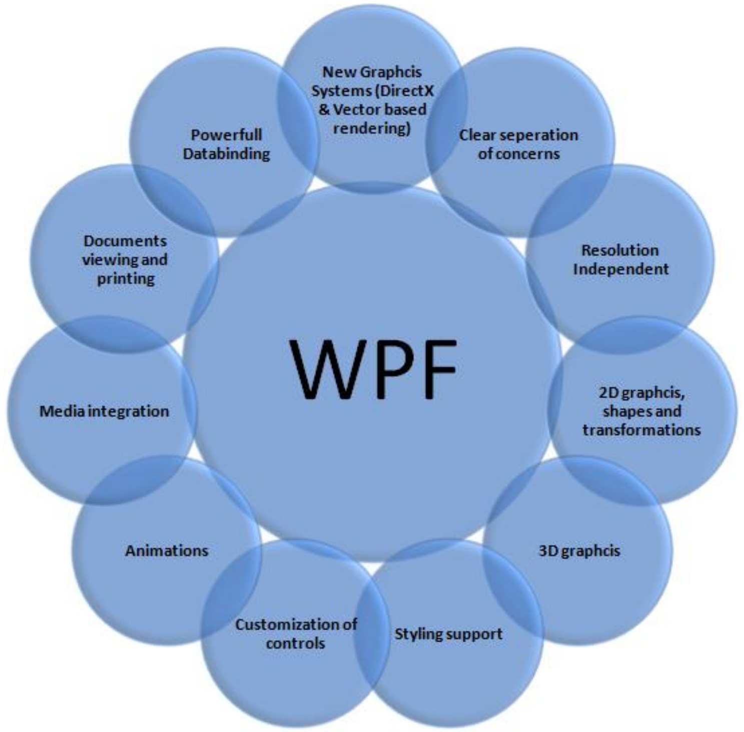
Advanced Programming 2
Recitation 4 – WPF

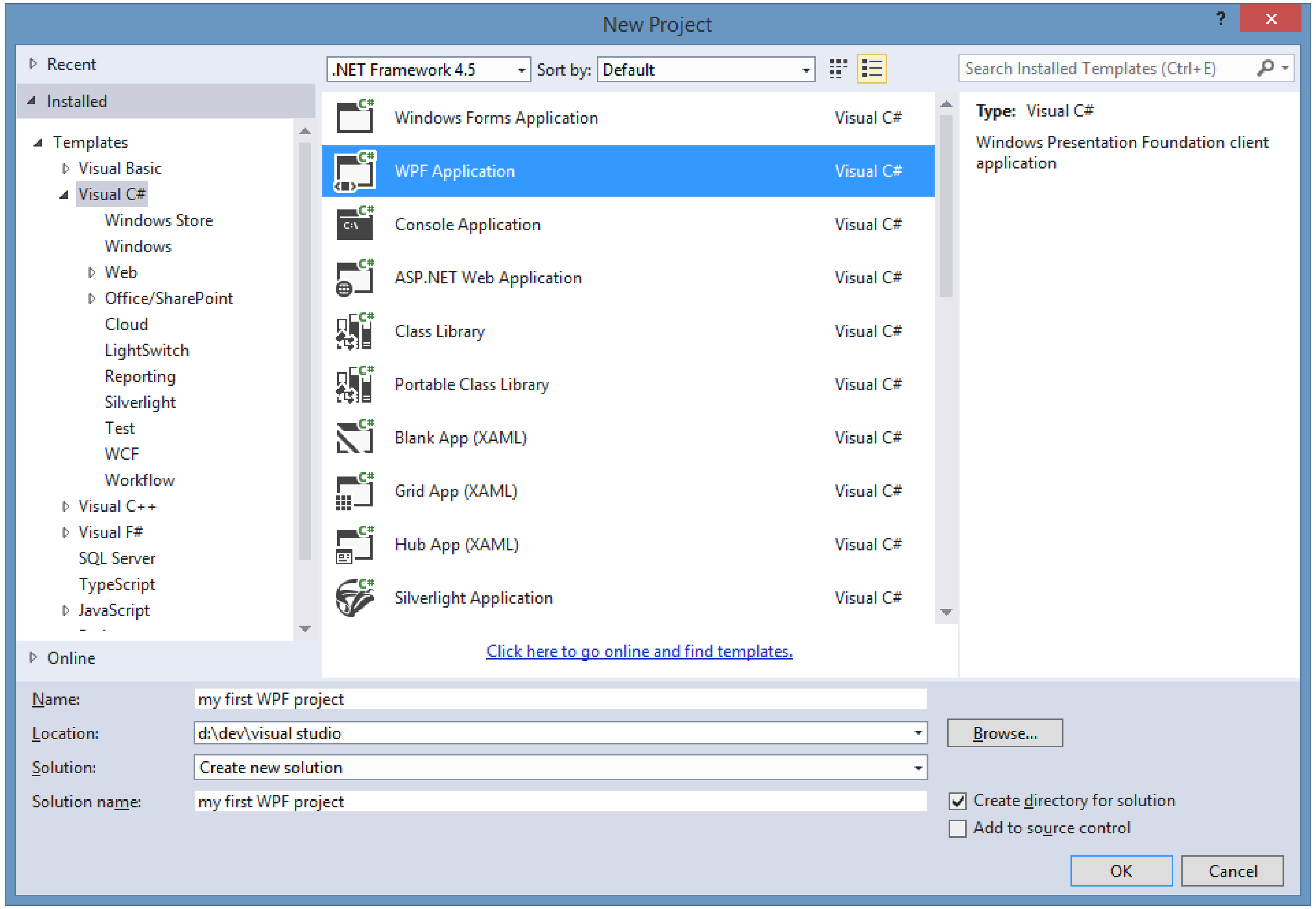
Roi Yehoshua
2017

What is WPF?

