

UWP-025 - Common XAML Controls - Part 2

In this lesson we will add more controls to our repertoire taking the approach from our previous examination of XAML controls whereby I copied and pasted XAML defining the control into the XAML editor, talked about the Control itself, explained how to access the features and functions of that Control and so on.

I'll begin by creating a new project named ControlsExample2, then in the MainPage.xaml I add Row and Column Definitions to create a structure for our MainPage.

```
10 <Grid Margin="20,20,0,0">
11   <Grid.RowDefinitions>
12     <RowDefinition Height="Auto" />
13     <RowDefinition Height="Auto" />
14     <RowDefinition Height="Auto" />
15     <RowDefinition Height="Auto" />
16     <RowDefinition Height="Auto" />
17     <RowDefinition Height="Auto" />
18     <RowDefinition Height="Auto" />
19     <RowDefinition Height="Auto" />
20     <RowDefinition Height="Auto" />
21     <RowDefinition Height="*" />
22   </Grid.RowDefinitions>
23   <Grid.ColumnDefinitions>
24     <ColumnDefinition Width="Auto" />
25     <ColumnDefinition Width="*" />
26   </Grid.ColumnDefinitions>
27
28 </Grid>
29 </Page>
```

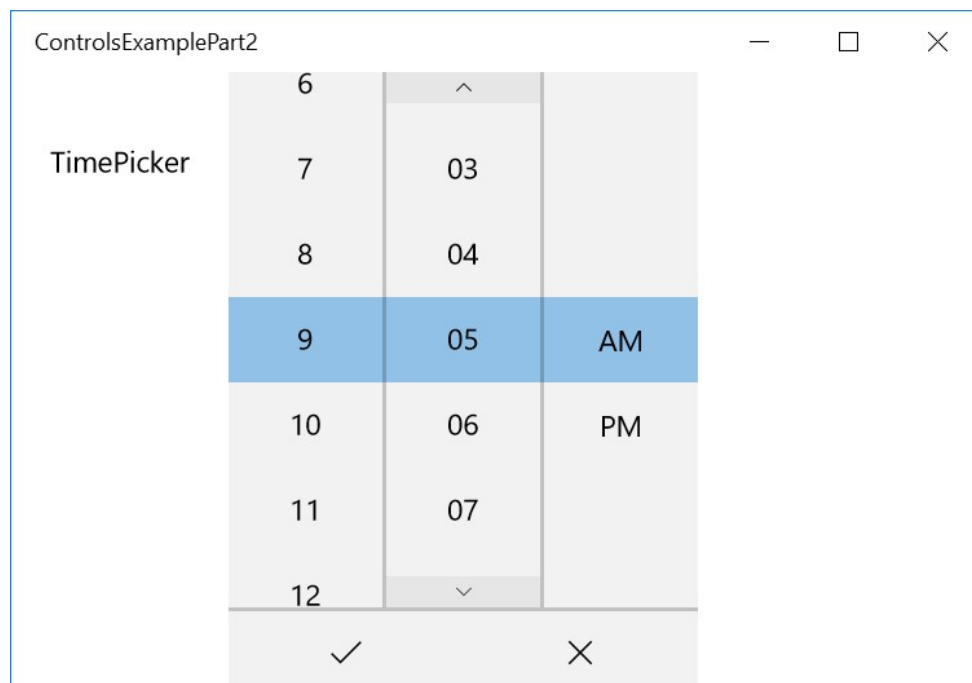
We will start by talking about Controls that allow user input for times and dates. The first Control that I want to call your attention to is the TimePicker. This allows a user to select an hour and a minute as input for your application. Suppose I want to create an app that allows the user to create a reminder at 9:30 PM. The TimePicker control allows your user to easily select that whether with touch or with mouse.

```

28 <TextBlock Grid.Row="0" Text="TimePicker" VerticalAlignment="Center" />
29 <TimePicker Grid.Row="0"
30             Grid.Column="1"
31             ClockIdentifier="12HourClock"
32             Margin="20,0,0,20" />

```

The most interesting property is the `ClockIdentifier` property which allows for two potential values: either a `12HourClock` that will allow the user to select AM/PM, or a `24HourClock`, which in the United States, is known as military time. So I choose the 12-hour clock, and run debug the app:



As I select the Control itself, it shows a nice pop-up that would look good in a mobile device, as well as a desktop. And here I can select different times and choose AM/PM and then click the check mark for acceptance or the X to cancel.

