ROUTED EVENTS

Chapter 5 of Pro WPF : By Matthew MacDonald

Assist Lect. Wadhah R. Baiee . College of IT – Univ. of Babylon - 2014

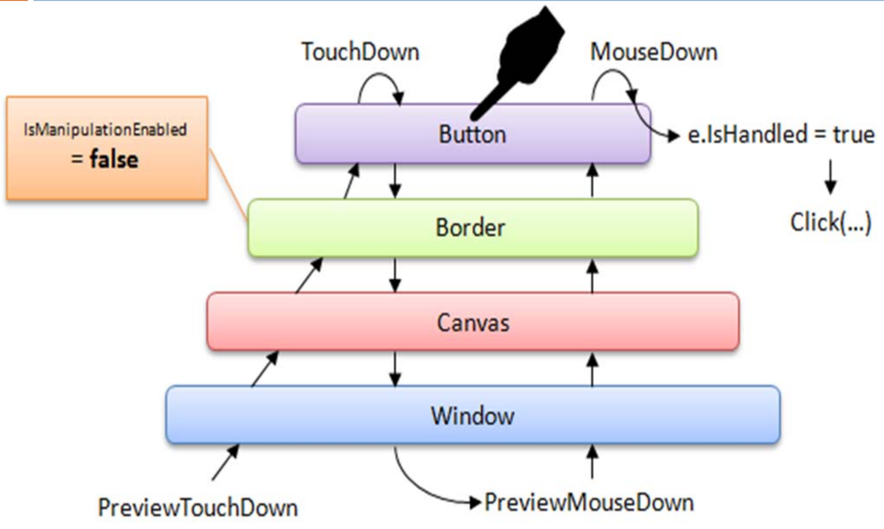
Introduction

- *Routed events* are events with more traveling power—they can tunnel down or bubble up the element tree and be processed by event handlers along the way.
- A routed event can be handled on one element (such as a label) even though it originates on another (such as an image inside that label).

Introduction

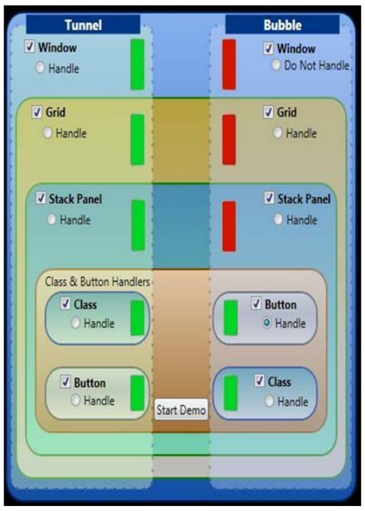
- Every .NET developer is familiar with the idea of *events*—messages that are sent by an object .
- WPF enhances the .NET event model with the concept of event routing.
- Event routing allows an event to originate in one element but be raised by another one.
- For example, event routing allows a click that begins in a toolbar button to rise up to the toolbar and then to the containing window before it's handled by your code.

Introduction



Event Routing

- Many controls in WPF are content controls, and content controls can hold any type and amount of nested content.
- For example, you can build a graphical button out of shapes, create a label that mixes text and pictures, or put content in a specialized container to get a scrollable or collapsible display. You can even repeat this nesting process to go as many layers deep as you want.



Event Routing

```
<Label BorderBrush="Black" BorderThickness="1">
  <StackPanel>
    <TextBlock Margin="3">
     Image and text label</TextBlock>
    <Image Source="happyface.jpg" Stretch="None" />
    <TextBlock Margin="3">
     Courtesy of the StackPanel</TextBlock>
  </StackPanel>
</Label>
```

- imagine you have a label like this one, which contains a StackPanel that brings together two blocks of text and an image
- consider what happens when you click the image part of the fancy label shown here.
- Clearly, it makes sense for the Image.MouseDown and Image.MouseUp events to fire. But what if you want to treat all label clicks in the same way?

Routed events come in the following three flavors:

- *Direct events*: These are like ordinary .NET events. They originate in one element and don't pass to any other. For example, MouseEnter (which fires when the mouse pointer moves over an eleent) is a direct event.
- *Bubbling events:* These events travel *up* the containment hierarchy. For example, MouseDown is a bubbling event. It's raised first by the element that is clicked. Next, it's raised by that element's parent, then by *that* element's parent, and so on, until WPF reaches the top of the element tree.

Routed events come in the following three flavors:

- *Tunneling events* These events travel *down* the containment hierarchy. They give you the chance to preview (and possibly stop) an event before it reaches the appropriate control. For example, PreviewKeyDown allows you to intercept a key press, first at the window level and then in increasingly more-specific containers until you reach the element that had focus when the key was pressed.
- Example p111-113 on your textbook.

Attached Events

- many controls have their own more specialized events. The button is one example—it adds a Click event that isn't defined by any base class.
- Imagine that you wrap a stack of buttons in a StackPanel. You want to handle all the button clicks in one event handler.
- You can handle all the button clicks by handling the Click event at a higher level (such as the containing StackPanel).