- WPF is a (relatively) new platform for UI, media and documents
- Integrates all UX elements
- Broad integration
  - 2D, 3D, video, animations, documents
- Resolution independence
- Hardware acceleration
  - Rendering engine uses DirectX
- Declarative programming
  - Via XAML
- Rich composition and customization
- Data binding throughout the system


WPF
New Graphcis Systems (DirectX & Vector based rendering)
Clear seperation of concerns
Resolution Independent
2D graphcis, shapes and transformations
3D graphcis
Styling support
Customization of controls
Animations
Media integration
Documents viewing and printing
Powerfull Databinding

Creating New WPF Project


New Project
Recent
Installed
Templates
Visual Basic
Visual C#
Windows Store
Windows
Web
Office/SharePoint
Cloud
LightSwitch
Reporting
Silverlight
Test
WCF
Workflow
Visual C++
Visual F#
SQL Server
TypeScript
JavaScript
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Application Visual C#
Class Library Visual C#
Portable Class Library Visual C#
Blank App (XAML) Visual C#
Grid App (XAML) Visual C#
Hub App (XAML) Visual C#
Silverlight Application Visual C#
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual C#

Windows Presentation Foundation client application

Name: my first WPF project

Location: d:\dev\visual studio

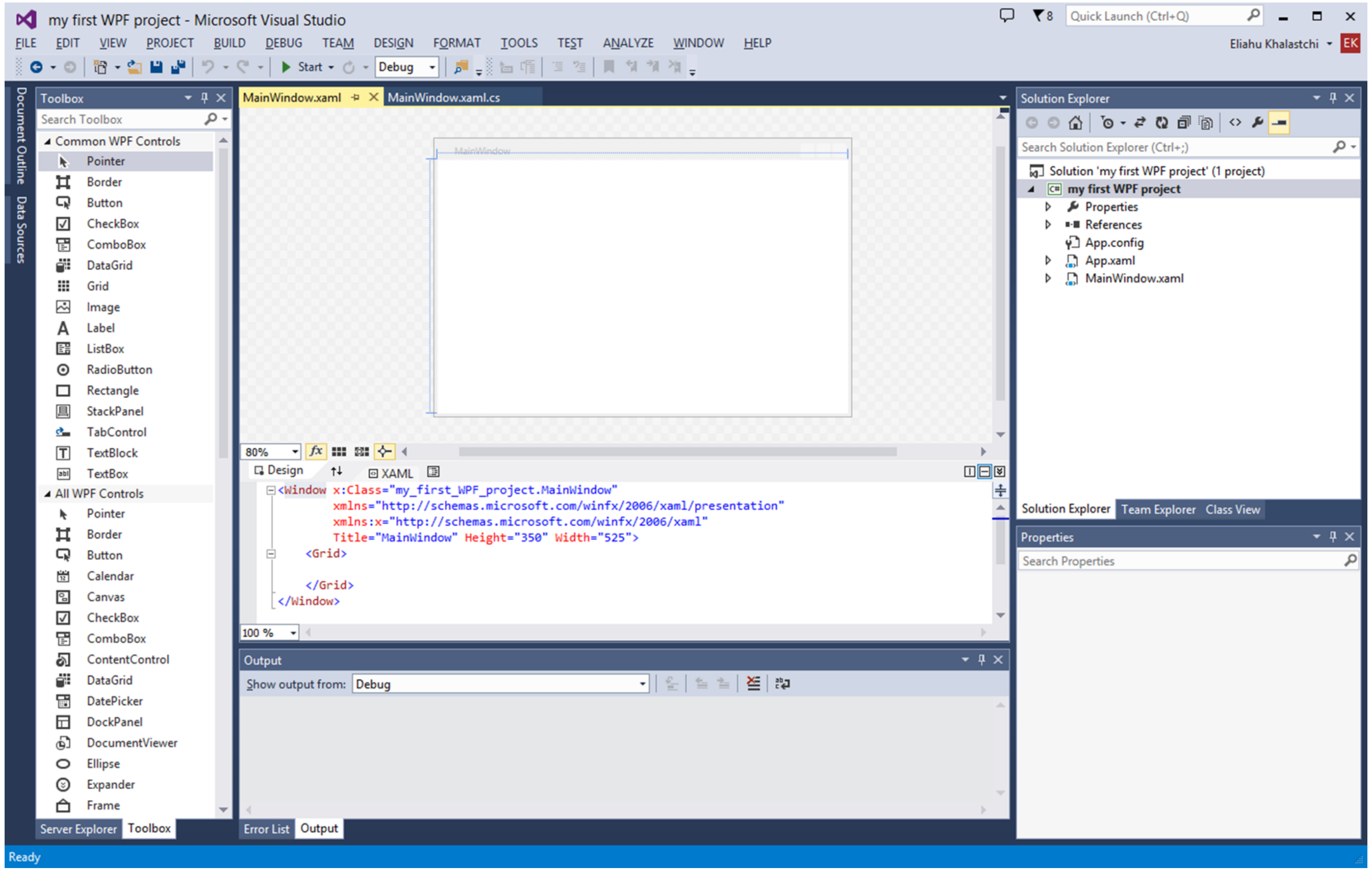
Solution: Create new solution

Solution name: my first WPF project

☑ Create directory for solution

☐ Add to source control

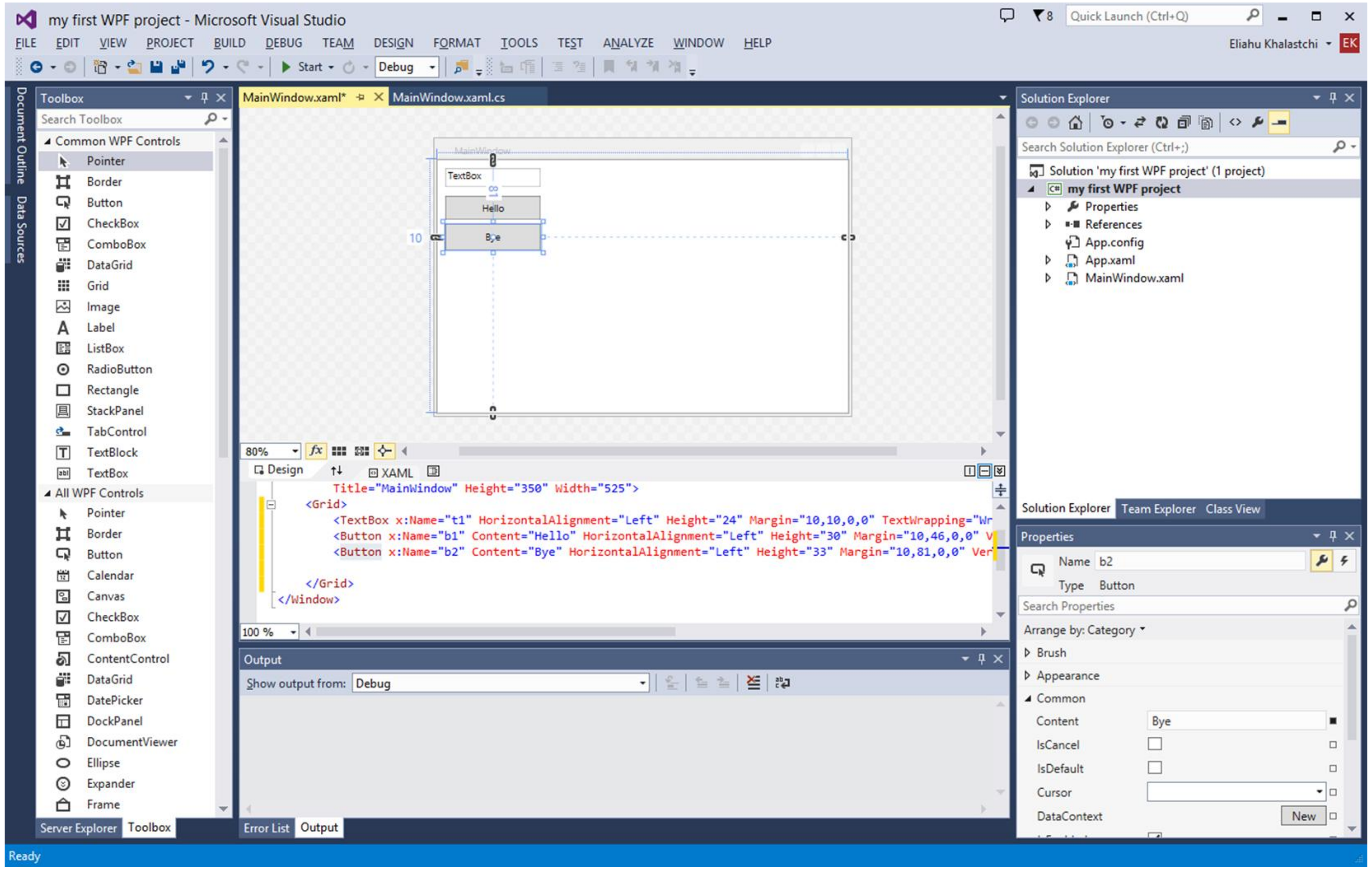
XAML Editor



```
<Window x:Class="my_first_WPF_project.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

XAML Editor

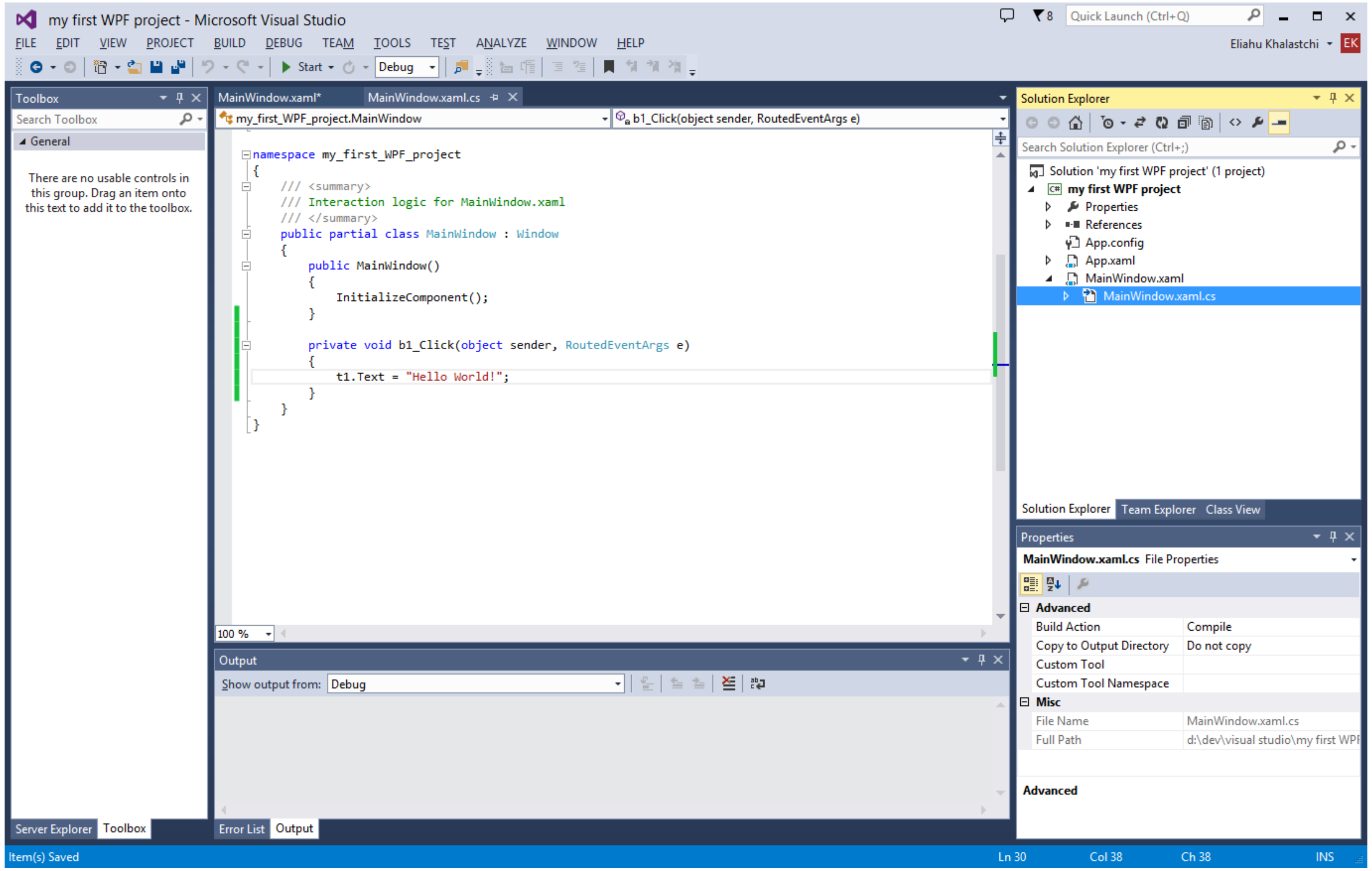


