screen would be the same. For proportional widths, it doesn’t matter what the numbers are. It only matters what fractional portion a column’s number makes up of the sum for proportional columns.

A common practice for setting the number for proportional columns is to use numbers that add up to 100. That way, you can immediately tell percentage of available width each column gets.

The exact same sizing options are available for rows. The only difference is that you use the `Height` property of a `RowDefinition` in place of the `Width` property of a `ColumnDefinition`. I won’t bore you by going through a virtually identical example for rows.

Designing Grids in Visual Studio

The XAML designer in Visual Studio gives you the capability to place rows and columns in a `Grid`. When you click on a `Grid`, a light blue bar appears on the top and left-hand sides of the grid. Clicking in the bar on the top causes a column to be defined, and clicking in the left-hand side bar defines a row.

You can also set the size of rows and columns. Edges of rows and columns are shown with light blue lines inside the grid. Each line has a corresponding small triangle in the light blue bars. Clicking the triangle associated with a column or row border allows you to drag and change the size of the column or row. Figure 2-5 shows a `Grid` in Visual Studio with several rows and columns defined.

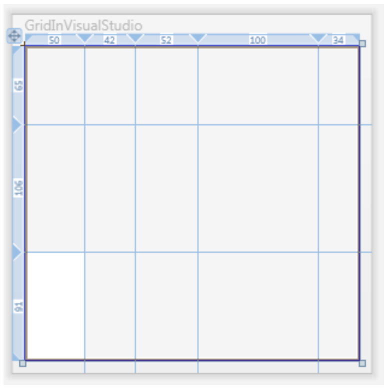


Figure 2-5 Designing a Grid in Visual Studio

If you hover over a column size at the top, or a row size on the left, you will see a drop-down with other options for width or height. Figure xx shows the drop-down, both as it initially appears, and as it looks when you drop it down.

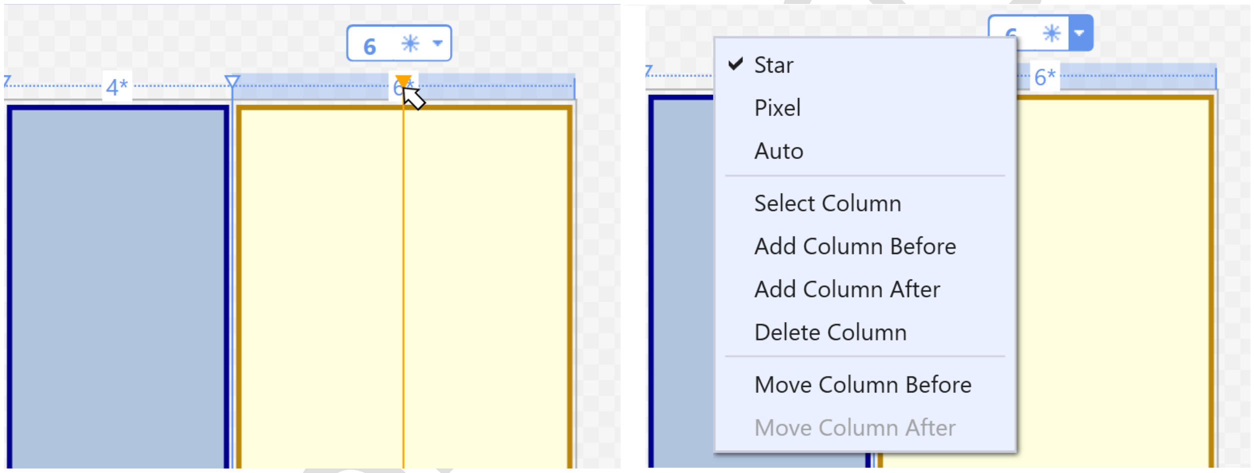
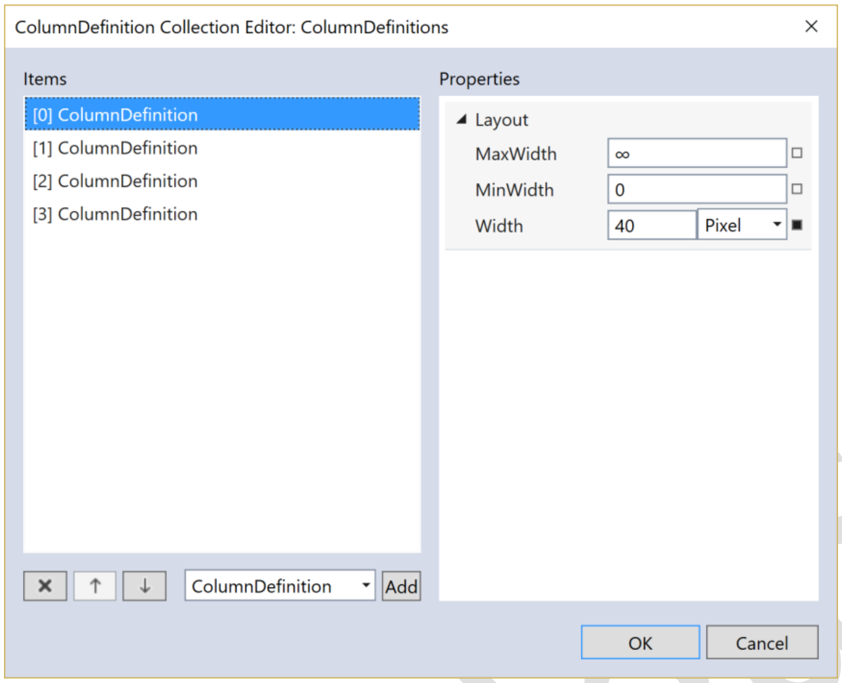


Figure 2-6 Setting a column width while designing a Grid in Visual Studio

I find, though, that I rarely use the drag-and-drop Grid designer in Visual Studio. Drag and drop sizing is imprecise for star values, and I find that it's tedious to move rows and columns using this designer.

Instead, I routinely use the editors for the `ColumnDefinitions` and `RowDefinitions` collections. You can get to those editors in the Properties window for the `Grid`. The entry in the Properties window for `ColumnDefinitions` has a button on the right with an ellipsis. Clicking that button brings up the `ColumnDefinitions` editor, which gives you precise control over all the different properties of each `ColumnDefinition`.

Figure 2-6 shows the `ColumnDefinition` editor. Using the editors, you can set the size of rows and columns directly. You can also access other useful properties of rows and columns.



MaxWidth ∞
MinWidth 0
Width 40 Pixel

Figure 2-6 The ColumnDefinitions editor in Visual Studio

For columns, the `MinWidth` and `MaxWidth` properties allow you to constrain the size of automatically sized rows. That includes both columns set with "Auto" and columns with proportional sizing. Rows have a corresponding `MinHeight` and `MaxHeight` property.

You can also easily move rows and columns using the editor. You do that with the buttons that contain up and down arrows, in the lower right of the editor.

Placing Elements in a Grid

Now that you've seen how to set up the Grid with rows and columns, you're ready to place visual elements inside a Grid. That's done with attached properties. We discussed the attached properties for Grid positioning in the first chapter, but here's a quick review, along with some guidance on using the drag and drop designer to use these properties.

To specify the column and row for an element, you use the `Grid.Column` and `Grid.Row` attached properties. The numbering for rows and columns is zero-based, just like all collections in .NET. To place a `Button` in the second row and third column of a grid, the `Button` would be defined inside a `Grid` with a line of XAML like this:

```
<Button Grid.Row="1" Grid.Column="2">Button text</Button>
```

If you drag an element into a cell of a `Grid` using the Visual Studio designer, the `Grid.Row` and `Grid.Column` attached properties are set for you. But, as I'll discuss in detail below, dragging elements into a `Grid` can have unexpected side effects.

The default for `Grid.Row` and `Grid.Column` are zero. Elements that don't set a row or column end up in the top left cell of the `Grid`.

Spanning Columns and Rows

By default, an element in placed in a single cell of a grid. However, an element can begin in the cell assigned by `Grid.Column` and `Grid.Row`, and then span additional rows and/or columns.

The attached properties `Grid.ColumnSpan` and `Grid.RowSpan` determine the number of columns and rows an element spans. Their default is, of course, 1. You can set either or both for an element. Setting only `Grid.ColumnSpan` restricts an element to a single row, but extends the element into extra columns. Similarly, setting only `Grid.RowSpan` extends through multiple rows in the same column. Setting both causes an element to span a rectangular block of cells.

To see `Grid.Column`, `Grid.Row`, `Grid.ColumnSpan`, and `Grid.RowSpan` in action, let's look at an example. Listing 2-2 shows the XAML for a Grid with several elements, using varying rows, columns, and spans. I'm using an element called Frame that has not been discussed yet, but it's a convenient element to use for this example because it automatically spans its block of cells, and I can put some text inside of it.

Figure 2-7 shows what that Grid would look like in design view in Visual Studio. I'm showing the design view so that you can see the blue lines that define rows and columns, and how buttons span the rows and columns based on their settings in XAML.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="50*" />
        <RowDefinition Height="20*" />
        <RowDefinition Height="30*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="20*" />
        <ColumnDefinition Width="20*" />
        <ColumnDefinition Width="45*" />
        <ColumnDefinition Width="15*" />
    </Grid.ColumnDefinitions>
    <Frame Background="LightSeaGreen">
        1 cell
    </Frame>
    <Frame Grid.Column="1" Grid.RowSpan="2"
           Background="LightSteelBlue" >
        2 cells
    </Frame>
    <Frame Grid.Row="2" Grid.ColumnSpan="2" Background="LightSalmon">
        2 cells
    </Frame>
    <Frame Grid.Column="2" Grid.ColumnSpan="2"
           Grid.RowSpan="2" Background="LightYellow">
        4 cells
    </Frame>
    <Frame Grid.Column="3" Grid.Row="2" Background="LightCyan">
        1 cell
    </Frame>

