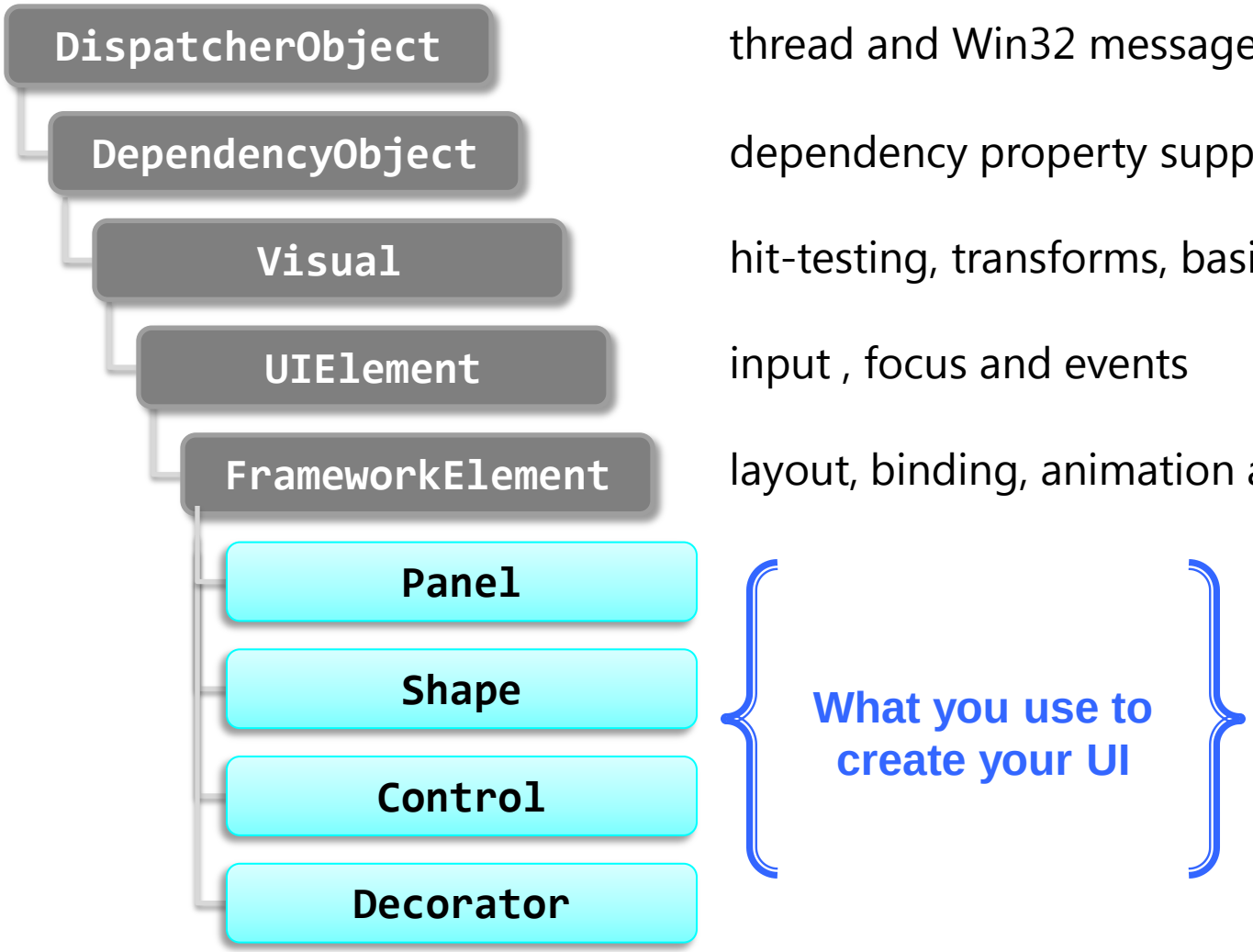
Framework Architecture

DEVELOPMENTOR

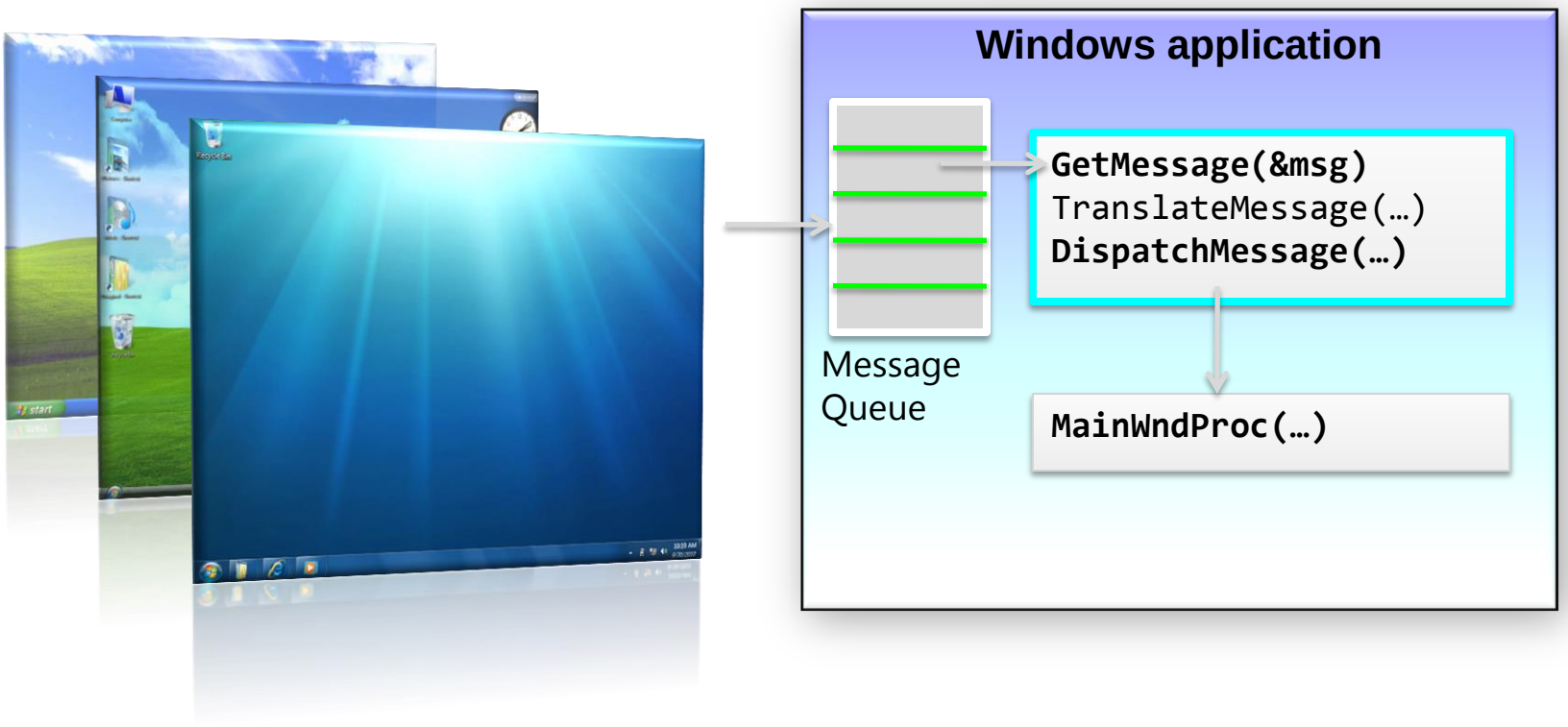
DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Understanding the class hierarchy



