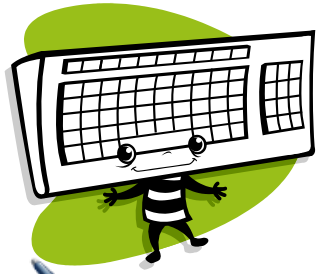
Routed Events

Agenda

- Overview
- Routed Events
- Mouse Input
- Keyboard Input

Input Sources

- There are 4 main kinds of user input for a Windows application:
  - Mouse
  - Keyboard
  - Stylus events
  - Multitouch events



Input Receivers

- Any user interface element in WPF can receive input
  - not just controls

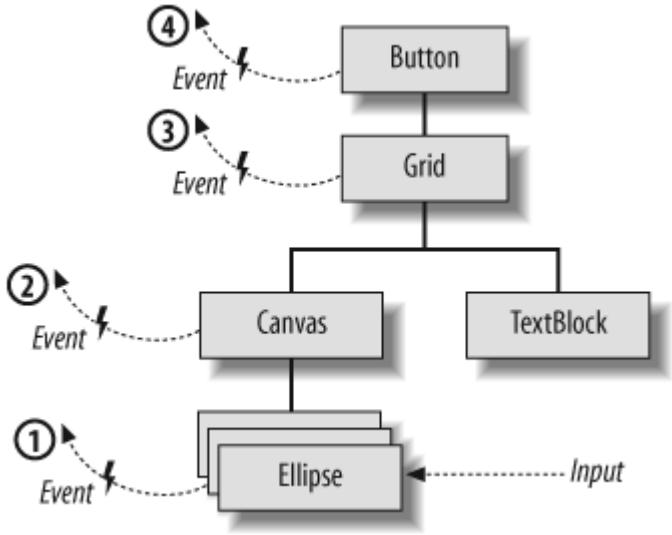

```
<Ellipse PreviewMouseDown="PreviewMouseDownEllipse"
    MouseDown="MouseDownEllipse"
    x:Name="myEllipse"
    Canvas.Left="1" Canvas.Top="1" Width="16" Height="16"
    Fill="Yellow" Stroke="Black" />
```

Action types

- WPF has three types of actions:
  - Events
    - Raw user input is delivered to your code through WPF's *routed event* mechanism
  - Commands
    - A higher-level concept
      A particular action that might be accessible through several different inputs such as keyboard shortcuts, toolbar buttons, and menu items
  - Triggers
    - Are best suited for superficial responses, such as making a button change color when the mouse moves over it
    - It is often possible to create the visual feedback you require entirely within the user interface markup (xaml) by use of triggers. Triggers offer a declarative approach, where WPF does more of the work for you

Routed Events

- WPF uses *routed events*
  - Instead of just calling handlers attached to the element that raised the event (as the normal .Net event does)
  - **WPF walks the tree of user interface elements**,
    - calling all handlers for the routed event attached to any node from the originating element right up to the root of the user interface tree
- This behavior is the defining feature of routed events
  - And is at the heart of event handling in WPF



Routed Events

- A routed event can either be:
  - **Bubbling**
  - **Tunneling**
  - **Direct**
- A **bubbling** event starts by looking for event handlers attached to the target element that raised the event
  - and then looks at its parent and then its parent's parent, and so on until it reaches the root of the tree
- A **tunneling** event works in reverse
  - it looks for handlers at the root of the tree first and works its way down, finishing with the originating element
- **Direct** events work like normal .NET events
  - Only handlers attached directly to the originating element are notified - no real routing occurs