</Grid>
```

Listing 2-2 XAML for a Grid containing Frame controls with various settings for row and column, and various settings for spanning of rows and columns.

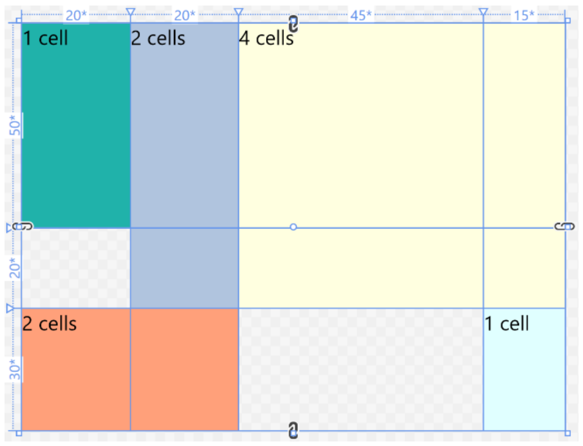


Figure 2-7 A design view of the Grid defined in Listing 2-2.

The first Frame is simple. It has default values for all attached properties, so it is positioned in the first cell and spans no additional cells.

The second Frame has a `Grid.Column` setting of 1, so it is positioned in the second column. It also has a `Grid.RowSpan` setting of 2, so it spans both the first and second cells in the second column.

The third Frame has a `Grid.Column` setting of 2, so it starts in the third column, first row. Both `Grid.RowSpan` and `Grid.ColumnSpan` are set to 2. The resulting Frame spans a rectangular block of four cells.

I'll leave it as an exercise for you to understand the positioning and size of the last two Frame elements.

Arranging Elements in a Cell or Set of Cells

You can place more than one element into a cell of a grid. With default settings, that will usually cause one element to be on top of the other.

I use this capability quite often, and to be a fluent XAML author, you should too. For example, you could place a shaded rectangle in a particular row of a `Grid`, using `Grid.ColumnSpan` to cause the rectangle to span all the columns in the row. Then you could place other elements in cells in that row, and all of them would sit on top of the shaded rectangle.

Effectively, the Rectangle would furnish a background color for the entire row. Grid rows, columns, and cells don't have an explicit property for setting a background. Including a Rectangle as the first element in a cell or set of cells is the easiest way to get that visual result.[7]

In later chapters, you'll see various examples of stacking elements in a Grid cell. In some cases, translucency is used to allow elements in the back to "show through" elements in front. We'll look at some examples in Chapter X that talks about graphical effects.[8] We will also do an example that depends heavily on layering towards the end of this chapter.

[7] You might be worried that adding such elements just to get certain visual effects would raise performance issues. For routine usage, it does not. Shapes such as Rectangles are extremely lightweight, and are some of the most optimized elements to render.

[8] Layering is an important part of XAML's composition model. In older UI stacks, the rendering engines had layering and translucency capabilities that were weak or non-existent. The way you think about how a screen or view is laid out probably reflects those constraints. One of my secondary goals in this book is to loosen up your thinking about layering elements. You can do some amazing things with it.

In other cases, you might not want layering. You may need to place multiple elements in a grid cell, and ensure that they do not overlap. Let's consider two ways you can do that, one of which I don't recommend that you use because it doesn't leverage composition in XAML.

You already know the way I recommend because we did an example in Chapter 1. You can place another panel inside the cell of a `Grid`, and then place elements in that panel. If you placed a `StackPanel`, for example, in a `Grid` cell, and then placed elements in the `StackPanel`, they would be stacked and not overlapping.

A alternative is to use the `Margin` property of the elements. You can set appropriate `Margin` and alignment settings on different elements to make both visible, and to have no overlap. Figure XX shows an example of the technique.

```
<Button Grid.Row="1"
        Grid.Column="1"
        VerticalAlignment="Top" >
    Top button
</Button>

<Button Grid.Row="1"
        Grid.Column="1"
        VerticalAlignment="Top"
        Margin="0,39,0,0" >
    Bottom button
</Button>
```

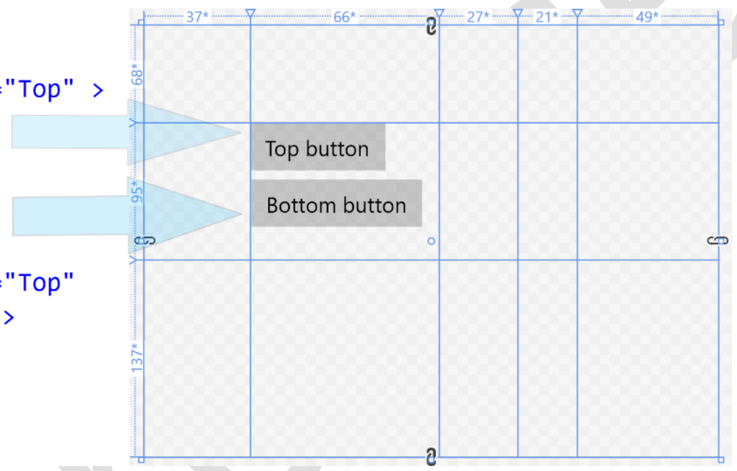


Figure 2-8 Two Button elements positioned in the same Grid cell using alignment and Margin properties.

If this technique isn't recommended, why am I showing it to you? I'm doing it because you'll see results like this when using the drag-and-drop visual designer for XAML in Visual Studio.

If you use the visual designer to drag elements such as Buttons into a Grid cell and position them so that they don't overlap, the designer inserts alignment and Margin settings, much like the example above, to position the element where you dropped it.

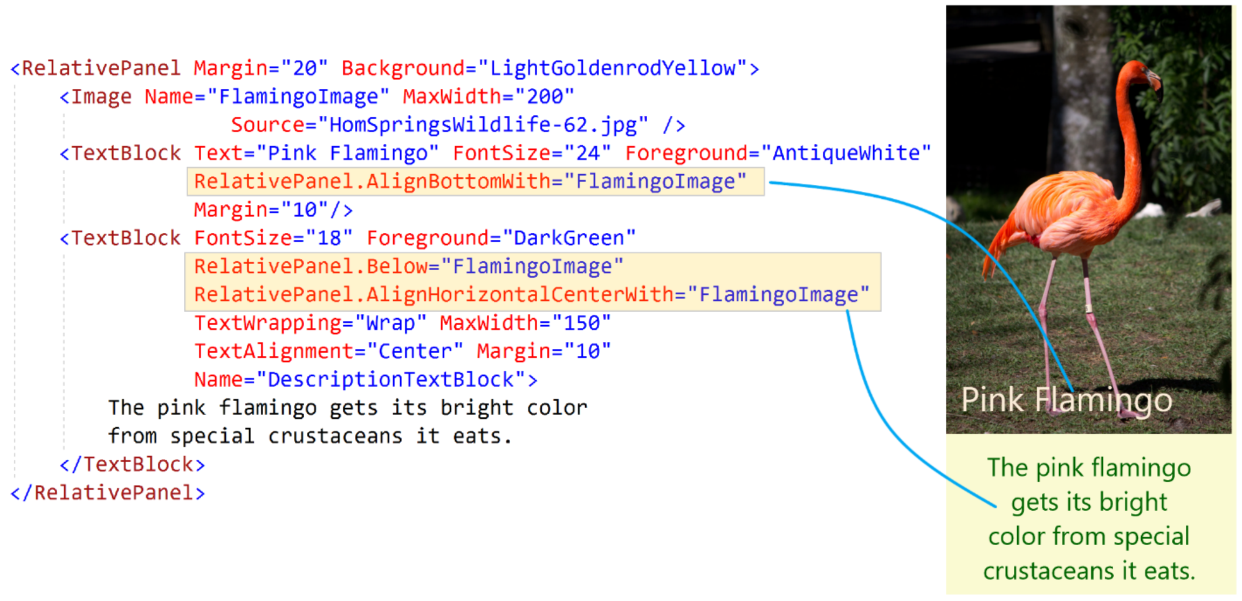
I wish the designer didn't work that way. It's a pretty silly way of positioning elements just to keep them from overlapping. It's tedious, and can lead to unexpected results when the size of the application Window changes. Most experienced XAML authors realize quickly that they will get the results they want faster if they don't drag and drop from the Toolbox. Instead, they use techniques that give more control, such as putting a StackPanel or RelativePanel into a Grid cell or bank of cells, and then letting that panel arrange elements without overlap.

Don't let the visual designer lead you into thinking that using Margin to position elements in a Grid cell is a good idea. It usually isn't.

RelativePanel

Some layouts need the ability to position items relative to one another. You can spoof that kind of layout with a Grid, but a RelativePanel has options for such positioning built in.[9]

Figure xx shows a simple example in XAML.