Win32 history lesson

- WPF applications must run on top of Windows
  - must follow the rules of Win32 programming
- At the core of every Windows program is a message pump



What does this job in WPF?

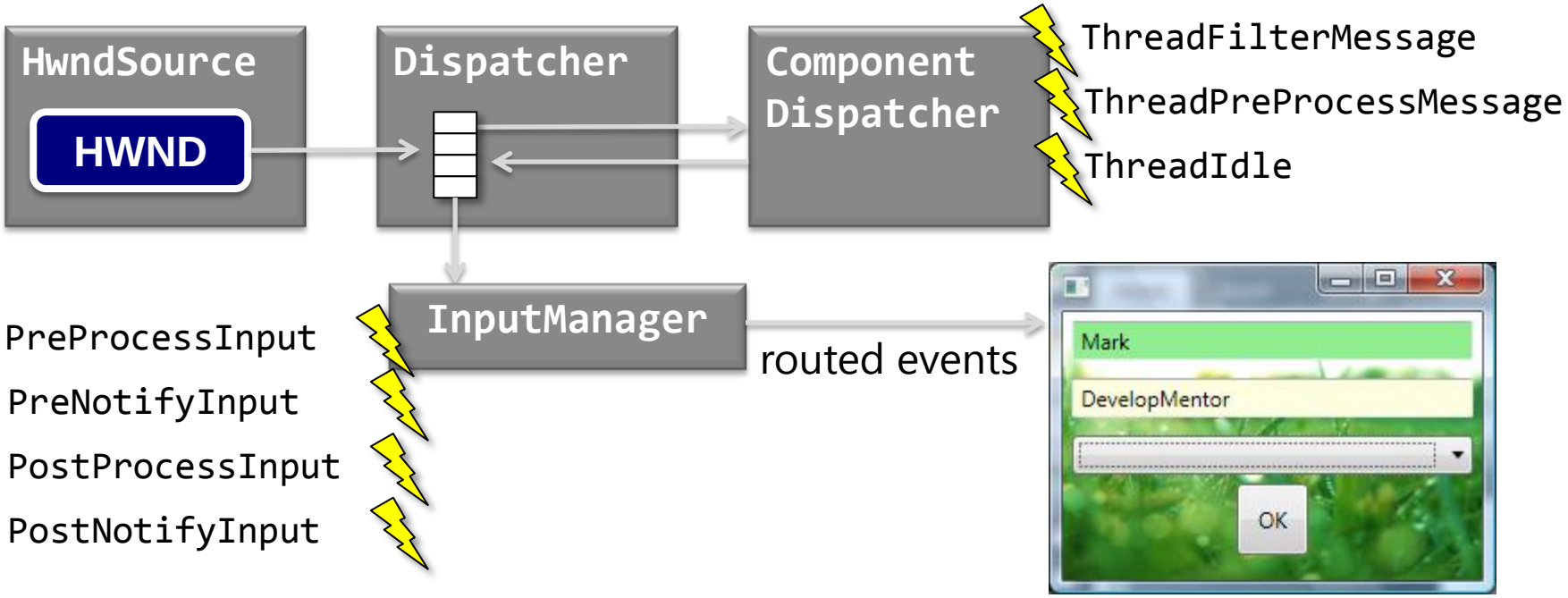
- The `Dispatcher` pumps Win32 messages for the application
  - restricted to a single thread (it's creator)
  - UI elements belong to a single dispatcher
  - inbound events are queued for dispatch to appropriate element

```
public sealed class Dispatcher
{
   public static Dispatcher CurrentDispatcher { get; }
   public DispatcherHooks Hooks { get; }
   public Thread Thread { get; }

   public event ... ShutdownFinished;
   public event ... ShutdownStarted;
   public event ... UnhandledException;
   public event ... UnhandledExceptionFilter;
  ...
}
```

WPF threading architecture

- Message Dispatch is run on primary (creating) UI thread
  - created as part of **`Window`** initialization



DispatcherObject

- Base class for thread-affinitive objects is `DispatcherObject`
  - provides support for identifying and accessing proper thread
  - all **`UIElements`** derive from this class
  - interacts with **`Dispatcher`** to provide single-threaded behavior
  - all properties and methods must be called on UI thread[1]

```
public abstract class DispatcherObject
{
   protected DispatcherObject();

   public Dispatcher Dispatcher { get; }

   public bool CheckAccess();
   public void VerifyAccess();
}
```

What if I need multiple threads?