Once I've made my selections and clicked the checkmark, the pop-up selector disappears and the chosen time is displayed.

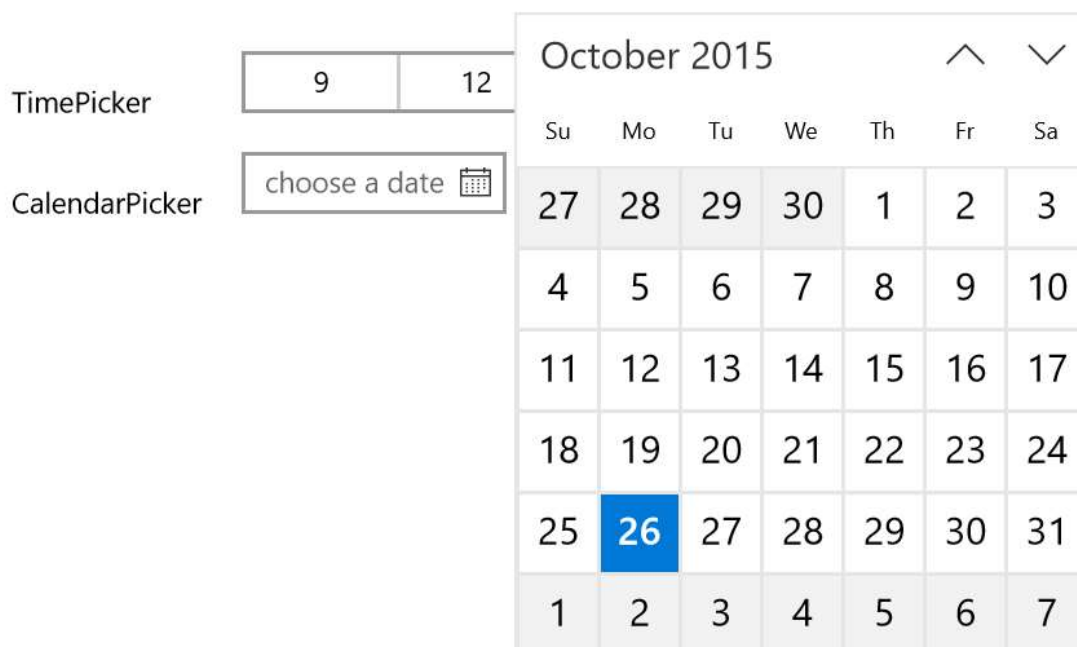
The next two controls relate to dates. The first is called a `CalendarDatePicker`. When you hear the term "Picker", typically you're going to see a fly-out, or rather, a pop-up appear. In other words, you're not going to see the entire body of the Control, you'll just see the default value, but when click on that control it will show you the pop-up with all available selections. In this case, you'll see a Calendar pop up.

```

34 <TextBlock Grid.Row="1"
35           Text="CalendarPicker"
36           VerticalAlignment="Center"/>
37 <CalendarDatePicker
38     Grid.Row="1"
39     Grid.Column="1"
40     Margin="20,0,0,20"
41     PlaceholderText="choose a date" />
42

```

When we run the app you can see the default value displayed. In this case, we set the PlaceholderText attribute. When you tap the control it displays a calendar fly-out:



In the Properties area, you can see that there are a number of interesting Properties like the type of Calendar that we will use. By default, in the United States, it's set to GregorianCalendar. However, you can see that there are many different types of Calendars for various Asian countries. There is a HebrewCalendar and others as well.

