Introduction to WPF

Introduction

- **WPF** (Windows Presentation Foundation) is a UI framework to create applications with a rich user experience.
- It is part of the .NET framework 3.0 and higher.
- It includes application UI, 2D graphics, 3D graphics and multimedia.
- WPF makes the UI faster, scalable and resolution independent.
- It offers easy separation between UI and the business logic.
- WPF is a vector-based rendering engine that uses hardware acceleration of modern graphics cards which makes the UI faster and highly scalable.

Features of WPF

- **Resolution Independence:**
  - WPF is resolution independence since all measures in WPF are logical units not pixels.
  - A logical unit is a 1/96 of an inch. So, with changing the screen resolution setting in WPF each control will look same for each resolution.
  - It is not based on Dots per inch (DPI) setting of the device.

- **Separation of Appearance and Behaviors:**
  - WPF separates the appearance of an UI from its behavior.
  - The appearance is specified by XAML and behavior is specified by a managed programming language like C# or VB.

- **Built-in Support for Graphics and Animation:**
  - WPF applications run within DirectX environment, hence it has major support of graphics and animation capabilities.
  - WPF has a separate set of classes that specifically deal with animation effects and graphics.

Features of WPF

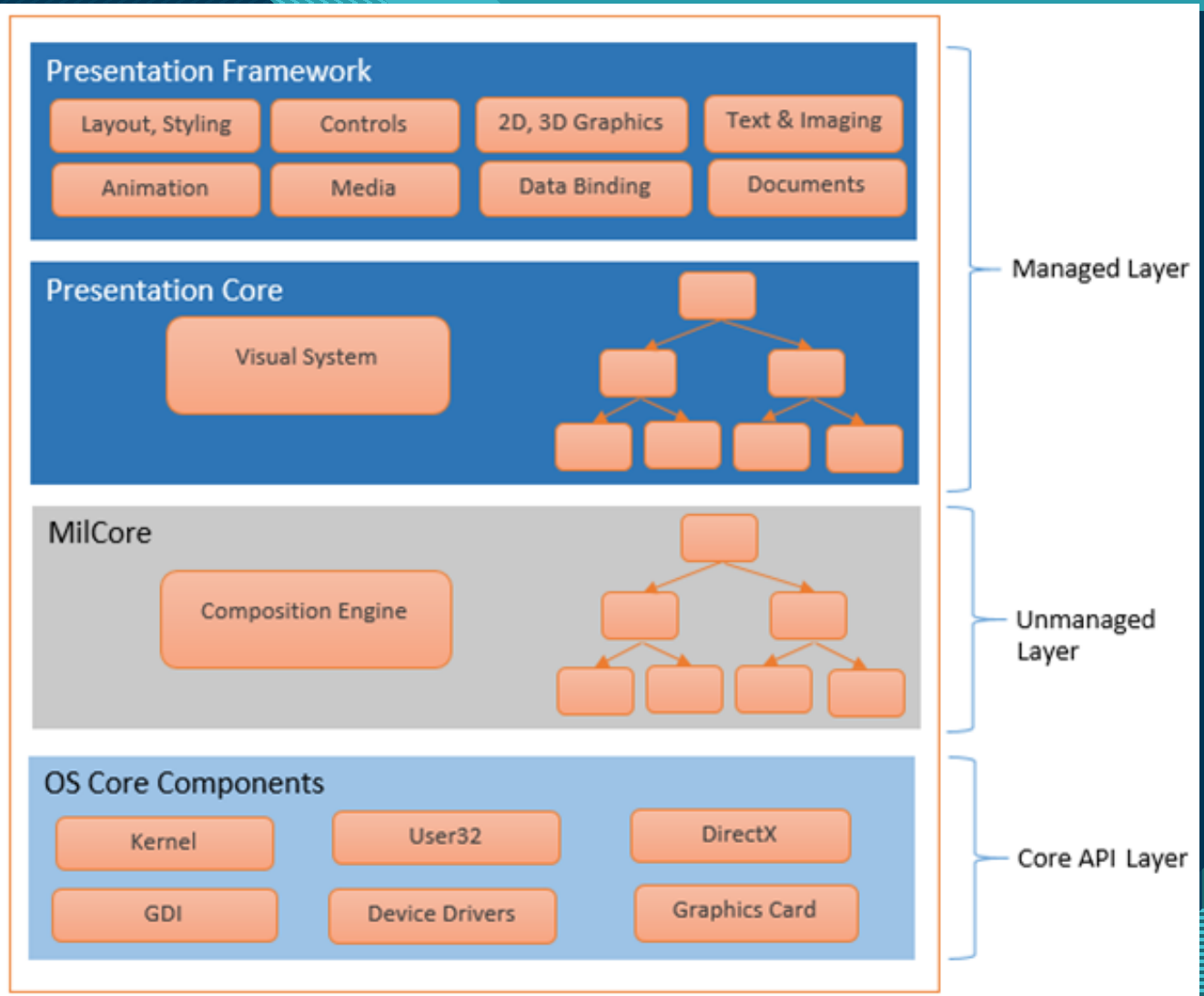
- **Support for Audio and Video:**
  - WPF has support for playing any audio or video file supported by Windows Media Player.
  - It also gives you the tools to integrate video content into your rich UI such as placing a video window on a spinning 3-D cube.

- **Highly Customizable:**
  - WPF supports separation of appearance and behaviors; hence you can easily change the look of a control or a set of controls.
  - This concept of styling controls in WPF, is almost like CSS in HTML.
  - In WPF, you can store styles, controls, animations, and even any object as a resource and you may associate that resource to the controls.
  - Each resource is declared once when the form loads itself.

WPF Architecture

WPF architecture is a layered architecture which have Managed, Unmanaged and Core API layers:

- **Managed Layer:**
  - Managed layer has two main components:
    - **Presentation Framework** provides the required functionalities that we need to build the WPF applications such as controls, data bindings, styling, shapes, media, documents, animation and more.
    - PresentationFamework.dll is responsible for this purpose.
    - **Presentation Core** acts as a managed wrapper around MILCore and provides public interface for MIL.
    - Presentation Core is the home for WPF Visual System and provides classes for creating application visual tree.
    - The Visual System creates visual tree which contains applications Visual Elements and rendering instructions.
    - PresentationCore.dll is responsible for this purpose.

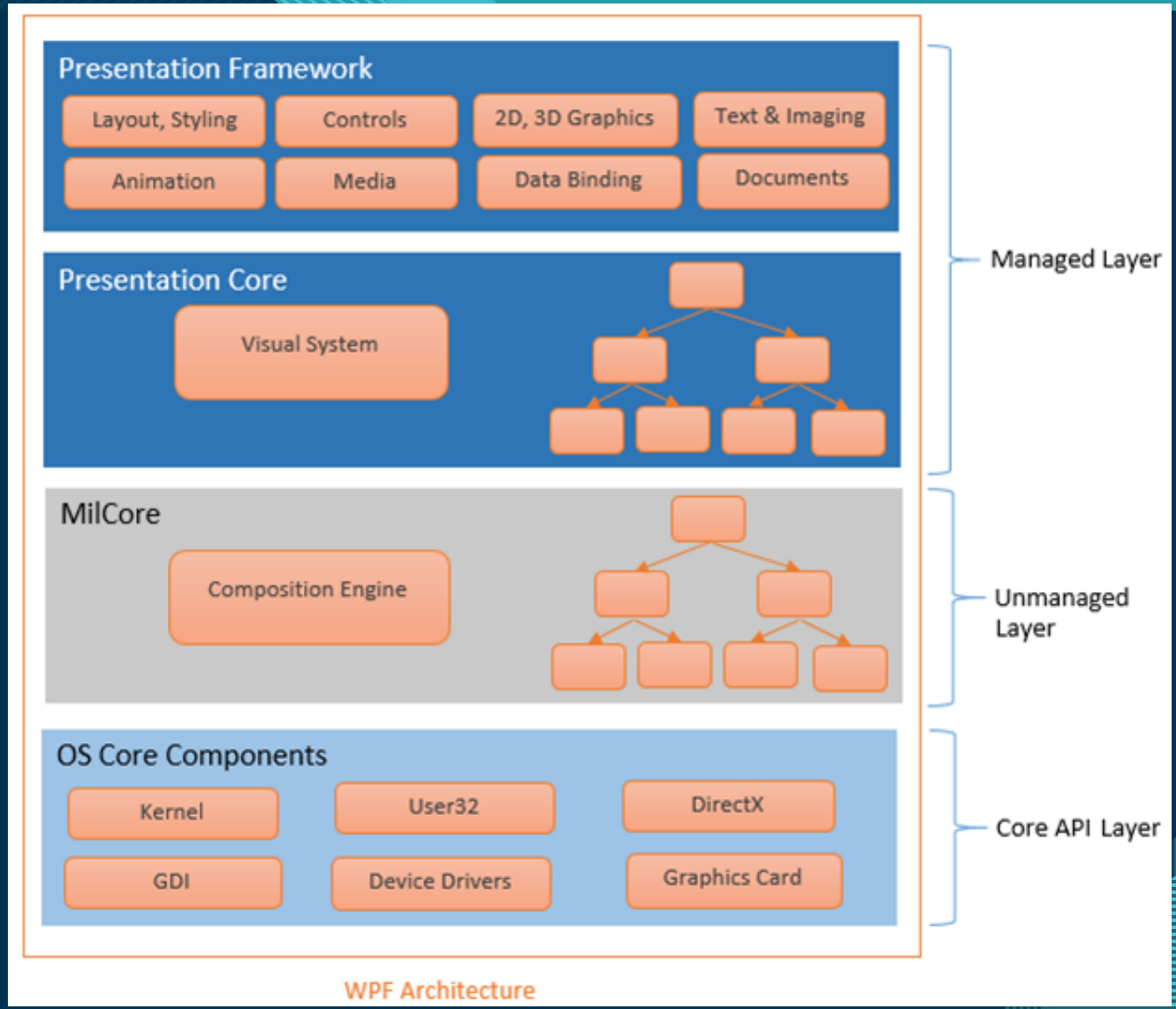


WPF Architecture

WPF Architecture

- **Unmanaged Layer:**
  - This layer is also called MilCore or Media Integration Library Core.
  - MilCore is written in unmanaged code in order to enable tight integration with DirectX.
  - DirectX engine is underlying technology used in WPF to display all graphics, allowing for efficient hardware and software rendering.
  - MIL has Composition System that receives rendering instructions from Visual System and translates into data that can be understood by DirectX to render user interface.