- `DispatcherObject` prohibits changes on other threads
  - must marshal activity to proper UI thread through **`Dispatcher`**
- `Dispatcher` methods provide fine-grained control
  - **`CheckAccess`** to see if (current thread == UI thread)
  - **`VerifyAccess`** to enforce (current thread == UI thread)
  - **`Invoke`** to call UI thread synchronously
  - **`BeginInvoke`** to call UI thread asynchronously
- Can also use `SynchronizationContext` for agility
  - works in Windows Forms and WPF
- For timer-based callbacks, use `DispatcherTimer`
  - properly marshals to UI thread for callback

Switching to the UI thread

- `Invoke` performs synchronous call to UI thread
  - pauses calling thread until UI thread services request
  - returns result of delegate call (null if no return value)
  - must be cautious of possible deadlock scenarios
- `BeginInvoke` performs asynchronous call to UI thread
  - returns **`DispatcherOperation`** to monitor progress, get results
- All method versions take Delegate and optional parameter(s)
  - no specific signature required
- Can also pass optional priority to `Invoke` / `BeginInvoke`
  - defaults to "Normal"

Example: using the Dispatcher

```
void ProcessData(byte[] data, int count)
{
   Dispatcher dp = textBox1.Dispatcher;
   if (dp.CheckAccess())
   {
      textBox1.Text = ConvertData(data, count);
   }
   else
   {
      dp.Invoke((Action)(() => ProcessData(data, count)));
   }
}
```

If method is called by background thread it will recursively call ProcessData – the second time will be on the UI thread

Getting visuals onto the screen

- WPF utilizes a Retained Mode graphics model
  - entire scene and dirty regions are maintained by WPF
  - apps change properties and WPF determines what is redrawn
- Contrast that to GDI's Immediate Mode graphics model
  - application maintains UI state and dirty regions
  - application decides what to redraw, most punt and redraw screen

Visual change notifications

- WPF caches property values used to render visuals
  - application paint code is only called when something *changes*
  - properties changing size impact layout and positioning

```
class Window
{
    public Brush Foreground
    {
        get { ... }
        set { ... }
    }
}
```

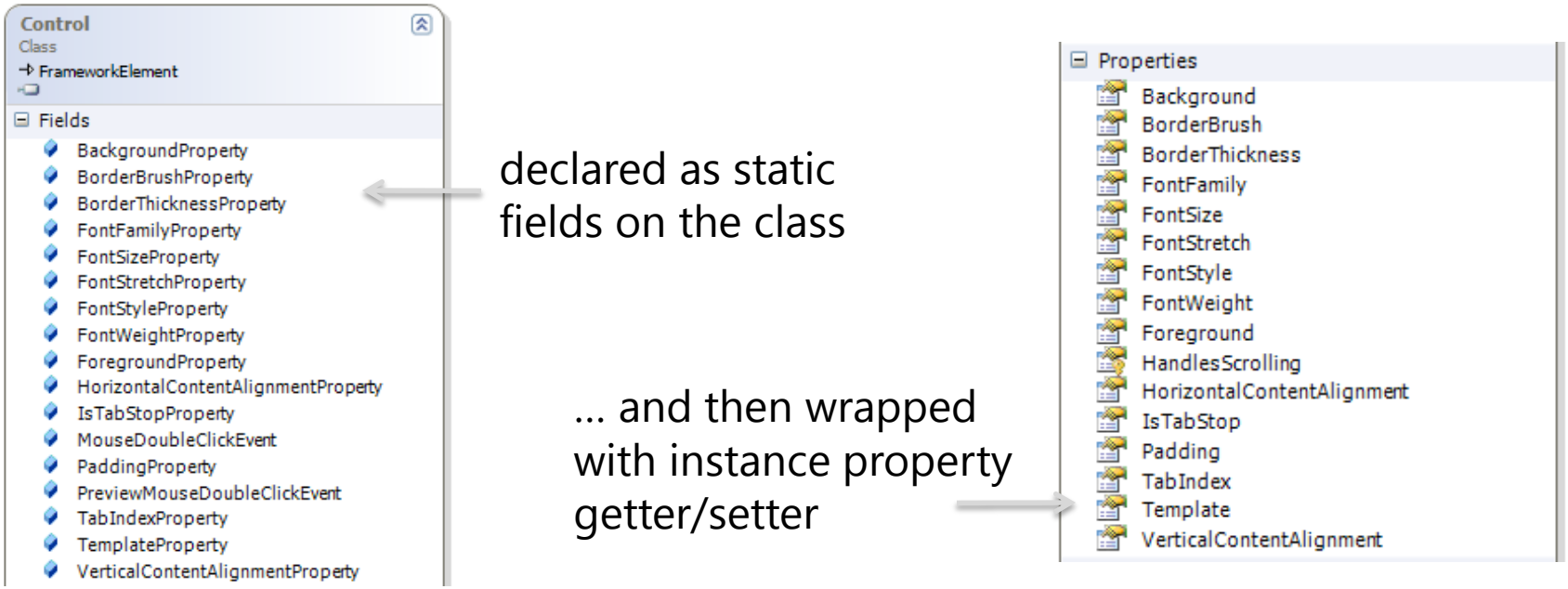
WPF needs to know when property value changes in order to invalidate cache

The cornerstone of WPF: Dependency Properties

- Dependency Properties enhance traditional properties with
  - default values
  - calculated values
  - change notification support
  - property value inheritance
  - adding properties dynamically to classes at runtime
  - metadata
- Used to implement most core WPF features
  - themes and styles
  - data binding
  - animations
  - …

Using Dependency Properties

- DPs look like a regular property to the consumer
  - because they have a normal .NET property "wrapper"
  - but the declaration is quite different