The other Calendar Control is the CalendarView. And when you hear the word “view”, typically, in XAML controls, it means that the interactive / selection area will be displayed all of the time instead of hidden in a fly-out.

```

43     <TextBlock Grid.Row="2" Text="CalendarView" VerticalAlignment="Center" />
44     <StackPanel Grid.Row="2"
45         Grid.Column="1"
46         Margin="20,0,0,20"
47         HorizontalAlignment="Left">
48         <CalendarView Name="MyCalendarView"
49             SelectionMode="Multiple"
50             SelectedDatesChanged="MyCalendarView_SelectedDatesChanged" />
51         <TextBlock Name="CalendarViewResultTextBlock" />
52     </StackPanel>
53

```

As you can see, I'm handling the SelectedDatesChanged event. I've also set the SelectionMode to "Multiple". I can retrieve all of the selected dates by retrieving the CalendarView's SelectedDates property. Using a clever LINQ statement I can project out the way I want the date to appear (in this case, as a string in the form Month/Day) and then call ToArray() to work with those selections as an array of strings:

```

34     private void MyCalendarView_SelectedDatesChanged(
35         CalendarView sender,
36         CalendarViewSelectedDatesChangedEventArgs args)
37     {
38         var selectedDates = sender.SelectedDates
39             .Select(p => p.Date.Month.ToString() + "/" + p.Date.Day.ToString())
40             .ToArray();
41
42         var values = string.Join(", ", selectedDates);
43         CalendarViewResultTextBlock.Text = values;
44     }
45

```

To display the values I'll use the string.Join method adding a comma as a separator.

Next we'll talk about the the Flyout Control. The Flyout will display a message box. Inside that message box you can add any XAML markup you want to display in the pop-up. The Flyout is displayed as a result of some trigger. Some controls have a Flyout property which produces the necessary trigger to display the flyout. In the following case, we'll use the Button's Flyout property in property element syntax to define the Flyout and the content inside of it:

```

55 <TextBlock Grid.Row="3" Text="Flyout" VerticalAlignment="Center" />
56 <Button Name="MyFlyoutButton"
57     Margin="20,0,0,20"
58     Grid.Row="3"
59     Grid.Column="1"
60     Content="Flyout">
61     <Button.Flyout>
62         <Flyout x:Name="MyFlyout">
63             <StackPanel Margin="20,20,20,20">
64                 <TextBlock Text="I just flew out to say I love you."
65                     Margin="0,0,0,10" />
66                 <Button Name="InnerFlyoutButton"
67                     HorizontalAlignment="Right"
68                     Content="OK"
69                     Click="InnerFlyoutButton_Click" />
70             </StackPanel>
71         </Flyout>
72     </Button.Flyout>
73 </Button>

```

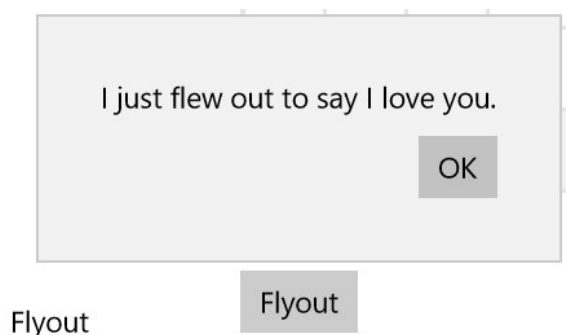
The Flyout can be displayed only using XAML, but you must make provisions for closing the Flyout inside of the Flyout itself. Here we add a Button called InnerFlyoutButton and handle its click event to hide the Flyout:

```

46 private void InnerFlyoutButton_Click(object sender, RoutedEventArgs e)
47 {
48     MyFlyout.Hide();
49 }

```

The result:



Similarly, we'll examine the Menu Flyout control. You've probably seen a Menu Flyout if you've been using Windows 10. The Menu Flyout is useful whenever you are trying to create a contextual menu for a given control.