```
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <TextBox x:Name="t1" HorizontalAlignment="Left" Height="24" Margin="10,10,0,0" TextWrapping="Wr
        <Button x:Name="b1" Content="Hello" HorizontalAlignment="Left" Height="30" Margin="10,46,0,0" V
        <Button x:Name="b2" Content="Bye" HorizontalAlignment="Left" Height="33" Margin="10,81,0,0" Ver

    </Grid>
</Window>
```

Properties
Name b2
Type Button
Arrange by: Category
Common
Content Bye

Code Behind File



There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

```
namespace my_first_WPF_project
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void b1_Click(object sender, RoutedEventArgs e)
        {
            t1.Text = "Hello World!";
        }
    }
}
```

Properties

MainWindow.xaml.cs File Properties

Advanced	
Build Action	Compile
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	
Misc	
File Name	MainWindow.xaml.cs
Full Path	d:\dev\visual studio\my first WPF

Advanced

XAML

What is XAML?

- XML based language
- Enable separation of UI and behavior (code)
- XAML allows
  - Creation of objects
  - Setting of properties
  - Connection to events
- Anything that can be done in XAML can be done in code
  - But not vice versa
    - e.g., XAML cannot call methods
- XAML is usually shorter and more concise than the equivalent code
  - Thanks to type converters and markup extensions
- XAML should be used for initial UI
- Code will handle events and change items dynamically

Simple XAML Example

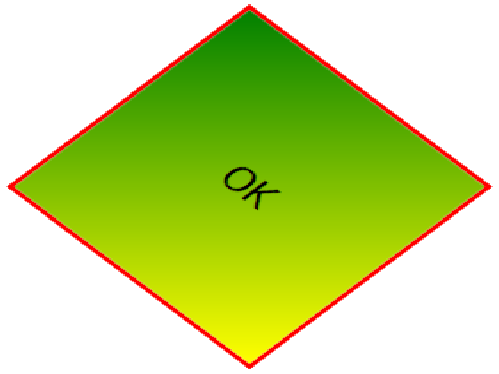
```xml
<Button Content="OK" />
```

```csharp
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Content = "OK";
```

XAML Property Elements

- You can assign other object elements to be the value of a property.
- Instead of the property being specified as an attribute, the property is specified using an opening element tag in *elementTypeName.propertyName* form, the value of the property is specified within, and then the property element is closed

```
<Button BorderBrush="Red" Content="OK" FontSize="25">
  <Button.LayoutTransform>
    <RotateTransform Angle="45" />
  </Button.LayoutTransform>
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Color="Green" Offset="0" />
      <GradientStop Color="Yellow" Offset="1" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```



Content Property

- A single property that is designated with the **ContentProperty** attribute on the type
- Allows shortening the markup

```
<Button Content="OK" >
</Button>
```

⟺

```
<Button>
    OK
</Button>
```

```
<Button>
    <Button.Content>
        <Rectangle Fill="Blue" />
    </Button.Content>
</Button>
```

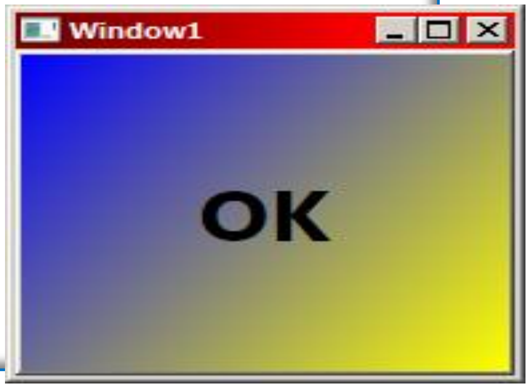
⟺

```
<Button>
    <Rectangle Fill="Blue" />
</Button>
```

XAML And Code Behind

- Code behind file
  - The **x:Class** XAML attribute specifies the code behind class that inherits from the top level element

```
<Window x:Class="Demo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Button Content="OK" FontSize="35" FontWeight="Bold">
        <Button.Background>
             <LinearGradientBrush>
                 <GradientStop Color="Blue" Offset="0" />
                 <GradientStop Color="Yellow" Offset="1" />
             </LinearGradientBrush>
        </Button.Background>
    </Button>
</Window>
```



XAML Namespaces

- The default X(A)ML namespace is mapped to the WPF .NET namespaces
- An “x” namespace is mapped to **System.Windows.Markup**
  - (the XAML namespace within WPF)
- You can map other XML namespaces to .NET namespaces using the xmlns prefix
  - This will enable you to use the public types within that namespace as elements

```
<Window x:Class="MazeGUI.SinglePlayerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:controls="clr-namespace:MazeGUI.Controls”
        Title="{Binding Path=Maze.Name}" Height="650" Width="600"
WindowStartupLocation="CenterScreen" Closing="Window_Closing">
    <Grid TextBlock.FontSize="14" x:Name="grid1">
        …
        <controls:MazeBoard x:Name="mazeBoard"></controls:MazeBoard>
    </Grid>
</Window>
```

Note that the Name attribute belongs to the “x” namespace

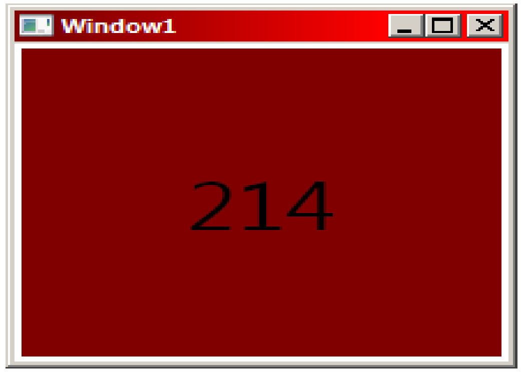
Type Converters

- Property values in XAML may be translated
  - E.g. “Red” translated to the static property **`System.Windows.Media.Brushes.Red`**
- XAML parser looks for a converter from a string to the required type
- Converters inherit **`System.ComponentModel.TypeConverter`**
  - E.g. **`BrushConverter`**, **`ColorConverter`**

