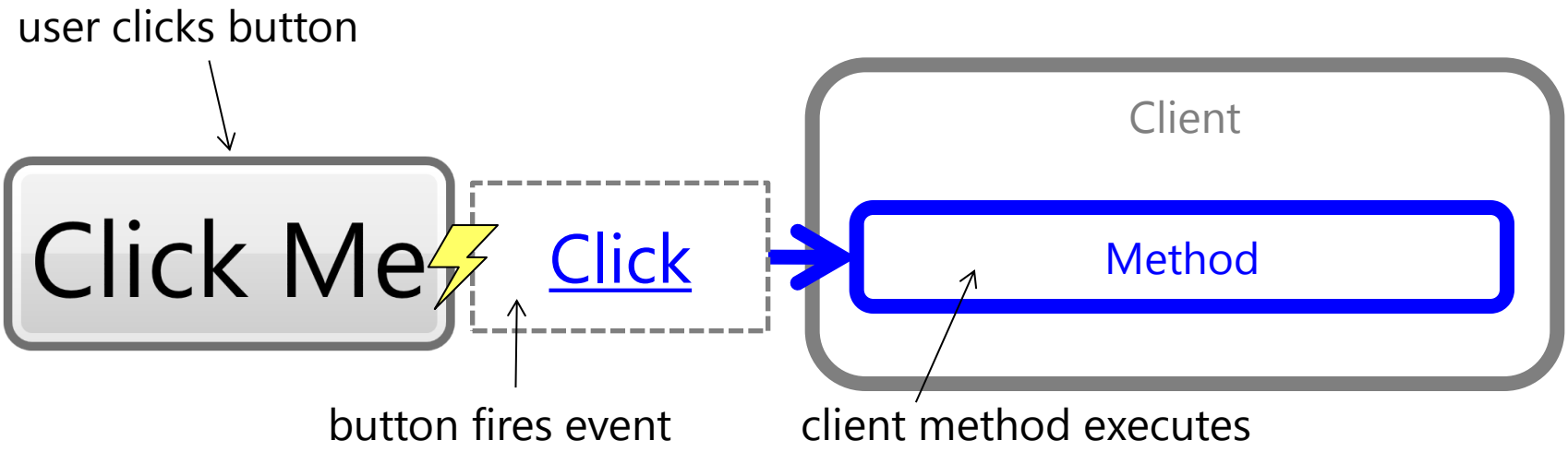
Input Management

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Event review

- UI controls commonly declare events to expose activity
  - raised in response to keyboard, mouse or stylus input



WPF events

- WPF controls use events to report activity

Button calls `Click` event

TextBox uses `TextChanged` event

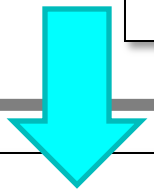
ComboBox uses `SelectionChanged` event

Handling events

- Events are connected to handlers to provide behavior

```
theButton.Click += OnClick;
```

```
<Button
    x:Name="theButton"
    Content="Main Button"
    Click="OnClick" />
```

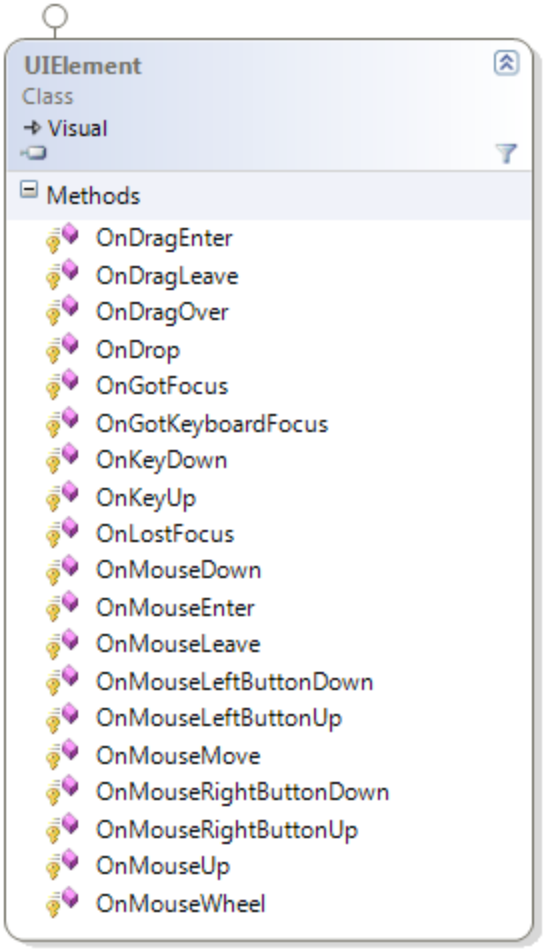


```
void OnClick(object sender, RoutedEventArgs e)
{
    Button button = (Button) sender;
    button.Content= "Clicked";
}
```