Creating dependency properties

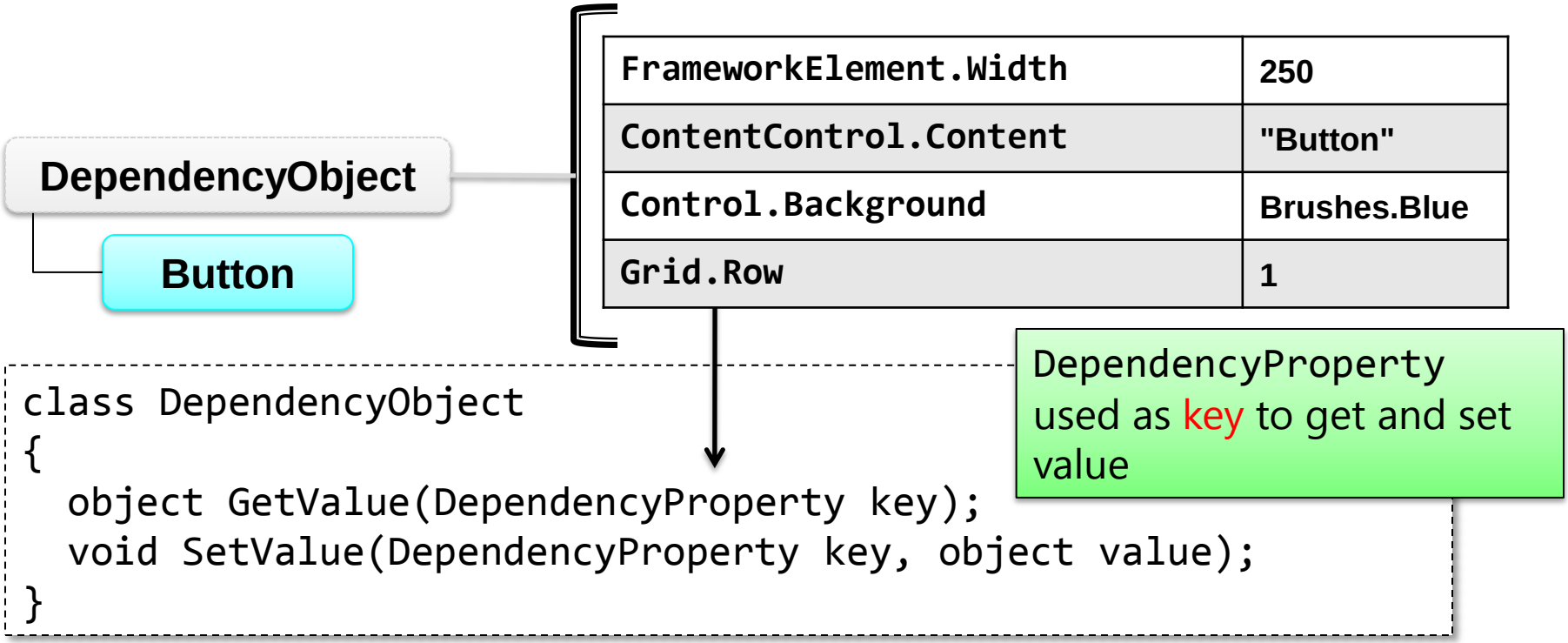
- DependencyProperty class has static Register methods
  - name must be unique on owner type
  - storage type must have public default constructor
  - result generally stored in **public static readonly** field with **Property** suffix

```
public partial class Control
{
  public static readonly DependencyProperty
     BackgroundProperty = DependencyProperty.Register(
            "Background",
            typeof(Brush),
            typeof(Control));
  ...
}
```

name
storage type
owner type

Dependency Property Support

- Fundamental support provided in `DependencyObject`
  - maintains collection of properties that have *set* values[1]
  - provides **`GetValue`** and **`SetValue`** method to alter collection



Providing traditional property wrapper

- Property wrapper gets/sets value in base class collection
  - property value is then managed by **`DependencyObject`**

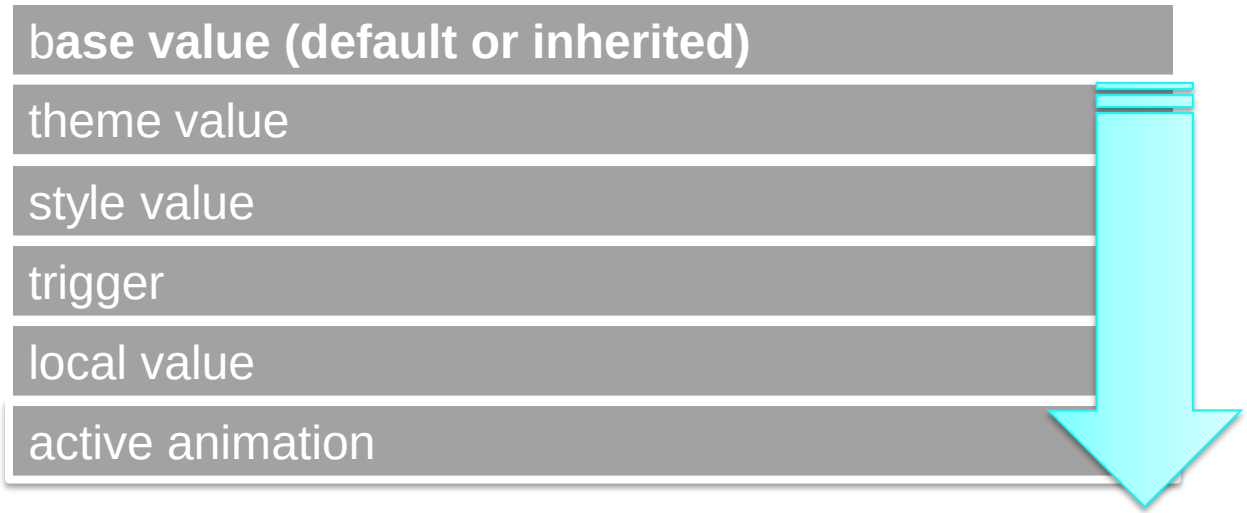
```
public partial class Control
{
  public static readonly DependencyProperty BackgroundProperty;
   ...
  public Brush Background
  {
     get { return (Brush) base.GetValue(BackgroundProperty); }
     set { base.SetValue(BackgroundProperty, value); }
  }
}
```

static `DependencyProperty` used as key into collection

Getting the value of a dependency property

- Current value can be retrieved through property wrapper
  - or directly[1] through **`DependencyObject.GetValue`**
- WPF determines value at runtime when `GetValue` is called
  - can be influenced by a variety of factors

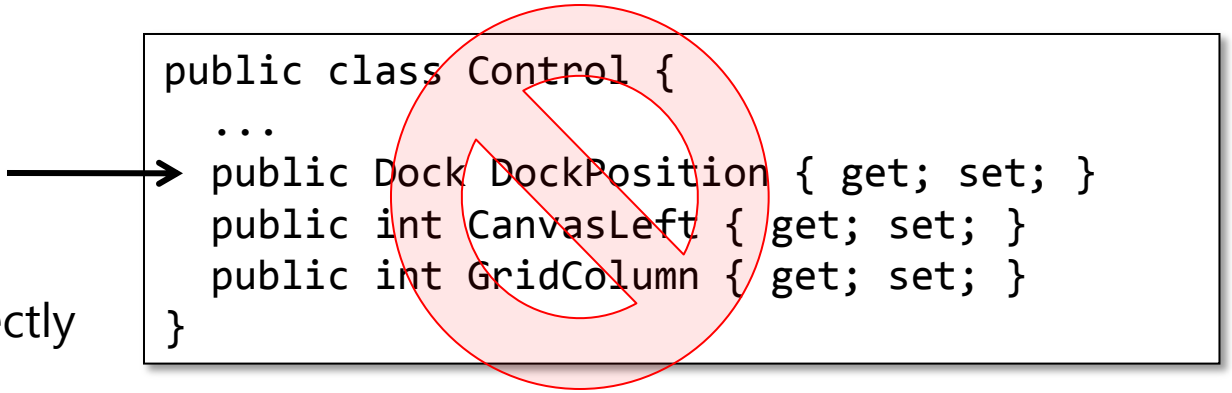
```
Brush bkBrush = (Brush)
button1.GetValue(Control.BackgroundProperty);
```