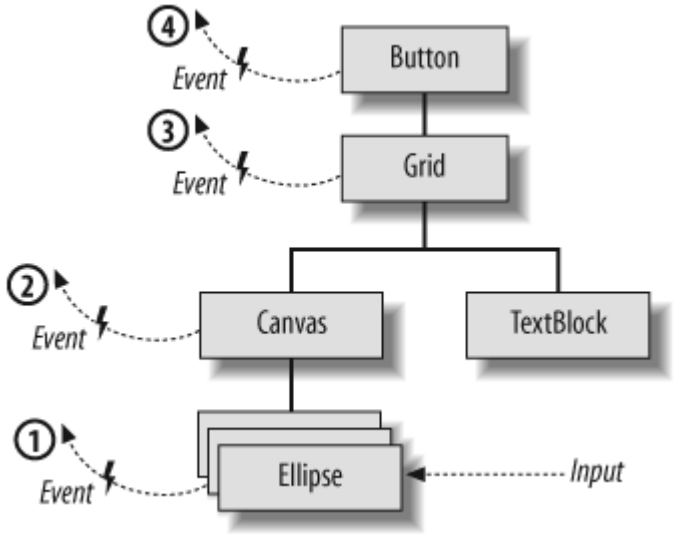
Routed Event VS Ordinary CLR Event

- The main difference is that with a direct routed event, WPF provides the underlying implementation, whereas if you were to use the normal C# event syntax to declare an event, the C# compiler would provide the implementation
- The C# compiler would generate a hidden private field to hold the event handler, meaning that you pay a per-object overhead for each event whether or not any handlers are attached
- With WPF's event implementation, event handlers are managed in such a way that you pay an overhead only for events to which handlers are attached
- In a UI with thousands of elements each offering tens of events, most of which don't have handlers attached, this starts to add up

Routed Events Pairs

- WPF defines most routed events in pairs:
  - one bubbling and
  - one tunneling
- The tunneling event name always begins with Preview and is raised first
- The tunneling preview event is followed directly by a bubbling event

```
PreviewMouseDownButton
PreviewMouseDownGrid
PreviewMouseDownCanvas
PreviewMouseDownEllipse
MouseDownEllipse
MouseDownCanvas
MouseDownGrid
```



Halting Event Routing

- There are some situations in which you might not want events to bubble up
- For example, you may wish to convert the event into something else
  - the Button element effectively converts a MouseDown event followed by a MouseUp event into a single Click event
  - It suppresses the more primitive mouse button events so that only the Click event bubbles up out of the control
- Any handler can prevent further processing of a routed event by setting the Handled property of the RoutedEventArgs

Determining the Target

- Although it is convenient to be able to handle events from a group of elements in a single place, your handler might need to know which element caused the event to be raised
- You might think that this is the purpose of the sender parameter of your handler
  - but the sender always refers to the object to which you attached the event handler
  - In the case of bubbled and tunneled events, this often isn't the element that caused the event to be raised
- The handler has a RoutedEventArgs parameter, which offers a **Source** property for this purpose

Routed Events and Normal Events

- Normal .NET events offer one advantage over routed events:
  - many .NET languages have built-in support for handling CLR events
  - Because of this, WPF provides wrappers for routed events, making them look just like normal CLR events
  - This provides the best of both worlds:
    - you can use your favorite language's event handling syntax while taking advantage of the extra functionality offered by routed events

Attaching event handlers in code

- When you use these CLR event wrappers, WPF uses the routed event system on your behalf.

```
...
public Window1( ) { InitializeComponent( );
myEllipse.MouseDown += MouseDownEllipse;
myEllipse.PreviewMouseDown += PreviewMouseDownEllipse; }
...
```