Handling events within the class

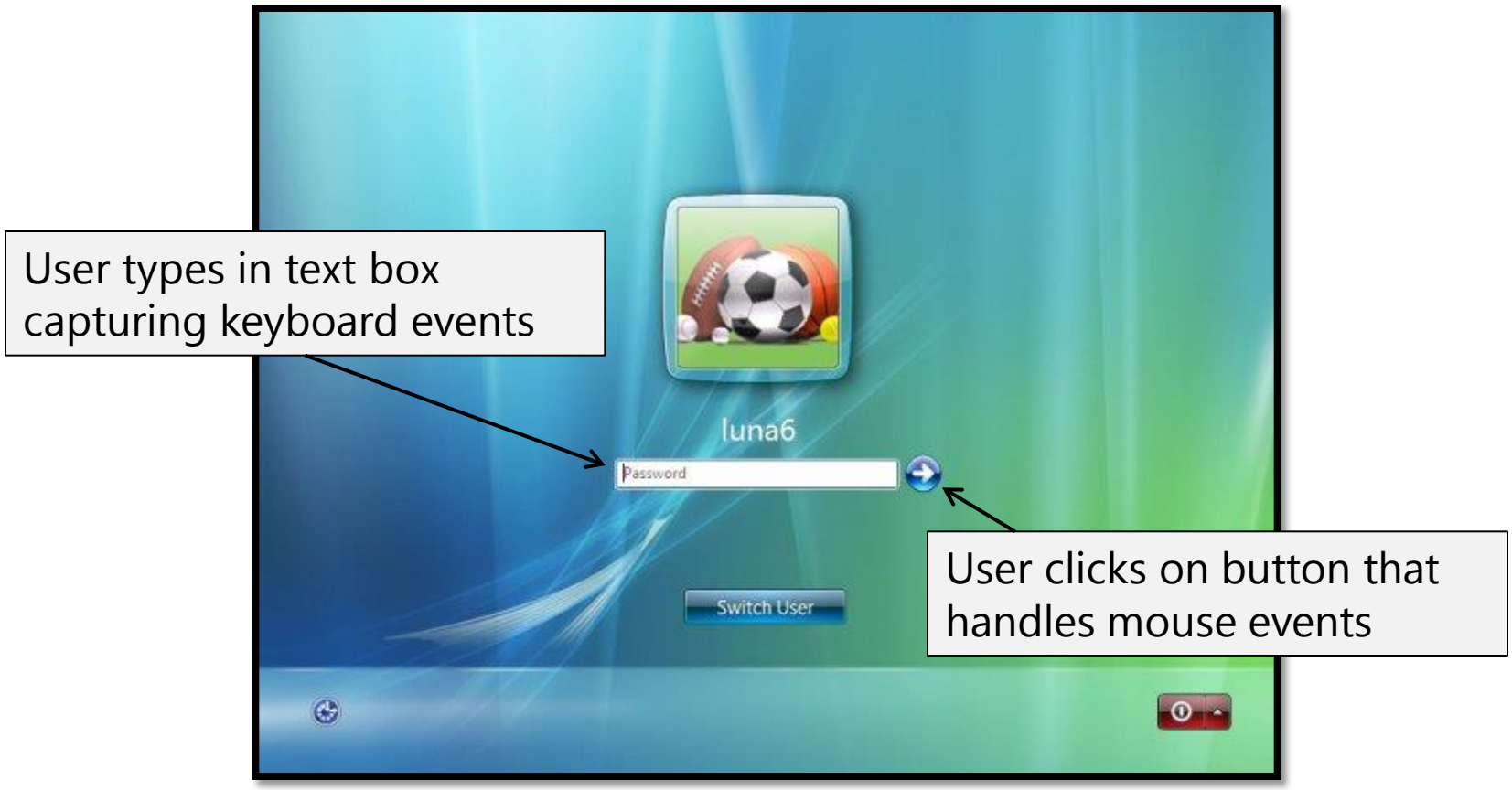
- Derived classes can also override virtual methods for events
  - generally raised *before* event variety
  - should call base implementation to ensure proper control behavior

```
partial class MainWindow : Window
{
    protected override void OnMouseDown(
                MouseButtonEventArgs e)
    {
        base.OnMouseDown(e);
    }
}
```



Windows input dispatching

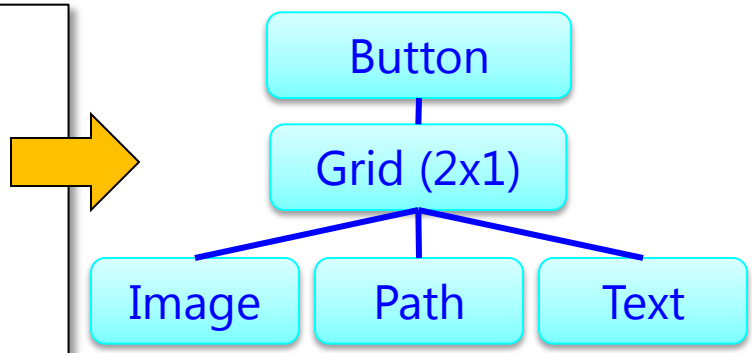
- Windows sends input to UI control with focus or under cursor



Reminder: WPF controls can be composed

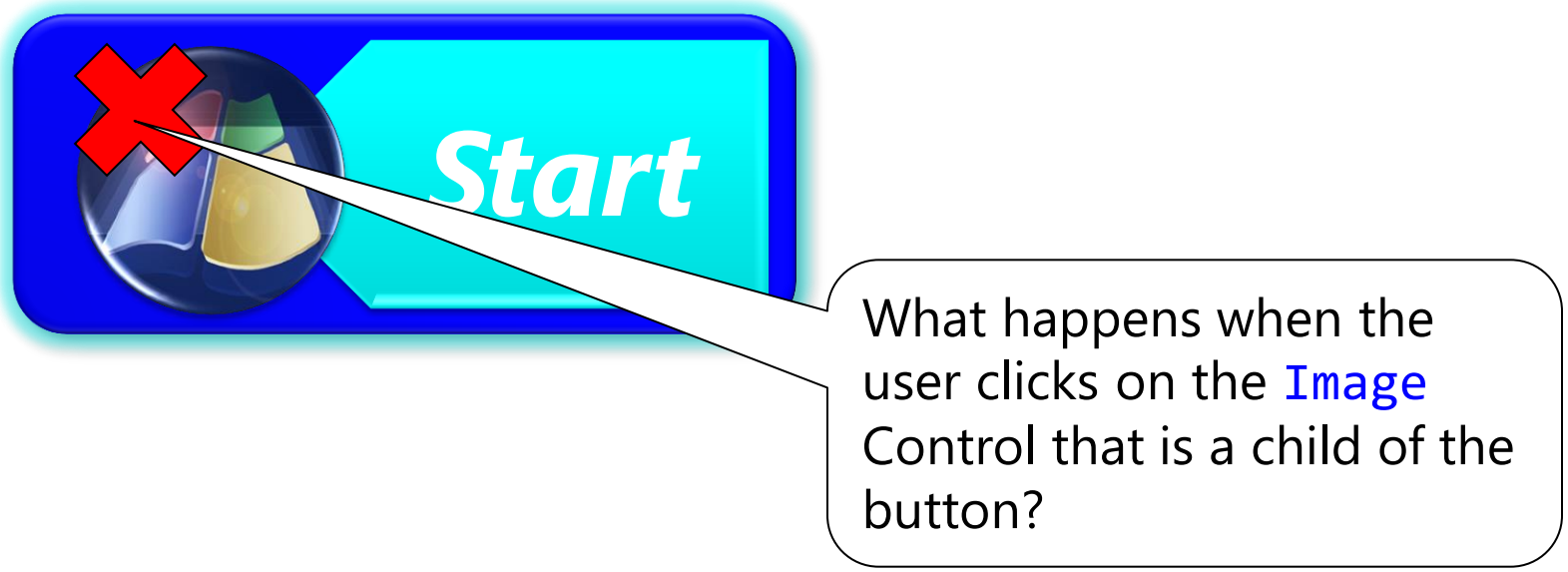
- WPF controls can be built from a variety of content
  - including other controls


```
<Button>
   <Grid>
      <Image Grid.Column="0" ... />
      <Path Grid.Column="1" ... />
      <TextBlock Grid.Column="1" ... />
   </Grid>
</Button>
```



Problem [what should get the input?]