Markup Extensions

- Special extenders to XAML
- Classes deriving from **System.Windows.Markup.MarkupExtension**
  - Usually end with the word **Extension**
- Used with curly braces
- Constructor parameters inserted after the markup extension type
  - Named properties may be added next

```
<Button Background="{x:Static SystemColors.ActiveCaptionBrush}"
    FontSize="40" BorderBrush="{x:Null}"
    Content="{Binding Path=ActualWidth, RelativeSource={RelativeSource Self}}">
</Button>
```



Child Elements

- Child elements (that are not property elements) can be one of
  - The **Content** property of the object
    - A property adorned with the attribute `System.Windows.Markup.ContentProperty`
  - Collection items
    - The object implements `IList` or `IDictionary`
  - A value that can be type-converted to its parent (and the parent has no properties)

Collection Items

- List (`IList`)

```
<ListBox>
    <ListBox.Items>
        <ListBoxItem Content="Item 1"/>
        <ListBoxItem Content="Item 2"/>
    </ListBox.Items>
</ListBox>
```

- Dictionary (`IDictionary`)

```
<ResourceDictionary>
    <SolidColorBrush x:Key="br1" Color="Aqua" />
    <Rectangle x:Key="rc1" Fill="Brown" />
</ResourceDictionary>
```

Basic Concepts

Dependency Properties

- “Normal” .NET properties are usually simple wrappers around private fields
- Dependency properties allow
  - Property value change notifications
    - E.g. for data binding, animations
  - Property value inheritance
    - Down the logical / visual tree
  - Multiple providers affecting the final value
  - Memory conservation

How a Property Value is Determined

- Lowest to highest precedence

Default value set by FrameworkPropertyMetadata object

The inherited value, if the Inherited flag is true and a value was applied up in the element hierarchy

The value from a Style

The local value (set explicitly)

If it's an expression – evaluate (currently: Resources or Data Binding)

If it's the target of an animation – apply animation

Execute CoerceValueCallback and ValidateValueCallback

Dependency Property Declaration

- Can use the “propdp” code snippet in Visual Studio

```
public int MyProperty
{
    get { return (int)GetValue(MyPropertyProperty); }
    set { SetValue(MyPropertyProperty, value); }
}

// Using a DependencyProperty as the backing store for MyProperty.  This enables animation,
styling, binding, etc...
public static readonly DependencyProperty MyPropertyProperty =
    DependencyProperty.Register("MyProperty", typeof(int), typeof(ownerclass), new
PropertyMetadata(0));
```

Attached Properties

- Special kind of dependency properties
- May be “attached” to objects of different types than the declaring type
  - Declared with the static method **`DependencyProperty.RegisterAttached`**
- Allows “context” properties
  - E.g. **`Canvas.Left`** for elements that happen to be in a `Canvas` element
  - Can be set on *any* object
- XAML
  - An attribute with **`Type.Property`** syntax is used
- In code
  - The type exposes a `SetXxx` and a `GetXxx` with the element reference

Attached Properties Example

- XAML

```
<Canvas>
     <Button x:Name="cmdOK" Canvas.Left="30" Canvas.Top="20"
             Content="OK" Padding="10" FontSize="26">
     </Button>
</Canvas>
```

- Code

```
Canvas.SetLeft(cmdOK, 30);
Canvas.SetTop(cmdOK, 20);
```

⇕

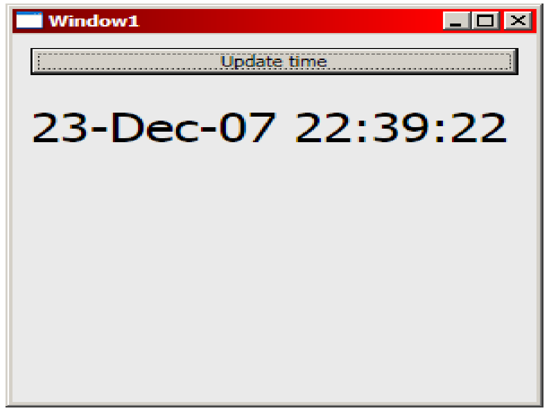
```
cmdOK.SetValue(Canvas.LeftProperty, 30);
cmdOK.SetValue(Canvas.TopProperty, 30);
```

Routed Events

- WPF events are not implemented via the default .NET event implementation
- Routed events implemented similarly to dependency properties
- Routing strategies (`RoutingStrategy` enum)
  - `Bubbling` event – upwards the tree
  - `Tunneling` event – downward the tree
  - `Direct` event – only on the source element

```
<StackPanel>
      <Button Margin="10" Content="Update time" Click="Button_Click" />
      <TextBlock x:Name="text" Text="" FontSize="30" Margin="10" />
</StackPanel>
```

```
private void Button_Click(object sender, RoutedEventArgs e) {
    text.Text = DateTime.Now.ToString();
}
```



23-Dec-07 22:39:22

Routed Event Delegate

```
public delegate void RoutedEventHandler(object sender, RoutedEventArgs e);
```

- **RoutedEventArgs**
  - Derives from **System.EventArgs**
  - Properties
    - **Source**
      - the element in the logical tree that raised the event
    - **OriginalSource**
      - Usually the same as Source
      - Sometimes the element in the visual tree that raised the event
    - **Handled**
      - indicates whether to stop tunneling/bubbling
    - **RoutedEvent**
      - the routed event object itself