```xml
<RelativePanel Margin="20" Background="LightGoldenrodYellow">
    <Image Name="FlamingoImage" MaxWidth="200"
                  Source="HomSpringsWildlife-62.jpg" />
    <TextBlock Text="Pink Flamingo" FontSize="24" Foreground="AntiqueWhite"
               RelativePanel.AlignBottomWith="FlamingoImage"
               Margin="10"/>
    <TextBlock FontSize="18" Foreground="DarkGreen"
               RelativePanel.Below="FlamingoImage"
               RelativePanel.AlignHorizontalCenterWith="FlamingoImage"
               TextWrapping="Wrap" MaxWidth="150"
               TextAlignment="Center" Margin="10"
               Name="DescriptionTextBlock">
        The pink flamingo gets its bright color
        from special crustaceans it eats.
    </TextBlock>
</RelativePanel>
```

Pink Flamingo
The pink flamingo gets its bright color from special crustaceans it eats.

Figure 2-8 The RelativePanel positions elements in relationship to one another, using attached properties for various positioning possibilities.

The first element is an Image control holding a photo of a pink flamingo. It serves as the anchor element, and the two TextBlock elements are positioned relative to the Image, using attached properties of the RelativePanel. The XAML that sets those attached properties is highlighted in Figure xx.

These are just three of over a dozen attached properties that can be used for positioning in a RelativePanel. Most of them have names that are self-explanatory, so I won't bore you by going through all of them. Instead, I'll just list them in a table, and invite you to play around with them to gain proficiency with the RelativePanel. Don't forget to try combinations; particularly the "AlignHorizontal…" and the "AlignVertical…" properties work in combination with others, as shown in the example in figure xx.

AlignBottomWith	AlignRightWith	Above
AlignBottomWithPanel	AlignRightWithPanel	Below

[9] RelativePanel is the only Panel I'll be discussing that is not present in earlier versions of XAML, such as WPF and Silverlight.

AlignHorizontalCenterWith	AlignTopWith	LeftOf
AlignHorizontalCenterWithPanel	AlignTopWithPanel	RightOf
AlignLeftWith	AlignVerticalCenterWith	
AlignLeftWithPanel	AlignVerticalCenterWithPanel	

The RelativePanel also has an advanced capability to enable even more dynamic layout. It allows you to create *adaptive triggers* that change the layout based on the size of the RelativePanel, and will move items around when the RelativePanel gets smaller or bigger. While there are some scenarios where this is quite useful, it's not something you will use constantly, so I'm leaving it out of this initial discussion of panels and layout.

Canvas

The `Canvas` panel is easier to understand than Grid or RelativePanel. I've placed it at the end of the panel discussion, though, because it should be used sparingly in XAML interfaces.

If you're a Windows Forms developer, you might feel an affinity for the `Canvas` panel. It positions children in almost exactly the way as a `Form` in Windows Forms. That may tempt you to rely on the `Canvas` so that you can avoid understanding the complexity of the other panels.

You should resist that temptation. While the `Canvas` certainly has valid uses, it should not normally be the dominant element on a XAML user interface. If it is, the interface will not possess many of the best characteristics of a XAML interface. It won't automatically adjust children to fit different aspect ratios, for example.

However, `Canvas` is a good choice for several application scenarios. Here are a few examples:

- Graphing and charting surfaces
- Positioning elements with some real world arrangement, such as bays in a warehouse or gates at an airport
- Surfaces that allow users to move elements around to create their own layout

Positioning Child Elements on a Canvas

The attached properties for position child elements of a Canvas have nice, familiar names:

- `Canvas.Top`
- `Canvas.Left`

If set, `Canvas.Top` and `Canvas.Left` work exactly as the `Top` and `Left` properties of a control in Windows Forms. These two settings determine the position of the top left corner of a child element.

The Canvas does not size its child elements. There are not any logical bounds to use for sizing, so the Canvas just lets elements be whatever size they want to be. As we will see when we talk about layout below, that size is usually determined by what's inside the elements.

Of course, some elements don't have anything inside them. Shapes such as Rectangle are examples. These elements will default to zero size, so if you put one in a Canvas and don't give it a size, it will be zero size and won't be visible.

Using Panels in Combination

In Chapter 1, one of the examples had a `StackPanel` nested inside a `Grid` to provide a convenient way to stack some buttons in a Grid cell. That is merely a simple example of a general technique.

Thinking in XAML means using panels in combination to achieve the visual results you want. It means taking advantage of the automatic sizing and layout options of panels to provide a dynamic, responsive user interface that can adjust itself to a variety of display and windows sizes.

Even routine XAML pages with responsive design often involve nesting panels four or five levels deep. Let's look at an example that uses nesting to provide flexible and dynamic layout.

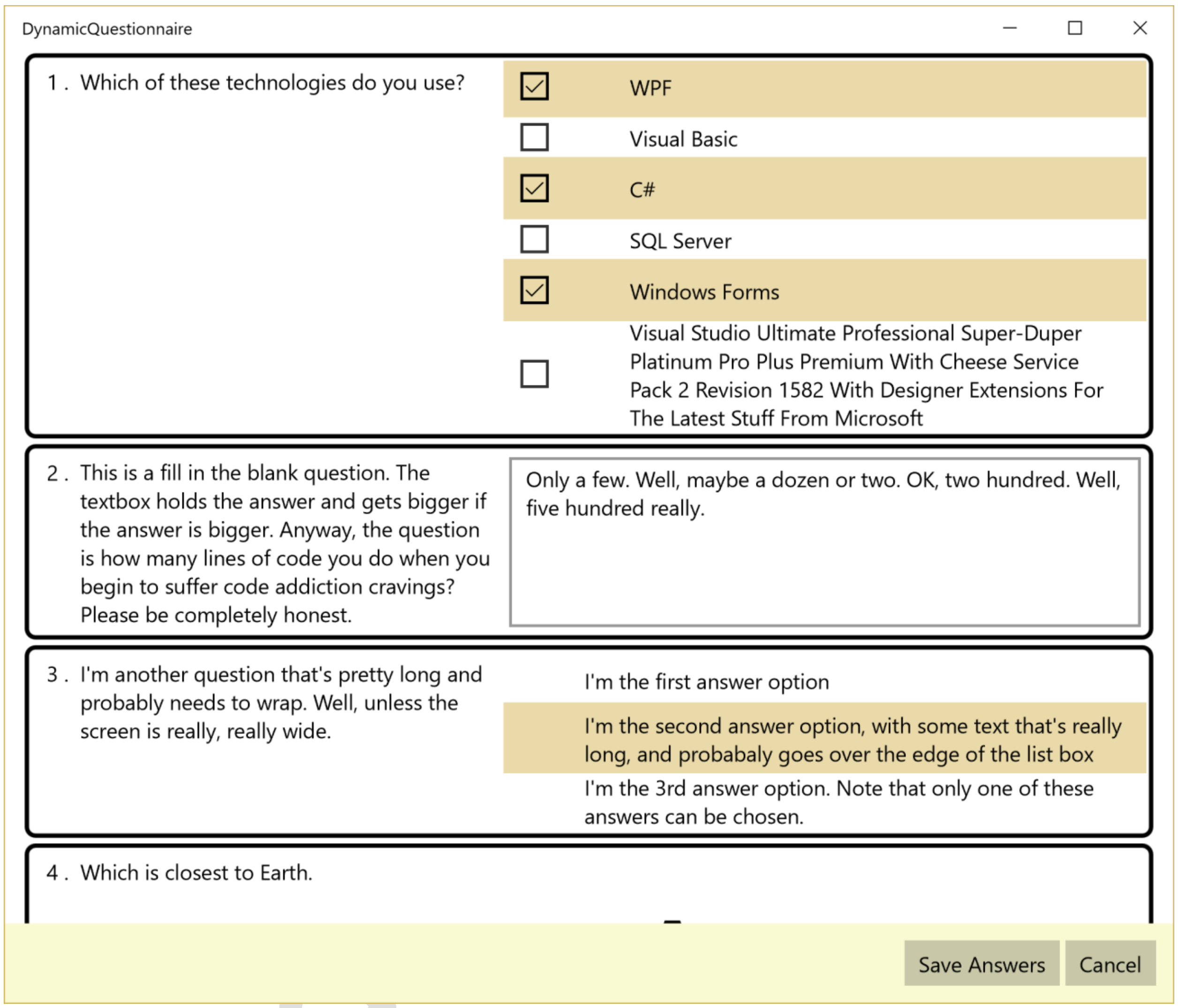
Figure XX shows a view for a questionnaire. Each question in the view has:

- A question number
- The text of the question
- Answers to be filled in. Different types of questions have different types of answers, and some questions allow the user to choose one or many options from a list of possible answers.

The bottom of the view contains buttons for typical actions.

A static image in a book can't show the dynamic layout furnished by the arrangement of panels in this app. The full application shown in Figure xx is included with the downloads for this book, and you might want to load that project (DynamicQuestionnaire), resize the window for the application, and notice the dynamic nature of the view.

When the view is resized, text fields for questions and answers wrap automatically, and change the number of lines they need depending on screen size. Sometimes the question is longer than the answer, and sometimes the answer or list of answers is longer. The dynamic layout works fine in either case. The questions will scroll if they can't fit on the screen, but the bottom section with the buttons remains there for any size of the window.


DynamicQuestionnaire

1 . Which of these technologies do you use?

- [✓] WPF
- [ ] Visual Basic
- [✓] C#
- [ ] SQL Server
- [✓] Windows Forms
- [ ] Visual Studio Ultimate Professional Super-Duper Platinum Pro Plus Premium With Cheese Service Pack 2 Revision 1582 With Designer Extensions For The Latest Stuff From Microsoft

2 . This is a fill in the blank question. The textbox holds the answer and gets bigger if the answer is bigger. Anyway, the question is how many lines of code you do when you begin to suffer code addiction cravings? Please be completely honest.

Only a few. Well, maybe a dozen or two. OK, two hundred. Well, five hundred really.

3 . I'm another question that's pretty long and probably needs to wrap. Well, unless the screen is really, really wide.

I'm the first answer option

I'm the second answer option, with some text that's really long, and probabaly goes over the edge of the list box

I'm the 3rd answer option. Note that only one of these answers can be chosen.

4 . Which is closest to Earth.

Figure xx – A dynamic questionnaire, with layout that adapts to the size of the window by rearranging and resizing visual elements. The application is included in the downloads for the book. If you run the application and resize the window, you will see the visual elements rearrange themselves.

Let’s walk through the panels that are used to achieve this layout.

The outer panel is a Grid, with an automatically sized row at the bottom for the two Buttons in the lower right, and one other row that takes up the rest of the Grid. The RowDefinitions for that outer Grid look like this:

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

The individual questions are included within a Border that draws the black outline around the question. There are four questions, and they are in a vertical stack managed by a StackPanel. The bottom question is truncated because the StackPanel does not have enough room to display all questions.

Inside each individual question is a Grid with three columns. The first column holds the question number, the second holds the question text, and the third holds the area where the user answers the question.

That Grid's ColumnDefinitions collection looks like this in XAML:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="2*" />
    <ColumnDefinition Width="3*" />
</Grid.ColumnDefinitions>
```

The column with the question number is a fixed size, and the question text and answer columns split the remaining width in a ratio of 2 to 3. If you run the program and resize the window for it, you'll see that proportion being preserved.

Finally, some of the questions have a stack of answers. Those stacks, in the first and third questions are arranged by a StackPanel.

This user interface isn't that complex, but it already has four levels of nested panels to achieve the desired arrangement. Complex business apps will often have more than double this number of levels. Nesting panels is one of the keys to leveraging the power of XAML because they are the mainstays of dynamic layout.

Sizing and Layout of XAML Elements

As I noted in the Introduction, one of the primary goals of XAML is allowing creation of interfaces that intelligently use the space they are given by resizing and rearranging elements on the fly. With widely varying resolutions, plus the fact that an application window can be in a wide range of sizes, many different factors affect the final rendered size and position of a XAML element.

In this section, I'd like to give you an overview of the major factors that affect sizing and layout of XAML elements. I won't try to tell you everything there is to know about the subject – just enough to handle the common cases you will run into during application programming.

This section isn't required reading. You will eventually develop some intuition about the layout process. But if you would like to speed up the development of that intuition, I think this section will help. It fits the "thinking in XAML" theme of the book.

I apologize in advance for the fact that you may have to read this section a couple of times to sort out all of factors involved in sizing and layout. Some of it probably won't be clear until you try the techniques in your own projects. Layout in XAML a complex subject, but a necessary one to understand if you're going to design anything beyond very simple interfaces.

What is Layout?

In XAML, *layout* means sizing and arranging the children of a panel. This is a complicated, recursive process. Since panels may contain other panels as children, as in the nested panels example above, sizing and positioning must be done at several "levels" before a complete `Page` can be rendered. The process starts in the root element, and goes down through the tree of child elements. At each level, there is interaction between child elements and their parent to decide on size and position.

If you don't understand some of the mechanisms used for this process, you'll constantly be producing XAML interfaces that don't act the way you expect. That happened to me more times than I can remember while learning XAML. It will happen to you too, but I hope this section will mean it happens fewer times.

First step – Measuring the Child Elements

Measurement is the process of determining the size that an element wants to be when it is rendered. That size is usually called the desired size.[10] It may or may not be the actual rendered size; that depends on several other factors, including whether the height and width values are hard-coded, and what the container holding the element wants to do with the element.

The base `FrameworkElement` class contains several properties that furnish information to the measurement process, including:

- `Height` and `Width`
- `MinHeight`, `MaxHeight`, `MinWidth`, and `MaxWidth`
- `Margin`

Using Height and Width Properties

`Height` and `Width` work exactly as you would expect. They hard-code the size of an element, just as they do in other user interface technologies. If an element with hard-coded size is placed in a panel, the panel will respect that size and will not change it. If there's not enough room to show the element, the panel will just truncate the element.[11]

Since these two properties work in a way you'll find familiar, at first you may be tempted to bypass the whole process of understanding XAML's complex sizing process, and just set the sizes you want. For most XAML applications, that's a mistake. It requires giving up much of the flexibility of XAML to respond to different conditions. You should use explicit settings for `Height` and `Width` rarely; only do it if automatic sizing doesn't fit your needs. It should be your last resort, not your first one.

One common place to explicitly set `Height` and `Width` is on the root element of your interface. In fact, when you create a new `Page` in Visual Studio, it will have `Height` and `Width` explicitly set to default values. If you are using the normal Visual Studio template for a `Page`, a new one will be set to 300 by 300 units.

Applying Minimum and Maximum Values

The next level of control over size is to set a range of values for the height and width of an element. The four properties `MinHeight`, `MaxHeight`, `MinWidth`, and `MaxWidth` contain the range settings. I'll discuss in terms of width, but the exact same discussion applies to height.

If `MinWidth` is set for an element, it won't have a width less than that amount. If a `MaxWidth` is set, the element won't be made wider than that amount.[12]

[10] In fact, the UIElement class has a property called DesiredSize. It holds the desired size value during the layout process. However, until you begin writing XAML controls, it's unlikely that you'll need to use that property.
[11] This isn't strictly true; you could create a panel that violated this convention. However, the panels that come with XAML do respect hard-coded height and width.
[12] The property system in XAML has a feature called coercion, which constrains property values. In this case, MaxWidth is constrained to be larger than MinWidth.

Within that range, an element will be sized based on other factors, including the space available within the element’s container. If the element is a content control, then the size will also be dependent on the desired size of the content. I’ll have more to say on that a bit further down.

Applying a Margin

For the purposes of fitting an element within its container, the element’s `Margin` setting must also be taken into account. The container will have a certain size available for the element; that size must then be reduced by any `Margin` on the element. If an element has a `Margin`, room is always made for it, even if that means truncating the element.