- WPF control that receives event from Windows may not be prepared to handle it



One possible solution

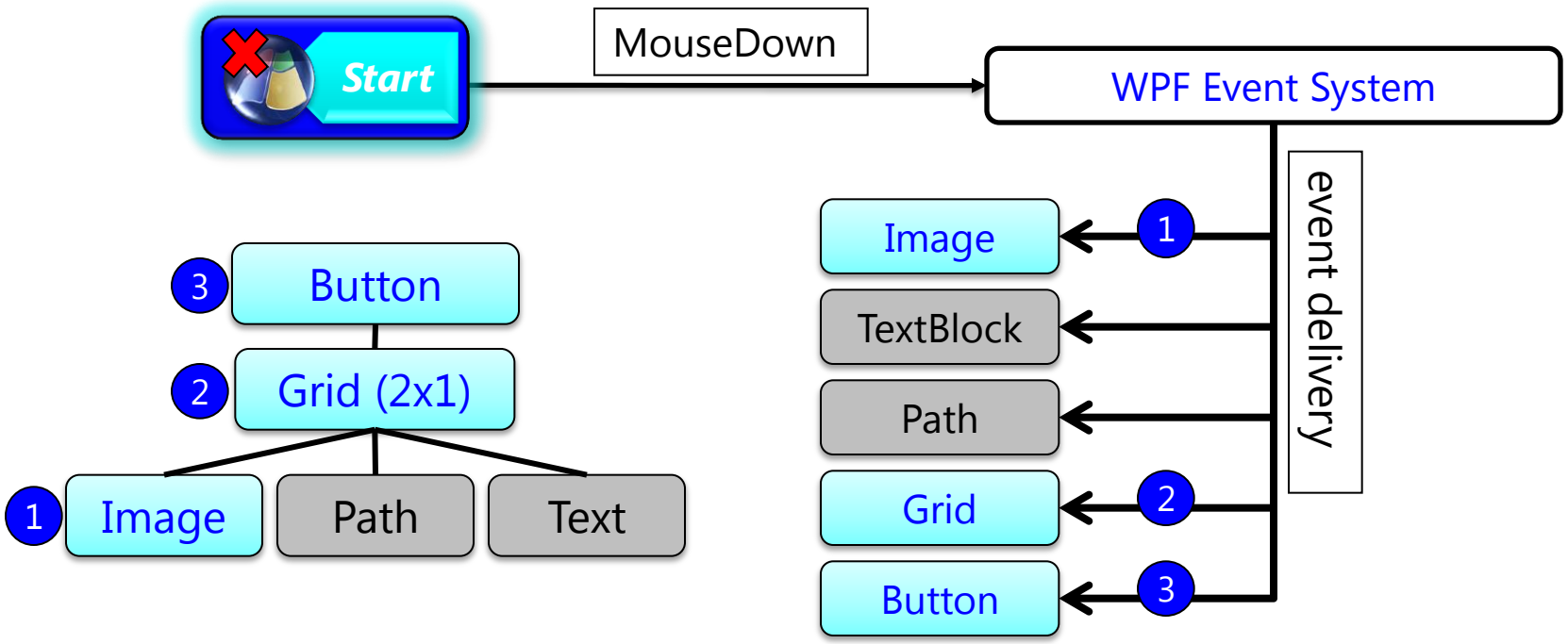
- Could have handlers on each element that raises events
  - not very elegant

You could do this …

```
theButton.Click += OnClickHandler;
imageChild.MouseButtonDown += OnClickHandler;
textChild.MouseButtonDown += OnClickHandler;
pathChild.MouseButtonDown += OnClickHandler;
```

Introducing Routed Input Events

- WPF events are sent to child *and* parent
  - ensure proper behavior when complex content in element
  - walks the visual tree sending event to each interested element



`Image` and `Button` will both see `MouseDown` event

New solutions to old problems

- Event handlers are commonly consolidated to a single function
  - handler function specified on every control


```
<GroupBox Header="Background">
   <StackPanel>
      <RadioButton Click="ChangeBkgnd">
            Red</RadioButton>
      <RadioButton Click="ChangeBkgnd">
            Green</RadioButton>
      <RadioButton Click="ChangeBkgnd">
            Blue</RadioButton>
   </StackPanel>
</GroupBox>
```

it would be more convenient to wire up event handler
in one place instead of every radio button

Listening to child events

- `TypeName.Event` *attaches* event on parent in XAML
  - parent elements can listen for child events directly

each radio button `Click` will be handled by `GroupBox` method

```
<GroupBox Header="Background"
          RadioButton.Click="ChangeBkgnd">
   <StackPanel>
      <RadioButton>Red</RadioButton>
      <RadioButton>Green</RadioButton>
      <RadioButton>Blue</RadioButton>
      <Button>Reset to Default</Button>
   </StackPanel>
</GroupBox>
```

```
void ChangeBkgnd(object sender, EventArgs e) {...}
```

Be careful with child event handling

- Often events are inherited from base classes …
  - **`RadioButton.Click`** is actually **`ButtonBase.Click`**
  - same event used for **`Button`**, **`CheckBox`** and **`ToggleButton`**
  - **worse** – some controls (**`Slider`**, **`TreeView`**) use buttons too!

```
<GroupBox Header="Background"
          RadioButton.Click="ChangeBkgnd">
   <StackPanel>
      <RadioButton>Red</RadioButton>
      <RadioButton>Green</RadioButton>
      <RadioButton>Blue</RadioButton>
      <Button>Reset to Default</Button>
   </StackPanel>
</GroupBox>
```

Can lead to unexpected handling of events from children

What originated the event?

- Sender will always be element where event is hooked up
  - handler likely needs to know which control generated event

Sender will be Window not Button!