- **Core API Layer:**
  - This layer has OS core components like Kernel, User32, GDI, Device Drivers, Graphic cards etc.
  - These components are used by the application to access low level APIs.



WPF Architecture

The Evolution of Windows Graphics

- Before WPF, Windows developers spent nearly 15 years using essentially the same display technology.
- That's because every traditional, pre-WPF Windows application relied on two well-worn parts of the Windows operating system to create its user interface:
  - **User32:** This provides the traditional Windows look and feel for elements such as windows, buttons, text boxes, and so on.
  - **GDI/GDI+:** This provides drawing support for rendering shapes, text, and images at the cost of additional complexity and performance.

DirectX: The New Graphics Engine

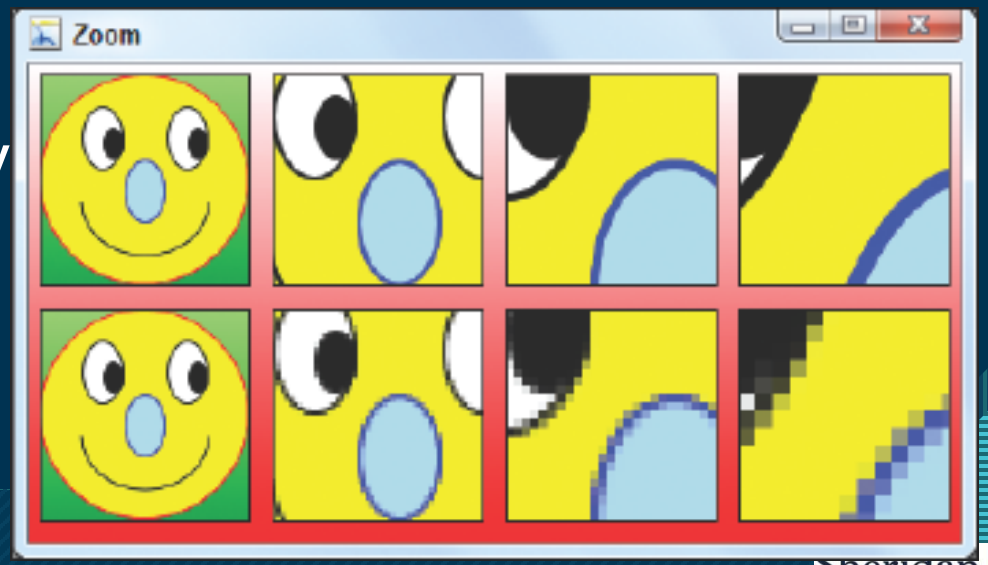
- Microsoft created a way around the limitations of the User32 and GDI/GDI+ libraries: DirectX.
- DirectX began as a cobbled-together, error-prone toolkit for creating games on the Windows platform.
- Its design mandate was speed, and so Microsoft worked closely with video card vendors to give DirectX the hardware acceleration needed for complex textures, special effects such as partial transparency, and three-dimensional graphics.
- Over the years since it was first introduced (shortly after Windows 95), DirectX has matured.
- It's now an integral part of Windows, with support for all modern video cards.
- However, the programming API for DirectX still reflects its roots as a game developer's toolkit.
- Because of its raw complexity, DirectX is almost never used in traditional types of Windows applications (such as business software).

DirectX: The New Graphics Engine

- WPF changes all this.
- In WPF, the underlying graphics technology isn't GDI/GDI+, instead it's DirectX.
- In fact, WPF applications use DirectX no matter what type of user interface you create.
- That means whether you're designing complex three-dimensional graphics or just drawing buttons and plain text, all the drawing work travels through the DirectX pipeline.
- As a result, even the most mundane business applications can use rich effects such as transparency and anti-aliasing.
- You also benefit from hardware acceleration, which simply means DirectX hands off as much work as possible to the graphics processing unit (GPU), which is the dedicated processor on the video card.

Bitmap and Vector Graphics

- WPF takes care of making sure that everything has the right size automatically.
- However, if you plan to incorporate images into your application, you can't be quite as casual.
- For example, in traditional Windows applications, developers use tiny bitmaps for toolbar commands.
- In a WPF application, this approach is not ideal because the bitmap may become blurry as it's scaled up or down according to the system DPI.
- Instead, when designing a WPF user interface, even the smallest icon is generally implemented as a *vector graphic*.
- Vector graphics are defined as a set of shapes, and as such they can be easily scaled to any size.



Separate User Interface and Code-Behind

- One of the biggest goals of WPF was to separate the user interface from the code that lies behind it.
- Using XAML to define the user interface and C# or Visual Basic code to provide the application's functions lets you separate the two tasks of interface construction and programming.

Introduction to XAML

- XAML (pronounced *zammel*) stands for:

eXtensible Application Markup Language

- It is an extension of XML (eXtensible Markup Language).
- Microsoft invented XAML to represent WPF user interfaces in a static language, much as HTML represents the contents of a web page.
- It defines special tokens to represent windows, controls, resources, styles, and other WPF objects.

Introduction to XAML

- All of the usual XML rules apply to XAML files.
- XAML files must have a single root element that contains all of the other elements in the file.
- What element you use as the root element depends on the type of project you are building.
- For example: In a compiled application, the root element is a `Window` that represents the window displayed on the desktop.
- In contrast, a loose XAML page is displayed in a web browser, so the browser plays the role of the window.
- In that case, the root element is typically some container control such as a `Grid` or `StackPanel` that can hold all of the other elements.

- **Note:**
  - These controls are described in detail later, but for now, know that a `Grid` arranges controls in rows and columns, and a `StackPanel` arranges controls in a single row either vertically or horizontally.

Control Sneak Peek

- Each opening element must have a corresponding closing element with the same name but beginning with a slash.
- For example, the following code snippet defines a `StackPanel`:

```
<StackPanel>
</StackPanel>
```

- If an element doesn't need to contain any other elements, you can use a special shorthand and end the opening element with a slash instead of a separate closing element.
- The following snippet shows an `Image` object.
- It doesn't contain any other items, so it uses the shorthand notation.

```
<Image Margin="10" Width="75" Height="75" Source="Volleyball.jpg" />
```

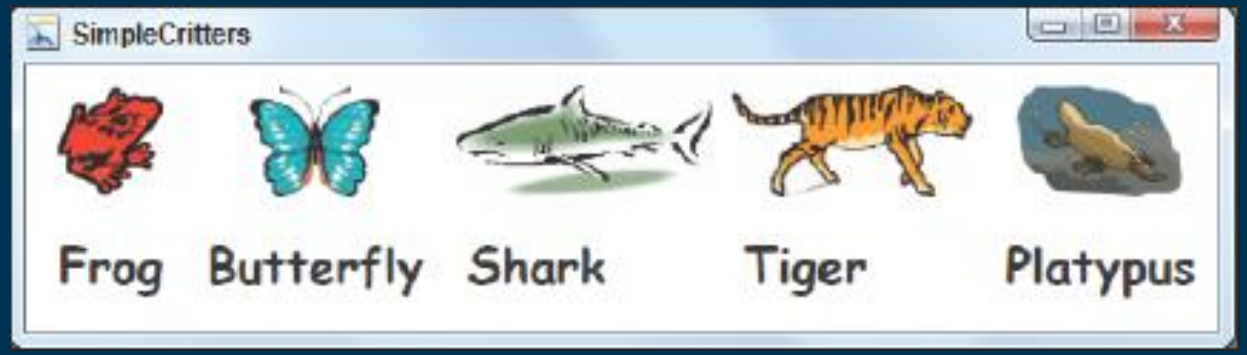
Control Sneak Peek

```
<Image Margin="10" Width="75" Height="75" Source="Volleyball.jpg" />
```

- The preceding snippet also demonstrates attributes.
- A XAML *attribute* is a value contained inside an item's opening tag.
- In this snippet, the `Image` object has attributes `Margin`, `Width`, `Height`, and `Source` with values `10`, `75`, `75`, and `Volleyball.jpg`.

