عنوان مضمون

Visual Programming-II

توسط : صفری

بهار1398

Basic Desktop Programming

creating user interfaces

- Over the past 10 years, Visual Studio has provided the Windows developers with a **couple of choices for creating user interfaces:**
- **Windows Forms**

  which is a basic tool for creating applications that target classic Windows,
- **Windows Presentation Foundations (WPF)**

  which provide a wider range of application types and attempts to solve a number of problems with Windows Forms.
- WPF is technically platform-independent, and some of its flexibility can be seen in the fact that a subset of WPF called Silverlight is used to create interactive web applications.

graphical Windows applications

- At the heart of the development of most graphical Windows applications is the Window Designer.
- You create a user interface by dragging and dropping controls from a Toolbox to your window, placing them where you want them to appear when you run the application.
- With WPF this is only partly true, as the user interface is in fact written entirely in another language called **Extensible Application Markup Language** (**XAML**, pronounced *zammel*).
- Visual Studio allows you to do both and as you get more comfortable with WPF, you are likely going to combine **dragging and dropping controls** with **writing raw XAML.**

XAML

- XAML is a language that uses XML syntax and enables controls to be added to a user interface in a declarative, hierarchical way.
- That is to say, you can add controls in the form of XML elements, and specify control properties with XML attributes.
- You can also have controls that contain other controls, which is essential for both layout and functionality

- XAML is designed with today’s powerful graphics cards in mind, and as such it enables you to use all the advanced capabilities that these graphics cards offer through DirectX. The following lists some of these capabilities:

➤ Floating-point coordinates and vector graphics to provide layout that can be scaled, rotated, and otherwise transformed with no loss of quality

➤ 2D and 3D capabilities for advanced rendering

➤ Advanced font processing and rendering

➤ Solid, gradient, and texture fills with optional transparency for UI objects

➤ Animation storyboarding that can be used in all manner of situations, including user-triggered events such as mouse clicks on buttons

➤ Reusable resources that you can use to dynamically style controls

Separation of Concerns

- **Problem:**
  - One problem that exists with **maintaining Windows applications** that has been written over the years is that they very often mix the **code that generates the user interface** and the **code that executes based on users actions**.
  - This makes it difficult for multiple developers and designers to work on the same project.
- WPF solves this in two ways.
- First, by using XAML

  to describe the GUI rather than C#, the GUI becomes platform independent, and you can in fact render XAML without any code whatsoever.
- Second, this means that it feels natural to place the C# code in a different file than you place the GUI code.

  Visual Studio utilizes something called *code-behind* files, which are C# files that are dynamically linked to the XAML files.

- Because the GUI is separated from the code, it is possible to create tailor-made applications for designing the GUI, and this is exactly what Microsoft has done.
- Microsoft provides two important tools for WPF application development:

1. Visual Studio
2. Expression Blend

    is the favored tool used by designers when creating GUIs for WPF.

    - This tool can load the same projects as Visual Studio, but where Visual Studio targets the developer more than the designer, the opposite is true in Expression Blend.
    - This means that on large projects with designers and developers, everyone can work together on the same project, using their preferred tool without fear of inadvertently influencing the others.

XAML in Action

- As stated, XAML is XML, which means that as long as the files are fairly small, it is possible to see immediately what it is describing. Take a look at this small example and see if you can tell what it does:

```
<Window x:Class="Ch15Ex01.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>
    <Button Content="Hello World"
            HorizontalAlignment="Left"
            Margin="220,151,0,0"
            VerticalAlignment="Top"
            Width="75"/>
  </Grid>
</Window>
```

Namespaces

- The Window element of the previous example is the root element of the XAML file.
- This element usually includes a number of namespace declarations. By default, the Visual Studio designer includes two namespaces that you should be aware of:
- http://schemas.microsoft.com/winfx/2006/xaml/presentation
  - is the default namespace of WPF and declares a lot of controls that you are going to use to create user interfaces.
- http://schemas.microsoft.com/winfx/2006/xaml.
  - declares the XAML language itself.