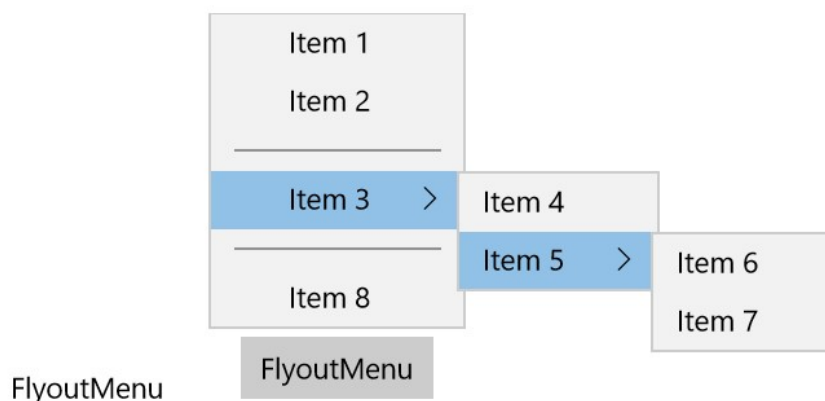
Here again I'll use a Button, however you'll see that many different controls have a Flyout property inside of which we can implement a Menu Flyout.

```

74
75 <TextBlock Grid.Row="4" Text="FlyoutMenu" VerticalAlignment="Center" />
76 <Button Grid.Row="4"
77     Margin="20,0,0,20"
78     Grid.Column="1"
79     Content="FlyoutMenu">
80     <Button.Flyout>
81         <MenuFlyout Placement="Bottom">
82             <MenuFlyoutItem Text="Item 1" />
83             <MenuFlyoutItem Text="Item 2" />
84             <MenuFlyoutSeparator />
85             <MenuFlyoutSubItem Text="Item 3">
86                 <MenuFlyoutItem Text="Item 4" />
87                 <MenuFlyoutSubItem Text="Item 5">
88                     <MenuFlyoutItem Text="Item 6" />
89                     <MenuFlyoutItem Text="Item 7" />
90                 </MenuFlyoutSubItem>
91             </MenuFlyoutSubItem>
92             <MenuFlyoutSeparator />
93             <ToggleMenuFlyoutItem Text="Item 8" />
94         </MenuFlyout>
95     </Button.Flyout>
96 </Button>

```

The MenuFlyout consists of one or more MenuFlyoutItems, MenuFlyoutSubItems, ToggleMenuFlyoutItems or MenuFlyoutSeparators. Each of their purposes will become apparent simply by running and examining their XAML code and the result.



In this case, notice that I set the Placement attribute to "Bottom", however the Menu Flyout is displayed above the button. The XAML layout engine attempts to fill our request, however it is bound by the boundaries of the application's overall width and height and when necessary will move the Menu Flyout to an available area.

This example doesn't handle the Click event, however you would handle it for each menu items as you deem fit.

The ToggleMenuFlyoutItem has an "on" and "off" state perfect for enabling features of your app. You can check the user's select using the IsChecked property:



The next control we'll examine is the AutoSuggestBox which will become very helpful to us whenever we're building real applications that include the search feature like a hamburger-style navigation that we saw previously. This will allow us to utilize any collection or array as a source of potential items that are displayed and filtered as the user types into the text box:

```
101 <TextBlock Grid.Row="5" Text="AutosuggestBox" VerticalAlignment="Center" />
102 <AutoSuggestBox Name="MyAutoSuggestBox"
103     Margin="20,0,0,20"
104     Grid.Row="5"
105     Grid.Column="1"
106     HorizontalAlignment="Left"
107     QueryIcon="Find"
108     PlaceholderText="Find Something"
109     Width="200"
110     TextChanged="MyAutoSuggestBox_TextChanged" />
111
```

To implement this control, you'll need a data source. This could come from a database or some hard-coded list of potential values. In this case I'll create an array of strings containing names called selectionItems. In a more full-featured app, you would retrieve this from a web service, database, flat file, etc.

Inside of the TextChanged event handler I use a clever LINQ statement to retrieve all of the items from the list that start with the letters the user typed in.

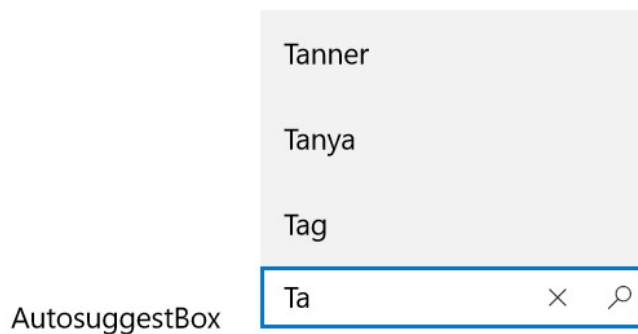
Finally, I set the ItemsSource property of the AutoSuggestBox to the filtered array of strings.


```

51
52     private string[] selectionItems = new string[]
53     { "Ferdinand", "Frank", "Frida", "Nigel", "Tag", "Tanya", "Tanner", "Todd" };
54
55     private void MyAutoSuggestBox_TextChanged(
56         AutoSuggestBox sender,
57         AutoSuggestBoxTextChangedEventArgs args)
58     {
59         var autoSuggestBox = (AutoSuggestBox)sender;
60         var filtered = selectionItems
61             .Where(p => p.StartsWith(autoSuggestBox.Text))
62             .ToArray();
63
64         autoSuggestBox.ItemsSource = filtered;
65     }
66

