



# .NET Technologies using C#

C # --- Inside WPF

**Instructor:** Mahboob Ali

**Email:** mahboob.ali@sheridancollege.ca

**Course:** PROG32356

Adapted from: Gursharan Singh Tatla

# Introduction

- Most modern UI frameworks are event driven and so is WPF.
- All of the controls expose a range of events that you may subscribe to.
- You can subscribe to (in other words *handle*) these events, which means that your application will be notified when they occur, and you may react to them.
- There are many types of events, but some of the most commonly used are using the mouse or the keyboard.
- On most controls, you will find events like `KeyDown`, `KeyUp`, `Click`, `MouseDown`, `MouseEnter`, `MouseLeave`, `MouseUp` and several others.
- For example: Your app will need to take action when the user clicks buttons, selects menu items, and clicks tools in the toolbar.
- In WPF, the code that sits behind the user interface, responding to control events and performing other processing, is called the *code-behind*.

# Introduction

- Some of the common events we'll cover are:
  - Keyboard Events
  - Mouse Events
  - Stylus Events
  - Touch Events

# Creating Event Handlers

- Visual Studio provides three methods for creating event handlers:
  - Double-clicking a control
  - Using the Properties window
  - Using XAML IntelliSense.



# Double-Clicking a Control

- The first method for creating an event handler is to simply *double-click* on the control on the Window Designer.
- This creates an event handler for the control's default event.
- For example: The default event for a `Button` is `Click`.
- Visual Studio adds an appropriate name attribute to the XAML code and creates a stub for the event handler in the code-behind file.

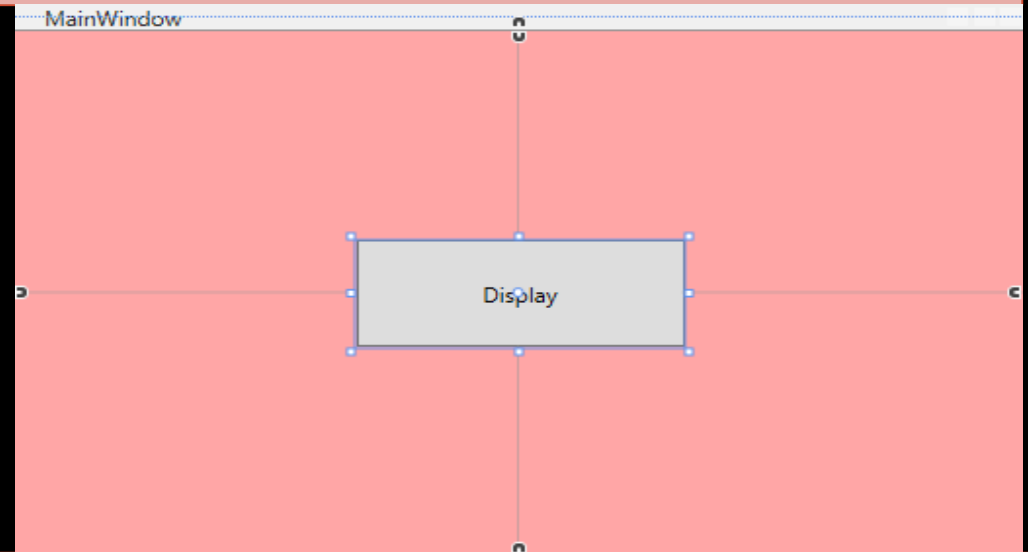
# Double-Clicking a Control

- XAML Code:

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

- C# Code Behind:

```
private void btnDisplay_Click(object sender, RoutedEventArgs e)  
{  
    MessageBox.Show("Hello world");  
}
```