Namespaces don’t have to be declared on the root tag, but doing so ensures that their content can be easily accessed throughout the XAML file, so there is rarely any need to move the declarations.

- The last namespace that you will see quite often is the system namespace: xmlns:sys=″clr-namespace:System;assembly=mscorlib″.
- This namespace allows you to use the built-in types of the .NET Framework in your XAML. By doing this, the markup you write can explicitly declare the types of elements you are creating. For example, it is possible to declare an array in markup and state that the members of the array are strings:

```
<Window.Resources>
  <ResourceDictionary>
    <x:Array Type="sys:String" x:Key="localArray">
      <sys:String>"Karli Watson"</sys:String>
      <sys:String>"Jacob Vibe Hammer"</sys:String>
      <sys:String>"Christian Nagel"</sys:String>
      <sys:String>"Job D. Reid"</sys:String>
      <sys:String>"Morgan Skinner"</sys:String>
    </x:Array>
  </ResourceDictionary>
</Window.Resources>
```

Code-Behind Files

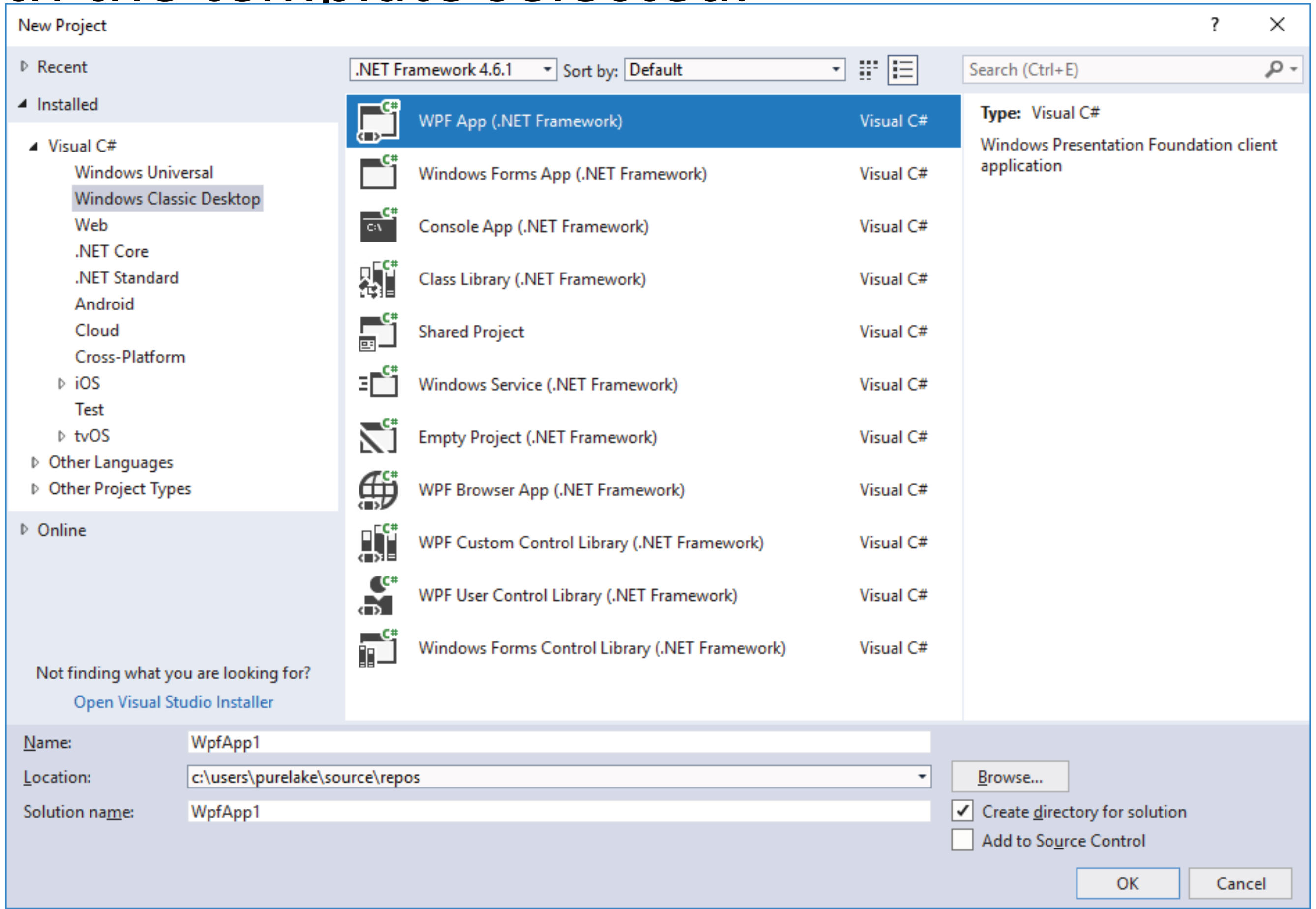
- Although XAML is a powerful way to declare user interfaces, it is not a programming language.
- Whenever you want to do more than presentation, you need C#. It is possible to embed C# code directly into XAML, but mixing code and markup is never recommended and you will not see it done in this book.
- What you will see quite a lot is the use of code-behind files.
- These files are normal C# files that have the same name as the XAML file, plus a .cs extension. Although you can call them whatever you like, it's best to stick to this naming convention.
- Visual Studio creates code behind files automatically when you create a new window in your application, because it expects you to add code to the window. It also adds the x:Class property to the Window tag in the XAML:

```
<Window x:Class="Ch14Ex01.MainWindow"
```