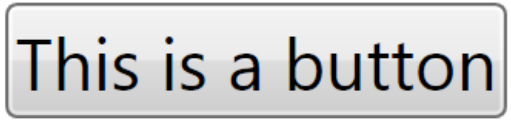
```
<Window Button.Click="WatchMeCrash">
   <Button>
      <Grid>
         <Image Grid.Column="0" ... />
         <Path Grid.Column="1" ... />
         <TextBlock Grid.Column="1" ... />
      </Grid>
   </Button>
</Window>
```

```
void WatchMeCrash(object sender, EventArgs e)
{
   Button button = (Button) sender;
   ...
}
```

Proper event sender determination

- Identity of event source is passed via `RoutedEventArgs`
  - **`Source`** identifies *logical tree* element raising event
  - **`OriginalSource`** identifies *visual tree* element raising event

```
<Button MouseRightButtonDown="RightButton_Down">
   This is a button
</Button>
```




```
void RightButton_Down(object sender, RoutedEventArgs e)
{
   object source = e.Source;
   object original = e.OriginalSource;
   ...
}
```

`Source` will be Button

`OriginalSource` could be TextBlock or ButtonChrome

Hooking events in code behind

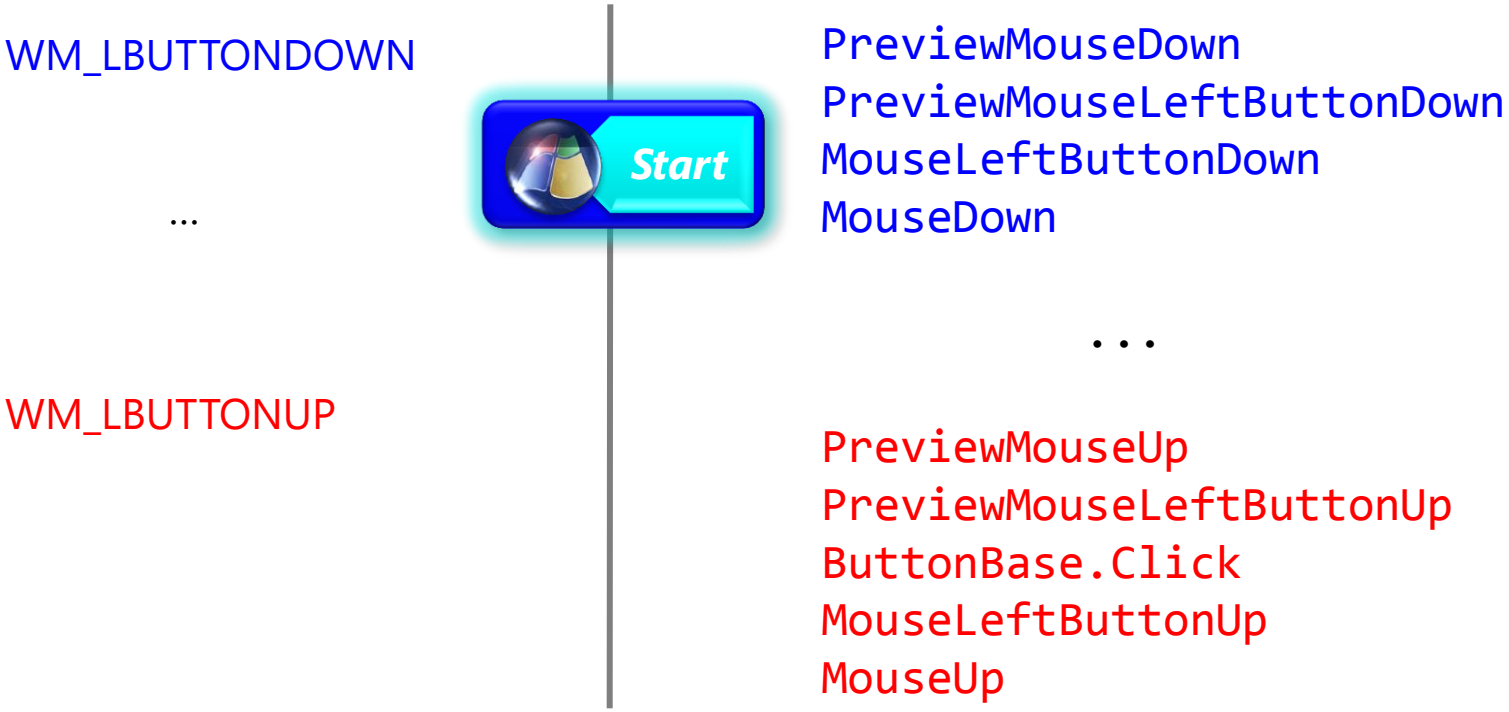
- Wiring up events in XAML mixes design and behavior
  - should prefer to add handlers in code

```
partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        button1.Click += OnClick;

        stackPanel.AddHandler(Button.ClickEvent,
                    new RoutedEventHandler(OnClick));
    }
}
```