Tunneling vs. Bubbling

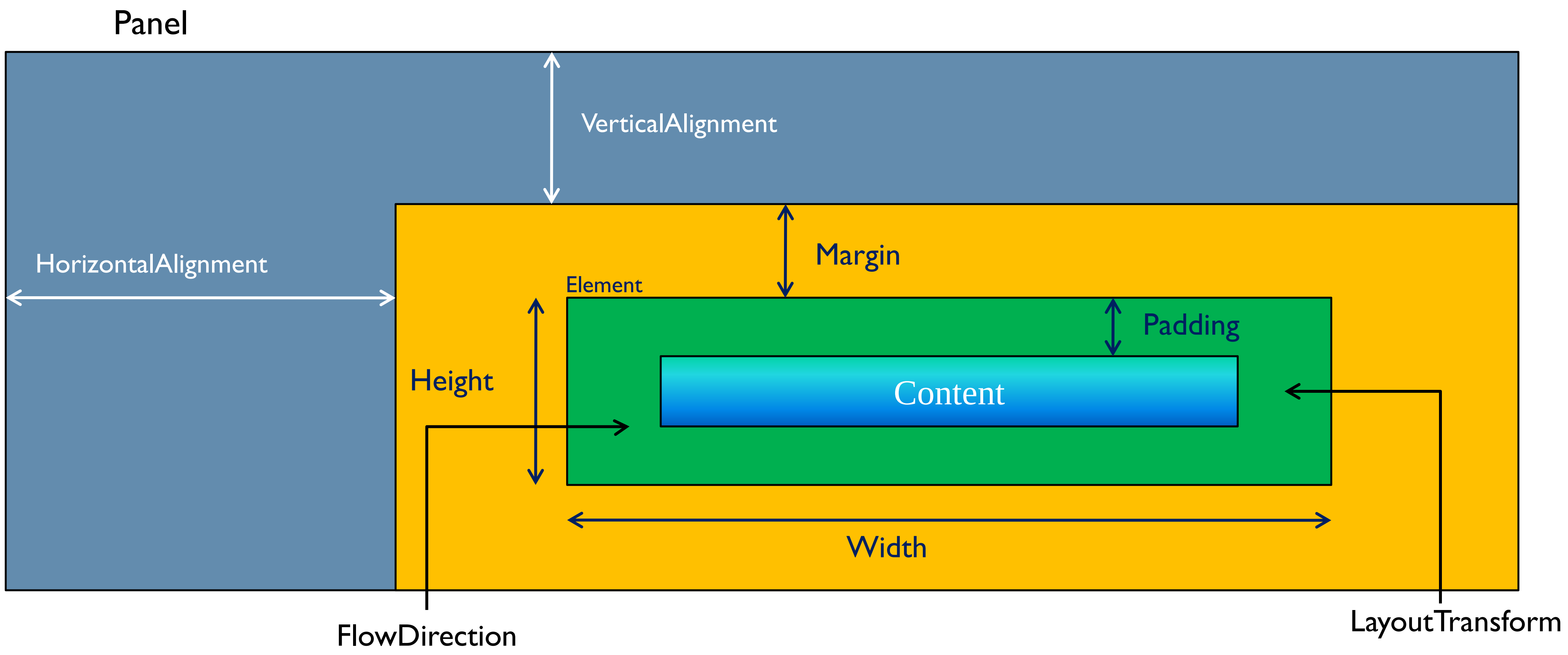
- Tunneling and bubbling events are usually paired
- The tunneling event fires first
  - Marking it as "handled" cancels its bubbling paired event
- By convention, the tunneling event name begins with "Preview"
- Example
  - `PreviewMouseLeftButtonDown` (tunneling), `MouseLeftButtonDown` (bubbling)

Layout and Panels

Layout

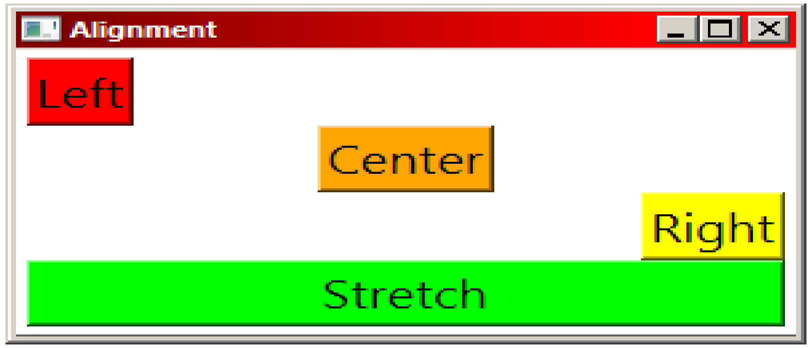
- Layout is the arranging of user interface elements within some container
- Older technologies (e.g. Windows Forms) mostly used exact position and sizes
  - Limited in flexibility and adaptability
- WPF provides several layout panels that can control dynamically size and placement of elements
- Element sizing and positioning is determined by the element itself and its logical parent
- A child element may request various settings
- The parent panel does not have to comply

Element Layout Properties



Alignment

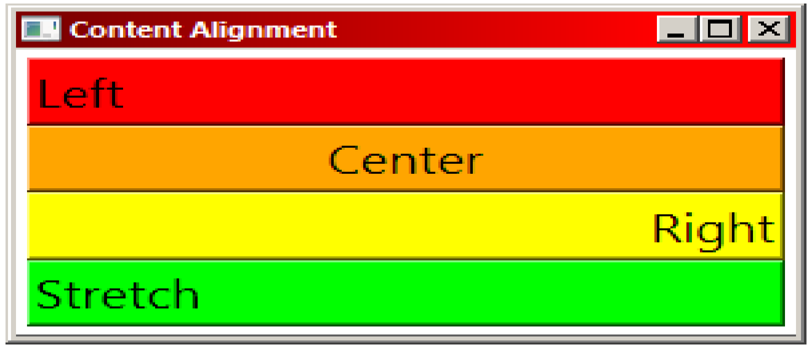
- Alignment indicates what should be done with any extra space given to an element
- **HorizontalAlignment**
  - **Left**, **Right**, **Center**, **Stretch**
- **VerticalAlignment**
  - **Top**, **Bottom**, **Center**, **Stretch**



```
<StackPanel TextBlock.FontSize="20" Margin="4">
    <Button HorizontalAlignment="Left" Background="Red">Left</Button>
    <Button HorizontalAlignment="Center" Background="Orange">Center</Button>
    <Button HorizontalAlignment="Right" Background="Yellow">Right</Button>
    <Button HorizontalAlignment="Stretch" Background="Lime">Stretch</Button>
</StackPanel>
```