Attached Events

```
<StackPanel Click="DoSomething" Margin="5">
  <Button Name="cmd1">Command 1</Button>
  <Button Name="cmd2">Command 2</Button>
  <Button Name="cmd3">Command 3</Button>
  ...
</StackPanel>
```

- The problem is that the StackPanel doesn't include a Click event, so this is interpreted by the XAML parser as an error. The solution is to use a different attached-event syntax in the form *ClassName.EventName*.

```
<StackPanel Button.Click="DoSomething" Margin="5">
  <Button Name="cmd1">Command 1</Button>
  <Button Name="cmd2">Command 2</Button>
  <Button Name="cmd3">Command 3</Button>
  ...
</StackPanel>
```

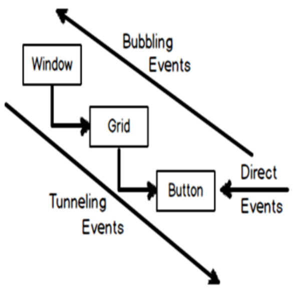
Attached Events

- In the DoSomething() event handler, you have several options for determining which button fired the event. You can compare its text (which will cause problems for localization) or its name (which is fragile because you won't catch mistyped names when you build the application).

```
private void DoSomething(object sender, RoutedEventArgs e)
{
    if (e.Source == cmd1)
    { ... }
    else if (e.Source == cmd2)
    { ... }
    else if (e.Source == cmd3)
    { ... }
}
```

Tunneling Events

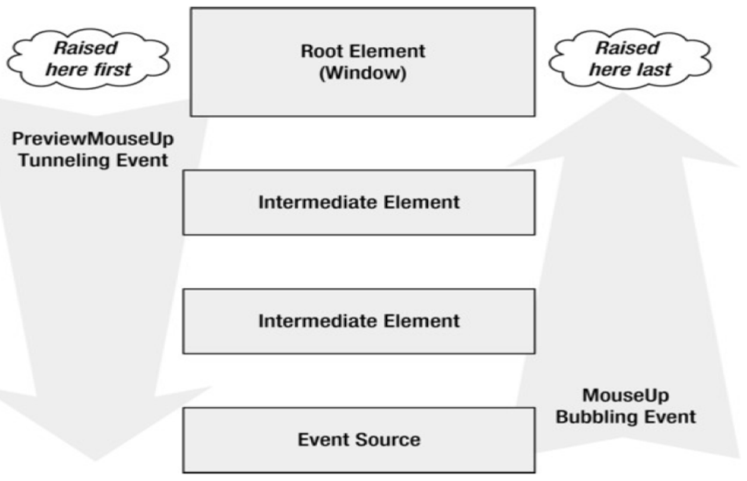
- Tunneling events work the same as bubbling events but in the opposite direction. For example,
- If MouseUp was a tunneled event (which it isn't), clicking the image in the fancy label example would cause
  - MouseUp to fire first in the window,
  - then in the Grid,
  - then in the StackPanel,
  - and so on,
- until it reaches the actual source, which is the image in the label.



Tunneling Events

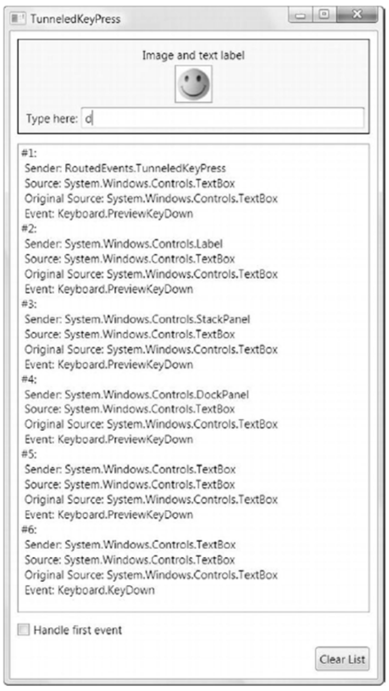
- Tunneling events are easy to recognize because they begin with the word *Preview*.
- Furthermore, WPF usually defines bubbling and tunneling events in pairs.
- That means if you find a bubbling MouseUp event, you can probably also find a tunneling PreviewMouseUp event.

Tunneling Events



Tunneling Events

- Tunneling events are useful if you need to perform some preprocessing that acts on certain keystrokes or filters out certain mouse actions.



```
#1:
 Sender: RoutedEvents.TunneledKeyPress
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#2:
 Sender: System.Windows.Controls.Label
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#3:
 Sender: System.Windows.Controls.StackPanel
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#4:
 Sender: System.Windows.Controls.DockPanel
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#5:
 Sender: System.Windows.Controls.TextBox
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#6:
 Sender: System.Windows.Controls.TextBox
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.KeyDown
```