`AddHandler` allows any element to register for propagated events

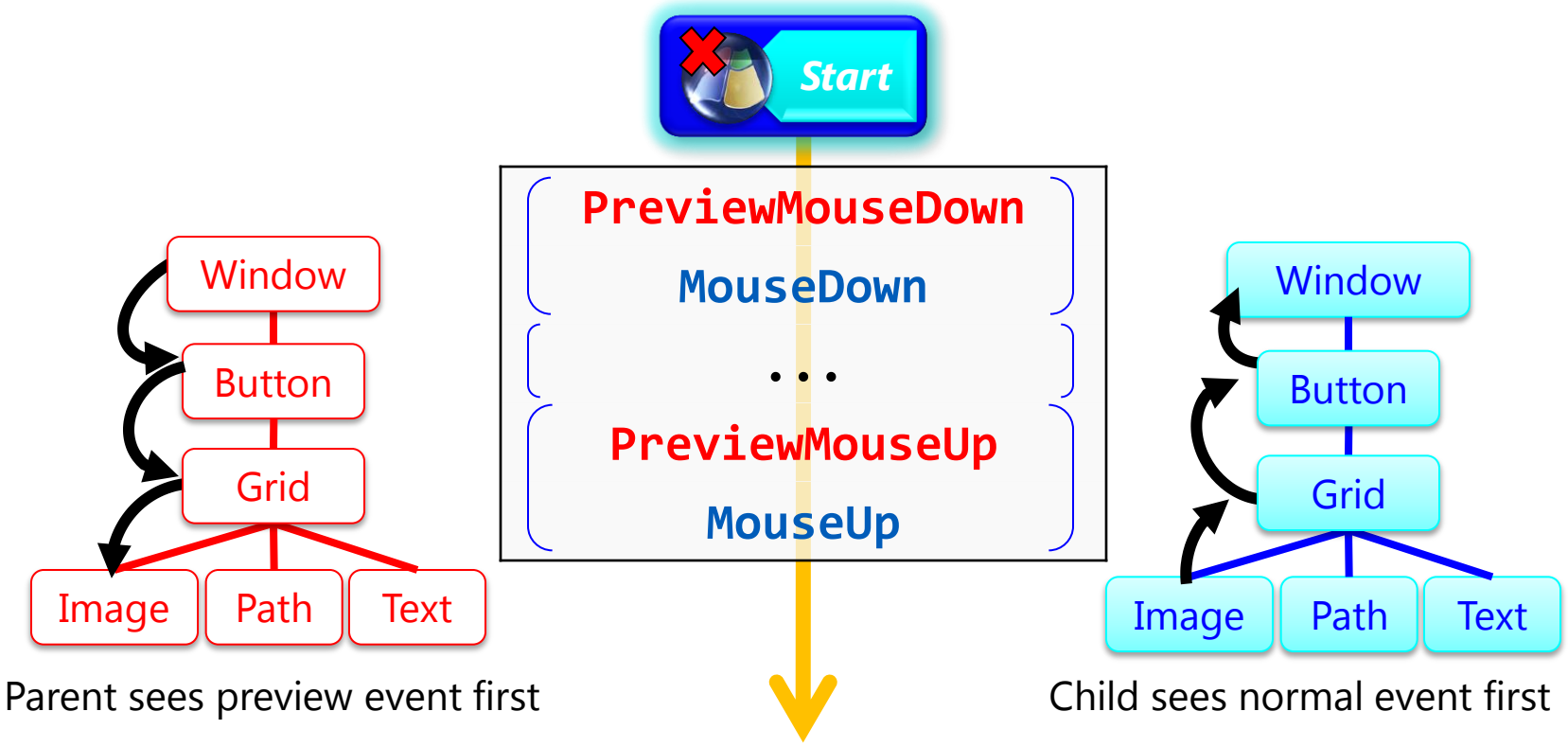
Windows input => WPF events

- WPF adds a rich event model on top of the raw input events
  - provides flexible handling model



Event ordering

- Raw input events are generally sent in pairs
  - preview event sent to parent then child (tunnel)
  - normal event sent to child followed by parent (bubble)



Stopping Event Propagation

- Sometimes it is desirable to "eat" events in the handler
  - **`RoutedEventArgs.Handled`** allows handler to stop event

```
<Window ContentElement.PreviewKeyDown="CheckHelpKey">
   <StackPanel>
       ...
      <TextBox />
   </StackPanel>
</Window>
```

```
void CheckHelpKey(object sender, KeyEventArgs e)
{
   if (e.Key == Key.F1)
   {
      ShowHelp();
      e.Handled= true;
   }
}
```

TextBox will never see F1 key because the parent stops the Preview event

How does this all work?

- WPF implements a completely new eventing model
  - **`RoutedEvent`** is a class registered with WPF and is used as a *key* to raise and respond to event
- `Control` and `Shape` have support for manipulating events
  - **`RaiseEvent`** used to send notification through visual tree
  - **`AddHandler`** used to register handler for event
- Applications should expose activity through routed events
  - makes it consistent with the rest of WPF
  - allows for preview and parent element handling
  - XAML can wire up to the normal .NET event style too

Creating a custom RoutedEvent [Step 1]

- `RoutedEvent` is registered with WPF as a public static field
  - base support included with all **`Control`** and **`Shape`** classes

```
partial class ColorChooserControl
{
   public static readonly RoutedEvent ChooseColorEvent =
      EventManager.RegisterRoutedEvent(
         "ChooseColor",
         RoutingStrategy.Bubble,
         typeof(EventHandler<ColorChooserEventArgs>),
         typeof(ColorChooserControl));
}
```

Creating a custom RoutedEvent [Step 2]

- **Create custom `EventArgs` type [optional]**
  - must derive from **`RoutedEventArgs`**

```
public class ColorChooserEventArgs : RoutedEventArgs
{
   public readonly Color Color;

   public ColorChooserEventArgs(
        Color clr, RoutedEvent re, object source)
          : base(re, source)
   {
       Color = clr;
   }
}
```

Creating a custom RoutedEvent [Step 3]

- **Event wrappers expose routed events for procedural code**
  - allow normal CLR event register and unregister style

```
partial class ColorChooserControl
{
    public event
      EventHandler<ColorChooserEventArgs> ChooseColor
    {
        add
        {
            base.AddHandler(ChooseColorEvent, value);
        }
        remove
        {
            base.RemoveHandler(ChooseColorEvent, value);
        }
    }
}
```

Creating a custom RoutedEvent [Step 4]

- Control raises event through WPF routed event system
  - can check the **`Handled`** flag to see if it was handled by app

```
partial class ColorChooserControl
{
   bool RaiseChooseColorEvent(Color color)
   {
      ChooseColorEventArgs ea =
         new ChooseColorEventArgs(color,
                        ChooseColorEvent, this);
      base.RaiseEvent(ea);
      return ea.Handled;
   }
}
```

Capturing input

- Input events are often managed in groups
  - **xxxDown**, [**xxxMove**], **xxxUp**
- Should capture input device on initial event to ensure delivery
  - otherwise you might not receive entire group
  - **CaptureMouse**, **CaptureTouch**, **CaptureStylus**
- Must `ReleasexxxCapture` when finished with operation