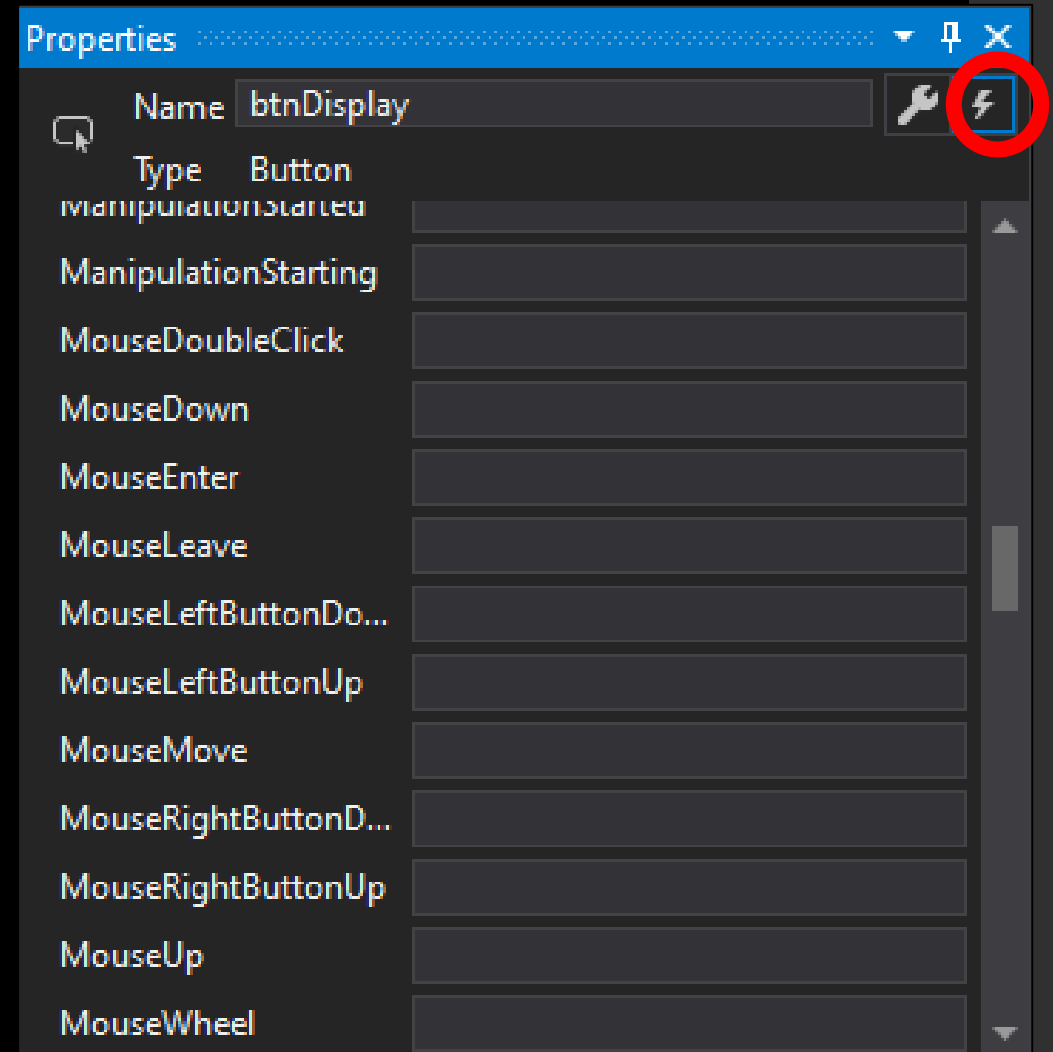


# Name the Controls First

- Visual Studio uses the control's name to build a name for the event handler.
- For example: If you double-click on a Button named btnSave, then Visual Studio creates an event handler named btnSave\_Click.
- If the control doesn't have a name, Visual Studio gives it a name and then names the event handler after it.
- For example: It might give the Button the name button1 and then name the event handler button1\_Click.
- If you want the control and event handler to have nice names, give the control a good name before you let Visual Studio create the event handler.

# Using the Properties Window

- The second way to make an event handler uses the Properties window.
- First select the control in the Window Designer.
- Next click on the Events icon in the Properties window (it looks like a lightning bolt) to see a list of that control's events and find the event that you want to handle.