- This tells the compiler that it can find the code for this window in, not a file, but the class Ch14Ex01.MainWindow . Because you can specify only the fully qualified class name, and not the assembly in which the class is found, it is not possible to put the code-behind file somewhere outside of the project in which the XAML is defined. Visual Studio puts the code-behind files in the same directory as the XAML files so you never have to worry about this while working in Visual Studio.

THE PLAYGROUND

- Start by creating a new WPF project by selecting File ➪ New ➪ Project. From the New Project dialog box, select the project template WPF App (.Net Framework).follow Figure shows the New Project Dialog with the template selected.



New Project

Recent
Installed
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Android
Cloud
Cross-Platform
iOS
Test
tvOS
Other Languages
Other Project Types
Online

Not finding what you are looking for?
Open Visual Studio Installer

.NET Framework 4.6.1 Sort by: Default

WPF App (.NET Framework)	Visual C#
Windows Forms App (.NET Framework)	Visual C#
Console App (.NET Framework)	Visual C#
Class Library (.NET Framework)	Visual C#
Shared Project	Visual C#
Windows Service (.NET Framework)	Visual C#
Empty Project (.NET Framework)	Visual C#
WPF Browser App (.NET Framework)	Visual C#
WPF Custom Control Library (.NET Framework)	Visual C#
WPF User Control Library (.NET Framework)	Visual C#
Windows Forms Control Library (.NET Framework)	Visual C#

Search (Ctrl+E)