```

When I run the application I can start to filter all of the possible names by typing a couple of letters. I can then use the keyboard's arrow keys to make the final selection:



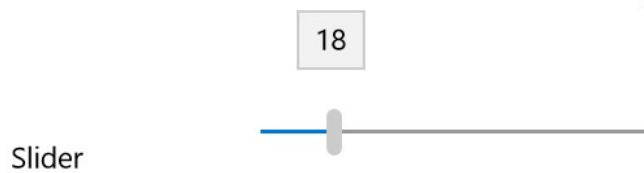
Next, the Slider Control which, again, you'll see often used in Windows 10. The Slider Control will allow the user to make a numerical selection between a certain range by dragging the head of the slider to the right or left.

```

113
114     <TextBlock Text="Slider" Grid.Row="6" VerticalAlignment="Center" />
115     <Slider Name="MySlider"
116         Margin="20,0,0,20"
117         Grid.Row="6"
118         Grid.Column="1"
119         HorizontalAlignment="Left"
120         Maximum="100"
121         Minimum="0"
122         Width="200" />
123

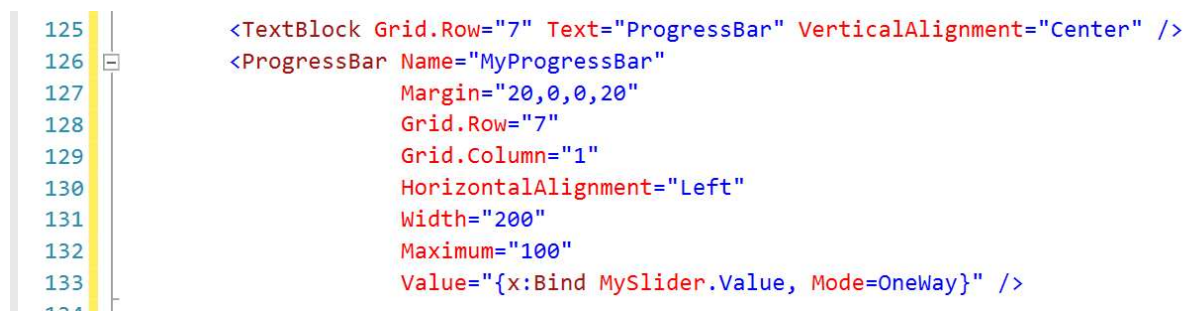
```

Here I set a Minimum and Maximum value. There are other attributes that can affect the "steps" between 0 and 100, and other properties of the Slider.



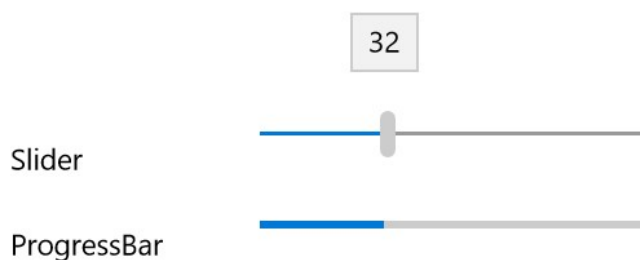
As the user drags the head on the Slider, the Value attribute is changed. You can see the current Value in a small tooltip above the head.

Next, we'll talk about the ProgressBar which provides feedback to the user for long-running operations.



In this example, I am using a special syntax to set the Value. Sure, I could set it in C# or XAML by setting the Value property to something like "57". In this case I wanted to show something a bit more advanced as a segue into the topic of data binding later in this series.

I'm using a special binding syntax which will bind the value of the Progress Bar control to the value of the Slider control.



Admittedly, this isn't a very practical example. Typically you would use the Progress Bar to give the user feedback on some long running operation by setting the Value property, as well as the Minimum and Maximum value to display a percentage complete.