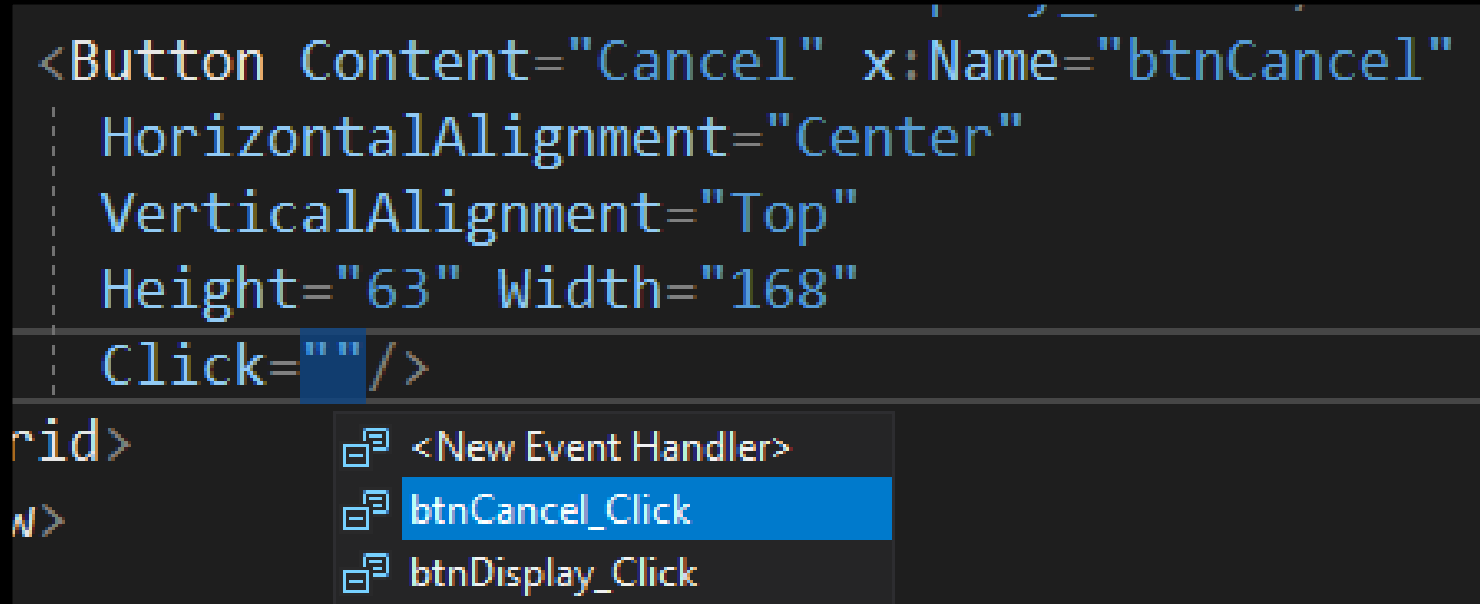
# Using the Properties Window

- Now you have two options:
  - To create a new event handler with a default name, double-click on the event.
  - To create a new event handler with a name of your choosing, type the name next to the event and then double-click on the event.

# Using XAML IntelliSense

- The third way to make an event handler in Visual Studio is to use the XAML Code Editor's IntelliSense.
- Start typing the event name attribute in the XAML Code Editor.
- When you type the equals sign, IntelliSense displays a list of existing event handlers that could catch the event.
- At the top of the list is the special entry <New Event Handler>.
- If you select this entry, Visual Studio invents an event handler name and creates an event handler stub.

```
<Button Content="Cancel" x:Name="btnCancel"
    HorizontalAlignment="Center"
    VerticalAlignment="Top"
    Height="63" Width="168"
    Click="" />
```



The screenshot shows the XAML code editor with a Button element. The Click attribute is being edited, and the IntelliSense dropdown is open, displaying a list of event handlers. The first option is '<New Event Handler>', followed by 'btnCancel\_Click' (which is highlighted), and 'btnDisplay\_Click'.

# Event Handlers at Run Time

- You can also use code-behind to attach event handlers to controls at run time.
- To use this technique, you don't need to add any reference to the event handler in the XAML code.
- Then, in the code-behind, you add code to attach the event handlers.

```
public MainWindow()  
{  
    InitializeComponent();  
  
    // Attach the event handlers.  
    btnDisplay.Click += btnDisplay_Click;  
    btnCancel.Click += btnCancel_Click;  
}
```