Second Step – Arranging the Child Elements in a Panel

After the measurement phase, an element has either an explicit size or a desired size, and possibly a `Margin` that must be included. Using that information, the parent panel then arranges each child element.

There are two sub-phases to arrangement. First, the panel decides how much space it can allocate to a child element. Next, it positions the child element within the allocated space, with truncation or resizing of the element if necessary.

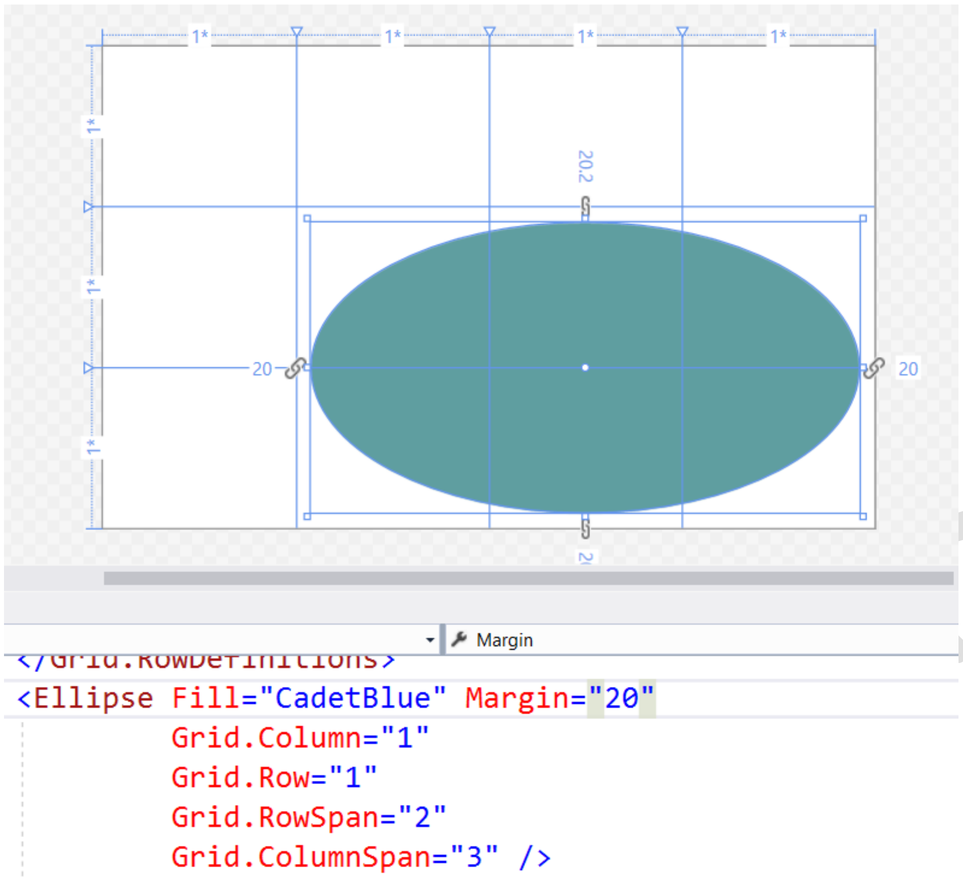
The total number of combinations of type of panel, alignment settings, and available space is quite large. I will focus on common cases. While the resulting explanation isn’t complete, it should be enough for you to understand what happens for most of the situations you are likely to encounter as you begin developing XAML interfaces.

Each panel has a different arrangement process, so it’s necessary to look at the arrangement phase one panel at a time. I’ll start with the Grid, since it’s the panel you’ll probably use the most, and has a fairly straight-forward arrangement process.

To illustrate arrangement, I’ll use a Button with a large FontSize. I’ll explicitly declare the VerticalAlignment in every case, even when it is the default of Center. You will be able to see the same basic element used in various circumstances, so that you can see how it varies in size and position after arrangement in a panel.

Arranging children in a Grid

A `Grid` allocates space for an element based on element’s `Grid.Row`, `Grid.Column`, `Grid.Rowspan`, and `Grid.Columnspan` properties. The allocated space is a rectangular block of cells. Figure xx shows this graphically.



```
<Ellipse Fill="CadetBlue" Margin="20"
         Grid.Column="1"
         Grid.Row="1"
         Grid.RowSpan="2"
         Grid.ColumnSpan="3" />
```

Figure xx – XAML for an ellipse with settings for attached properties in a Grid with four columns and three rows, and the visual results in the Visual Studio designer. The allocated area for the ellipse is a rectangular block of cells in the lower right of the Grid. The ellipse plus its margin will, by default, spread out over the entire allocated area.

Once the Grid knows the cell or block of cells that make up the allocated space, the Grid must place the element within that space. How it does that depends on three main factors:

- Whether the allocated space is too big, just right, or too small for the child element's desired size
- The child element's settings for HorizontalAlignment and VerticalAlignment
- Whether the child element is automatically sized or has a specific size

Each dimension is done independently. That is, an element may be clipped in width because it's too wide, but not clipped in height because it's not tall enough to need it.

It's easiest to understand the possibilities if you see them visually. I have created three figures that show typical cases, showing the results for every combination of HorizontalAlignment and VerticalAlignment:

Figure xx shows what happens when the allocated space is too big, but the element is automatically sized.

Figure xx shows what happens when the allocated space is too big, but the element has a defined size.

Figure yy shows what happens when the allocated space is too small. In this case, it doesn't matter whether the size is automatic or not.

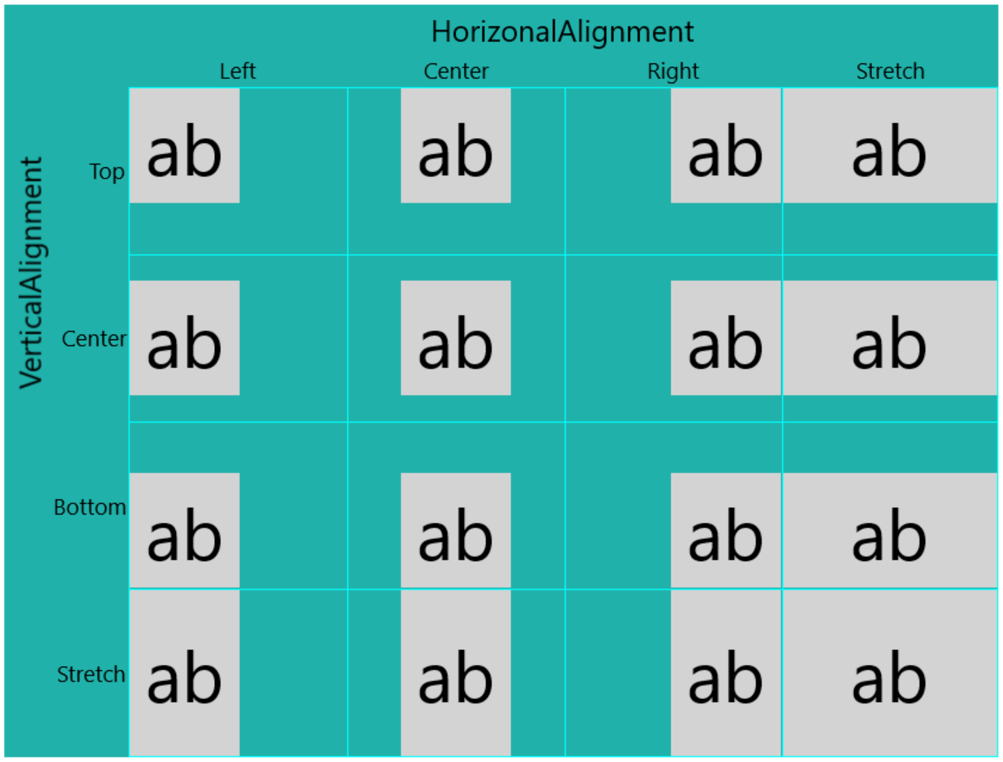


Figure xx - The arrangement results for various alignments when the available room for a button is bigger than the button needs and the button has a Height and Width of Auto.

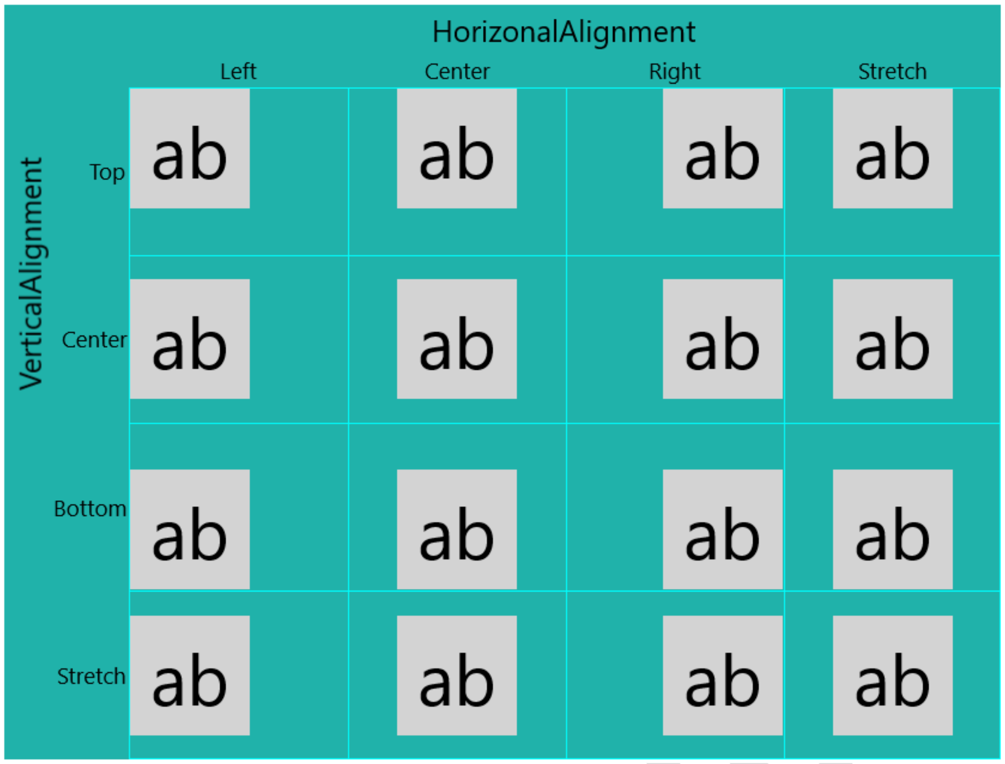


Figure xx - The arrangement results for various alignments when the available room for a button is bigger than the button needs and the button has a Height and Width both set to exactly 80. The results vary from Figure xx the most for one or both alignments set to Stretch.