The final XAML control I'll feature is the `ProgressRing` which is another type of progress control, however it doesn't give the user any specific feedback on exactly how far along they are in a long running process. The only feedback they get is that the app is "still working":

```
135 <TextBlock Grid.Row="8" Text="Progress Ring" VerticalAlignment="Center" />
136 <ProgressRing Name="MyProgressRing"
137             Margin="20,0,0,20"
138             Grid.Row="8"
139             Grid.Column="1"
140             HorizontalAlignment="Left"
141             Width="50"
142             Height="50"
143             IsActive="True" />
144
```

By setting `IsActive` to "True" the `ProgressRing` spins displaying the familiar Windows 10 orbiting circles:

Progress Ring



UWP-026 - Working with the ScrollViewer

The ScrollViewer is a Layout allows you to add a scrollable area into your application. This scrollable area could be for the entire viewable area or just for sub-sections. In this lesson I'll demonstrate its use by adding it to specific Cells of a Grid.

I have created a new project called ScrollViewerExample.

First, I'll create four Grid cells:

```
10 <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11   <Grid.RowDefinitions>
12     <RowDefinition Height="*" />
13     <RowDefinition Height="*" />
14   </Grid.RowDefinitions>
15   <Grid.ColumnDefinitions>
16     <ColumnDefinition Width="*" />
17     <ColumnDefinition Width="*" />
18   </Grid.ColumnDefinitions>
19
```

I'll add the ScrollViewer into Row zero, Column zero.

```
20 <ScrollViewer
21   HorizontalScrollBarVisibility="Auto"
22   VerticalScrollBarVisibility="Auto">
23   <Image Source="Assets/Financial.png" Height="800" Stretch="None" />
24 </ScrollViewer>
```

This ScrollViewer contains an Image control with its Source property set to a .png image we've worked with previously. It's a large image with a Height="800" therefore I'm setting its Stretch="None". I've chosen this so that we can see our ScrollViewer at work.



A ScrollView is just that -- a viewable area that can be scrolled. By adding more content inside of the ScrollView than can be viewed without scrolling, we've created a condition where the ScrollView can shine. By setting the HorizontalScrollBarVisibility and VerticalScrollBarVisibility to "Auto" we're allowing the ScrollView to determine whether we need scrollbars across the right or bottom sides (respectively). We could programmatically change those Visibility properties to Hidden, Disabled or Visible as well to force the scrollbars to appear and be functional.

I've demonstrated the primary usage of the ScrollView. However, I do want to demonstrate one of the little "gotcha's" when using the ScrollView: If you enclose a ScrollView with a StackPanel you essentially eliminate the ScrollView from operating. In this example, I add a StackPanel in the second row, first Column. Inside of that a ScrollView I've set the HorizontalScrollBarVisibility and VerticalScrollBarVisibility to "Auto". I am employing the same Image control as the previous example. In fact, everything is same as the previous example -- just enclosed in a StackPanel.

```

26 <!-- Stack panel kills a child ScrollView -->
27 <StackPanel Grid.Row="1" Grid.Column="0">
28     <ScrollView
29         HorizontalScrollBarVisibility="Auto"
30         VerticalScrollBarVisibility="Auto">
31         <Image Source="Assets/Financial.png" Height="800" Stretch="None" />
32     </ScrollView>
33 </StackPanel>

```

When I run the application, I get no scroll bars whatsoever.

However, when I use a StackPanel inside of a ScrollView, all is well:

```

35 <!-- ScrollView can contain the stack panel -->
36 <ScrollView
37     Grid.Row="1"
38     Grid.Column="1"
39     HorizontalScrollBarVisibility="Auto"
40     VerticalScrollBarVisibility="Auto">
41     <StackPanel>
42         <Image Source="Assets/Financial.png" Height="800" Stretch="None" />
43         <Image Source="Assets/Financial.png" Height="800" Stretch="None" />
44     </StackPanel>
45 </ScrollView>
46

```

In this case, I've added two Image controls inside of a StackPanel, inside of a ScrollView and the result is that we can scroll around the StackPanel and its contents just fine.

Admittedly, this is a convoluted example for the purpose of experimentation. You would typically use it to display the entire viewable area inside of the "chrome" of the application's window or the entire application itself (in the case of the phone form factor). The ScrollView is very important and we'll use it whenever we need to scroll through a lot of data.