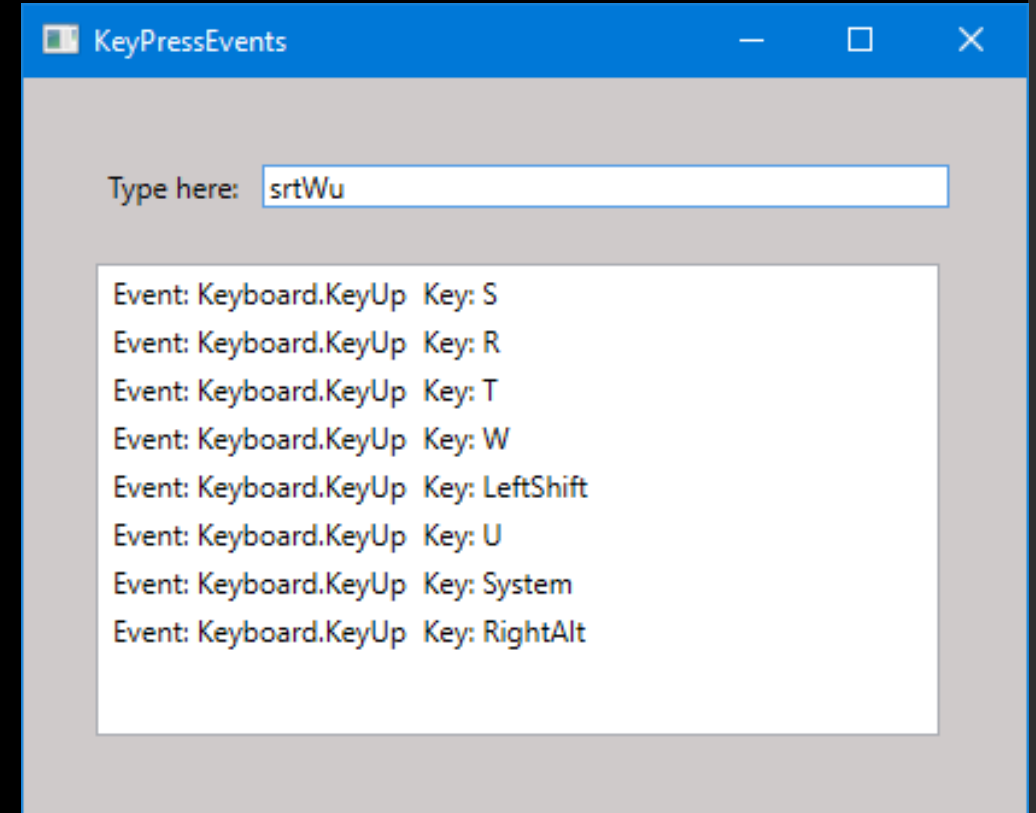
# Keyboard Events

- Common keyboard events:

Name	Description
KeyDown	Occurs when a key is pressed.
KeyUp	Occurs when a key is released.
TextInput	<p>Occurs when a keystroke is complete and the element is receiving the text input.</p> <p>This event isn't fired for keystrokes that don't result in text being "<i>typed</i>" (for example, Ctrl, Shift, Backspace, arrow keys, function keys etc.).</p>

# Example 1

- This example monitors a text box for all the possible key events and reports when they occur.



```
private void txtInput_KeyUp(object sender, KeyEventArgs e)
{
    string message = "Event: " + e.RoutedEvent + " " + "Key: " + e.Key;
    lstMessages.Items.Add(message);
}
```

# Mouse Events

- Common Mouse events:

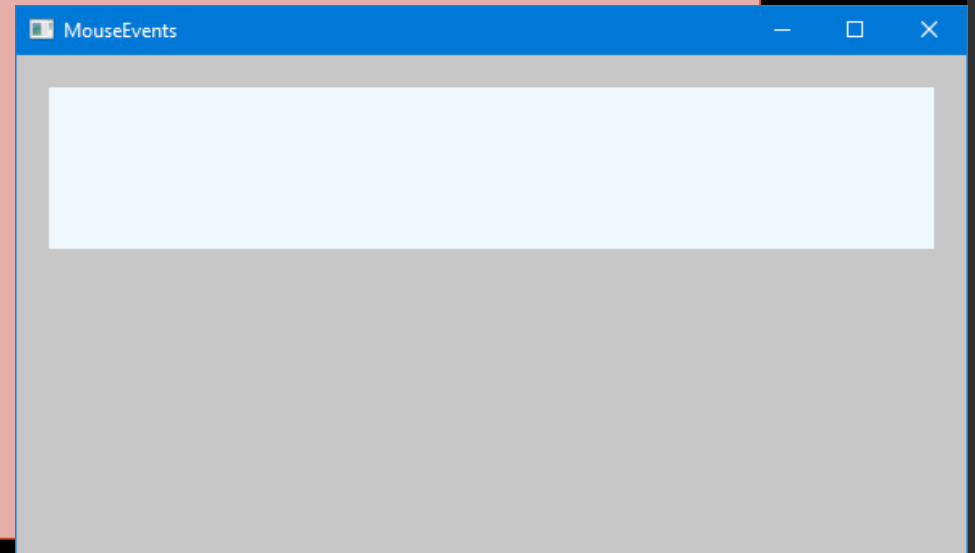
Name	Description
MouseDown and MouseUp	Occurs when a mouse button is pressed.
MouseLeftButtonDown and MouseRightButtonDown	Occurs when a mouse button is released.
MouseLeftButtonUp and MouseRightButtonUp	Occurs when mouse pointer moves into element.
MouseEnter	Occurs when mouse pointer moves out of element.
MouseLeave	Occurs when mouse button is pressed/released while pointer inside element (raised for any mouse button).
MouseDown and MouseUp	Occurs when a mouse pointer moves while pointer inside element.
MouseMove	



# Example 2

- XAML Code:

```
<StackPanel>  
    <Rectangle Name="recMouseEvents" Fill="AliceBlue"  
        MouseEnter="recMouseEvents_MouseEnter"  
        MouseLeave="recMouseEvents_MouseLeave"  
        MouseMove="recMouseEvents_MouseMove"  
        MouseDown="recMouseEvents_MouseDown"  
        Height="100" Margin="20">  
    </Rectangle>  
  
    <Label Name="lb11"/>  
    <Label Name="lb12"/>  
    <Label Name="lb13"/>  
</StackPanel>
```