Motivation [dynamic properties]

- Some properties are only valid in certain context
  - adding all known combinations bloats object size
  - cannot foresee all possible combinations

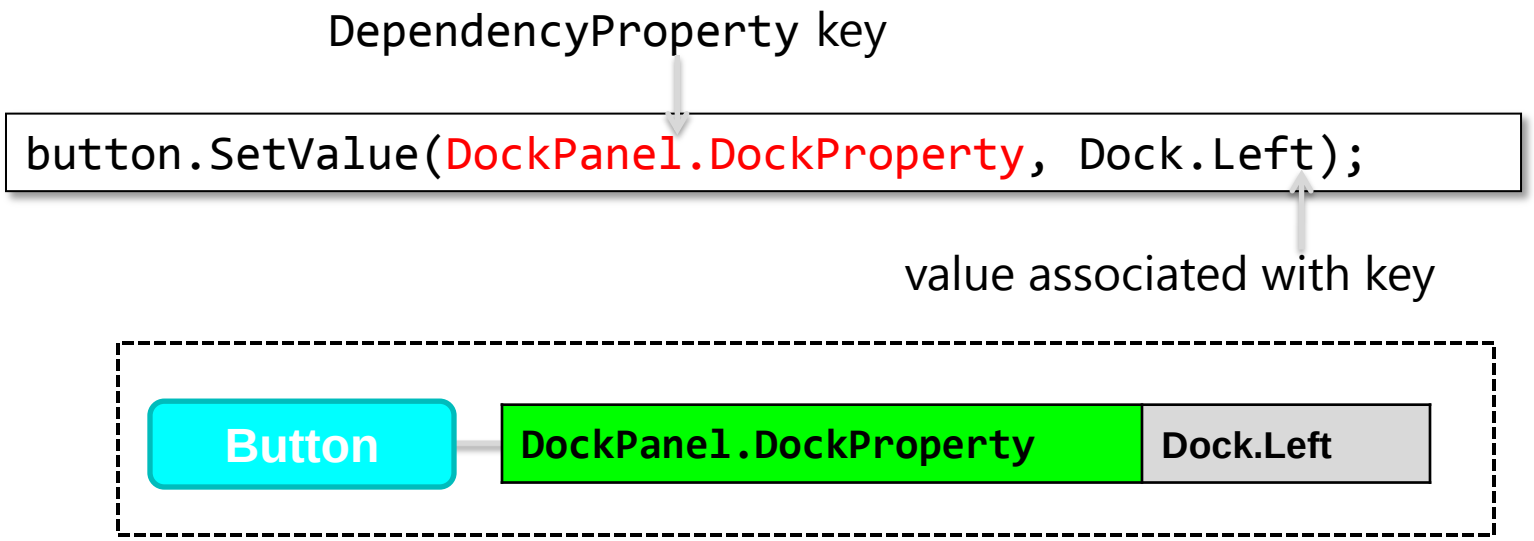
inefficient
and inflexible
to store panel
properties directly



```
public class Control {
  ...
  public Dock DockPosition { get; set; }
  public int CanvasLeft { get; set; }
  public int GridColumn { get; set; }
}
```

Attaching properties at runtime

- Properties can be attached to `DependencyObjects` at runtime
  - provides dynamic extensible property mechanism for WPF
  - used to store context-specific data and extend class behavior
- Values associated with specific object instances
  - … but property definition is defined on separate class



Attached Property static helper functions

- Owner class defines static helpers to get/set the value
  - should be named **`Get[Property]`** and **`Set[Property]`**
  - provides compile-time type safety

```
Button button = new Button();
DockPanel.SetDock(button, Dock.Left);
Canvas.SetLeft(button, 100);
Grid.SetColumn(button, 1);
```

```
public class DockPanel : Panel
{
   public static readonly DependencyProperty DockProperty;
   public static Dock GetDock(UIElement element) {
        (Dock) element.GetValue(DockPanel.DockProperty);
   }
   public static void SetDock(UIElement element, Dock dock) {
        element.SetValue(DockPanel.DockProperty, dock);
   }
}
```

Attached Properties in XAML

- Attached properties accessed using `Type.PropertyName`
  - associates the value with the given **`DependencyProperty`** key

```
<TextBox Name="myTextBox"
    SpellCheck.IsEnabled="True"
    Text="This is mispelled" />
```


```
TextBox myTextBox = new TextBox();
myTextBox.Text = "This is mispelled";

SpellCheck.SetIsEnabled(myTextBox, true);
    or
myTextBox.SetValue(SpellCheck.IsEnabledProperty, true);
```

Creating an attached property

- `DependencyProperty.RegisterAttached` creates property
  - associates property metadata with all **`DependencyObjects`**
  - step can be omitted if not storing any data on types

```
public partial class SpellCheck
{
   public static readonly DependencyProperty
     IsEnabledProperty =
          DependencyProperty.RegisterAttached(
          "IsEnabled",
          typeof(bool),
          typeof(SpellCheck));
     ...
}
```