UWP-027 - Canvas and Shapes

In this lesson we're going to talk about the Canvas Layout Control as well as various Shape controls. While they're not inextricably tied together, you'll see that Shapes and the Canvas are complementary in nature.

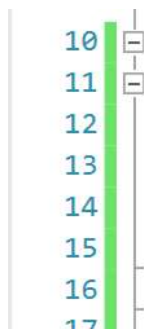
The Canvas allows for the absolute placement of other controls inside the boundaries of the Canvas. Typically, when creating layouts for Windows apps we do not want absolute positioning because the user resizes the application we would have to either write a lot of display logic and perform a lot of math to determine how to resize and position each item appropriately, or we would essentially limit the form factors that can consume our application.

However, there are some "edge cases" where utilizing the Canvas for absolute positioning would be helpful. So, in the interest of making sure you have a broad knowledge regarding layout I wanted to cover both the Canvas and the various Shape controls.

There are several areas types of application that could benefit from some simple Shape and Canvas usage such as:

- Apps that work with music, musical staves, musical notation, etc.
- Apps that deal with math requiring special symbols, placement of numbers and symbols
- Apps that have some graphical presentation of data, such as analog clocks, graphs, charts, etc.
- Apps for diagraming, drawing, or perhaps some sort of domain specific language generator which requires the user to drag and drop two symbols on to a design surface and then draw a line between them that has some meaning inside of your particular industry
- Apps that deal with bar readers, or create bars for scanners

First, we'll discuss the Line control. Admittedly, it doesn't need to be hosted by a Canvas control – you can add it to a Grid.

```
10  
11
12
13
14
15
16
17
    <Grid>
        <Line X1="10" X2="200"
            Y1="10" Y2="10"
            Stroke="Black" Fill="Black"
            StrokeThickness="5"
            StrokeEndLineCap="Triangle" />
    </Grid>
```

Here we're putting it inside of the default Cell, we're just specifying the X and Y coordinates for the beginning and the end. So here's the beginning point. X="10" and Y="10", so that gives it this left-most corner here. And then the second point would be X="200", Y="10", so that's going to give us a nice vertical line because the Y's don't change, only the X values.

You can see that we've said the `Stroke="Black"` and the `StrokeThickness="5"`, and then we also set the `StrokeEndLineCap="Triangle"` `StrokeEndLineCap="Triangle"` There are a couple different options here, you can see that we can create a Flat, Round, Square or Triangle. By choosing Triangle we get that little arrowhead on the end.

Next, I created a triangle by creating three more Lines and making sure that they intersect at just the right spot, making sure that their `StrokeStartLineCap` is set to Round, and that the `StrokeEndLineCap` is set to Round so they have a nice rounded appearance on the corners and they come together nicely.

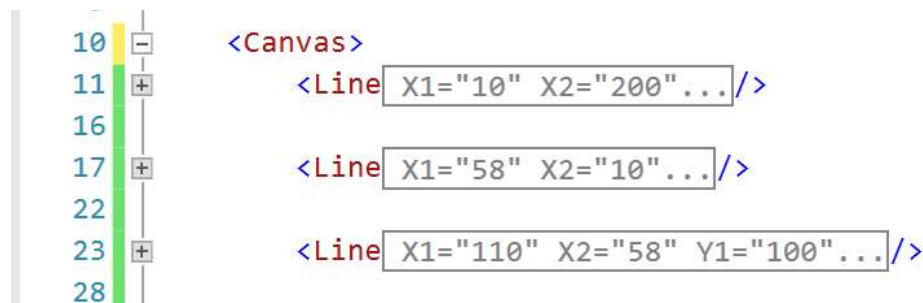
```
17 <Line X1="58" X2="10"  
18     Y1="25" Y2="100"  
19     Stroke="Black" Fill="Black"  
20     StrokeThickness="5"  
21     StrokeEndLineCap="Round" StrokeStartLineCap="Round" />  
22  
23 <Line X1="110" X2="58" Y1="100"  
24     Y2="25" Stroke="Black"  
25     Fill="Black"  
26     StrokeThickness="5"  
27     StrokeLineJoin="Round" StrokeStartLineCap="Round" />  
28  
29 <Line X1="110" X2="10"  
30     Y1="100" Y2="100"  
31     Stroke="Black" Fill="Black"  
32     StrokeThickness="5"  
33     StrokeLineJoin="Round" StrokeStartLineCap="Round" />  
34
```