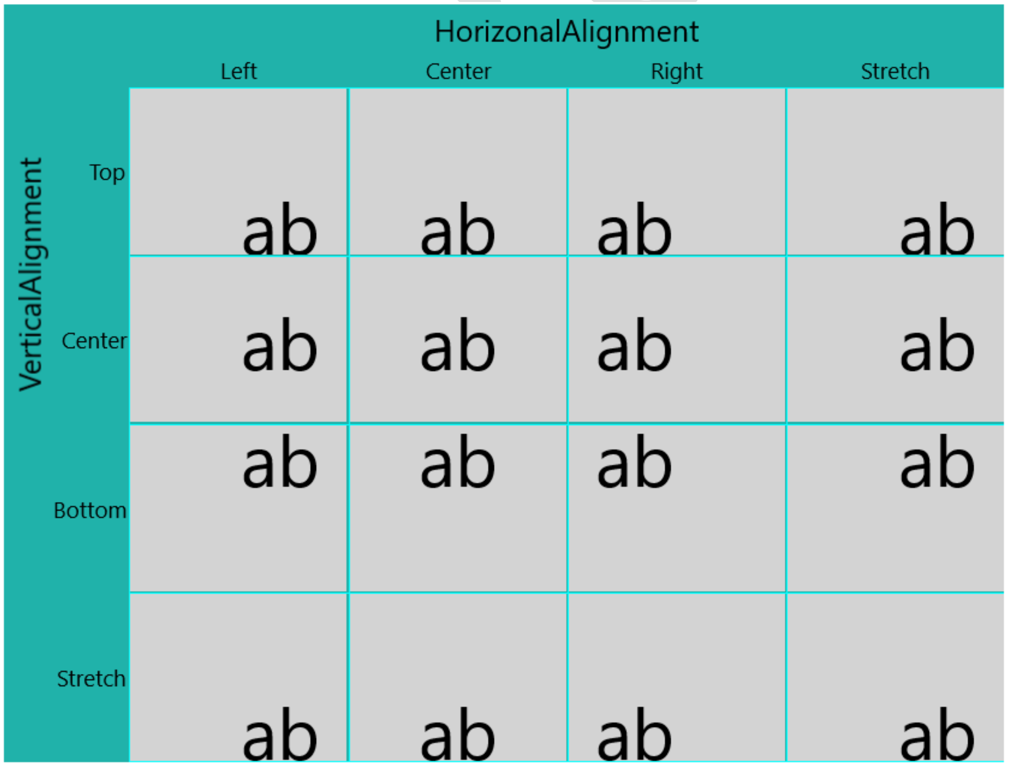


Figure xx - The arrangement results for various alignments when the available room for a button is smaller than the button needs because each button has a Height of 180 and a Width of 200. Each button fills up its cell, but the part of the button that is truncated varies depending on the alignment settings.

The arrangement we've just discussed used rows and columns that were proportionally sized. Remember that rows and columns have the option of being automatically sized to their largest element.

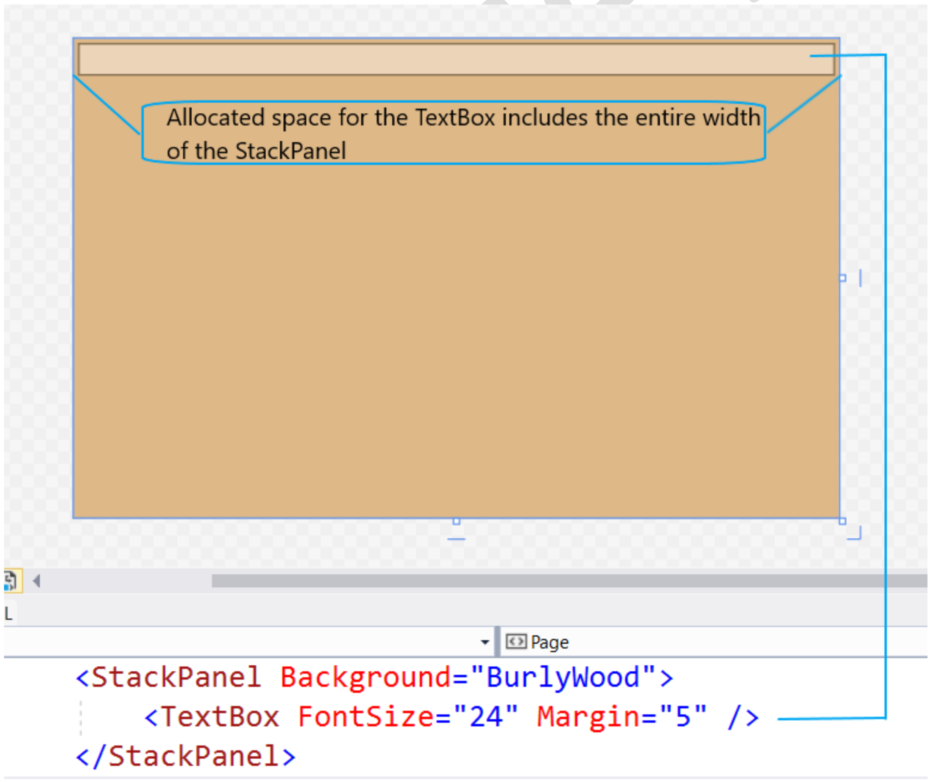
However, once that size is decided by the largest elements, smaller elements in the column or row are still sized and arranged depending on their alignment settings and sizes, similar to the results in Figures xx, xx, and xx. So an element in a column that is not as wide as the widest element in the column, but which has a HorizontalAlignment of Stretch, will still stretch to the width of the column.

Just as a reminder, you can put multiple elements in a Grid cell, and they will layer on top of one another. Each element in such a layered stack is sized and arranged independently, based on its own settings.

Arranging Children in a StackPanel

A StackPanel has somewhat simpler arrangement of children, but it also has some behavior you might not find intuitive at first. Because it stacks elements vertically or horizontally, and does not layer elements as a Grid does, it has to approach HorizontalAlignment and VerticalAlignment differently than a Grid.

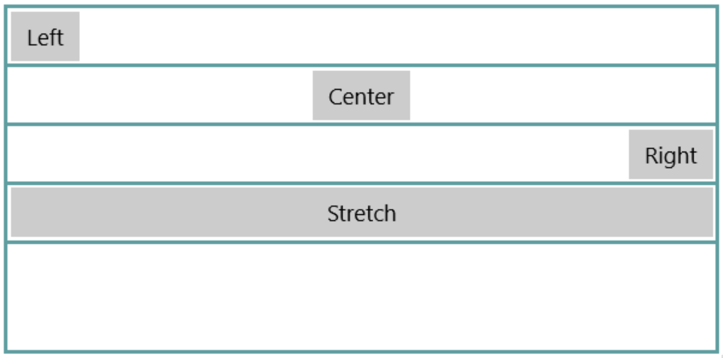
By default, a vertical StackPanel will allocate width for a child element up to the width of the StackPanel itself.


Allocated space for the TextBox includes the entire width of the StackPanel
Page

```
<StackPanel Background="BurlyWood">
    <TextBox FontSize="24" Margin="5" />
</StackPanel>
```

Figure xx – A TextBox in a StackPanel. The StackPanel allocates as much height as the TextBox requests, but the allocated width is the width of the StackPanel itself.

HorizontalAlignment is used in a vertical StackPanel the same way it's used in a Grid. If the allocated width is more than the element needs, the element is positioned or stretched based on HorizontalAlignment. Figure xx shows examples of each HorizontalAlignment setting, using a Button with each setting.



Buttons in a StackPanel with various HorizontalAlignment settings

Figure xx – Four buttons in a vertical StackPanel, each with a different HorizontalAlignment setting. Cosmetic elements have been added to outline the StackPanel and separate the Buttons.

Thus, HorizontalAlignment works as you would expect. But what if we change the VerticalAlignment settings for those buttons? You might naively think that changing VerticalAlignment for the top button to Bottom would move the button to the end of the stack. It does not. The StackPanel does not change the stacking order based on VerticalAlignment.

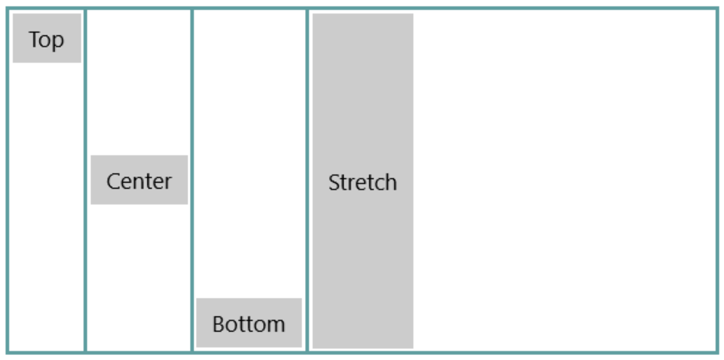
In fact, the StackPanel ignores VerticalAlignment completely! For a vertical stack, it doesn't make sense to do anything with a VerticalAlignment setting. So you can change VerticalAlignment for an element in a vertical stack to any of its permitted values, and you will see no change in the visual result at all.

This can be disconcerting to developers who have experience with some older technologies. For most older, bit-mapped technologies, if you changed a property setting, you would see a visual change. But in XAML, if circumstance make a property irrelevant, its value is just ignored. You might change a property value and see no change whatsoever if that property simply doesn't apply to the current situation.

A vertical StackPanel places no limits on allocation of height. It will allocate as much height as the element requests in its desired size, up to the amount of vertical space available in the StackPanel. The allocated space can be as much as the entire height of the StackPanel, minus any space already allocated to other elements placed in the stack. If an element takes up all the space in the StackPanel, then any later elements have no allocated space at all, and won't appear.

When using a vertical StackPanel, you have the responsibility for making sure that either the elements placed in the stack don't demand so much space that they hide other elements, or that scrolling capability is available. I'll take up scrolling, using a control named ScrollViewer, later in the chapter.

If a StackPanel has Orientation set to Horizontal, then the discussion above is the same except that the dimensions are switched, and the roles of HorizontalAlignment and VerticalAlignment are switched. Here is an equivalent of Figure xx for a horizontal StackPanel with various settings for VerticalAlignment.



Buttons in a horizontal StackPanel with various VerticalAlignment settings

Figure xx – A StackPanel with Orientation="Horizontal" and with a horizontal stack of buttons, each with a different VerticalAlignment.

Arrangement by a RelativePanel

Of all the panels, RelativePanel has the most complex arrangement phase. That's because some elements are not positioned individually; they are positioned relative to other elements.

The number of combinations of relative positioning makes it impractical for me to summarize all the possibilities for arrangement in a RelativePanel, but a good summary is that usually one or two "base" elements are positioned without respect to other elements, and then other elements are arranged with positioning relative to those "base" elements.

There is often a chain effect, in which there some elements are positioned relative to an element that itself is positioned relative to another element. A change in the positioning for one element can alter the arrangement for several elements.