Providing helper methods for attached properties

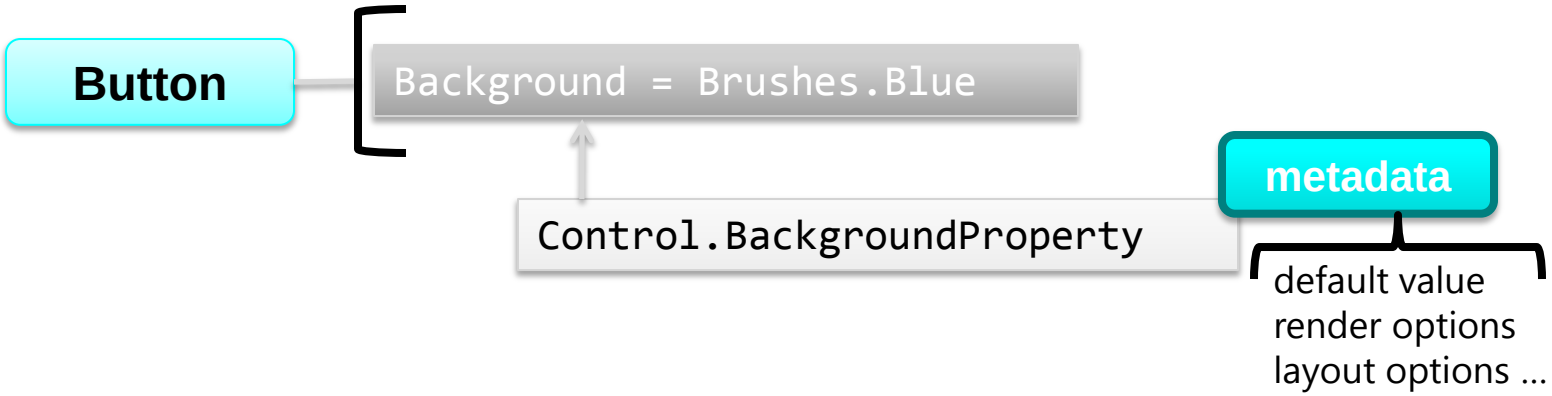
- Helper methods provide type safety for getter and setter
  - **`DependencyObject`** or derived type as first parameter
  - XAML bypasses methods by default[1] but they must be defined

suffix should match property name[1]

```
partial class SpellCheck
{
   public static bool GetIsEnabled(UIElement ui)
   {
      return (bool) ui.GetValue(IsEnabledProperty);
   }
   public static void SetIsEnabled(UIElement ui, bool b)
   {
      ui.SetValue(IsEnabledProperty, b);
   }
}
```

Back to visuals – how do these change the UI?

- Property change likely has a desired behavior
  - does changing the property affect the rendering?
  - does changing the property affect the layout?
  - does the property have a default value?
  - can the property be inherited by children?
- All this is specified using metadata
  - additional "bits" of information tied to the property description



Defining property metadata

- Metadata is defined in the form of **`PropertyMetdata`**

```
public partial class FrameworkPropertyMetadata : UIPropertyMetadata
{
   public object DefaultValue { get; set; }
   public bool AffectsArrange { get; set; }
   public bool AffectsMeasure { get; set; }
   public bool AffectsParentArrange { get; set; }
   public bool AffectsParentMeasure { get; set; }
   public bool AffectsRender { get; set; }
   public bool BindsTwoWayByDefault { get; set; }
   public UpdateSourceTrigger DefaultUpdateSourceTrigger { get; set; }
   public bool Inherits { get; set; }
   public bool IsDataBindingAllowed { get; }
     ...
   public CoerceValueCallback CoerceValueCallback { get; set; }
   public PropertyChangedCallback PropertyChangedCallback { get; set; }
}
```

Example: setting the metadata properties

- DP.Register{Attached} takes a PropertyMetadata
  - associates metadata to property definition

```
partial class FrameworkElement {

public static readonly DependencyProperty LayoutTransformProperty =
       DependencyProperty.Register("LayoutTransform",
                                   typeof(Transform),
                                   typeof(FrameworkElement),
    new FrameworkPropertyMetadata(
           Transform.Identity,   // Default value
           FrameworkPropertyMetadataOptions.AffectsMeasure,
           new PropertyChangedCallback(OnLayoutTransformChanged))
    );
}
```

Visual classes

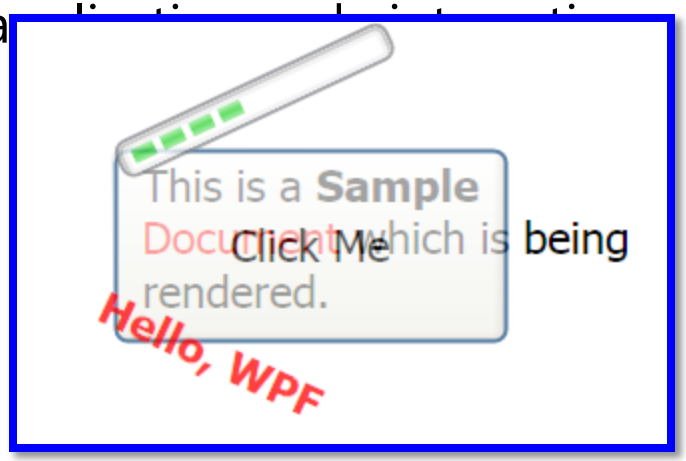
- Rendering starts with `Visual`
  - abstract class which provides support for hit-testing and coordinate transforms
- `UIElement` adds "IFE", input, focus, event support
  - provides concrete rendering support through **`OnRender`**
- `FrameworkElement` adds layout and data binding support
  - this is what everything generally extends

```
public class Cross : FrameworkElement
{
   protected override void OnRender(DrawingContext dc)
   {
      double w = ActualWidth, h = ActualHeight;
      dc.DrawRectangle(Brushes.Black, null, new Rect(w/2-w/8,2,w/4,h-2));
      dc.DrawRectangle(Brushes.Black, null, new Rect(2, h/2-h/4, w-2, h/4));
   }
}
```

Rendering details

- Entire scene composed in memory and painted in one operation
  - creates layers (tree) of objects and draws bottom to top
  - overlapping objects can affect output on a pixel-by-pixel basis
- Drawing commands are buffered and repainted automatically
  - screen can be repainted without a

VS.