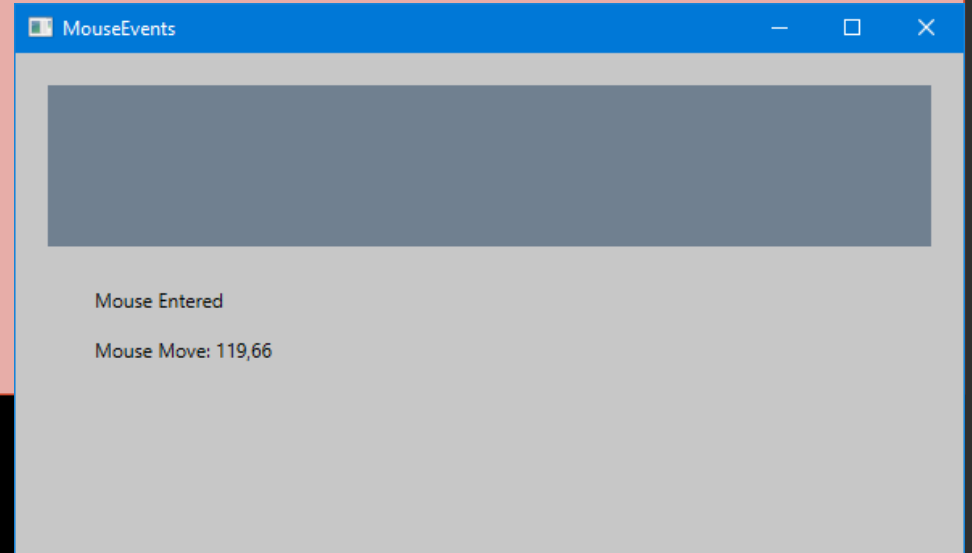


# Example 2 continued

- C# Code Behind:

```
private void recMouseEvents_MouseEnter(object sender, MouseEventArgs e)
{
    Rectangle source = e.Source as Rectangle;
    if (source != null)
        source.Fill = Brushes.SlateGray;

    lbl1.Content = "Mouse Entered";
}
```



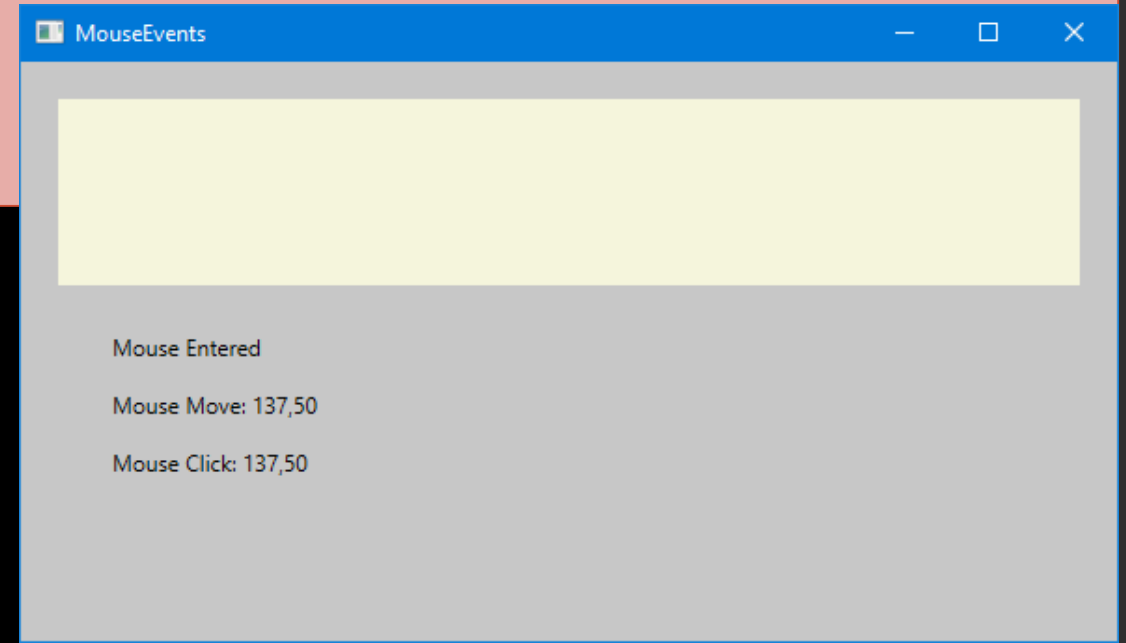
```
private void recMouseEvents_MouseMove(object sender, MouseEventArgs e)
{
    Point pnt = e.GetPosition(recMouseEvents);
    lbl2.Content = "Mouse Move: " + pnt.ToString();
}
```