Arrangement by a Canvas

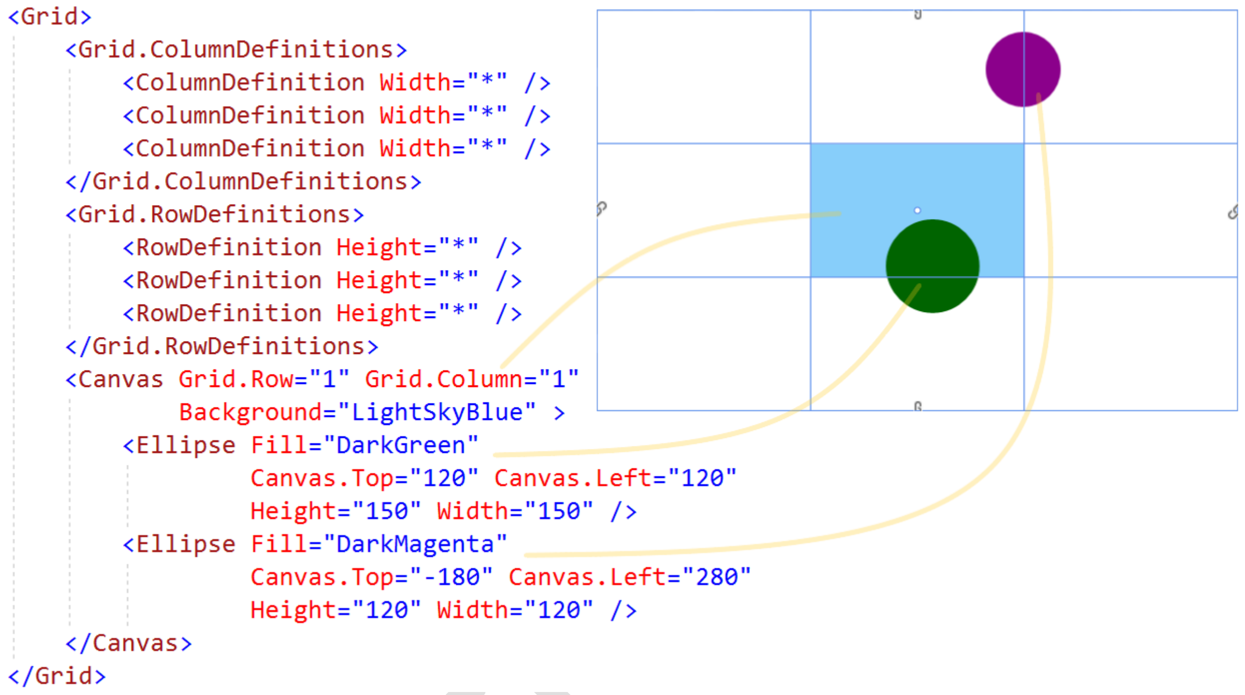
The Canvas panel is the odd man out for the panels included with XAML. It does not do the complex arrangement process outlined above. Canvas just gives an element its explicit or desired size, based on the measurement phase, and places it at the position indicated by Canvas.Top and Canvas.Left. If that results in overlapping elements, so be it. Canvas also completely ignores the element's settings for `VerticalAlignment` and `HorizontalAlignment`.

This leaves you with total control over the sizing and arrangement. You are probably tempted to use that total control[13], and depend on a Canvas to get some precise arrangement that you want. You must resist that temptation! Remember, your app will probably need to run in various sizes, resolutions, and aspect ratios. You need the dynamic responsiveness of the other panels. Suck it up, and learn how to use them, or you will never truly be able to think in XAML.

[13] You are probably especially tempted if your experience has been mostly in classic Visual Basic or in Windows Forms.

There is one aspect of arrangement on a Canvas that is unexpected, and sometimes useful. The Canvas does not clip its children. If they want to hang off the edge of the Canvas, or even render outside the Canvas completely, the Canvas is perfectly fine with that. Other panels or the window for the app might clip those children, but the Canvas won't do the clipping.

Figure xx shows an example, including the XAML that places a Canvas in a Grid cell and two Ellipses as children of the Canvas. One Ellipse hangs over the edge of the Canvas, and the other renders completely outside the Canvas.



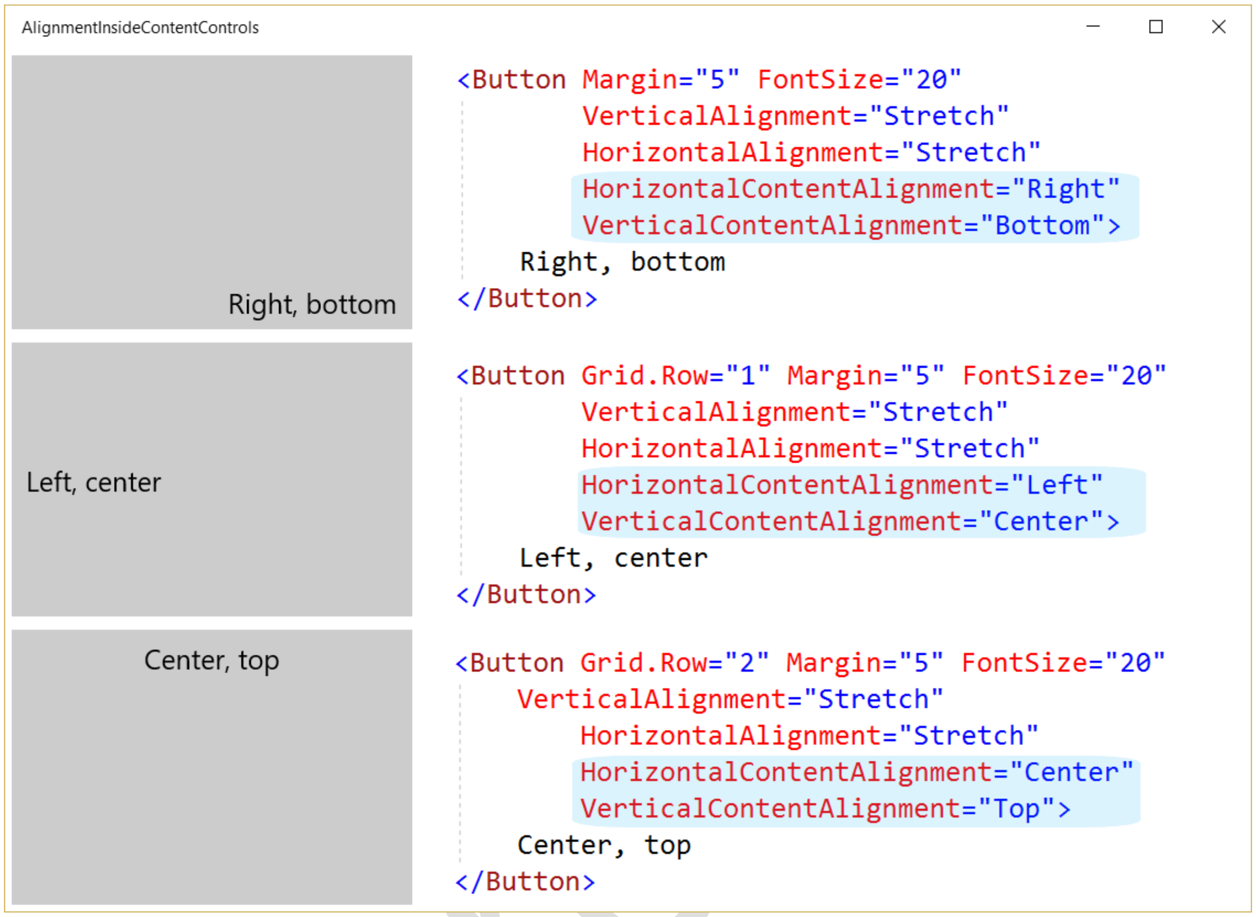
```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Canvas Grid.Row="1" Grid.Column="1"
            Background="LightSkyBlue" >
        <Ellipse Fill="DarkGreen"
                 Canvas.Top="120" Canvas.Left="120"
                 Height="150" Width="150" />
        <Ellipse Fill="DarkMagenta"
                 Canvas.Top="-180" Canvas.Left="280"
                 Height="120" Width="120" />
    </Canvas>
</Grid>
```

Figure xx – A Canvas in a Grid cell, with two children that are not clipped by the Canvas. The Grid is shown in design view so that you can see the cells of the Grid.

Figure xx shows that a Canvas can be used as a coordinate system, with both positive and negative coordinates, and with the top left corner of the Canvas as the origin. That is true even if the Canvas has no size and is itself invisible. Children of the Canvas still render as long as some other container isn't clipping them.

Arrangement inside ContentControls

ContentControls are also containers, although for just one item. Their arrangement is usually simple, though it varies depending on the control template for the control, and depending on the settings for HorizontalContentAlignment and VerticalContentAlignment. These take the same settings as HorizontalAlignment and VerticalAlignment. However, instead of a child element telling a parent what alignment to use, these settings are on the parent ContentControl, and determine the alignment of the content inside. Figure xx shows content in various alignments inside a button.



```
<Button Margin="5" FontSize="20"
        VerticalAlignment="Stretch"
        HorizontalAlignment="Stretch"
        HorizontalContentAlignment="Right"
        VerticalContentAlignment="Bottom">
    Right, bottom
</Button>

<Button Grid.Row="1" Margin="5" FontSize="20"
        VerticalAlignment="Stretch"
        HorizontalAlignment="Stretch"
        HorizontalContentAlignment="Left"
        VerticalContentAlignment="Center">
    Left, center
</Button>

<Button Grid.Row="2" Margin="5" FontSize="20"
    VerticalAlignment="Stretch"
        HorizontalAlignment="Stretch"
        HorizontalContentAlignment="Center"
        VerticalContentAlignment="Top">
    Center, top
</Button>
```

Figure xx – Three buttons with different settings for content alignment, along with the XAML for each button. The settings in the XAML for content alignment are highlighted.

Content controls attempt, within the constraints of their maximum and minimum sizes, to size themselves to their content. During layout, their content goes through the usual measurement process internally, and then tells the content control a desired size. The content control takes that desired size into account when deciding upon its own desired size.

However, if content requests a size for a content control that is bigger than the space allocated to the content control by a panel, the content will be truncated inside the content control.

More about Content in XAML

In Chapter 1, I introduced you to the concept of *content* in XAML. To briefly recap, most controls in XAML have a property named `Content`. It is of type `Object`. You can set the `Content` property to any .NET object, and XAML will attempt to render it. In Chapter 1, you saw a basic example of setting the Content property.

In Figure 2-1 early in this chapter, I showed you a partial class hierarchy of XAML classes. In the lower left of that diagram is the `ContentControl` class, together with a few of the classes that descend from it. As mentioned in Chapter 1, controls that descend from `ContentControl` and thereby have a `Content` property are sometimes referred to as *content controls*.

Understanding what you can do with content in a control is the key to a lot of XAML's functionality. Let's start by answering an obvious question. If content can be any object, how does XAML decide how to visually represent objects placed in the `Content` property?

How XAML Renders Content

Even though the `Content` property can be set to any object, from the standpoint of rendering content, objects only fall into two categories:

- objects that descend from the `FrameworkElement` class
- objects that don't

For general objects that do not descend from `FrameworkElement`, their rendering is simple. XAML calls the `ToString` method of the object, and uses the result as string-based content. XAML automatically creates a `TextBlock` to hold and render the string.

That's the type of rendering used in of the string content examples in Chapter 1. Since `String` is just another .NET type, XAML can call the ToString method (which just returns the string!) and places that in the `TextBlock` created for the purpose.

For objects that do descend from `FrameworkElement`, XAML already knows how to render them. They all have rendering behavior built in. So the ContentControl just passes off responsibility for rendering the content to the rendering engine.