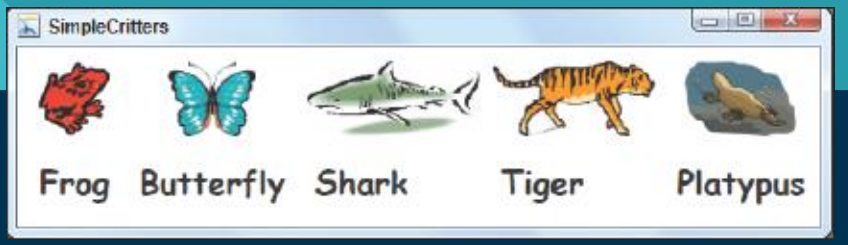
XAML Example

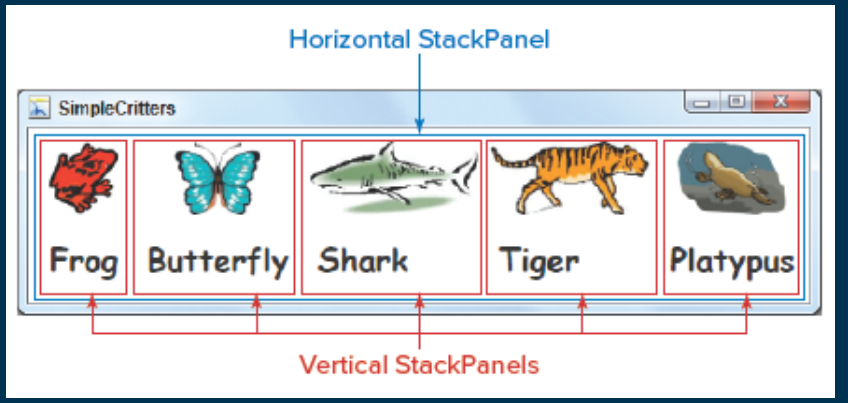
```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="WpfApp.MainWindow"
    x:Name="Window"
    Title="SimpleCritters"
    Width="557" Height="156"
    FontSize="22" FontWeight="Bold" FontFamily="Comic Sans MS">
    <StackPanel Orientation="Horizontal" Margin="5">
        <StackPanel>
            <Image Margin="5" Height="50" Source="Frog.jpg"/>
            <Label Margin="5" Content="Frog"/>
        </StackPanel>
        <StackPanel>
            <Image Margin="5" Height="50" Source="Butterfly.jpg"/>
            <Label Margin="5" Content="Butterfly"/>
        </StackPanel>
        <StackPanel>
            <Image Margin="5" Height="50" Source="Shark.jpg"/>
            <Label Margin="5" Content="Shark"/>
        </StackPanel>
        <StackPanel>
            <Image Margin="5" Height="50" Source="Tiger.jpg"/>
            <Label Margin="5" Content="Tiger"/>
        </StackPanel>
        <StackPanel>
            <Image Margin="5" Height="50" Source="Platypus.jpg"/>
            <Label Margin="5" Content="Platypus"/>
        </StackPanel>
    </StackPanel>
</Window>
```

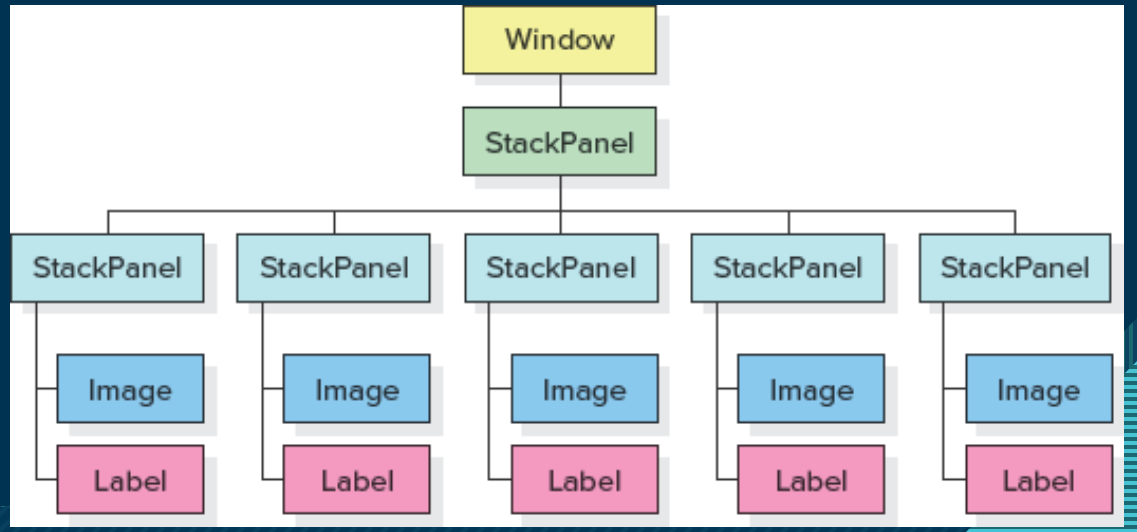


Object Trees

- See how the outer `StackPanel` arranges the inner `StackPanel`s horizontally and how the inner `StackPanel`s arrange their `Image`s and `Label`s vertically.
- The controls that make up a user interface such as the one shown here form a natural hierarchy with some controls containing others, which may then contain others.