# Example 2 continued

- C# Code Behind:

```
private void recMouseEvents_MouseDown(object sender, MouseButtonEventArgs e)
{
    Rectangle source = e.Source as Rectangle;
    Point pnt = e.GetPosition(recMouseEvents);
    lbl3.Content = "Mouse Click: " + pnt.ToString();

    if (source != null)
        source.Fill = Brushes.Beige;
}
```



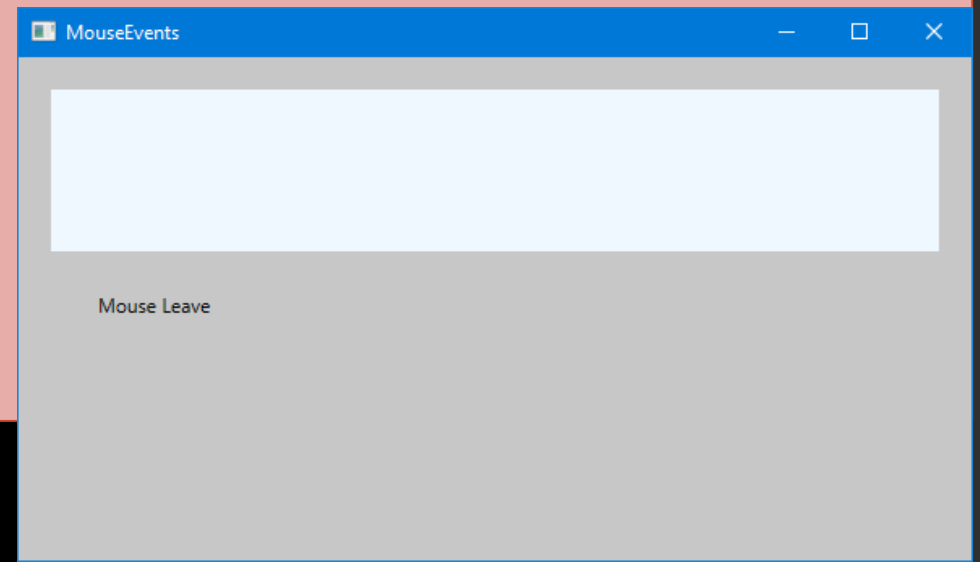
# Example 2 continued

- C# Code Behind:

```
private void recMouseEvents_MouseLeave(object sender, MouseEventArgs e)
{
    Rectangle source = e.Source as Rectangle;

    if (source != null)
    {
        source.Fill = Brushes.AliceBlue;
    }

    lbl1.Content = "Mouse Leave";
    lbl2.Content = "";
    lbl3.Content = "";
}
```



# Stylus Events

- WPF has special support for a pen digitizer, also known as a stylus, found on devices such as the Surface Pro.
  - This is sometimes referred to as “ink” support.
- If you don’t add any special support for a stylus in your application, it appears to act just like a mouse, raising all the relevant mouse events, such as `MouseDown`, `MouseMove`, and `MouseUp`.
- This behavior is essential for a stylus to be usable with programs that aren’t designed specifically for a stylus.
- However, if you want to provide an experience that is optimized for a stylus, you can interact with an instance of `System.Windows.Input.StylusDevice`.

# Stylus Events

- There are three ways to get an instance of `StylusDevice`:
  - You can use a `StylusDevice` property on `MouseEventArgs` to get an instance inside mouse event handlers.
    - (This property will be `null` if there is no stylus.)
  - You can use the static `System.Windows.Input.Stylus` class and its `CurrentStylusDevice` property to interact with the stylus at any time.
    - (This will also be `null` if there is no stylus.)
- You can handle a number of events specific to the stylus.



# Stylus Events

- **StylusDevice** contains a number of properties, including the following:
  - **Inverted:** A Boolean that reveals whether the stylus is being used as an eraser (with its back end against the screen).
  - **InAir:** A Boolean that indicates whether the stylus is in contact with the screen, because on some devices its movement can still be registered as long as it is close enough.
  - **StylusButtons:** A collection of **StylusButton** objects.  
Unlike with a mouse, there is no fixed list of possible buttons. Each **StylusButton** has a string **Name** and a **Guid** identifier, along with a **StylusButtonState** of **Up** or **Down**.
  - **TabletDevice:** A property of type **System.Windows.Input.TabletDevice** that provides detailed information about the current hardware and which stylus capabilities it provides (such as pressure-sensitivity or in-air movement). Its **Type** property is **Stylus** for a pen digitizer or **Touch** for a touch digitizer.

# Stylus Events

- The stylus-specific events are as follows:
  - StylusEnter and StylusLeave
  - StylusMove
  - StylusInAirMove
  - StylusDown, StylusUp
  - StylusButtonDown, StylusButtonUp
  - StylusSystemGesture
  - StylusInRange, StylusOutOfRange
  - GotStylusCapture and LostStylusCapture

# Touch Events

- The basic touch events look and act a lot like mouse events:
  - TouchEnter and TouchLeave
  - TouchMove
  - TouchDown and TouchUp
  - GotTouchCapture and LostTouchCapture
- When multiple fingers are touching simultaneously, these events get raised for each finger independently.