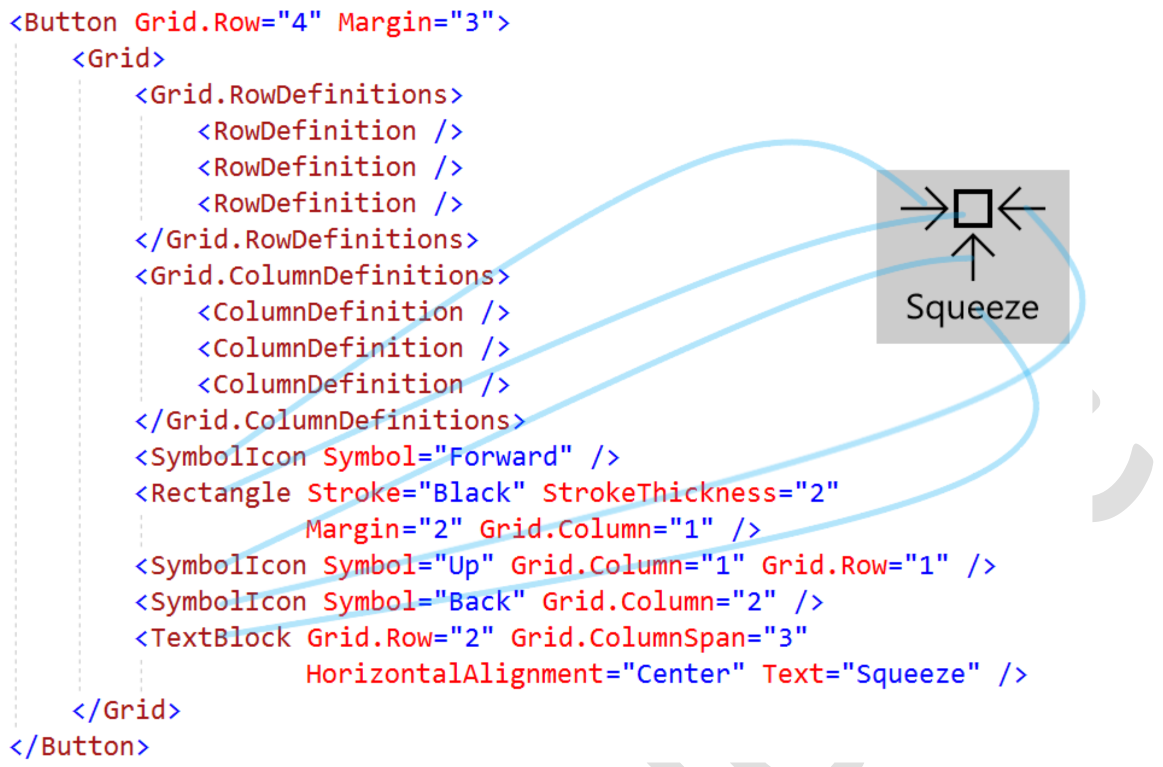
Complex Content

One of the examples in Chapter 1 used a SymbolIcon as content of a Button. This allows you to easily get something graphical in the Button instead of being forced to use boring text all the time. However, you can get a lot more flexibility by setting the `Content` property to a layout panel or other container. Then additional elements can be placed inside the container.

This is huge. If the Content property of a content control is set to a panel, then that panel can have as many children as you like. A content control such as a Button can leverage the entire layout system of XAML to make the visual appearance of the content as complex as you wish.[14]

Figure xx shows an example of a Button with content, and the resulting visual representation of the Button. The content consists of a Grid with five child elements to give the Button an evocative appearance for its function.

[14] It's interesting to contrast the approach for button content in Windows Forms vs. the approach in XAML. In Windows Forms, the "text and graphic in a button" scenario was never handled very effectively. Having flexible layout in a Windows Forms Button would require the Button to contain an entire layout engine inside itself. In XAML, however, the Button delegates the function of layout to other elements that already know how to do it. In a sense, a Button in XAML *does* have a layout engine inside it; that layout engine just happens to be the same layout engine used for all layout in XAML.



```
<Button Grid.Row="4" Margin="3">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <SymbolIcon Symbol="Forward" />
        <Rectangle Stroke="Black" StrokeThickness="2"
                   Margin="2" Grid.Column="1" />
        <SymbolIcon Symbol="Up" Grid.Column="1" Grid.Row="1" />
        <SymbolIcon Symbol="Back" Grid.Column="2" />
        <TextBlock Grid.Row="2" Grid.ColumnSpan="3"
                   HorizontalAlignment="Center" Text="Squeeze" />
    </Grid>
</Button>
```

Figure 2-10. A Button with a Grid inside is shown on the right. There are several elements in the Grid that combine to give the Button an interesting and evocative visual representation. The XAML for the Button is on the left, with connectors from the elements in XAML to their visual renderings.

Other Content Controls

XAML for the Uno Platform includes over a dozen controls that inherit from ContentControl. All of them have the capability to display complex content. Here is a list of the most important ones for routine development:

- ScrollViewer
- UserControl
- ToolTip
- Popup
- CheckBox
- RadioButton

The most lightweight control for content is the ContentControl base class. As we'll see in some later examples, it's a useful container in its own right.

An illustration of XAML composition and separation of concerns

ScrollViewer is an interesting example of a content control because it illustrates the separation of concerns built into XAML. I mentioned earlier that a StackPanel will truncate a stack of items if it runs out of room. You might think the StackPanel would have some kind of built-in scrolling capability for that case. It does not. StackPanel only knows how to stack items; it leaves any other functionality to other elements.

ScrollViewer provides scrolling capability for StackPanel and for anything else that needs scrolling. To scroll the child elements in a StackPanel, you just place the StackPanel inside a ScrollViewer. Here's the XAML:

```
<ScrollViewer>
    <StackPanel>
        <Button HorizontalAlignment="Stretch" FontSize="72" Margin="5">One</Button>
        <Button HorizontalAlignment="Stretch" FontSize="72" Margin="5">Two</Button>
        <Button HorizontalAlignment="Stretch" FontSize="72" Margin="5">Three</Button>
        <Button HorizontalAlignment="Stretch" FontSize="72" Margin="5">Four</Button>
        <Button HorizontalAlignment="Stretch" FontSize="72" Margin="5">Five</Button>
        <Button HorizontalAlignment="Stretch" FontSize="72" Margin="5">Six</Button>
    </StackPanel>
</ScrollViewer>
```

The StackPanel is the content of the ScrollViewer. The ScrollViewer scrolls any content placed in it, no matter how complex. You can see two scrolled positions for this ScrollViewer in Figure xx.

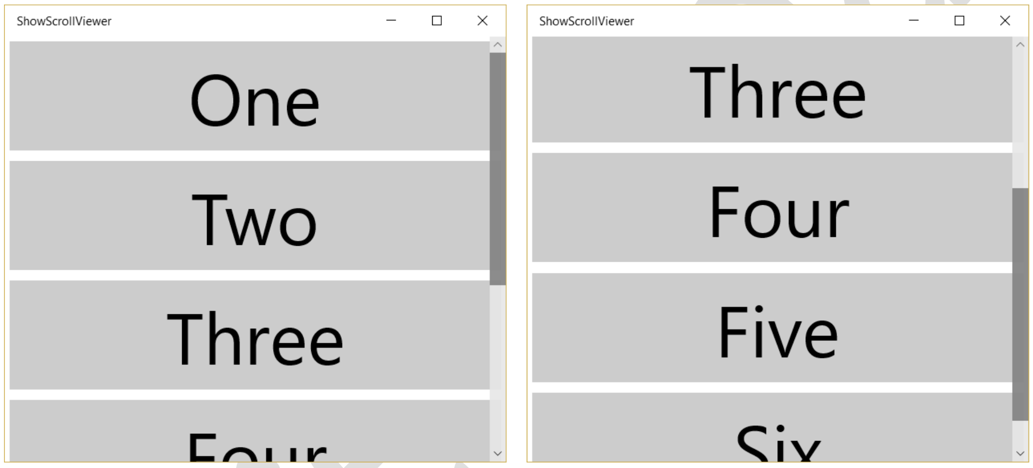


Figure xx – A ScrollViewer can scroll any content. In this case it scrolls a StackPanel that doesn't have enough height to display all its children at once.

By default, ScrollViewer scrolls vertically, and turns on the scroll bar when scrolling is needed. However, ScrollViewer has properties to control horizontal and vertical scrolling. They are named HorizontalScrollBarVisibility and VerticalScrollBarVisibility. Both have the following permitted values:

Auto – The scrollbar shows only when scrolling is available

Disabled – The scrollbar is shown but disabled, even if scrolling would be possible

Hidden – The scrollbar does not show at all

Visible – The scrollbar is always visible, but is only enabled when scrolling is available

We will see how these settings are used to get specific capabilities, such as automatic word wrapping, in some examples in later chapters.

You can have as many ScrollViewer controls as you need on your page. If you place a ScrollViewer inside a Grid cell, for example, it will scroll its content inside that Grid cell. You can thereby implement scrolling exactly where you need it, from scrolling an entire page to scrolling just a small Grid cell. Figure xx has both scenarios in the same page. Here is the XAML for that page:

```
<Page
    {namespace lines omitted}>

    <ScrollViewer FontSize="20">
        <Grid Height="900">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <Frame Background="LightSalmon"
                   VerticalContentAlignment="Center">
                I'm in the top left
            </Frame>
            <Frame Background="LightSteelBlue" Grid.Row="2" Grid.Column="2"
                   VerticalContentAlignment="Center">
                I'm in the bottom right
            </Frame>
            <ScrollViewer Grid.Row="1" Grid.Column="1">
                <StackPanel>
                    <Button HorizontalAlignment="Stretch"
                            FontSize="72" Margin="5">One</Button>
                    <Button HorizontalAlignment="Stretch"
                            FontSize="72" Margin="5">Two</Button>
                    <Button HorizontalAlignment="Stretch"
                            FontSize="72" Margin="5">Three</Button>
                    <Button HorizontalAlignment="Stretch"
                            FontSize="72" Margin="5">Four</Button>
                    <Button HorizontalAlignment="Stretch"
                            FontSize="72" Margin="5">Five</Button>
                    <Button HorizontalAlignment="Stretch"
                            FontSize="72" Margin="5">Six</Button>
                </StackPanel>
            </ScrollViewer>
        </Grid>
    </ScrollViewer>
</Page>
```

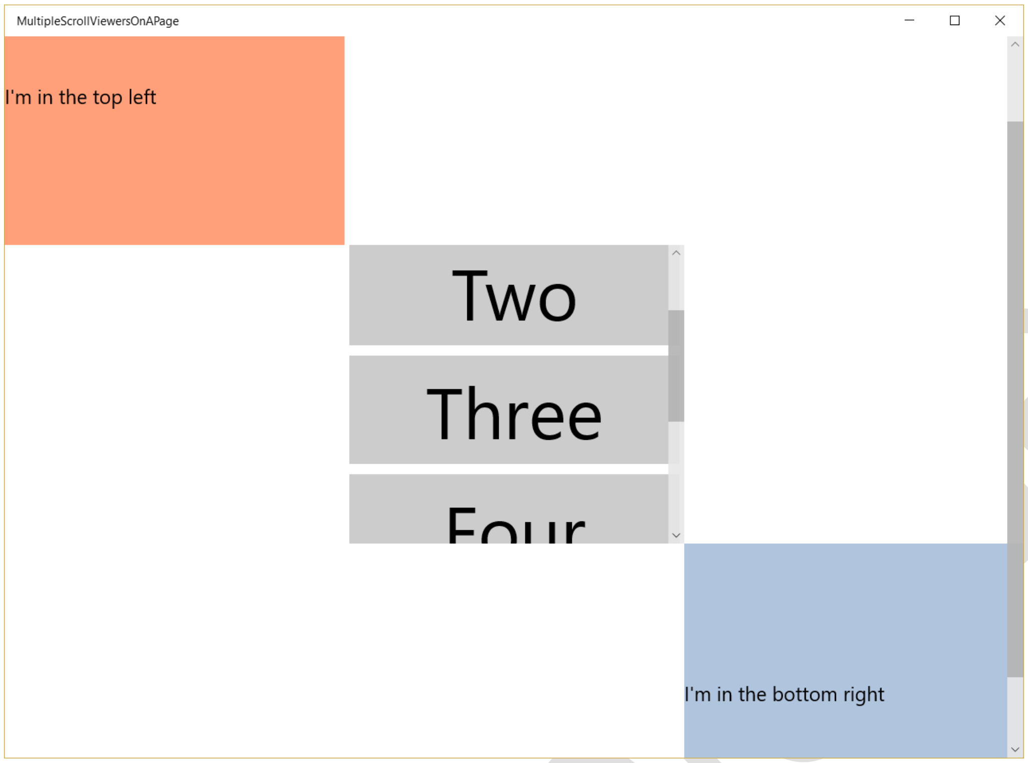


Figure xx – A page with a ScrollViewer that scrolls the whole page, and another ScrollViewer in the middle Grid cell that scrolls a StackPanel of buttons.