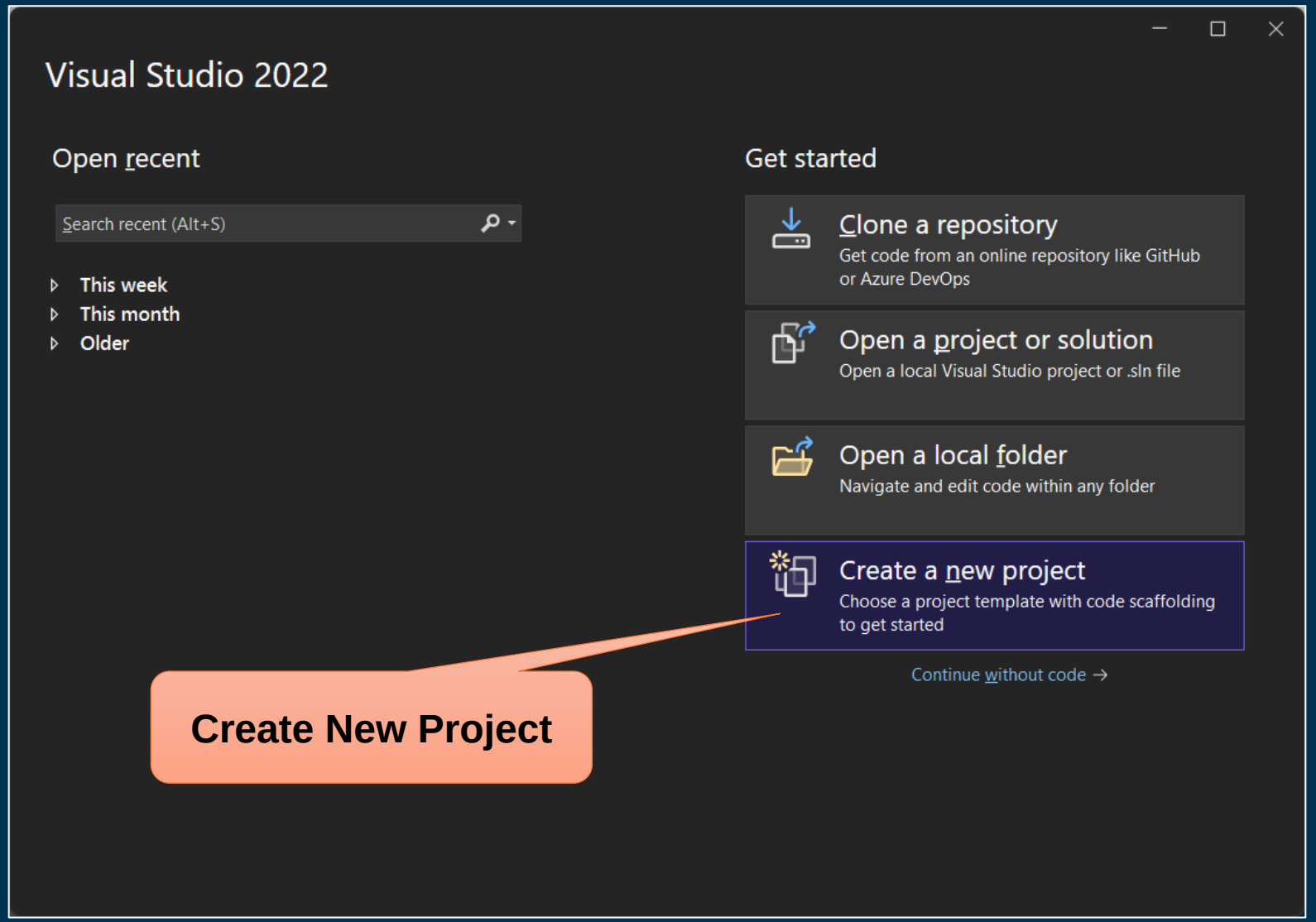




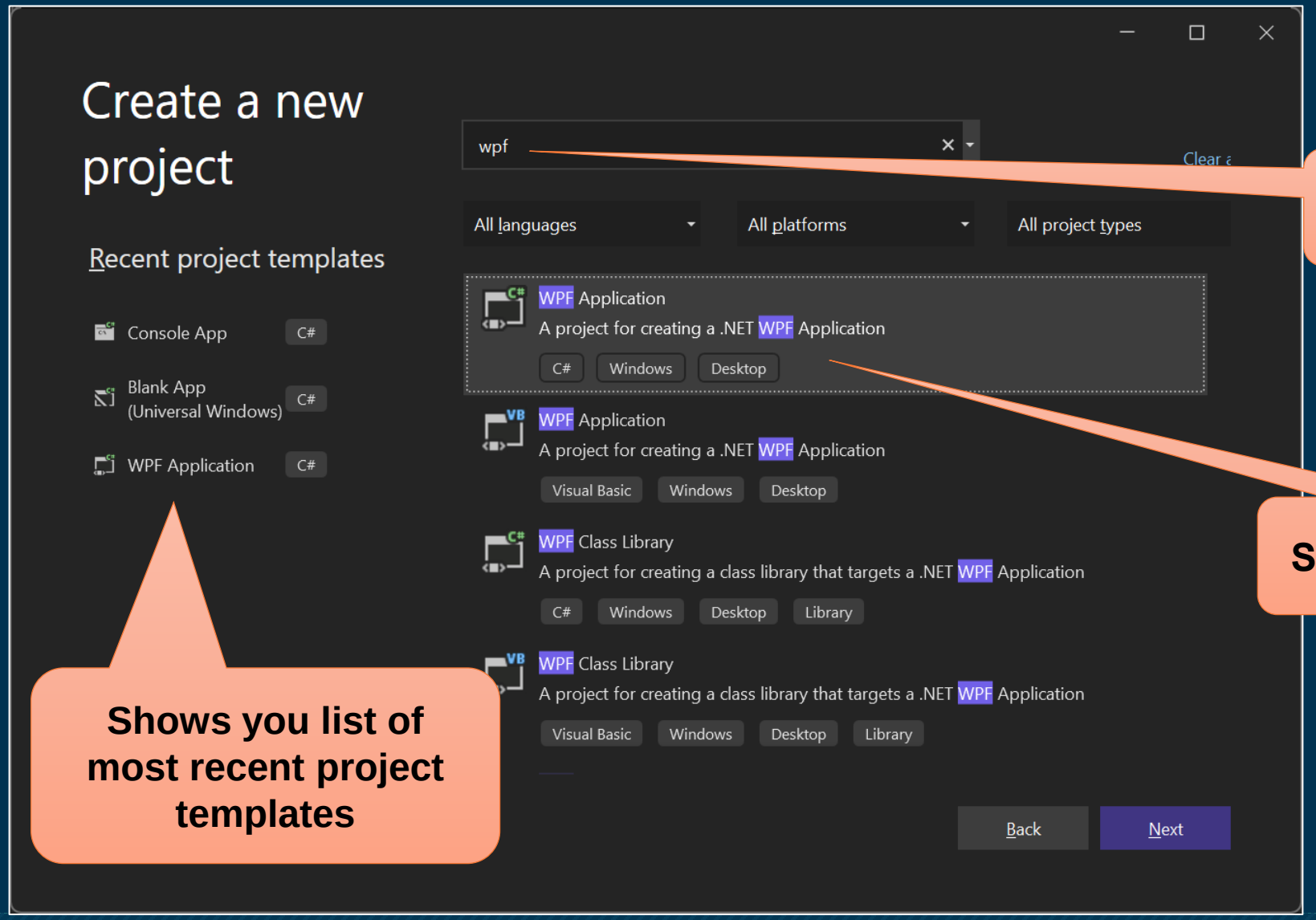
Create WPF Project in Visual Studio

- Visual Studio provides everything you need to build WPF applications.
- It has a WYSIWYG ("what you see is what you get") Window Designer that lets you create a XAML (Extensible Markup Language) interface by dragging controls onto a window.
- Its Code Editors let you write C# or Visual Basic code to sit behind the user interface (UI).
- It can also run the WPF application that you're building so you can see it in action.

Create New WPF Project



Create New WPF Project

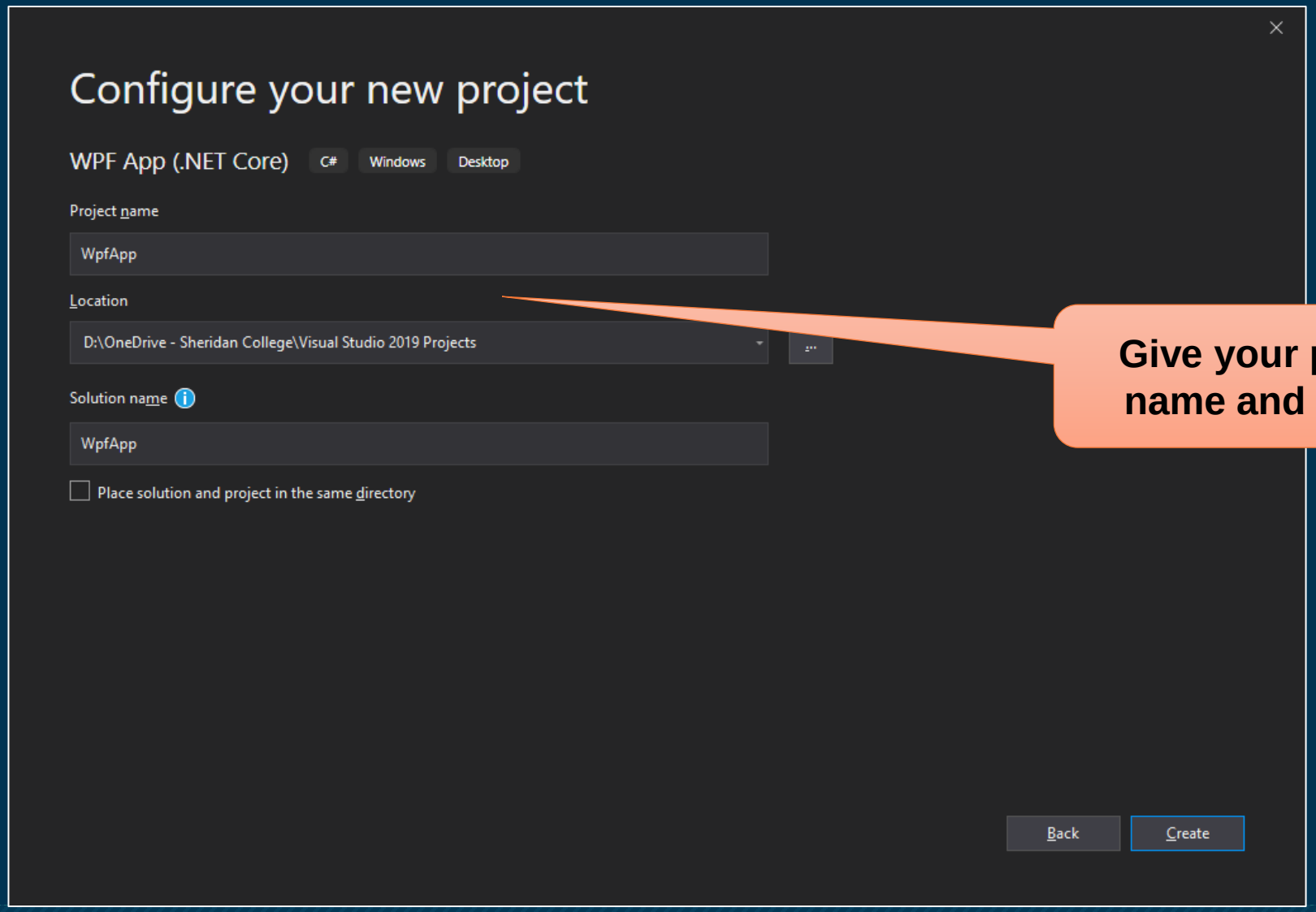


Search for "wpf"

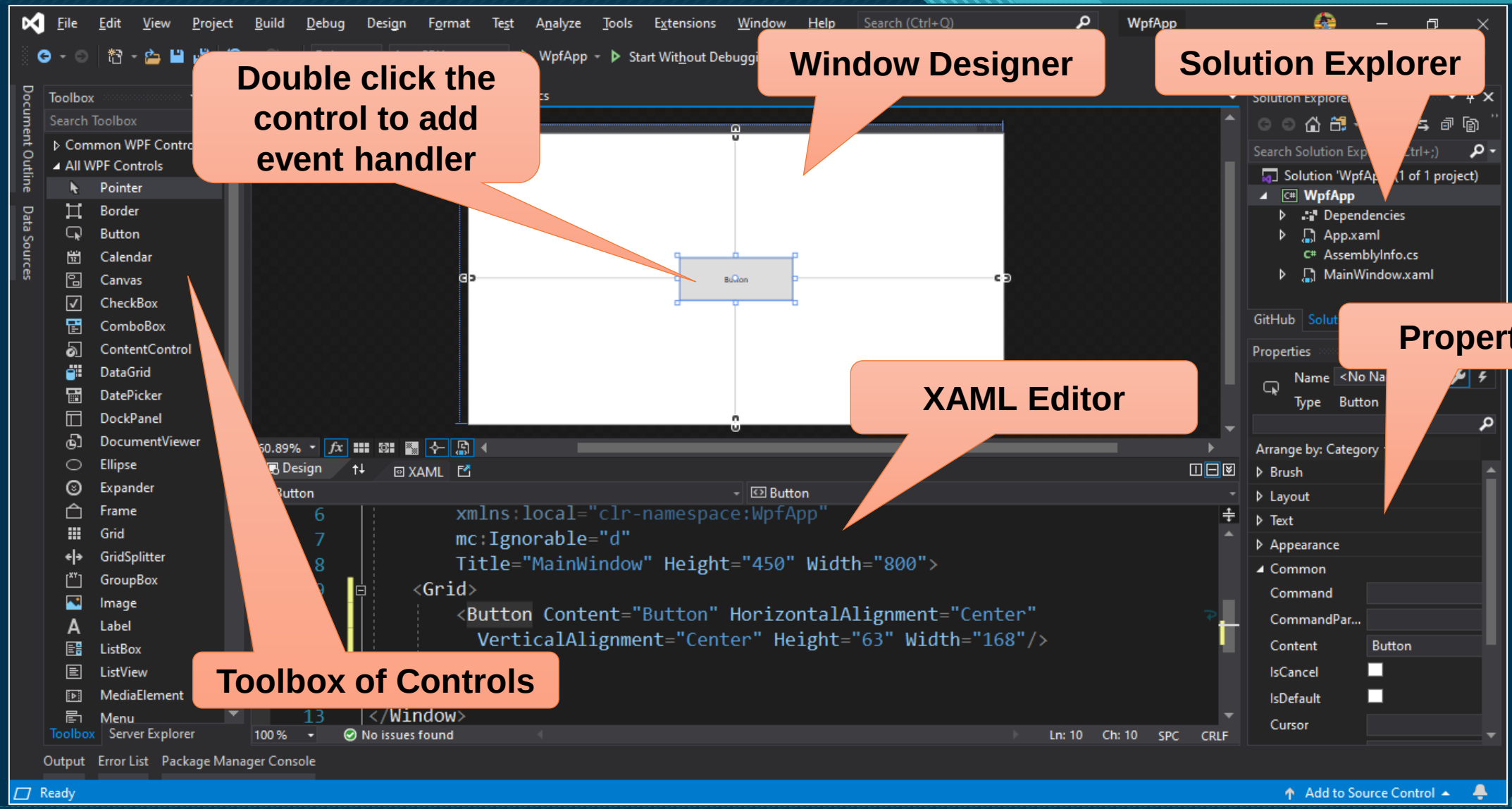
Select "WPF App"