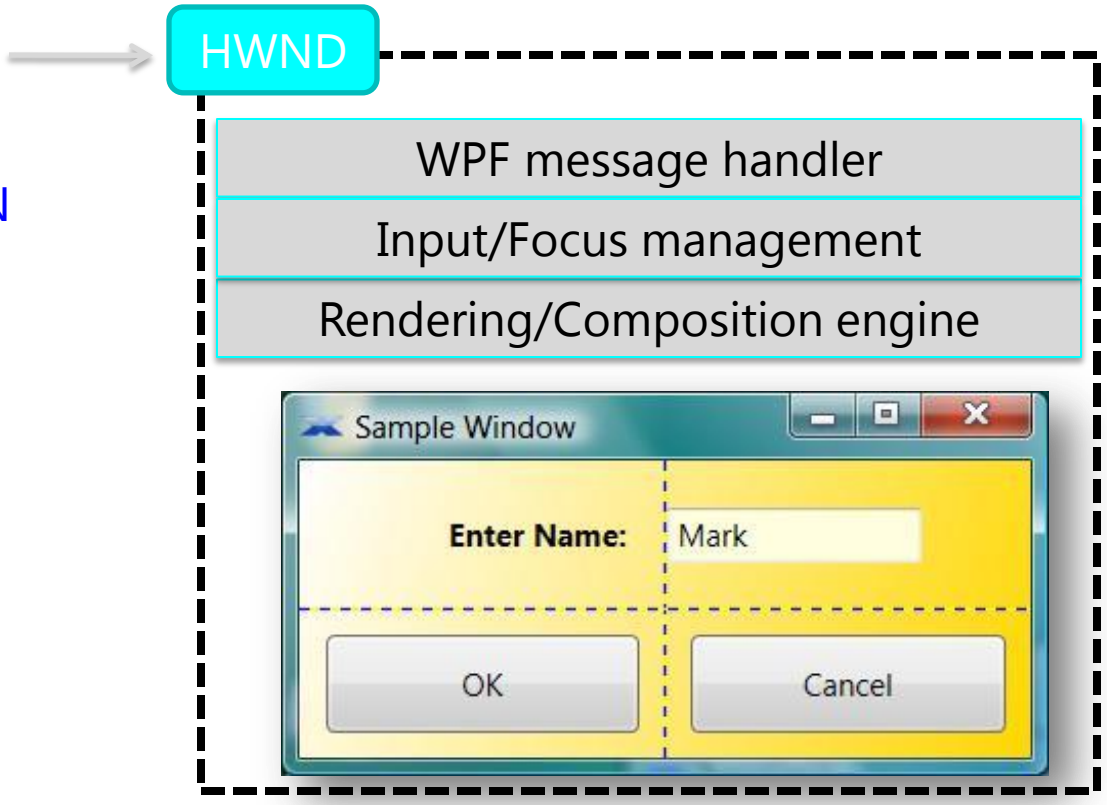
Overlapping controls with Windows Forms

Overlapping controls with WPF

How can it do that on Windows?

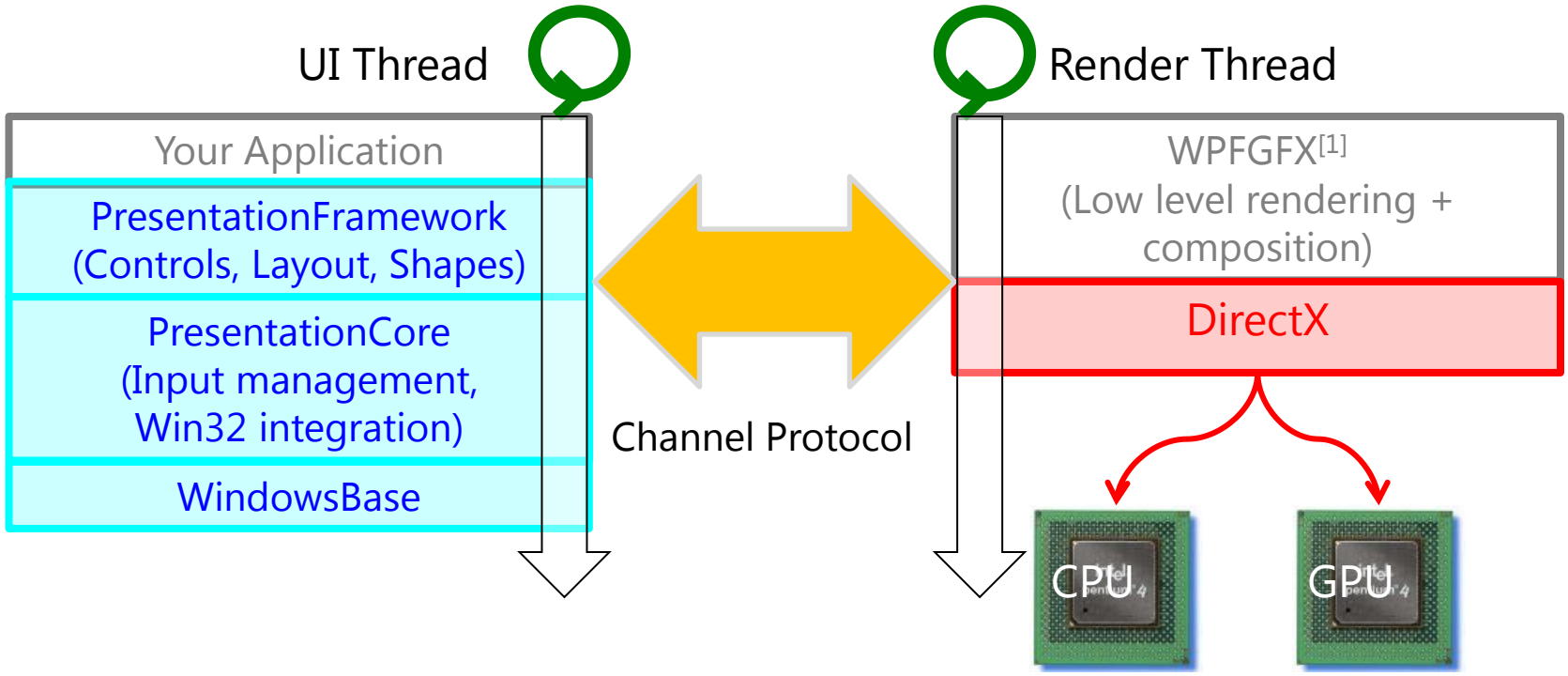
- WPF creates a single HWND and uses it to interact with Win32
  - input, focus, controls, rendering are all managed by WPF
  - window adornments still drawn and managed by Win32

WM_CREATE
WM_PAINT
WM_SETFOCUS
WM_LBUTTONDOWN
…



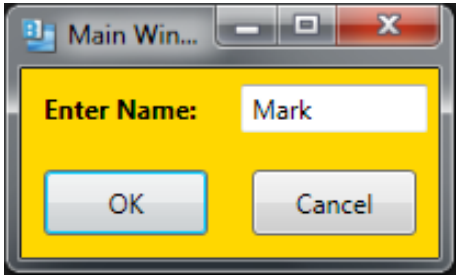
Rendering architecture

- Everything in WPF is rendered through DirectX
  - takes full advantage of GPU hardware and memory
  - dedicated thread does composition and DirectX rendering