Type: Visual C#
Windows Presentation Foundation client application

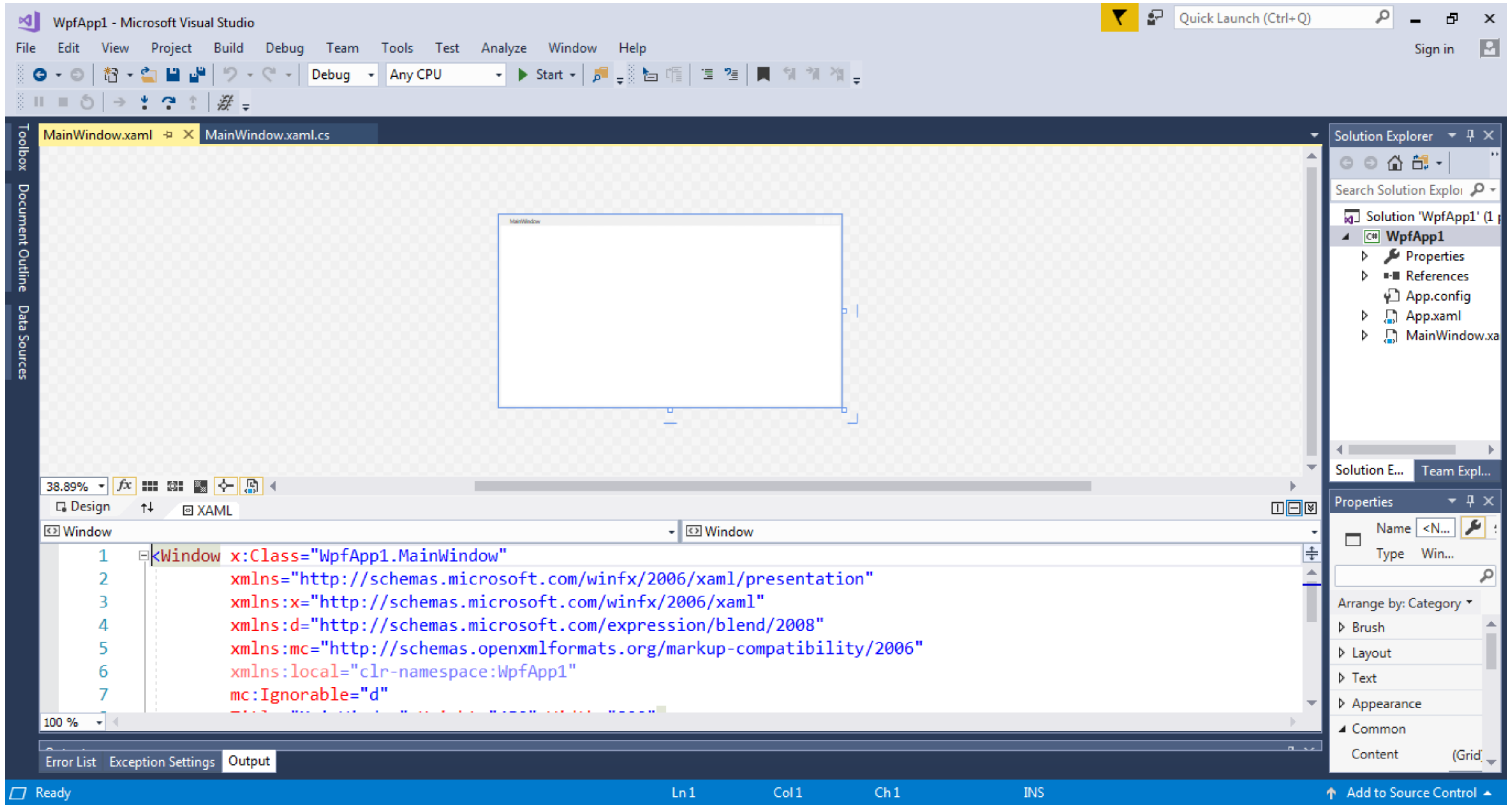
Name: WpfApp1
Location: c:\users\purelake\source\repos
Solution name: WpfApp1

Browse...
[✓] Create directory for solution
[ ] Add to Source Control

OK Cancel

- Visual Studio now displays an empty window and a number of panels around it.
- The greater part of the screen is divided in two sections.

1. The upper section,
   known as the Design View, displays a WYSIWYG (What You See Is What You Get) representation of the window you are designing
2. the lower section,
   known as the XAML View, displays a textual representation of the same window.



```xml
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApp1"
        mc:Ignorable="d"
```

WPF Controls

- Controls combine **prepackaged** code and **a GUI** that can be reused to create more complex applications.
- They can define how they draw themselves by default and a set of standard behaviors.
- Some controls, such as the Label, Button, and TextBox controls are easily recognizable and have been used in Windows applications for about 20 years.
- Others, such as Canvas and StackPanel, don’t display anything and simply help you organize the GUI.

dependency property

- WPF uses normal properties that you have seen before and adds a new type of property called a ***dependency property***. the properties of WPF do more than just get and set a value; for one, they are able to notify observers of changes.