Produces the following result:

ShapesExample

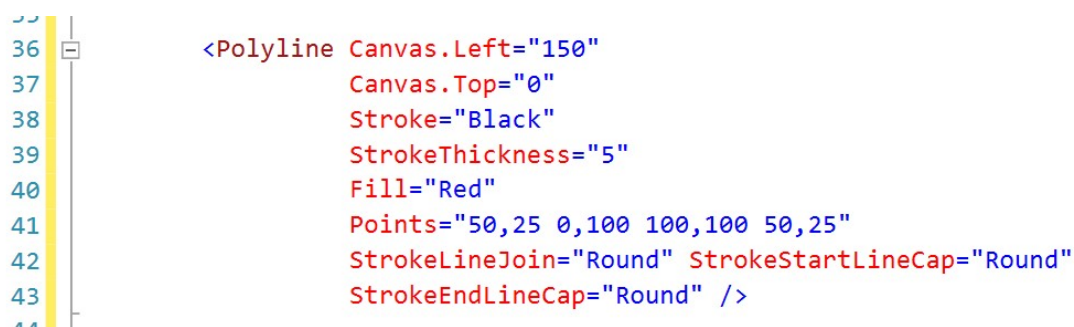


The point of the previous exercise was to prove that you can use the Line control in a Grid. However now let's change from a Grid to a Canvas:



With the Canvas we can use absolute placement by choosing an Left and Top position. Any child element inside the Canvas can be positioned using the attached property syntax: Canvas.Left and Canvas.Top.

I'll also introduce the Polyline object which greatly simplifies the task of creating a triangle:



Produces:

ShapesExample



In this example I am using attached properties to set the left and the top of the Polyline, and then I start creating a series of Points (X and Y pairs) that define the various points of my Triangle. Then I can even set the Fill color for my Polyline ... if it all joins together.

Notice that I'm placing this entire Polyline here at 150 Left, 0 Top, so all the Points are relative to that position that I defined.

Next, just to emphasize the absolute placement of items in a Canvas, I'm creating a TextBlock and I'm setting it to Canvas.Left and Canvas.Top to 50 and 150, respectively. This demonstrates that I can add any Control (not just Shapes) inside of a Canvas for absolute positioning.

```
44  
45 <TextBlock Name="HelloTextBlock"  
46     Canvas.Left="50"  
47     Canvas.Top="150"  
48     FontSize="24"  
49     Text="Shapes Example">  
50 </TextBlock>  
51
```

We've used the Rectangle in the past and for the sake of completeness I'll add it here, too. I define a yellow rectangle. Again, I am setting the Canvas.Top attached to 200 property and the Canvas.Left to zero.

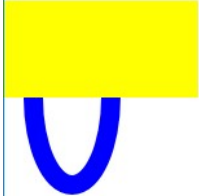
I've also added a Canvas.ZIndex property that will allow me to define the "stack order" of items should they overlap.

Next, I add an Ellipse intentionally overlapping them.

```
53 <Rectangle Canvas.Top="200"  
54     Canvas.Left="0"  
55     Height="50" Width="100"  
56     Fill="Yellow"  
57     Canvas.ZIndex="100" />  
58  
59 <Ellipse Stroke="Blue"  
60     Width="50" Height="100"  
61     Canvas.Left="10"  
62     Canvas.Top="200"  
63     StrokeThickness="10"  
64     Canvas.ZIndex="15" />  
65  
66 </Canvas>
```

In this case, I set the ZIndex of the Ellipse to 150 and the ZIndex of the yellow Rectangle to 100. If I change that, you can see that by changing the ZIndex to 15, it puts the yellow Rectangle on top of the Ellipse. So, we are talking in terms of X and Y-Coordinates, but also now adding Z-Coordinate in 3-D space, and so the ZIndex will set that stacking order for you.

Shapes Example



If you need to draw Shapes in your application.

Finally, most of the shapes have a Click or Tapped event you can handle in the event that the user were to try to interact with that given Shape.