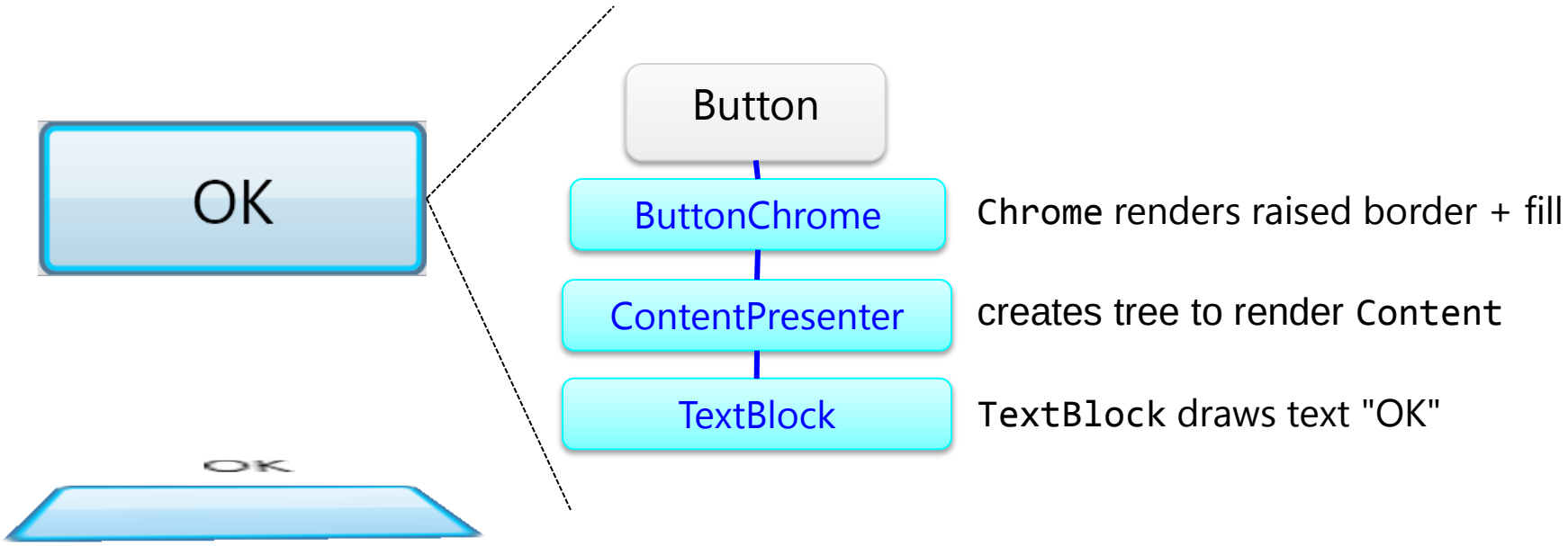
Logical Tree

- WPF builds Logical Tree to manage UI
  - identifies object relationships (parent to child)
  - follows the XAML structure



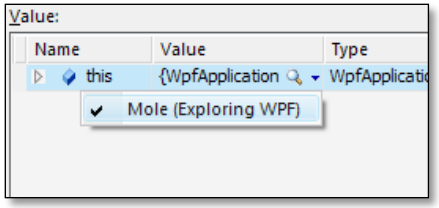
Visual Tree

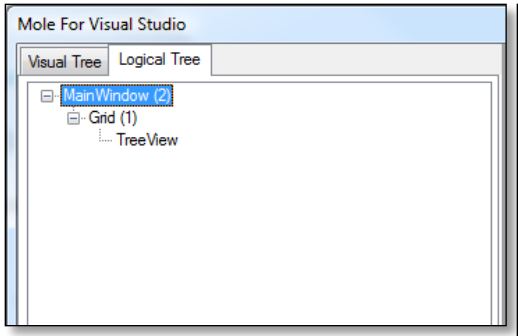
- WPF maintains Visual Tree to decide how to render items
  - builds on logical tree and adds visual information
  - enables complete replacement of visual representation ("lookless" controls)



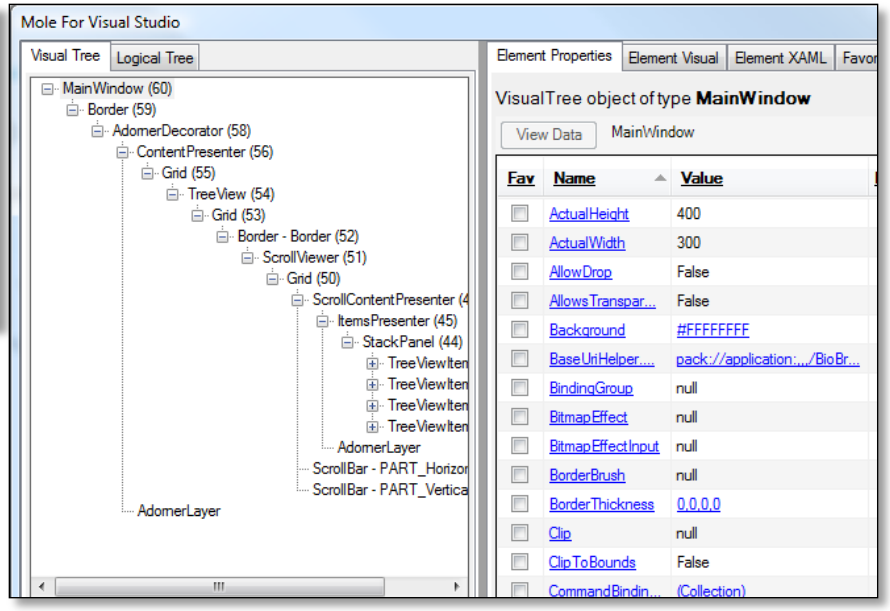
Exploring the logical and visual tree

- Mole is a VS.NET debugging visualizer for WPF
  - http://moleproject.com





logical tree matches the elements created in XAML



visual tree is what is actually rendered

Summary

- WPF provides a new architecture built on top of Win32
  - hides most of the abstraction away with a few exceptions
- Remember only one thread touches UI elements
  - use the Dispatcher to work with other threads
- Dependency Properties are core to the framework
  - you will see them again (and again)
- Rendering is performed through DirectX
  - allows full hardware acceleration support
  - uses retained model for ease of use
- WPF manages elements in logical and visual trees
  - will become more important when we talk about controls