Try It Out

- Adding Controls to a Window:

  you will add controls to the Design View by dragging them from the Toolbox panel or by typing the XAML manually.

1. Start by dragging a Button control from the Toolbox onto the Design View.

   Visual Studio will try to set properties and insert child elements to allow the controls to display themselves in a standard way.

2. Now drag another Button, but this time drop it in the XAML View below the first Button, but above the </Grid> tag.

   button expands to fill the entire window.

Try It Out

- Properties:
- As mentioned, all controls have a number of properties that are used to manipulate the behavior of the control.
- Some of these are easy to understand such as height and width, whereas others are less obvious such as RenderTransform.
- All of them can be set using:

1. the Properties panel,
2. directly in XAML,
3. by manipulating the control on the Design View.

- The following Try It Out demonstrates setting control properties in the Design View.

Try It Out

1. Start by selecting the second Button control in Design View; this is the button that is currently filling the entire window.
2. You can change the name of the control in the Properties panel at the very top. Change it to rotatedButton.
3. Under the Common node, change the Content to 2nd Button.
4. Under Layout, change width to 75 and height to 22.
5. Expand the Text node and change the text to bold by clicking the B icon.
6. Select the first button and drag it to a position above the second button. Visual Studio will assist with the positioning by snapping the control.
7. Select the second button again, and hover the mouse pointer over the top-left corner of it. The pointer changes to a quarter-circle with arrows on both ends. Drag down until the button is tilted down.

Dependency Properties

- Actions users take on a dialog, ➡ cause other controls to change and update their display or content.
- For the most part, normal .NET properties are simple getters and setters, which do not have the ability to inform other controls that they have changed.
- Enter Dependency Properties. A *dependency property* is a property that is registered with the WPF property system in such a way that it allows **extended functionality**.
- This extended functionality includes, but is not limited to, automatic property change notifications. Specifically, dependency properties have the following features:

➤ You can use styles to change the values of dependency properties.

➤ You can set the value of a dependency property by using resources or by data binding.

➤ You can change dependency property values in an animation.

➤ You can set dependency properties hierarchically in XAML—that is, a value for a dependency property that you set on a parent element can be used to set the default value for the same dependency property of its child elements.

➤ You can configure notifications for property value changes using a well-defined coding pattern.

➤ You can configure sets of related properties so that they all update in response to a change to one of them. This is known as *coercion*. The changed property is said to *coerce* the values of the other properties.

➤ You can apply metadata to a dependency property to specify other behavior characteristics. For example, you might

Events

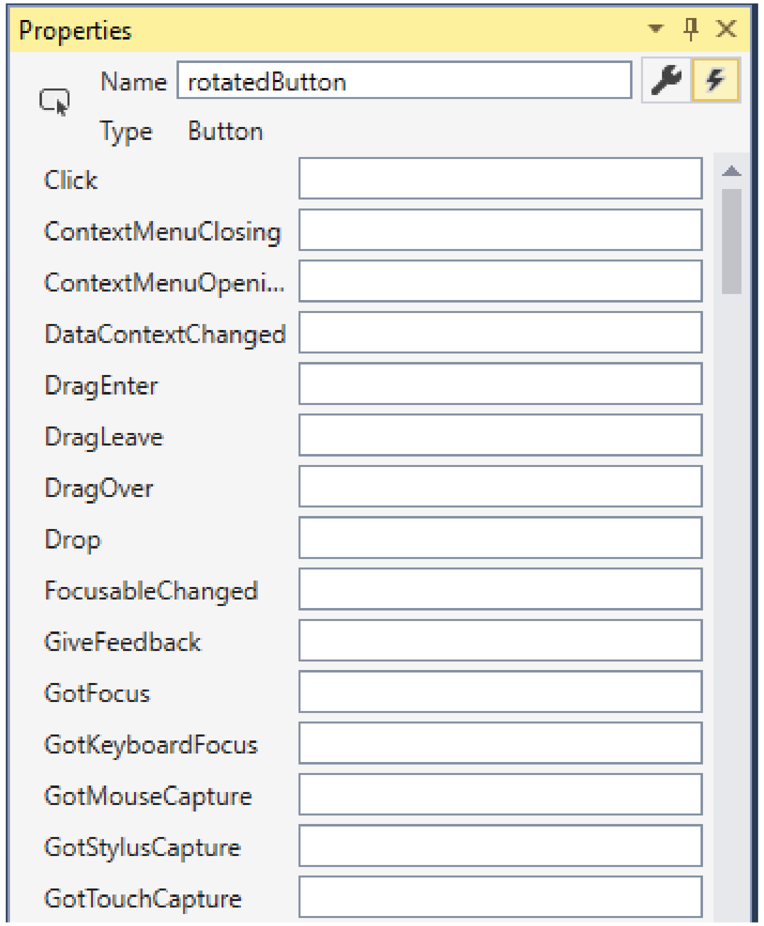
- This section covers particular kinds of events—specifically, the events generated by WPF controls—and introduces ***routed events****,* which are usually associated with user actions.
  - For example, when the user clicks a button, that button generates an event indicating what just happened to it. Handling the event is the means by which the programmer can provide some functionality for that button.
- Many of the events you handle are exposed by most of the controls that you work with in this book. This includes events such as LostFocus and MouseEnter. This is because the events themselves are inherited from base classes such as Control or ContentControl. Other events such as the CalendarOpened event of the DatePicker are more specific and only found on specialized controls.