Shows you list of most recent project templates

Create New WPF Project



Create New WPF Project



Window Designer

- The Window Designer lets you build windows graphically.
- It lets you add controls to the window, resize and arrange controls, place controls inside other controls, set control properties, and more.
- Use the Toolbox to select a control type.
- Then click and draw on the Window Designer to place the control on it.
- Alternatively, you can double-click on a tool in the Toolbox to place the control on the window at a default size and location.
- After you place a control on the Designer, you can click on it and drag it into a new position.
- If you create or drag a control on top of a container control such as a `Grid`, `StackPanel`, or `Frame`, the control will be placed inside the container.

Window Designer

Note:

- Be careful when you position controls on the window.
- It's easy to accidentally drag a control inside a container.
- When in doubt, look at the XAML code to see which control contains other controls.
- You can also use the XAML Editor to move controls into other containers if you have trouble dragging them where you want them to be.

XAML Editor

- The XAML Editor lets you manually edit the window's XAML code.
- Usually, it's easier to use the Window Designer to place controls on the window and let the designer generate the XAML for you, but there are occasions when you'll need to edit the XAML code directly.

- For example: Setting exact row and column sizes in a `Grid` can be tricky in the Window Designer, but it's easy in XAML.
- Define some rows and columns in the Window Designer.
- Then open the XAML Editor, find the `RowDefinition` and `ColumnDefinition` elements, and set their `Height` and `Width` attributes as needed.

XAML Editor

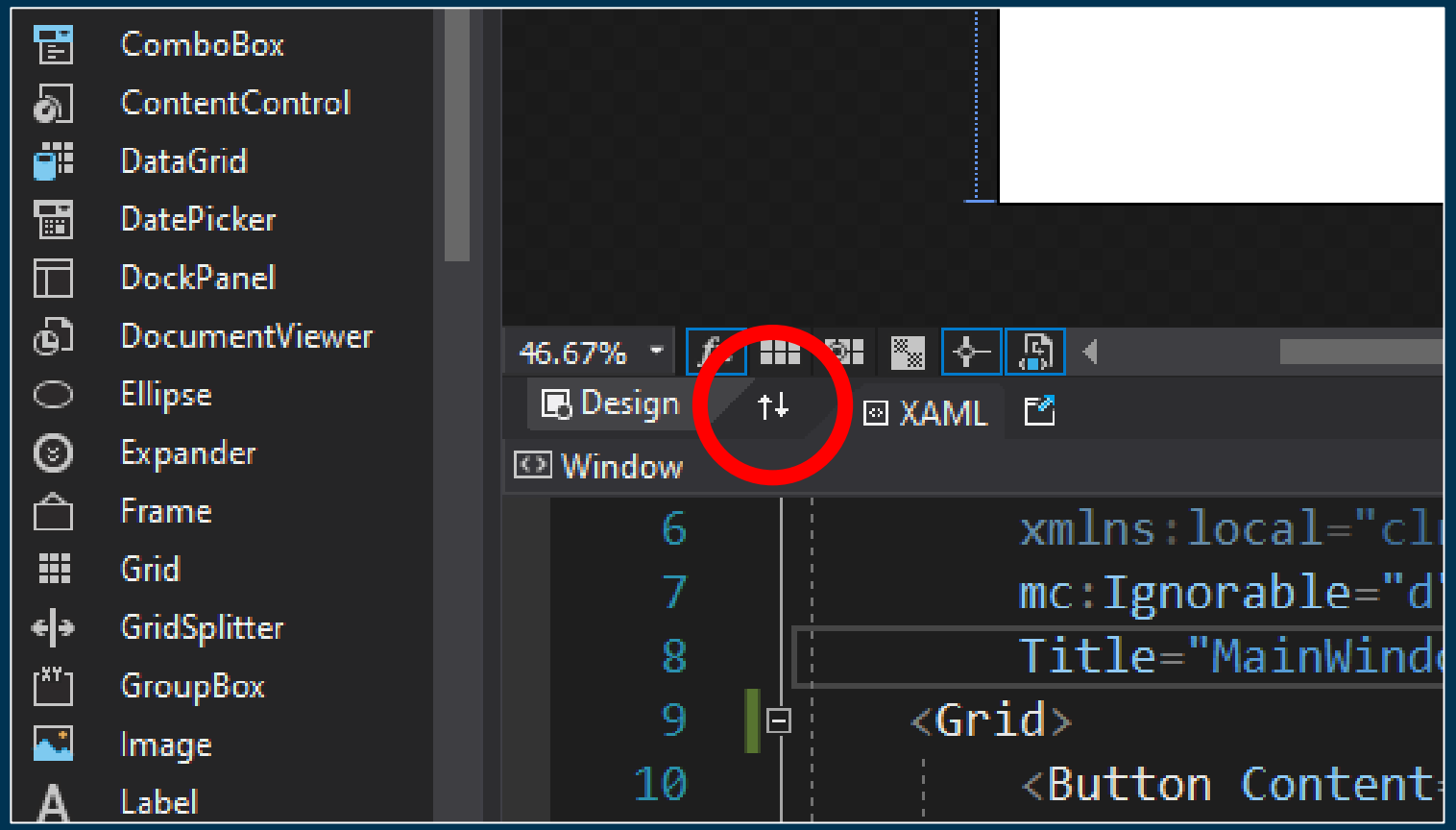
- For example: the following code gives a Grid two rows, one of height 30 pixels and one using all of the remaining vertical space.
- It gives the Grid three columns of equal widths.

```
<Grid>
     <Grid.RowDefinitions>
          <RowDefinition Height="30" />
          <RowDefinition Height="*" />
     </Grid.RowDefinitions>
     <Grid.ColumnDefinitions>
          <ColumnDefinition Width="33*" />
          <ColumnDefinition Width="33*" />
          <ColumnDefinition Width="33*" />
     </Grid.ColumnDefinitions>
</Grid>
```

- It is easier to set these values in XAML Editor rather than Window Designer.

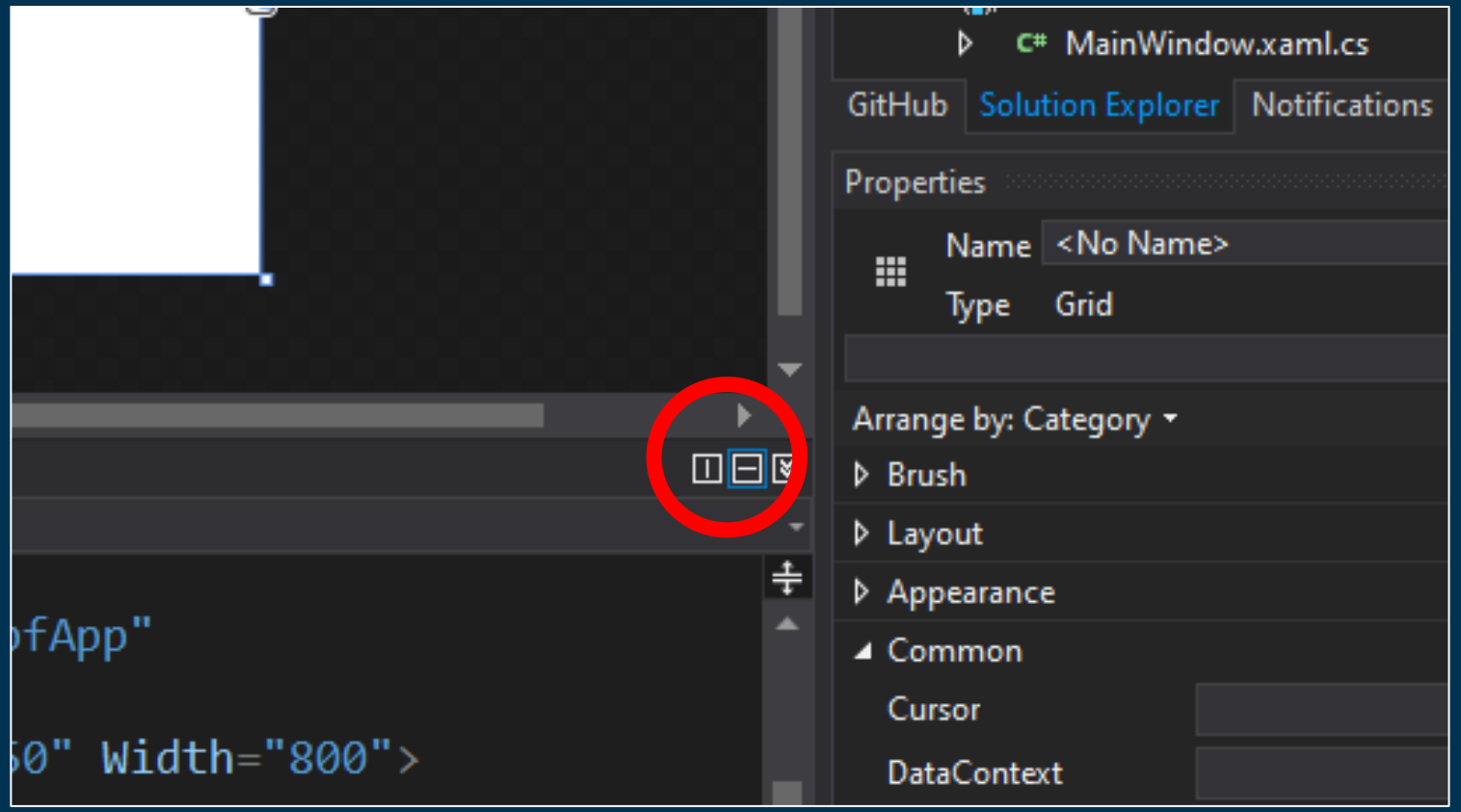
XAML Editor

- Click on the up/down arrow (↑↓) between the two panes to switch which is on top and which is on bottom.
- If you keep the top window large and the bottom one small, this makes it easy to switch between the two.