Using data binding to tie elements together

In chapter one, we discussed simple data binding. Data objects can be the source of values that are injected into visual elements through a binding. For example, if your program has a Book object with a Title property, then you can bind and use that Title value in a TextBox with this line of XAML:

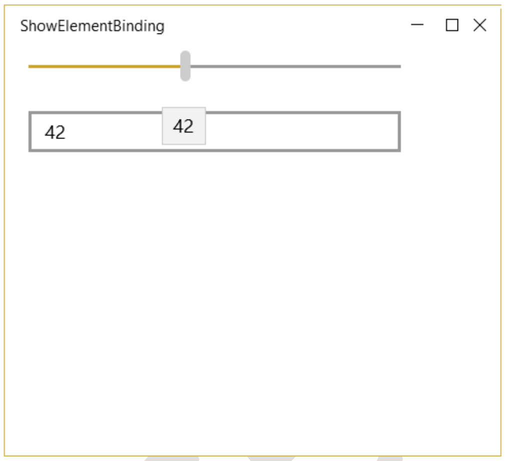
```
<TextBox Text="{Binding Title}" Name="TitleTextBox" />
```

This line would set the Text property of the TextBox to the Title property of the Book object if the Book object is available through the DataContext property of the TextBox.[15]

Data bindings are not required to get their values from data objects obtained by DataContext. Instead, the source object can be another element in the same page of XAML. The only change to the binding is setting the name of the element to be used as the source, using the ElementName property of the Binding. Figure xx shows XAML that puts the value of a Slider control into a TextBox, along with a screen capture of the Slider in action and the resulting value placed in the TextBox.

[15] Remember that DataContext is an inherited property, so it can be set on a container such as a Page or Grid, and is then available as the property value of DataContext for any elements inside the container.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Slider Margin="5" MaxHeight="100" Minimum="0"
            Value="20" Name="TopSlider" />
    <TextBox Margin="5" Grid.Row="1"
        Text="{Binding Value,ElementName=TopSlider}" />
</Grid>
```



}

Figure xx – A data binding from the Text property of a TextBox to the value of a Slider control. As the user moves the thumb on the Slider, the value in the TextBox automatically changes.

You can bind almost all of the properties of visual elements this way. For example, the Opacity of an element ranges from zero to one, with zero as completely transparent and one as completely opaque. Extending the XAML above, an ellipse can have its Opacity dynamically changed with the Slider control. Here is the extended version of the XAML, with a reference Ellipse that has normal Opacity.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Slider Margin="5" Maximum="100" Minimum="0"
            Value="20" Name="TopSlider" />
    <TextBox Margin="5" Grid.Row="1"
        Text="{Binding Value,ElementName=TopSlider}" />
    <Ellipse Height="50" Fill="DarkBlue"
             Grid.Row="2" />
    <Ellipse Height="50" Fill="DarkBlue"
             Grid.Row="3"
             Opacity="{Binding Value, ElementName=BottomSlider}"/>
    <Slider Margin="5" Maximum="1.0" Minimum="0.0" Grid.Row="4"
            StepFrequency=".01"
            Value=".4" Name="BottomSlider" />
</Grid>
```

Figure xx shows a screen capture, with both sliders having been moved from their default positions.

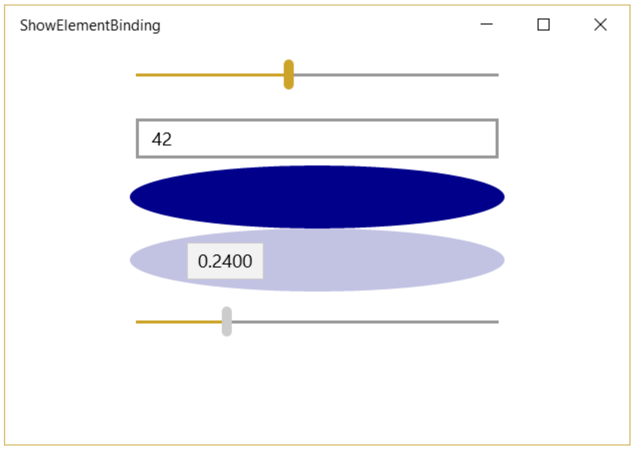


Figure xx – In the extended example, the TextBox is still getting its Text from the top Slider's Value property, and the second Ellipse is getting its Opacity from the bottom Slider's Value. The dark blue Ellipse is for reference comparison to the Ellipse with variable Opacity.

In the chapter on working with data (chapter 4), you will see many other capabilities of data binding. However, I thought it important for you to realize that data binding is one of the ways the pieces can be connected in a page of XAML. I also need this type of binding for an example I include at the end of the chapter.

You may think this capability to bind elements together with data binding is rather trivial, and would have no use in typical business applications. Open your mind, Padawan. Together with the layering and composition techniques discussed above, it is one of the doorways to compelling XAML applications.

An example of layering and composition

To see these capabilities come together, let's do a concrete example of data visualization. It is inspired by a real project, and particularly shows the power of layering elements in a single cell Grid.

The application scenario is to show a cross section of a fuel tank, with the level of the fuel as part of the cross section. Figure xx shows a screen capture of the final result, with a Slider used to set the level in the tank.[16]

[16] This reference example is a fixed size for simplicity. In the actual project, the fuel tank was completely resizable. However, that requires a fair amount of additional XAML and code. In this simplified example, we're primarily interested in the power of layering and composition.

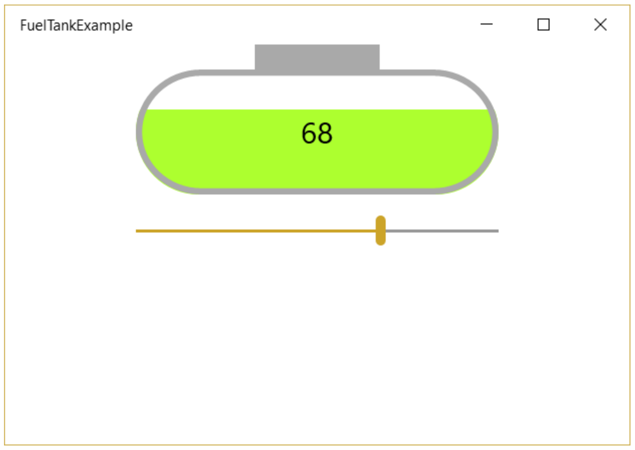


Figure xx – A data visualization in the form of a fuel tank. The level in the tank is controlled by the Slider, though of course in a real application the level in the tank would come from some data source. The tank part of the example is done with two Grid panels, two rounded Rectangle shapes plus a Rectangle for the block on top, and a TextBlock.

The XAML needed to get the interactive fuel tank in Figure xx is surprisingly concise. There's an outer Grid, an inner Grid used for sizing the fuel level, three rectangles, and a TextBlock. Here is the XAML listing:

```
 <Grid Width="300">
    <Grid.RowDefinitions>
        <RowDefinition Height="125" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Slider Name="FuelLevelSlider"
            Maximum="100"
            Minimum="0" Grid.Row="1"
            Value="40" Margin="5" />
    <Grid Margin="5,20,5,5" VerticalAlignment="Bottom"
          Height="{Binding Value, ElementName=FuelLevelSlider}">
        <Rectangle Height="100" RadiusX="50"
                   RadiusY="50" Fill="GreenYellow"
                   VerticalAlignment="Bottom"/>
    </Grid>
    <Rectangle Height="100" RadiusX="50" RadiusY="50"
               Margin="5,20,5,5" VerticalAlignment="Bottom"
               Stroke="DarkGray" StrokeThickness="5" />
    <Rectangle VerticalAlignment="Top" Height="20"
               Width="100" Fill="DarkGray" />
    <TextBlock Text="{Binding Value, ElementName=FuelLevelSlider}"
               Margin="5,20,5,5" VerticalAlignment="Center"
               HorizontalAlignment="Center" FontSize="24" />
</Grid>
```

Let's walk through this listing and see what each element in the XAML does.

1. The outer Grid has two rows. It holds a Slider in the first row, and the "fuel tank" in the cell of the second row.
2. The Slider has a maximum value of 100 and a minimum value of 0. The fuel level will be tied to the value of this Slider, which is named FuelLevelSlider.

3. The inner Grid holds a rounded Rectangle that shows the full fuel level. This Grid clips that Rectangle so that only the bottom part of the full fuel level is shown. The height of the Grid has a binding to the Slider's value, so this height varies as the Slider is changed.
4. The Rectangle inside the inner grid shows the full fuel level in a greenish color. It has VerticalAlignment to the bottom so that when it is clipped, only the bottom part will show. The RadiusX and RadiusY values give the Rectangle the rounded capsule shape.
5. The other Rectangle paints the outline of the tank. It is the same size as the "fuel level" Rectangle, and has the same rounding, but since it's not in the same Grid, it doesn't get clipped. It sits on top of the "fuel level" Rectangle, and has an outline but no Fill brush set. That allows the "fuel level" rectangle to show through inside this tank outline rectangle.
6. The TextBlock sits on top of everything else because it's the last element in the Children collection. It is centered. It gets its Text from the Value in FuelLevelSlider.

This fuel tank example illustrates thinking in XAML in several ways. Besides the layering and composition, which are based on principles discussed earlier in this chapter, the fuel tank is an example of how compelling XAML interfaces are usually designed. The fuel tank graphic was originally conceived with paper and pencil. Because XAML is so flexible and powerful, you can find a way to construct just about anything you can sketch.

I recommend that you *do not* design your interfaces while writing XAML in Visual Studio. You'll probably never come up with something like the fuel tank if you're trying to design while writing XAML. You'll more likely do the same old conventional stuff you have been doing in the past.

Instead, you should sketch your designs with paper and pencil, and refine them a few rounds before you begin to write the implementation in XAML. Some of your designs will likely be a challenge to create in XAML, but that's an excellent way to stretch yourself to become more fluent.

Wrap up

The compositional nature of XAML gives it extraordinary flexibility. XAML contains a set of pieces, each with narrow responsibilities, but it offers a structure in which those pieces can be combined in uncountable ways.

This gives degrees of freedom you simply don't have in other UI technologies. Once you master the art of composing user interfaces in XAML, you will be able to create applications that are far more productive, easier to understand, and which present data to users in meaningful ways.

This process will take time, however. With practice, the intuitive part of your mind will begin to realize interesting combinations that solve challenges you are facing in your development. When you begin to have these "shower thoughts" about how XAML can be used, you'll know you have made a major step towards learning to think in XAML.

Now that you've learned to start thinking about XAML in a compositional way, in the next chapter we'll look at some of the additional elements that make up your toolbox. Each one can be combined with others in interesting ways, using the compositional concepts we've covered in this chapter.