EVENT	DESCRIPTION
`Click`	Occurs when a control is clicked. In some cases, this event also occurs when a user presses the Enter or space keys.
`Drop`	Occurs when a drag-and-drop operation is completed—in other words, when an object has been dragged over the control, and the user releases the mouse button.
`DragEnter`	Occurs when an object being dragged enters the bounds of the control.
`DragLeave`	Occurs when an object being dragged leaves the bounds of the control.
`DragOver`	Occurs when an object has been dragged over the control.
`KeyDown`	Occurs when a key is pressed while the control has focus. This event always occurs before `KeyPress` and `KeyUp`.
`KeyUp`	Occurs when a key is released while a control has focus. This event always occurs after `KeyDown` event.
`GotFocus`	Occurs when a control receives focus. Do not use this event to perform validation of controls. Use `Validating` and `Validated` instead.
`LostFocus`	Occurs when a control loses focus. Do not use this event to perform validation of controls. Use `Validating` and `Validated` instead.
`MouseDoubleClick`	Occurs when a control is double-clicked.
`MouseDown`	Occurs when the mouse pointer is over a control and a mouse button is pressed. This is not the same as a `Click` event because `MouseDown` occurs as soon as the button is pressed and *before* it is released.
`MouseMove`	Occurs continually as the mouse travels over the control.
`MouseUp`	Occurs when the mouse pointer is over a control and a mouse button is released.

Handling Events

- There are **two basic** ways to add a handler for an event.

1. One way is to use the Events list in the Properties window, shown in follow Figure, which is displayed when you click the lightning bolt button.



Properties

Name	rotatedButton
Type	Button
Click	
ContextMenuClosing	
ContextMenuOpeni...	
DataContextChanged	
DragEnter	
DragLeave	
DragOver	
Drop	
FocusableChanged	
GiveFeedback	
GotFocus	
GotKeyboardFocus	
GotMouseCapture	
GotStylusCapture	
GotTouchCapture	

- To add a handler for a particular event, either type the name of the event and press Enter, or double- click to the right of the event name in the Events list.
- This causes the event to be added to the XAML tag. The method signature to handle the event is added to the C# code-behind file.

```
    <Button x:Name="rotatedButton" Content="2nd Button" Width="75"
            Height="22" FontWeight="Bold" Margin="218,138,224,159"
            RenderTransformOrigin="0.5,0.5"
            Click="rotatedButton_Click">

      ...
    </Button>
private void rotatedButton_Click(object sender, RoutedEventArgs e)
    {
    }
```

Handling Events

2. You can also type the name of the event directly in XAML and add the name of the handler there.
    - If you do this, Visual Studio will display a New Event Handler menu as you type. Selecting this will give the event the default name and create the handler in the code-behind file. If you type the name yourself, you can later right-click the event and select Go To Definition to generate the event handler in code.