Content Alignment

- Similar to element alignment
- What to do with extra space when the content is smaller than its control
- **`HorizontalContentAlignment`**
- **`VerticalContentAlignment`**



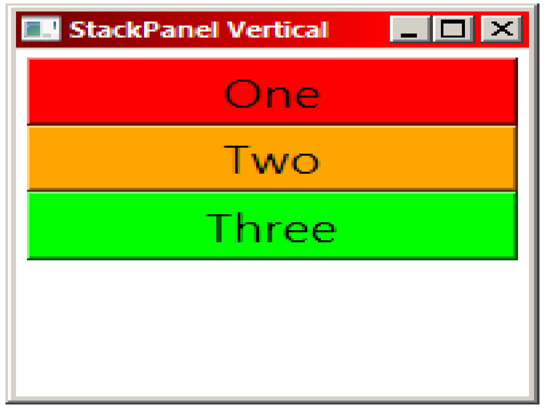
```
<StackPanel TextBlock.FontSize="20" Margin="4">
    <Button HorizontalContentAlignment="Left" Background="Red">Left</Button>
    <Button HorizontalContentAlignment="Center" Background="Orange">Center</Button>
    <Button HorizontalContentAlignment="Right" Background="Yellow">Right</Button>
    <Button HorizontalContentAlignment="Stretch" Background="Lime">Stretch</Button>
</StackPanel>
```

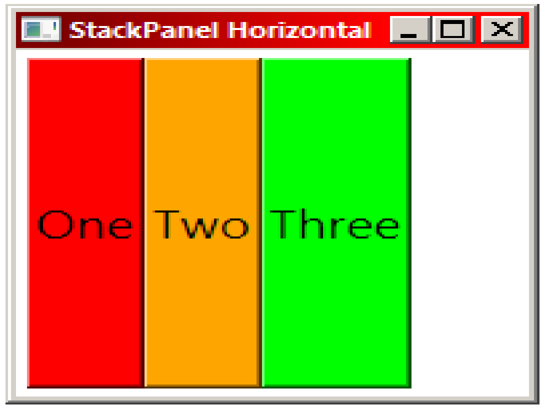
WPF Layout Panels

- Main layout panels
  - `Canvas`
    - Arranges children in a 2D coordinate system
  - `StackPanel`
    - Arranges children in a horizontal or vertical “stack”
  - `DockPanel`
    - Arranges children horizontally or vertically to each other towards the edges
  - `WrapPanel`
    - Arranges children continuously horizontally or vertically, flowing to the next row/column
  - `Grid`
    - Arranges children in a flexible grid

The StackPanel

- Stacks its elements in a vertical or horizontal “stack”
- **Orientation** property
  - **Vertical** (default) or **Horizontal**





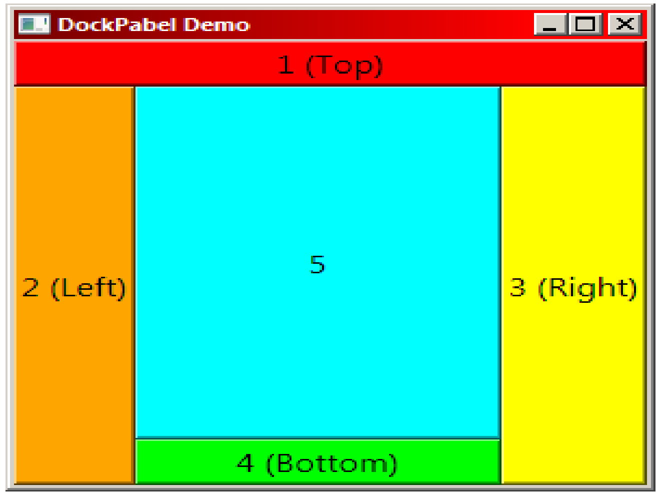
The `WrapPanel`

- Similar to a `StackPanel`, but wraps overflowing elements to the next row or column, depending on its orientation
  - Usually used as the panel for `ItemsControl` elements
- Properties
  - **`Orientation`**
    - **`Horizontal`** (default) or **`Vertical`**
  - **`ItemHeight`**
    - The maximum height for horizontal stacking
    - Default is **`Double.NaN`** – i.e. the tallest element
  - **`ItemWidth`**
    - Same concept as `ItemHeight` for the width

The DockPanel

- Enables easy docking of elements to the horizontal or vertical sides of the panel
- Allows the last child to fill the remaining space if the property **LastChildFill** is set to true (the default)
- Docking is done with the **DockPanel.Dock** attached property
  - **Left**, **Top**, **Right**, **Bottom**

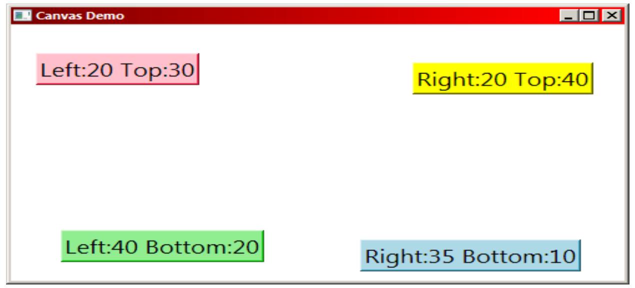
```
<DockPanel TextBlock.FontSize="16">
    <Button DockPanel.Dock="Top" Background="Red">1 (Top)</Button>
    <Button DockPanel.Dock="Left" Background="Orange">2 (Left)</Button>
    <Button DockPanel.Dock="Right" Background="Yellow">3 (Right)</Button>
    <Button DockPanel.Dock="Bottom" Background="Lime">4 (Bottom)</Button>
    <Button Background="Aqua">5</Button>
</DockPanel>
```