Use this style

- Attaching event handlers the long-winded way

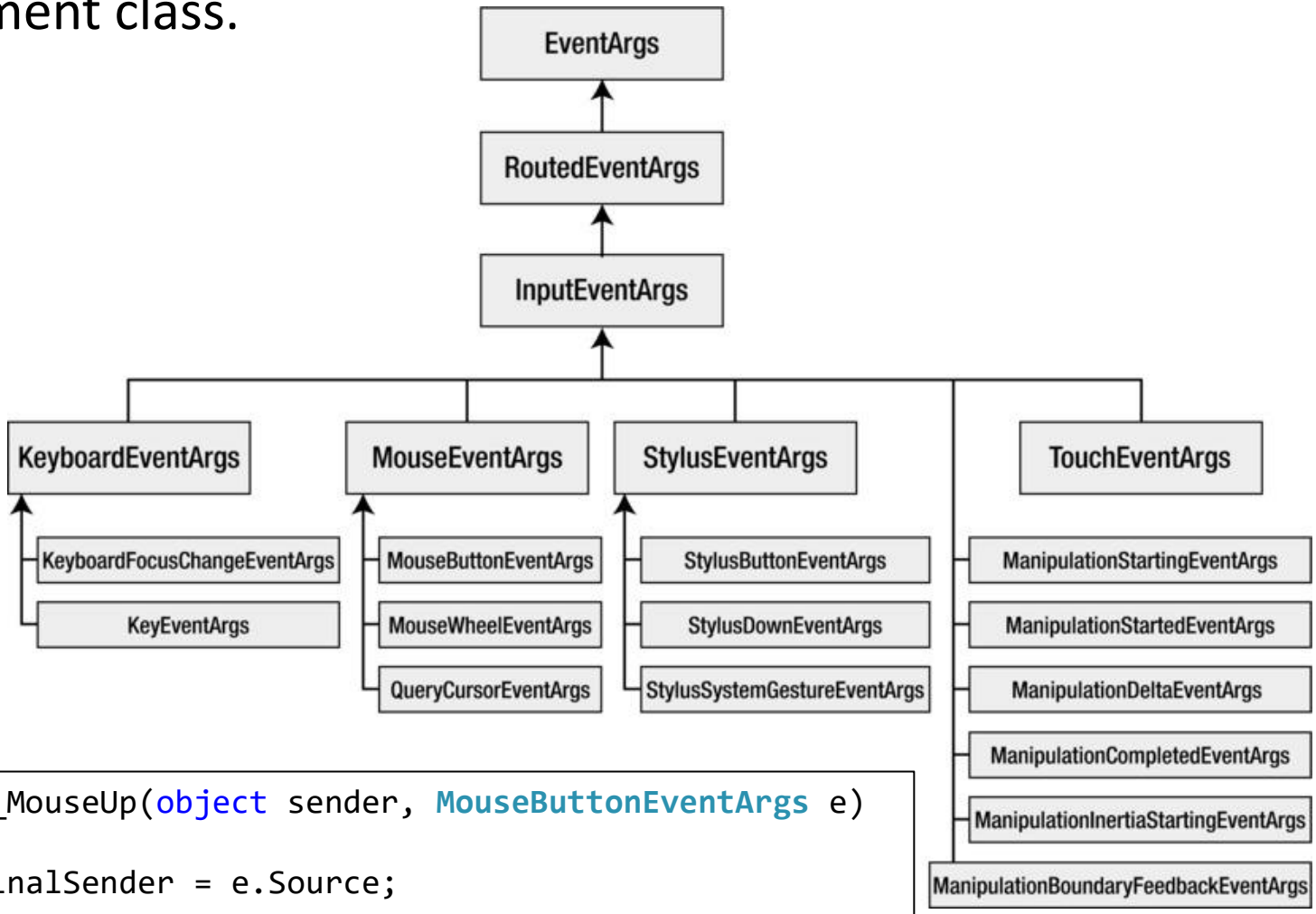
```
...
public Window1( )
{
   InitializeComponent( );
   myEllipse.AddHandler(Ellipse.MouseDownEvent,
      new MouseButtonEventHandler(MouseDownEllipse));
   myEllipse.AddHandler(Ellipse.PreviewMouseDownEvent,
      new MouseButtonEventHandler(PreviewMouseDownEllipse));
}
...
```

Events in Code or XAML?

- The code behind is usually the best place to attach event handlers
  - **Unless you use data binding**
- If your user interface has unusual and creative visuals, there's a good chance that the XAML file will effectively be owned by a graphic designer
- A designer shouldn't have to know what events a developer needs to handle, or what the handler functions are called
- Ideally, the designer will give elements names in the XAML and the developer will attach handlers in the code behind

EventArgs

- Input events can pass along extra information by using a custom event argument class.




```
private void Button_MouseUp(object sender, MouseButtonEventArgs e)
    {
        Object OriginalSender = e.Source;
    }
```

Mouse Input

Mouse Input

- Mouse input is directed to whichever element is directly under the mouse cursor
- All user interface elements derive from the UIElement base class, which defines a number of mouse input events

Event	Routing	Meaning
GotMouseCapture	Bubble	Element captured the mouse.
LostMouseCapture	Bubble	Element lost mouse capture.
MouseEnter	Direct	Mouse pointer moved into element.
MouseLeave	Direct	Mouse pointer moved out of element.
PreviewMouseLeftButtonDown, MouseLeftButtonDown	Tunnel, Bubble	Left mouse button pressed while pointer inside element.
PreviewMouseLeftButtonUp, MouseLeftButtonUp	Tunnel, Bubble	Left mouse button released while pointer inside element.
…(*Just an extract*)	…	…