XAML Editor

- You can also click on these icons to stack the panes horizontally or vertically.

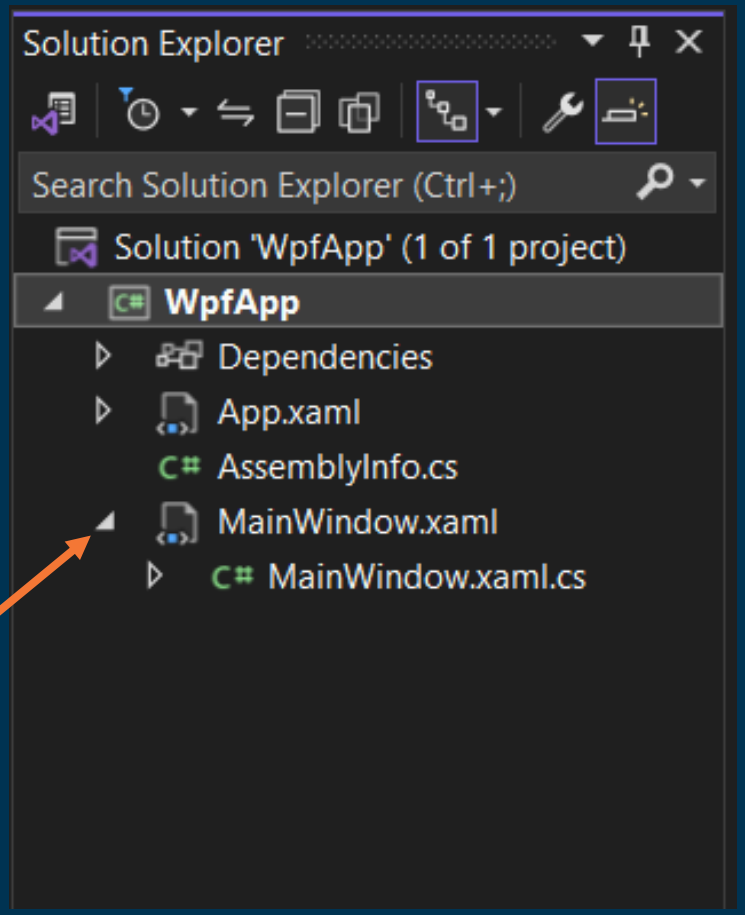


Toolbox

- The Toolbox holds controls that you can place on the window.
- Many of the controls, such as `Label` and `TextBox`, are self-explanatory, and you should have little trouble figuring out how to use them.
- Many of the controls are also similar to Windows Forms controls, so using them will be easy if you have experience with Windows Forms controls.

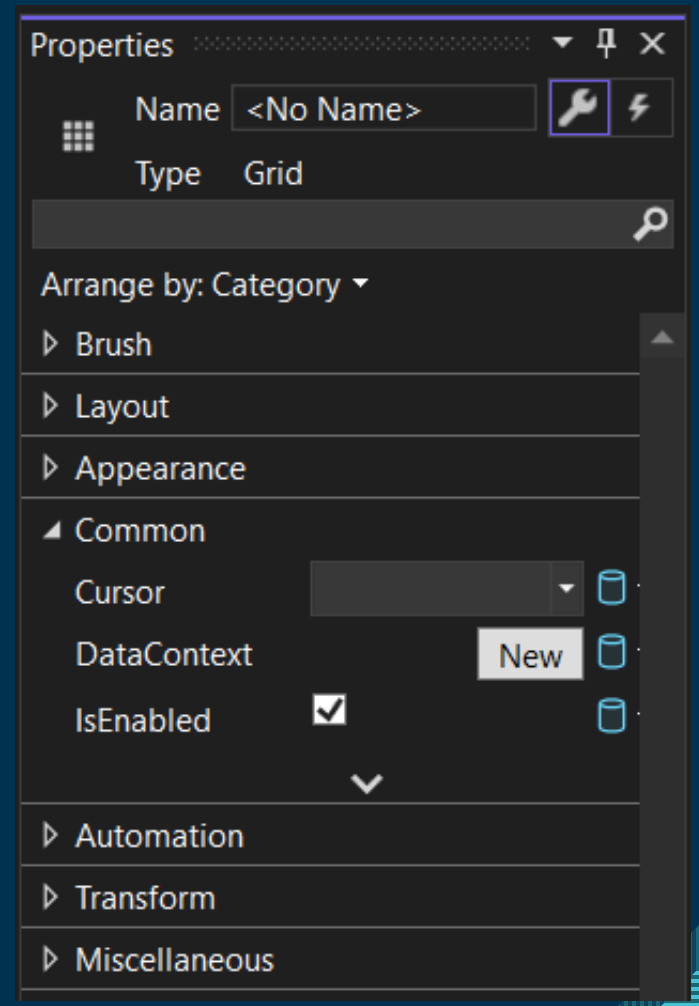
Solution Explorer

- The Solution Explorer lists the application's files.
- Double-click on a file to open it in the appropriate editor.
- For example: If you double-click MainWindow.xaml, Visual Studio opens the file in the Window Designer.
- You can use the Solution Explorer to find other project files such as code-behind files.
- For example: To open the code-behind file for MainWindow.xaml, click on the triangle to the left of that file in the Solution Explorer if necessary, and double-click on the MainWindow.xaml.cs.



Properties Window

- The Properties window lets you view and edit the properties values for the currently selected control in the Window Designer.
- The Properties window can only set simple kinds of properties.
- For example: It can set a control's `Background`, `Fill`, or `Stroke` property to solid colors such as Red, Light Blue, or #FFFF8000 (a dark orange).
- But you can handle the selected control's other properties through XAML code.



Naming Your Controls

- If you need to refer to a control in code, either the XAML code or the C#, then you should give the control a good name.
- Developers often don't give names to controls that are never referred to by the code such as `Label`s that don't change, `GroupBox`es and other decorative controls, layout controls such as `StackPanel`s, and so forth.
- Many C# developers, when naming the control, gives it a prefix that tells the kind of control followed by a descriptive name.
- For example, `txtFirstName` is a `TextBox` that holds a first name.
- Some developers put the type of control at the end of the name as in `firstNameTextBox`.

Code-Behind

- You'll need to associate program code with the UI elements.
- For example, the application will need to take action when the user clicks buttons and selects commands from menus.

Default Event Handlers

- The simplest way to attach code to the user interface is through Window Designer.
- If you want to add code to a control's default event handler (e.g., a `Button`'s `Click` event), simply double-click on the control.
- Visual Studio automatically creates an event handler for the event and opens it in the Code Editor.
- It also adds code to the XAML element to attach the control to the event handler.

Default Event Handlers

- For example, the following code shows the event handler generated when you double-click on a `Button` named `btnDisplay`.