The Canvas

- “Arranges” elements based on their coordinates and specific sizes
- The most primitive and fastest
- An element uses the **`Left`**, **`Top`**, **`Right`**, **`Bottom`** attached properties
  - Can position relative to the right or bottom edge, not just left or top
- Useful for custom drawn elements or other non-standard displays

```
<Canvas TextBlock.FontSize="20">
    <Button Canvas.Left="20" Canvas.Top="30" Content="Left:20 Top:30" Background="Pink" />
    <Button Canvas.Right="25" Canvas.Top="40" Content="Right:20 Top:40" Background="Yellow" />
    <Button Canvas.Left="40" Canvas.Bottom="20" Content="Left:40 Bottom:20“
      Background="LightGreen" />
    <Button Canvas.Right="35" Canvas.Bottom="10" Content="Right:35 Bottom:10"
      Background="LightBlue" />
  </Canvas>
```



The Grid

- The most versatile and useful panel
- Usually used as the top-level panel
- Arranges its children in a multi-row and multi-column way
- For rows
  - Set the **`RowDefinitions`** property
  - Add a **`RowDefinition`** object for each row
- For columns
  - Set the **`ColumnDefinitions`** property
  - Add a **`ColumnDefinition`** object for each column
- For each element
  - Set the **`Grid.Row`** and **`Grid.Column`** attached properties (default is 0, 0)

Sizing Rows and Columns

- By default, all rows are of equal height and all columns are of equal width
  - Can change the height of a row using the **`RowDefinition.Height`** property
  - Can change the width of a column using the **`ColumnDefinition.Width`** property
  - The unit is controlled by the **`GridUnitType`** property
    - **`Auto`** – size as required by content
    - **`Pixel`** – size is the number specified
    - **`Star`** – size is a weighted proportional (default)
      - “*”, “2*”, etc. in XAML
- Spanning
  - A row may span more than one column and vice versa
  - Can be set by the **`Grid.RowSpan`** and **`Grid.ColumnSpan`** attached properties
    - Default for both is 1

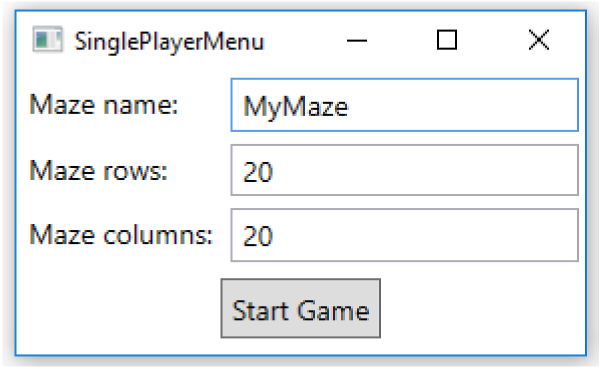
Grid Example

```
<Grid TextBlock.FontSize="14">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <TextBlock Padding="3" Margin="3">Maze name:</TextBlock>
    <TextBox x:Name="txtMazeName" Grid.Column="2" Padding="3" Margin="3"></TextBox>
    <TextBlock Grid.Row="1" Padding="3" Margin="3">Maze rows:</TextBlock>
    <TextBox x:Name="txtRows" Grid.Row="1" Grid.Column="2" Padding="3"
Margin="3"></TextBox>
    <TextBlock Grid.Row="2" Padding="3" Margin="3">Maze columns:</TextBlock>
    <TextBox x:Name="txtCols" Grid.Row="2" Grid.Column="2" Padding="3"
Margin="3"></TextBox>
    <Button x:Name="btnStart" Grid.Row="3" HorizontalAlignment="Center"
Grid.ColumnSpan="2" Click="btnStart_Click" Margin="5" Padding="5">Start Game</Button>
</Grid>
```

Notice the use of attached property

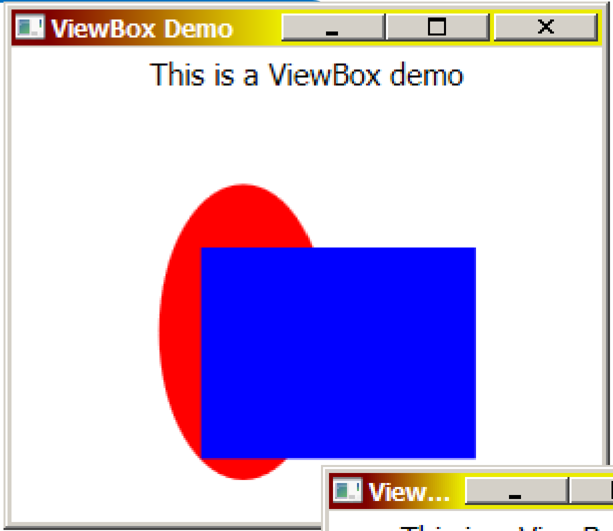
Allows the last column to fill the available space

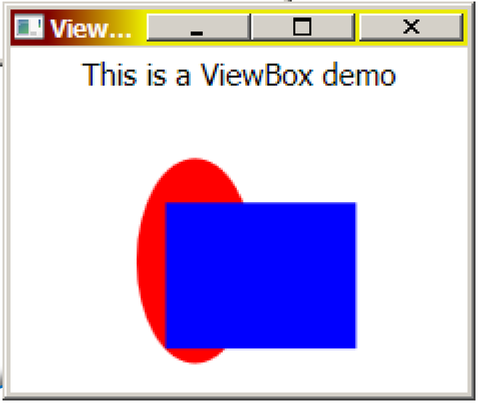


ViewBox

- Easy resizing can be achieved with a **ViewBox**
- A ViewBox has one child, which it stretches to fill available space

```
<Window x:Class="WpfApplication1.ViewBoxWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ViewBox Demo" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>
        <TextBlock Text="This is a ViewBox demo" Margin="4" FontSize="15"
             HorizontalAlignment="Center"/>
        <Viewbox Grid.Row="1">
            <Canvas Width="130" Height="100">
                <Ellipse Width="40" Height="70" Fill="Red" StrokeThickness="3"
             Canvas.Left="30" Canvas.Bottom="10"/>
                <Rectangle Width="65" Height="50" Fill="Blue" StrokeThickness="2"
                Canvas.Left="40" Canvas.Top="35" />
            </Canvas>
        </Viewbox>
    </Grid>
</Window>
```


ViewBox Demo
This is a ViewBox demo


View...
This is a ViewBox demo