WPF Events

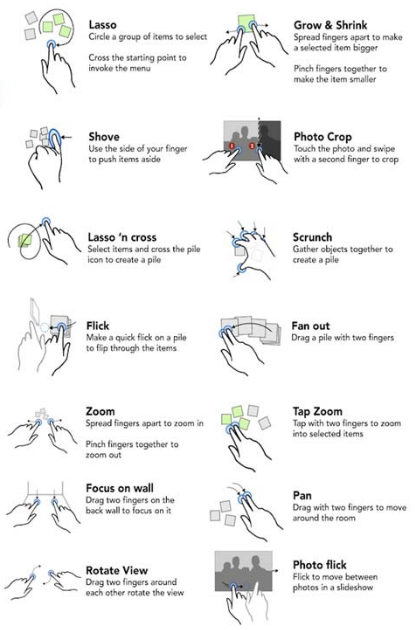
- The most important events usually fall into one of five categories:
  - *Lifetime events*: These events occur when the element is initialized, loaded, or unloaded.
  - *Mouse events*: These events are the result of mouse actions.
  - *Keyboard events*: These events are the result of keyboard actions (such as key presses).
  - *Stylus events*: These events are the result of using the pen-like stylus, which takes the place of a mouse on a Tablet PC.
  - *Multitouch events*: These events are the result of touching down with one or more fingers on a multitouch screen. They're supported only in Windows 7 and Windows 8.

Multitouch Input

- Multitouch is a way of interacting with an application by touching a screen.
- What distinguishes multitouch input from old-fashioned pen input is that multitouch recognizes *gestures*—specific ways the user can move more than one finger to perform a common operation.

Multitouch Input

- A simple multitouch enabled application might show multiple pictures on a virtual desktop and allow the user to drag, resize, and rotate each image to create a new arrangement.



Lasso
Circle a group of items to select
Cross the starting point to invoke the menu

Grow & Shrink
Spread fingers apart to make a selected item bigger
Pinch fingers together to make the item smaller

Shove
Use the side of your finger to push items aside

Photo Crop
Touch the photo and swipe with a second finger to crop

Lasso 'n cross
Select items and cross the pile icon to create a pile

Scrunch
Gather objects together to create a pile

Flick
Make a quick flick on a pile to flip through the items

Fan out
Drag a pile with two fingers

Zoom
Spread fingers apart to zoom in
Pinch fingers together to zoom out

Tap Zoom
Tap with two fingers to zoom into selected items

Focus on wall
Drag two fingers on the back wall to focus on it

Pan
Drag with two fingers to move around the room

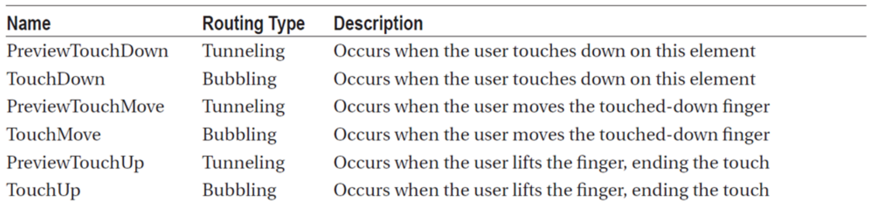
Rotate View
Drag two fingers around each other rotate the view

Photo flick
Flick to move between photos in a slideshow

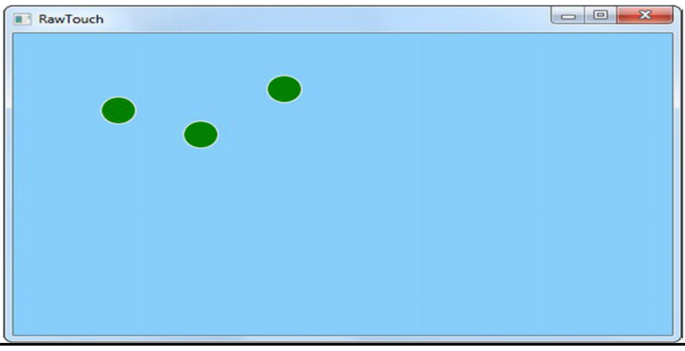
Multitouch : WPF provides three layers of multitouch support:

- *Raw touch*: This is the lowest level of support, and it gives you access to every touch the user makes.
- *Manipulation*: This is a convenient abstraction that translates raw multitouch input into meaningful gestures .The common gestures that WPF elements support include pan, zoom, rotate, and tap.
- *Built-in element support*: Some elements already react to multitouch events, with no code required. For example, scrollable controls such as the ListBox, ListView, DataGrid, TextBox, and ScrollViewer support touch panning.

Raw Touch



Name	Routing Type	Description
PreviewTouchDown	Tunneling	Occurs when the user touches down on this element
TouchDown	Bubbling	Occurs when the user touches down on this element
PreviewTouchMove	Tunneling	Occurs when the user moves the touched-down finger
TouchMove	Bubbling	Occurs when the user moves the touched-down finger
PreviewTouchUp	Tunneling	Occurs when the user lifts the finger, ending the touch
TouchUp	Bubbling	Occurs when the user lifts the finger, ending the touch



Raw Touch

What distinguishes this example from a similar mouse event test is that the user can touch down with several fingers at once, causing multiple ellipses to appear, each of which can be dragged about independently.
Exampl p134-136

Manipulation + Inertia

Sunday's Next Seminar