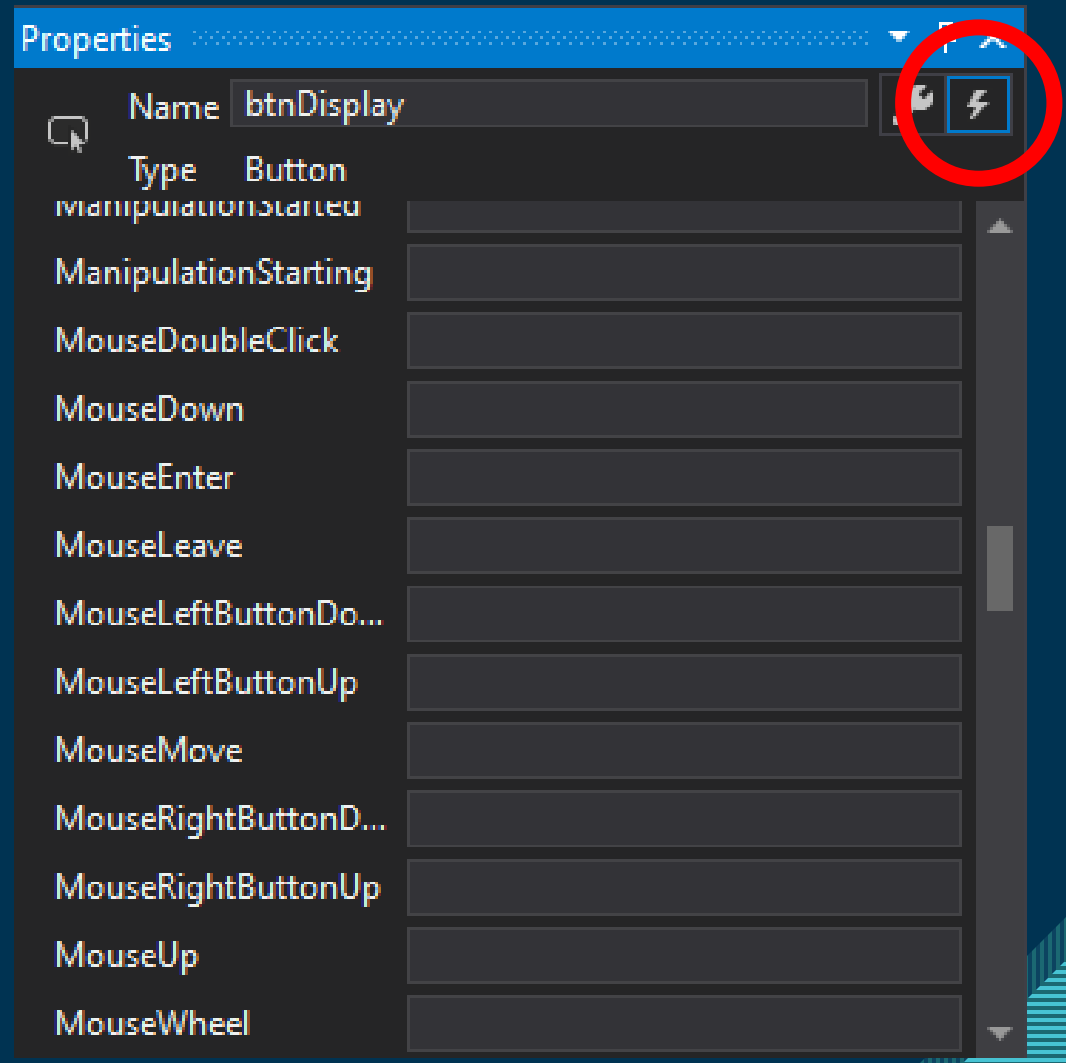
```
private void btnDisplay_Click(object sender, RoutedEventArgs e)
{

}
```

- The following XAML code shows the button's definition.
- Visual Studio automatically added the `Click="btnDisplay_Click"` attribute to tie the button to its event handler.

```
<Button Content="Display" Name="btnDisplay"
 HorizontalAlignment="Center" VerticalAlignment="Center"
 Height="63" Width="168"
 Click="btnDisplay_Click" />
```

Non-Default Event Handlers

- If you want to attach code to an event other than a control's default event (e.g., a `Button`'s `MouseEnter` event), select the control in the Window Designer.
- At the top-right-side of the Properties panel, click the lightning symbol to view the control's events.
- Now, if you want to create a new event handler, simply double-click on the event, and Visual Studio will create an appropriate event handler and open it in the Code Editor.



Common Properties

Size and Position

- Sizing and positioning properties can be very confusing because a control's geometry depends on several different properties, the way in which those properties interact, and even the control's container.
- For example: the same `Button` looks very different when it is placed inside a `Grid`, a `StackPanel`, or a `Canvas`.

Alignment Properties

- Four of the most important properties that determine a control's size are:
  - `Width`
  - `Height`
  - `HorizontalAlignment`
  - `VerticalAlignment`

- The `Width` and `Height` properties are fairly self-explanatory.
- They determine the control's size directly.

- Often, you don't really want to set a control's size explicitly.
- Instead, it would be better for the control to resize itself in some manner to fit either its contents or the area that contains it.
- That lets the control use as little space as possible or take best advantage of the space available.

Alignment Properties

- That's where the `HorizontalAlignment` and `VerticalAlignment` properties come into play.
- `HorizontalAlignment` can take the values:
  - `Left`
  - `Right`
  - `Center`
  - `Stretch`

- `VerticalAlignment` can take the values:
  - `Top`
  - `Bottom`
  - `Center`
  - `Stretch`

Alignment Properties

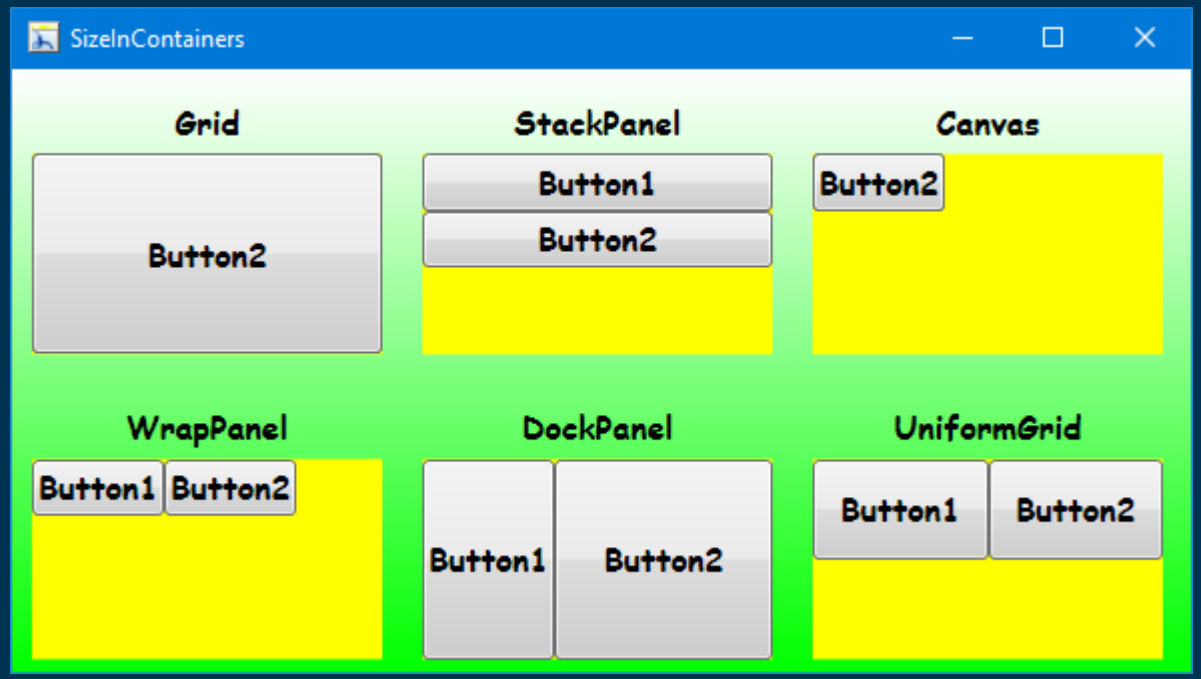
- The `Left`, `Right`, `Top`, `Bottom`, and `Center` values determine the control's position in an obvious way.
- For example, if a control has these properties set to `Left`/`Top`, then it is placed in the upper-left corner of its container.
- The `Stretch` value is a bit less intuitive.
- In general, this value means that the control should expand to fill its container, but the exact result depends on the type of the control's container.

Alignment Properties

- For example: If the container is a `Grid`, then the control stretches to fill its `Grid` cell as expected.
- In contrast, if the container is a `StackPanel` oriented vertically, then setting `HorizontalAlignment = Stretch` makes the control expand to fill the width of the `StackPanel`, but the `VerticalAlignment` property is ignored.
- If you don't explicitly set the control's `Height` property, then the control will be as short as possible while still displaying its contents.

Example

- This example displays `Button` controls inside various types of containers.
- The containers have yellow backgrounds where possible so you can see them.
- The `Buttons` in each container are the same, and all have `HorizontalAlignment` and `VerticalAlignment` set to `Stretch`.

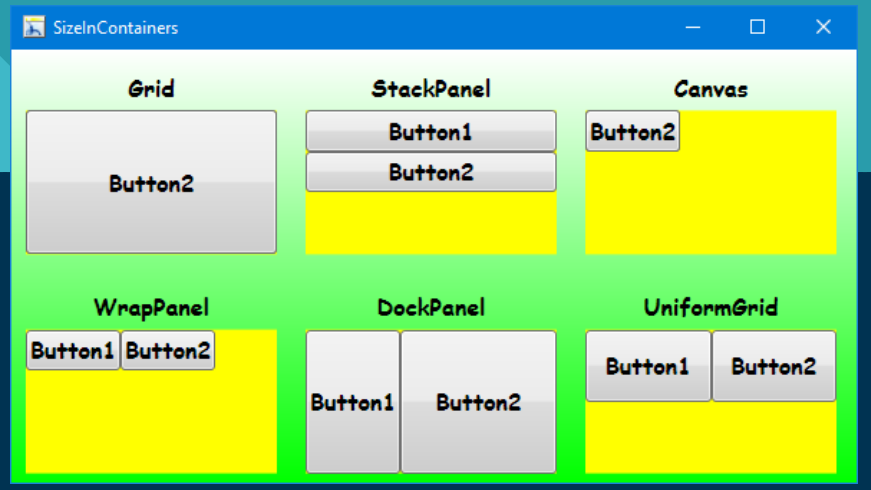


Example

- The following XAML code shows how the program creates its `Grid` container together with its `Label` and `Button`s.
- Notice that the `Grid`'s size is explicitly set with its `Width` and `Height` properties.
- The code for the other containers is similar except they use other controls in place of the `Grid`.