Mouse related properties

- In addition to the mouse-related events, UIElement also defines a pair of properties that indicate whether the mouse pointer is currently over the element:
- IsMouseOver
  - will be true if the cursor is over the element in question or over any of its child elements
- IsMouseDirectlyOver
  - will be true only if the cursor is over the element in question but not one of its children

The Click Event

- Clicks are a higher-level concept than basic mouse input
  - a button can be "clicked" with:
    - the mouse
    - the stylus
    - the keyboard
    - through the Windows accessibility API
- Clicking doesn't necessarily correspond directly to a single mouse event
  - Usually, the user has to press and release the mouse button while the mouse is over the control to register as a click
  - These higher-level events are provided by more specialized element types
  - The Control class adds a PreviewMouseDoubleClick and **MouseDoubleClick** event pair
  - ButtonBase (the base class of Button, CheckBox, and RadioButton) goes on to add a **Click event**

Mouse Input and Hit Testing

- WPF always takes the shapes of your elements into account when handling mouse input
- For example, if you create a donut-shaped control and click on the hole in the middle, the click will be delivered to whatever was visible behind your control through the hole
- Occasionally it is useful to subvert the standard hit testing behavior
  - You might wish to create a donut-shaped control with a visible hole, but which doesn't let clicks pass through it
  - Alternatively, you might want to create an element that is visible to the user, but transparent to the mouse
  - WPF lets you do both of these things

Mouse Input and Hit Testing

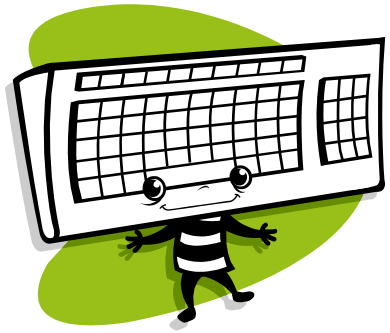
- To make an element transparent to the eye but opaque to the mouse
  - you can paint an object with a transparent brush
  - For example, an Ellipse with its Fill set to Transparent will be invisible to the eye, but not to the mouse
- If you want a shape with a transparent fill that does not receive mouse input, simply supply no Fill at all
- To create a visible object that is transparent to the mouse
  - Set the elements IsHitTestVisible property to false ensures that the element will not receive mouse input
  - instead, input will be delivered to whatever is under the element

Retrieving the mouse position

- The GetPosition method lets you discover the position of the mouse
- You must pass in a user interface element.
  - It will return the mouse position relative to the specified element (taking into account any transformations that may be in effect)

```
Point positionRelativeToEllipse = Mouse.GetPosition(myEllipse);
```

Keyboard Input



Keyboard Input

- At any given moment, a particular element is designated as having the ***focus***
  - meaning that it acts as the target for keyboard input
- The user sets the focus by clicking the control in question
  - with the mouse or stylus, or by using navigation keys such as the Tab and arrow keys
- The UIElement base class defines an IsFocused property
  - so in principle, any user interface element can receive the focus
- The Focusable property determines whether this feature is enabled on any particular element.
  - By default, this is true for controls, and false for other elements

Offered Keyboard Input Events

Event	Routing	Meaning
PreviewGotKeyboardFocus, GotKeyboardFocus	Tunnel, Bubble	Element received the keyboard focus.
PreviewLostKeyboardFocus, LostKeyboardFocus	Tunnel, Bubble	Element lost the keyboard focus.
GotFocus	Bubble	Element received the logical focus.
LostFocus	Bubble	Element lost the logical focus.
PreviewKeyDown, KeyDown	Tunnel, Bubble	Key pressed.
PreviewKeyUp, KeyUp	Tunnel, Bubble	Key released.
PreviewTextInput, TextInput	Tunnel, Bubble	Element received text input.

Keyboard State

- The Keyboard class provides a static property called **Modifiers**
- You can read this at any time to find out which modifier keys, such as the Alt, Shift, and Ctrl keys, are pressed.

```
if ((Keyboard.Modifiers & ModifierKeys.Control) != 0)
   { isCopy = true; }
```

- Keyboard also provides the IsKeyDown and IsKeyUp methods
  - which let you query the state of any individual key

```
bool homeKeyPressed = Keyboard.IsKeyDown(Key.Home);
```

- The state information returned by Keyboard does not represent the current state
  - It represents a snapshot of the state for the event currently being processed