# WPF Resources

- Resources are normally definitions connected with some object that you just anticipate to use more often than once.
- It is the ability to store data locally for controls or for the current window or globally for the entire applications.
- Resources are defined in resource dictionaries and any object can be defined as a resource effectively making it a shareable asset.
- Resources can be of two types:
  - `StaticResource`
  - `DynamicResource`
- A `StaticResource` is a onetime lookup, whereas a `DynamicResource` works more like a data binding.
- It remembers that a property is associated with a particular resource key.
- If the object associated with that key changes, dynamic resource will update the target property.

# Example 3

- Here's a simple application for the SolidColorBrush resource.
  - Create a new WPF project with the name WpfResources.
  - Drag two Rectangles and set their properties as shown in the following XAML code.

```
<Window.Resources>
    <SolidColorBrush x:Key="brushResourceLightGreen" Color="LightGreen" />
    <SolidColorBrush x:Key="brushResourceLightSalmon" Color="LightSalmon" />
</Window.Resources>

<StackPanel>
    <Rectangle Name="rect1" Height="50" Margin="20"
        Fill="{StaticResource brushResourceLightGreen}" />
    <Rectangle Name="rect2" Height="50" Margin="20"
        Fill="{DynamicResource brushResourceLightGreen}" />
    <Button x:Name="changeResourceButton"
        Content="Change Resource" Click="changeResourceButton_Click" />
</StackPanel>
```

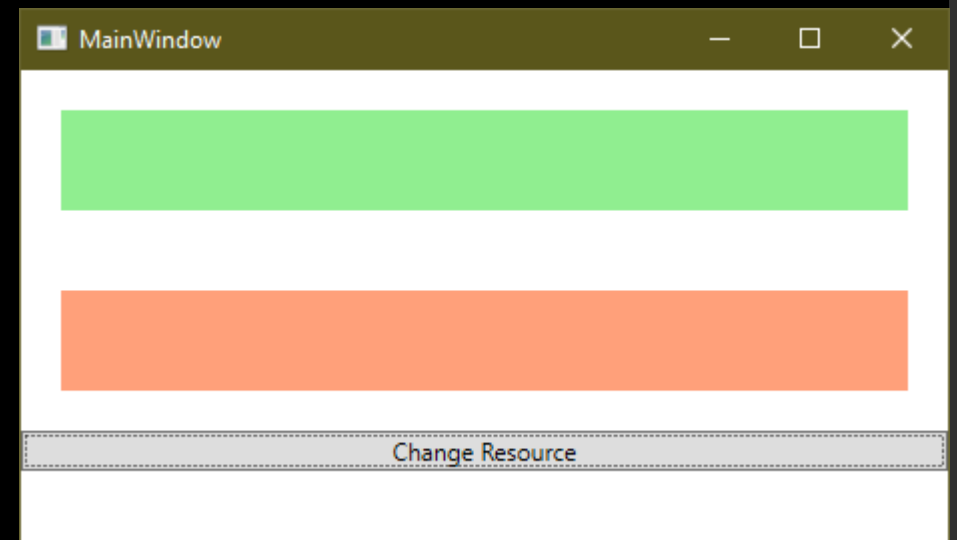
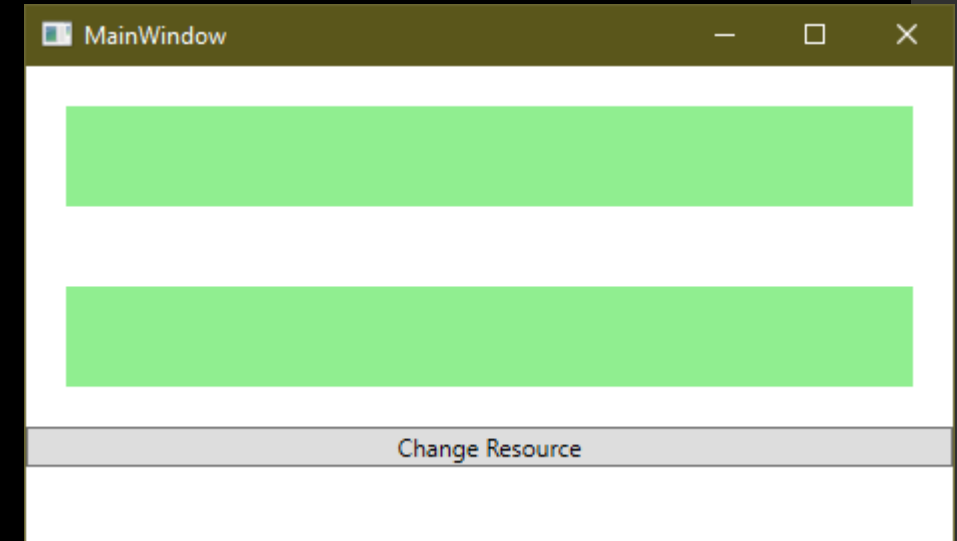
← XAML