Routed Events

- WPF uses events that are called *routed events*. A standard .NET event is handled by the code that has explicitly subscribed to it and it is sent only to those subscribers.
- Routed events are different in that they can send the event to all controls in the hierarchy in which the control participates.
- A routed event can travel up and down the hierarchy of the control on which the event occurred.
    - So, if you right-click a button, the MouseRightButtonDown event will first be sent to the button itself, then to the parent of the control—in the case of the earlier example, the Grid control. If this doesn't handle it, then the event is finally sent to the window. If, on the other hand you don't want the event to travel further up the hierarchy, then you simply set the RoutedEventArgs property Handled to true, and no additional calls will be made at that point. When an event travels up the control hierarchy like this, it is called a *bubbling event*.

- Routed events can also travel in the other direction, that is, from the root element to the control on which the action was performed. This is called a *tunneling event* and by convention all events like this are prefixed with the word Preview and always occur before their bubbling counterparts.
  - An example of this is the PreviewMouseRightButtonDown event. Finally, a routed event can behave exactly like a normal .NET event and only be sent to the control on which the action was made.

Routed Commands

- Routed commands serve much the same purpose as events in that they cause some code to execute.
- Where Events are bound directly to a single element in the XAML and a handler in the code, Routed Commands are more sophisticated.
- The key difference between events and commands is in their use.
  - An event should be used whenever you have a piece of code that has to respond to a user action that happens in only one place in your application. An example of such an event could be when the user clicks OK in a window to save and close it.
  - A command can be used when you have code that will be executed to respond to actions that happen in many locations. An example of this is when the content of an application is saved.
- There is often a menu with a Save command that can be selected, as well as a toolbar button for the same purpose. It is possible to use event handlers to do this, but it would mean implementing the same code in many locations—a command allows you to write the code just once.

Try It Out

1. Select the button rotatedButton and add the event KeyDown.
    1. You can do this by double-clicking the event in the Properties panel
    2. or by typing the XAML directly. If you type the name yourself, give it the name rotatedButton_KeyDown.
2. Select the Grid by clicking on the tag it in the XAML View, and add the same event to it. Name it Grid_KeyDown.
3. Select the Window tag in the XAML View and add the event again. Name it Window_KeyDown.
4. Repeat Steps 1 through 3, but replace the event with PreviewKeyDown and change the name of the event to reflect that it is the Preview handler.

Control Types

- WPF has a lot of controls to choose from. **Two types** of interest are **the Content and Items controls**.

1. Content controls, such as the Button control, have a Content property that can be set to any other control. This means that you can specify how the control is displayed, but you must specify zero or exactly one control directly in the content.
2. Items control, which is a control that allows you to insert multiple controls as content. An example of an Items control is the Grid control. When you are creating user interfaces, you are continually combining these two control types.

- In addition to Content and Items controls, there are a number of other types of controls that don’t allow you to use other controls as their content. One example of this is the Image control, which is used to display an image. Changing that behavior defeats the purpose of the control.

Alignment, Margins, Padding, and Dimensions

- Earlier examples used the Margin, HorizontalAlignment, and VerticalAlignment properties to position controls in a Grid container, but without going into much detail about their use.
- You have also seen how you can use Height and Width to specify dimensions. These properties, along with
- Padding, which you haven’t looked at yet, are useful for all of the layout controls (or most of them, as you will see), but in different ways. Different layout controls can also set default values for these properties. You’ll see a lot of this by example in subsequent sections, but before doing that, it is worth covering the basics.
- The two alignment properties, HorizontalAlignment and VerticalAlignment, determine how the control is aligned. HorizontalAlignment can be set to Left, Right, Center, or Stretch. Left and Right tend to position controls to the left or right edges of the container, Center positions controls in the middle, and Stretch changes the width of the control so that its edges reach to the sides of the container. VerticalAlignment is similar, and has the values Top, Bottom, Center, or Stretch.