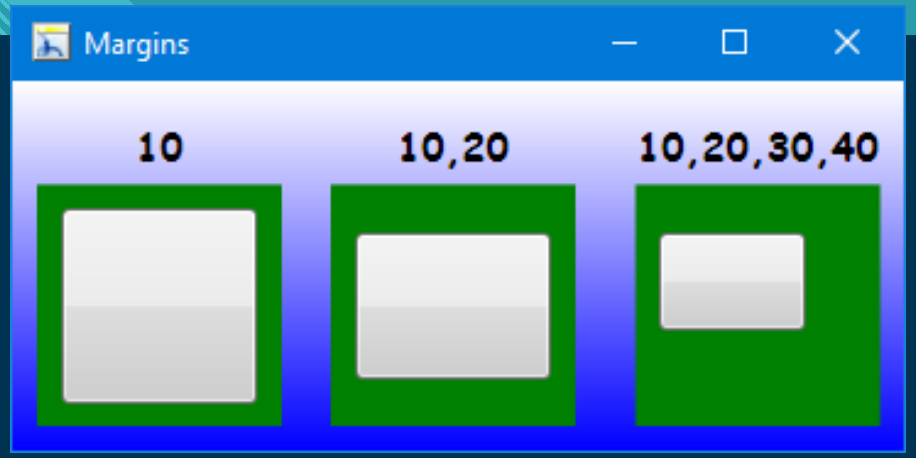
```
<Label Content="Grid" HorizontalAlignment="Center"/>

<Grid Background="Yellow" Width="175" Height="100">
   <Button Content="Button1"
           HorizontalAlignment="Stretch"
           VerticalAlignment="Stretch" />
   <Button Content="Button2"
           HorizontalAlignment="Stretch"
           VerticalAlignment="Stretch" />
</Grid>
```

Example



- The following list summarizes the results in the different containers:
  - `Grid`: The `Buttons` stretch to fill the `Grid`. The `Buttons` both fill the `Grid` so you only see the one on top: `Button2`.
  - `StackPanel`: The `Buttons` stretch to fill the `StackPanel`'s width (because its orientation is vertical) and are as short as they can be while holding their content.
  - `Canvas`: The `Buttons` are as small as possible while holding their content. Without other positioning information, the `Canvas` places them both in its upper-left corner so you can only see the one on top: `Button2`.
  - `WrapPanel`: The `Buttons` are as small as possible while holding their content. The `WrapPanel` arranges them in order so you can see them both.
  - `DockPanel`: By default, the first `Button` is docked to the control's left edge, so it is stretched to fill the `DockPanel` vertically, and it is as thin as possible while still holding its content. By default, the last control is stretched to fill the `DockPanel`'s remaining space.
  - `UniformGrid`: The `UniformGrid` divides its space up evenly into cells, places each `Button` in a cell, and stretches them to fill their cells.

The `Margin` Property



- The `Margin` property determines how much extra space is added around the control.
- That influences the control's position, and, if the control is stretching to fit its container, it also reduces the control's size.
- `Margin` can take the form of one, two, or four numbers.
  - If the value includes a single number, then all four edges of the control are given that much extra space.
  - If the value includes two numbers, the first gives the left and right margins, and the second gives the top and bottom margins.
  - If the value includes four numbers, they give the left, top, right, and bottom margins respectively.

The `Padding` Property

- Some controls such as `Border`, `Button`, and `TextBox` also provide a `Padding` property.
- This property has a syntax similar to that of `Margin`, but it specifies padding to add inside the control around its contents.
- For example: Setting `Padding = "10"` adds 10 pixels of extra space around the text inside a `TextBox`.

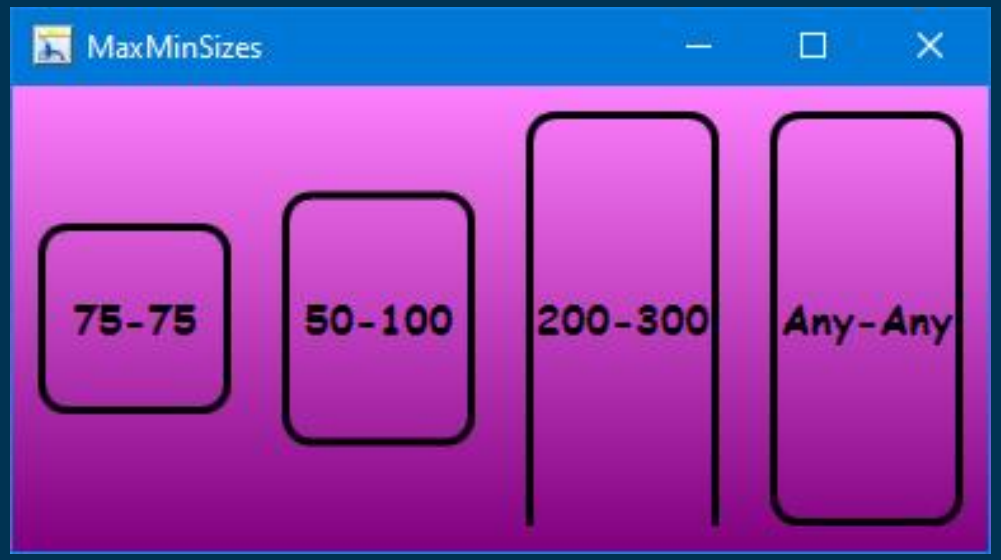
The Max and Min Height and Width Properties

- Four other properties that influence a control's size are:
  - `MaxWidth`
  - `MaxHeight`
  - `MinWidth`
  - `MinHeight`

- As you can guess from their names, they determine how big and how small a control can be when its container is resized.

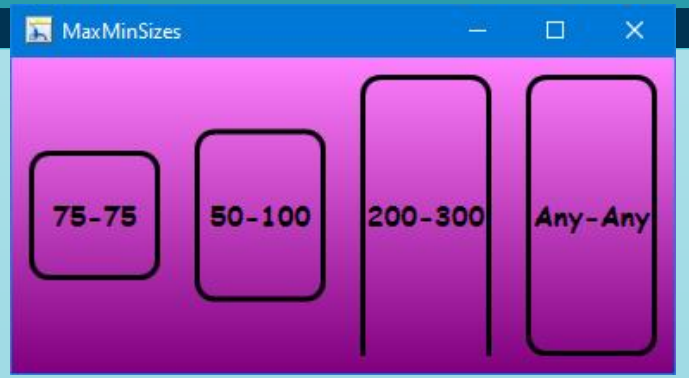
The Max and Min Height and Width Properties

- The first rectangle's `Height` is set to 75, so it always has that height.
- The second rectangle's `MaxHeight` value is set to 100.
- The window is big enough for the rectangle to be taller than this, but its `MaxHeight` value limits it.



- The third rectangle's `MinHeight` value is 200.
- That's too tall to fit completely on the window, so this rectangle is clipped.
- The final rectangle does not have any `MinHeight` or `MaxHeight` values, so it grows and shrinks to fit its part of the window.

The Max and Min Height and Width Properties




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

    <Label Content="75-75" Grid.Column="0" HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
    <Rectangle Grid.Column="0" Margin="10" Stroke="Black" Height="75"/>

    <Label Content="50-100" Grid.Column="1" HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
    <Rectangle Grid.Column="1" Margin="10" Stroke="Black" MinHeight="50" MaxHeight="100"/>

    <Label Content="200-300" Grid.Column="2" HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
    <Rectangle Grid.Column="2" Margin="10" Stroke="Black" MinHeight="200" MaxHeight="300"/>

    <Label Content="Any-Any" Grid.Column="3" HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
    <Rectangle Grid.Column="3" Margin="10" Stroke="Black" /> <!-- No MinHeight or MaxHeight set -->
</Grid>
```

The Font Properties

- The following list summarizes the most useful font properties:
  - `FontFamily`: Determines the text's font face. Some common values include Segoe (pronounced see-go, this is Microsoft's preferred font for Windows 10), Times New Roman, Courier New, and Arial.
  - `FontSize`: Determines the size of the font in pixels.
  - `FontStyle`: Determines the text's style. This property can take the values `Normal` and `Italic`.
  - `FontWeight`: Determines the text's density (whether it's bold or not). This property can take the values (in increasing order of density) `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, and `ExtraBlack`.

The Color Properties

- For drawing controls such as `Ellipse`, `Rectangle`, and `Path`:
  - `Stroke` property determines the brush used to draw the shape's outline.
  - `Fill` property determines how the shape is filled.

- For non-drawing controls such as `Label`, `TextBox`, and `StackPanel`:
  - `Background` property determines how the control's interior is filled.
  - `Foreground` property determines the text’s color.

Exercise 1

- Create a WPF app that asks the user for two numbers and displays the sum of those two numbers.
- Add two textboxes where the user can enter numbers.
- Add a label where the result will be displayed.
- Clicking on a button displays the result in a label.

- Extend the app so that it performs various arithmetic operations on those two number.
- Add buttons to perform subtraction, multiplication and division.
- Clicking on these buttons perform the respective arithmetic operations and displays the result.

Exercise 2

- Create a WPF app that asks the user for the discount percent and subtotal and finds the total payable amount.
- Add two textboxes where the user can enter the discount percent and subtotal.
- Add a button to find the total payable.
- Clicking on a button reads the inputs, performs the calculations and displays the result in a label.

Exercise 3

- Create a WPF app that converts Canadian Dollars to US Dollars.
- Ask the user to enter the Canadian Dollars in a textbox.
- Add a button Convert.
- Clicking on this button, read the input from the textbox, do the calculation and display the result in a message box.

- Assume the conversion rate as:
  - 1 Canadian Dollar = 0.75 US Dollar

- **Hint:**
  - `usDollar = caDollar * 0.75;`

Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

References

Material has been taken from:

- WPF Programmer's Reference: Windows Presentation Foundation with C# 2010 and .NET 4:
- https://learning.oreilly.com/library/view/wpf-programmers-reference/9780470477229/ch01.html

- WPF 4.5 Unleashed:
- https://learning.oreilly.com/library/view/wpf-45-unleashed/9780133497076/part01.html

- Exercises taken from Prof. Jawaad Sheikh’s material.