C#  
↓

```
private void changeResourceButton_Click(object sender, RoutedEventArgs e)
{
    rect2.Fill = Resources["brushResourceLightSalmon"] as SolidColorBrush;
}
```

# Example 3 continued

- In the above XAML code, you can see that one rectangle has `StaticResource` and the other one has `DynamicResource` and the color of `brushResource` is `LightGreen`.
- When you compile and execute the code, it will produce this `MainWindow`.
- When you click the `Change Resource` button, you will see that the rectangle with `DynamicResource` will change its color to `Red`.





# Resource Scope

- Resources are defined in resource dictionaries, but there are numerous places where a resource dictionary can be defined.
- In the previous example, a resource dictionary is defined on Window/page level.
- In what dictionary a resource is defined immediately limits the scope of that resource:
  - Define the resource in the resource dictionary of a grid and it's accessible by that grid and by its child elements only.
  - Define it on a window/page and it's accessible by all elements on that window/page.
  - The app root can be found in [App.xaml](#) resources dictionary. It's the root of our application, so the resources defined here are scoped to the entire application.



# Resource Dictionaries

- Resource dictionaries in XAML apps imply that the resource dictionaries are kept in separate files.
- Defining resources in separate files can have the following advantages:
  - Separation between defining resources in the resource dictionary and UI related code.
  - Defining all the resources in a separate file such as App.xaml would make them available across the app.
- To define resources in a resource dictionary in a separate file, add a new resource dictionary through Visual Studio by following these steps:
  - In the Solution Explorer, add a new folder and name it ResourceDictionaries.
  - Right-click on this folder and select Resource Dictionary from Add submenu item and name it DictionaryWithBrush.xaml.

# Example 4

- Let's now take the same example, but here, we will define the resource dictionary in app level.

```
<StackPanel>
  <Rectangle Name="rect1" Height="50" Margin="20"
    Fill="{StaticResource brushResourceLightGreen}" />
  <Rectangle Name="rect2" Height="50" Margin="20"
    Fill="{DynamicResource brushResourceLightGreen}" />
  <Button x:Name="changeResourceButton"
    Content="Change Resource" Click="changeResourceButton_Click" />
</StackPanel>
```

← MainWindow.xaml

DictionaryWithBrush.xaml



```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:WpfResources.ResourceDictionaries">

  <SolidColorBrush x:Key="brushResourceLightGreen" Color="LightGreen" />
  <SolidColorBrush x:Key="brushResourceLightSalmon" Color="LightSalmon" />
</ResourceDictionary>
```

# Example 4 continued

## App.xaml

```
<Application x:Class="WpfResources.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfResources"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary Source = "ResourceDictionaries\DictionaryWithBrush.xaml"/>
    </Application.Resources>
</Application>
```

## MainWindow.xaml.cs

```
private void changeResourceButton_Click(object sender, RoutedEventArgs e)
{
    rect2.Fill = Application.Current.Resources["brushResourceLightSalmon"] as SolidColorBrush;
}
```

# Exercise 1

- Create a WPF app to handle events. The app will have a slider between 0 and 100 and a button that says “click me”. Upon clicking on the button the slider will decrease. Add another button that says “touch me” where the slider will increase when the user mouses over the button.
- Add another label that says “Not Loaded” where the text for it will change when the Window loads.
- Add another label that says “Not in focus” where the text will change upon the window coming into focus.
- Add another label that says “No Key Pressed” where the text will change when the user presses any key.
- Add another label that says “Mouse Position: N/A” where the text will change as the mouse moves around the window.
- Add another label that says “Mouse Clicked: N/A” where the text will change when the user either left or right clicks on the window.

# Exercise 2

- Create a WPF app with 2 rectangles. The rectangle on the left will be filled with light green and the right with light blue.
- Create a context menu (a “right click” menu) and attach it to the Window with the menu item “Quit” (have a closing event asking “are you sure?”)
- Create a context menu and attach it to the light blue rectangle offering the options for “Light Blue”, “Light Green”, “Red” and “Yellow” and change that rectangle to the colour chosen.
- Create a context menu and attach it to the light green rectangle offering options for “Brown”, “Azure” and “Purple” and change that rectangle to the colour chosen.