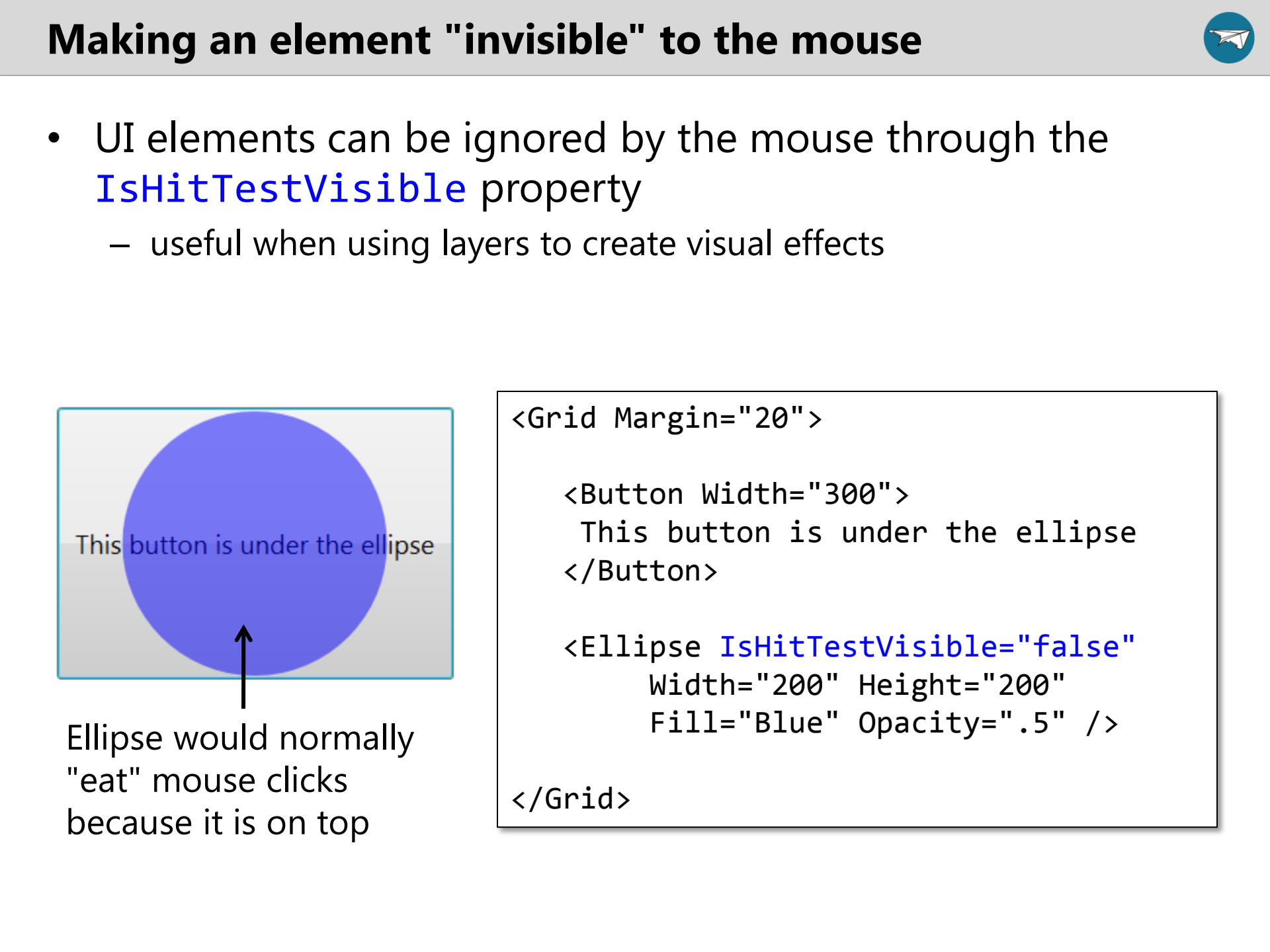
```
public void OnMouseLeftButtonDown(object s, MouseButtonEventArgs e)
{
   CaptureMouse();
}

public void OnMouseLeftButtonUp(object s, MouseButtonEventArgs e)
{
   ReleaseMouseCapture();
}

protected override virtual OnLostMouseCapture(MouseEventArgs e) { ... }
```

Hit-testing elements in your UI

- Identifying child or overlapping elements involves *hit-testing*
  - converts mouse coordinates to items being clicked on
- Several ways
  - **`UIElement.IsMouseOver`** set when mouse over element[1]
  - **`UIElement.InputHitTest`** method for single controls
  - **`VisualTreeHelper.HitTest`** for everything else[2]

```
public void OnMouseRightButtonDown(object sender,
                                   MouseButtonEventArgs e)
{
   Point pt = e.GetPosition(rootCanvas);
   UIElement currElement = rootCanvas.InputHitTest(pt) as UIElement
   if (currElement != null)
   {
      DoSelect(currElement);
   }
}
```

Making an element "invisible" to the mouse



- UI elements can be ignored by the mouse through the `IsHitTestVisible` property
  - useful when using layers to create visual effects

Ellipse would normally "eat" mouse clicks because it is on top

```
<Grid Margin="20">

   <Button Width="300">
    This button is under the ellipse
   </Button>

   <Ellipse IsHitTestVisible="false"
        Width="200" Height="200"
        Fill="Blue" Opacity=".5" />

</Grid>
```

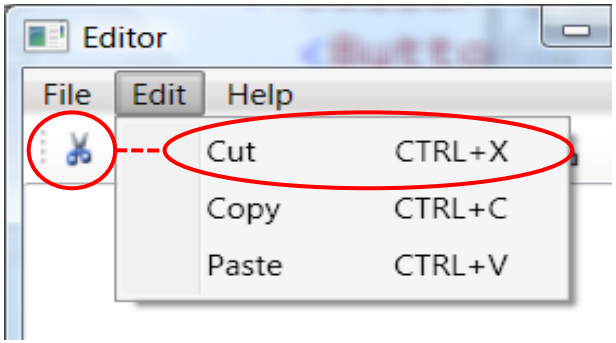
[Motivation] Centralizing event handlers

- Most applications have multiple ways to perform single action
  - toolbar, menu, accelerator keys

```
void OnMenuCut(…)
{
    Perfo
}
```

```
void OnToolbarCut(…)
{
   Perfor
}
```

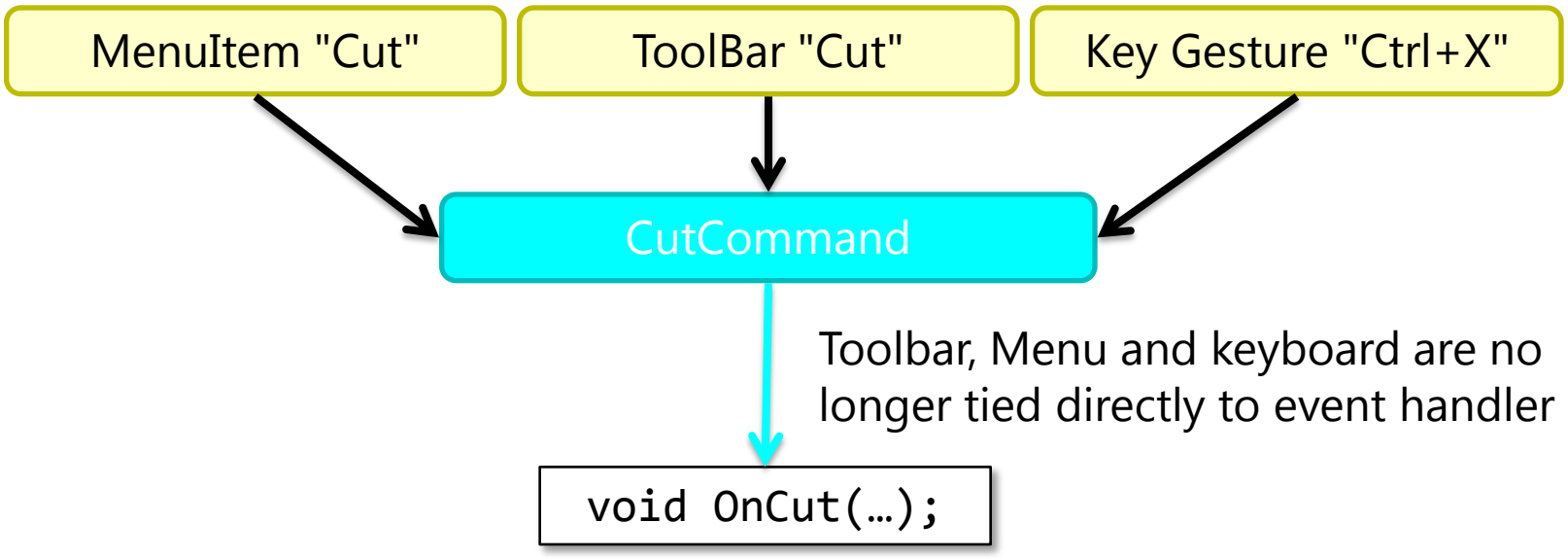
```
void OnCtrlX(…)
{
    PerformCut();
}
```



three event handlers are required to all do the same thing

Introducing the command pattern

- `ICommand` defines an action the user may execute
  - ties various input mechanisms to an implementation
  - promotes "loose" coupling between UI and handler
  - key ingredient to utilizing presentation model patterns



Commands in WPF

- `RoutedCommand`[1] implements `ICommand`
  - can be bound to control or input events
  - **`Execute`** method invokes "action"
  - **`CanExecute`** method decides current validity of action

```
public class RoutedCommand : ICommand
{
   public string Name { get; }
   public InputGestureCollection InputGestures { get; }

   public bool CanExecute(object parameter,
                          IInputElement target);
   public void Execute(object parameter,
                       IInputElement target);
}
```

Using Built-in commands

- Static classes provide built-in commands for your use

ApplicationCommands

- Cut, Copy, Paste
- Close, Open, Undo, Redo, Print, etc.

NavigationCommands

- NextPage, PreviousPage
- Zoom, GotoPage, Refresh, etc.

MediaCommands

- Pause, Play, Record, Rewind
- ChannelUp, ChannelDown, IncreaseVolume, etc.

EditingCommands

- Backspace, Delete, MoveDownByLine
- SelectLeftByWord, ToggleBold, etc.

Creating your own RoutedCommands

- Built by instantiating `RoutedCommand` or `RoutedUICommand`
  - should be **`public`** and **`static`**
  - applications that define multiple **`Commands`** should put them into a dedicated static class

```
public static class MyCommands
{
   public static readonly RoutedUICommand ShutdownApp =
      new RoutedUICommand("Shutdown", "ShutdownApp",
           typeof(MyCommands));
    ...
}
```

Associating Controls to Commands

- Many controls can be configured to invoke `ICommand`[1]
  - implementation of **`ICommandSource`**
  - event still raised prior to command invocation

```
<Button Command="ApplicationCommands.Cut"
        Content="_Cut" />

<Hyperlink Command="local:MyCommands.ShutdownApp">
   Click to shutdown the application
</Hyperlink>

<Menu>
   <MenuItem Command="Cut"/>
</Menu>
```

`TypeConverter` makes syntax convenient for built-in commands

controls are no longer associated directly with behavioral logic

Associating keyboard and mouse input to Commands

- Command can be triggered from keyboard or mouse input
  - input bindings added in code or XAML (preferred)
  - best practice to add bindings to root **`Window`** element

```
public MainWindow()
{
   this.InputBindings.Add(new InputBinding(
               ApplicationCommands.Undo,
               new MouseGesture(MouseAction.MiddleClick,
                         ModifierKeys.Control)));
    ...
}
```

```
<Window.InputBindings>
   <MouseBinding Command="ApplicationCommands.Undo"
                 Gesture="Control+MiddleClick" />
   <KeyBinding Command="local:MyCommands.ShutdownApp"
               Key="F3" />
</Window.InputBindings>
```

Associating actions with Commands

- `CommandBinding` ties actions to `RoutedCommand`
  - events raised in response to command invocation
  - event handler decides behavior to execute in response

```
partial class CommandBinding
{
    ICommand Command { get; set; }

    event CanExecuteRoutedEventHandler PreviewCanExecute;
    event CanExecuteRoutedEventHandler CanExecute;
    event ExecutedRoutedEventHandler PreviewExecuted;
    event ExecutedRoutedEventHandler Executed;
}
```

Associating actions to Commands

- `UIElement.CommandBindings` holds command bindings
  - commands are "bubbled" up visual tree starting with sender
  - can be wired up in code behind (preferred) or XAML

```
public class PrintWindow : Window
{
   public PrintWindow()
   {
      InitializeComponent();

      this.CommandBindings.Add(
        new CommandBinding(
            ApplicationCommands.Print,
            this.OnPrint,
            this.OnCanPrint));
   }
   ...
}
```

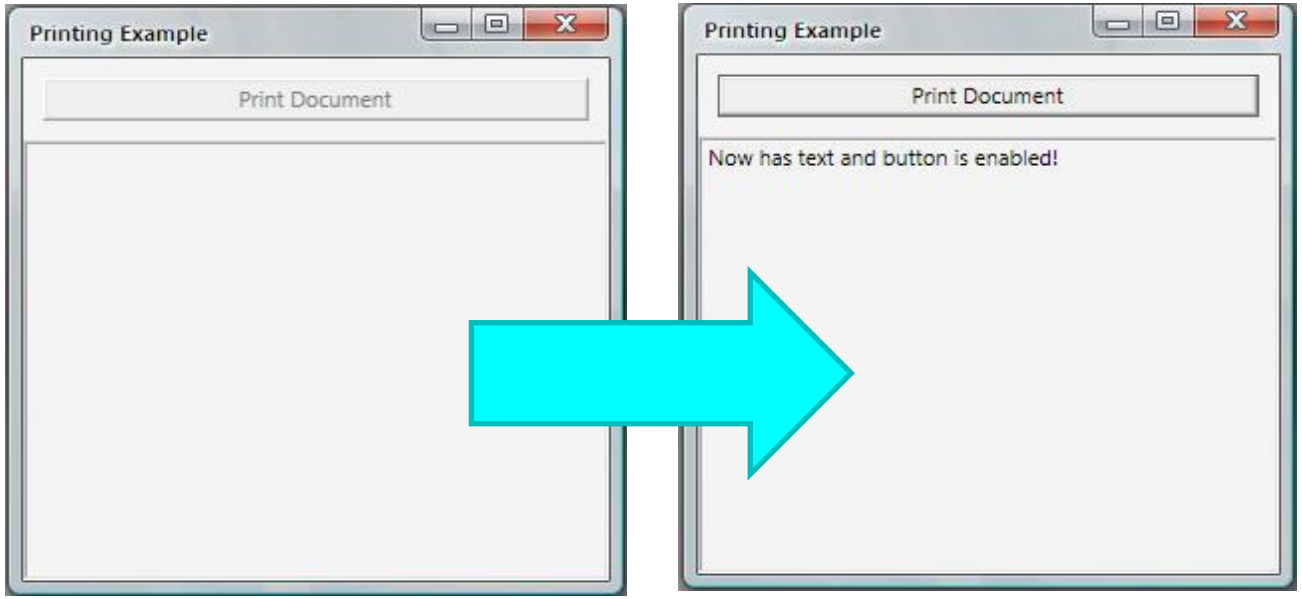
command to handle

"Executed" handler

"CanExecute" handler

Enabling and disabling command execution

- `CanExecute` handler determines whether command can run
  - controls bound to the command will be disabled automatically



`Button` wired to `ApplicationCommands.Print`
`CanExecute` handler returns whether text is present in `TextBox`

Creating command event handlers

- `CanExecute` event raised periodically to check state
  - typically in response to UI focus changes
- `Execute` event raised to invoke command logic

```
void OnCanPrint(object sender, CanExecuteRoutedEventArgs e)
{
   e.CanExecute = !isPrinting &&
                  textBox != null &&
                  textBox.Text.Length> 0;
   e.Handled = true;
}

void OnPrint(object sender, ExecutedRoutedEventArgs e)
{
   isPrinting = true;
}
```

Forcing WPF to query execution status

- Outside criteria may enable or disable commands
  - need to notify WPF that conditions have changed
- `CommandManager` static class provides commanding support
  - associate handlers to **`Type`** (vs. instances)
  - locate proper command for input gestures
  - notify commands that state has changed

```
void OnPrintingCompleted()
{
   isPrinting = false;
   CommandManager.InvalidateRequerySuggested();
}
```

command bindings will be re-evaluated and `Button` will re-enable

Passing parameters to command handlers

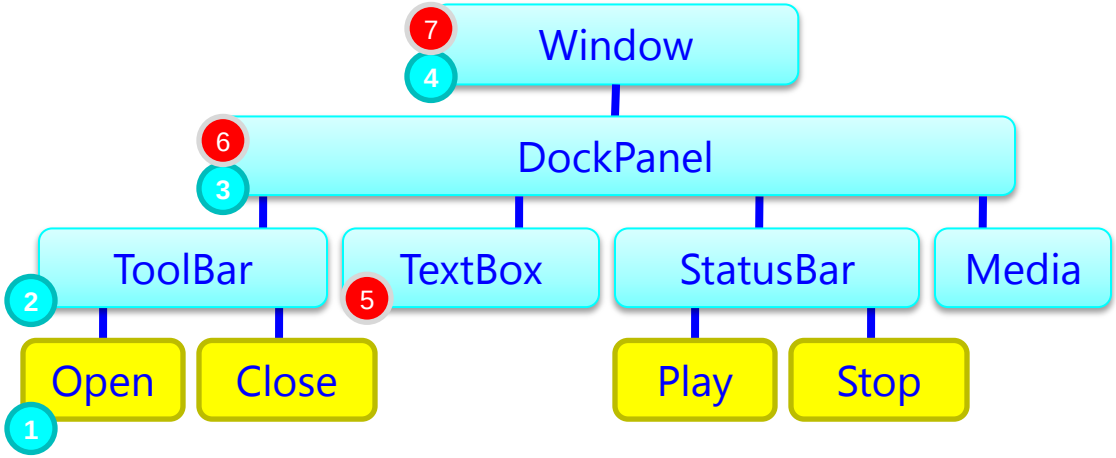
- Optional data can be passed using `CommandParameter`
  - target will receive parameter in **`Executed`** handler

```
<Window.CommandBindings>
   <CommandBinding Command="NavigationCommands.GoToPage"
                   Executed="OnNavigateUrl" />
</Window.CommandBindings>
...
<Button Command="NavigationCommands.GoToPage"
        CommandParameter="http://www.develop.com" />
<Button Command="NavigationCommands.GoToPage"
        CommandParameter="http://www.microsoft.com" />
```

```
void OnNavigateUrl(object sender, ExecutedRoutedEventArgs e)
{
   string url = (string)e.Parameter;
}
```

When the command is invoked …

- WPF searches for a `CommandBinding` to execute[1]
  - starts at command source, fires **Preview** + **Executed**
  - if no handler found, *might* re-raise events on focused element



first matching `CommandBinding` is executed – all others ignored

Last but not least

- Commands may be executed directly in code behind
  - useful when executing behavior from unrelated code
  - also useful for unit testing as we will see later

```
partial class MainWindow
{
   ...
   public override void OnLostFocus(RoutedEventArgse)
   {
      if (MyCommands.ShutdownApp.CanExecute(null, this))
          MyCommands.ShutdownApp.Execute(null, this);
   }
}
```

parameter for command

starting element for handler search, pass `null` for focused element

Summary

- Event architecture allows interception of events at any level
  - obsolesces necessity to override controls to change behavior
- Preview events allow parent to see event prior to children
  - parent can then stop regular event from occurring
- *Sender* is always where event handler was attached
  - use **`RoutedEventArgs`** properties to get actual source
- Commands provide a high level of abstraction
  - prefer to use these vs. binding to events directly
- WPF provides several pre-built Commands for you to use
  - some controls have default handlers for these commands