Adam Nathan

WPF 4.5

UNLEASHED

SAMS

Some Praise for the First Edition of *Windows® Presentation Foundation Unleashed*

"The Nathan book is brilliant—you'll love it. Publishers, take note: I'd sure be buying a heck of a lot more technical books if more of them were like this one."
—Jeff Atwood, codinghorror.com, cofounder of Stack Overflow

"*Windows Presentation Foundation Unleashed* is a must-have book for anyone interested in learning and using WPF. Buy it, read it, and keep it close to your computer."
—Josh Smith, Microsoft MVP

"As we built the feature team that delivered the new WPF presentation layer for Visual Studio 2010, *Windows Presentation Foundation Unleashed* quickly became our must-read WPF reference book of choice, over and above other books on WPF and indeed internal documentation. Highly recommended for any developer wanting to learn how to make the most of WPF."
—James Bartlett, senior lead program manager, Microsoft Visual Studio

"I've bought nearly all available WPF books, but the only one that's still on my desk is *Windows Presentation Foundation Unleashed*. It not only covers all WPF aspects, but it does it in the right, concise way so that reading it was a real pleasure."
—Corrado Cavalli, Codeworks

"*Windows Presentation Foundation Unleashed* is the most insightful WPF book there is. Don't be misled by its size; this book has the best introduction and deepest insights. This is the must-read for anyone getting started or wanting to get the most out of WPF."
—Jaime Rodriguez, Microsoft client evangelist for Windows, WPF, Silverlight, and Windows Phone

"I found *Windows Presentation Foundation Unleashed* to be an excellent and thorough introduction and guide to programming WPF. It is clearly written, easily understood, and yet still deep enough to get a good understanding of how WPF works and how to use it. Not a simple feat to accomplish! I heartily recommend it to all the students who take DevelopMentor's WPF course! Anyone serious about doing WPF work should have a copy in their library."
—Mark Smith, DevelopMentor instructor, author of DevelopMentor's Essential WPF course

"I have read *Windows Presentation Foundation Unleashed* from cover to cover and have found it to be really the most comprehensive material on WPF. I can't think of even a single instance when I have not been able to find the solution (or a pointer to one) every time that I have picked up the book to figure out the intricacies of WPF."
—Durgesh Nayak, team leader, Axis Technical Group

"*Windows Presentation Foundation Unleashed* is the book that made WPF make so much sense for me. Without Adam's work, WPF would still be a mystery to me and my team. The enthusiasm for WPF is evident from the offset and it really rubs off on the reader."
—Peter O'Hanlon, managing director, Lifestyle Computing Ltd

"Adam Nathan's *Windows Presentation Foundation Unleashed* must surely be considered one of the seminal books on WPF. It has everything you need to help you get to grips with the learning cliff that is WPF. It certainly taught me loads, and even now, after several years of full-time WPF development, *Windows Presentation Foundation Unleashed* is never far from my hand."
—Sacha Barber, Microsoft MVP, CodeProject MVP, author of many WPF articles

"Of all the books published about WPF, there are only three that I recommend. *Windows Presentation Foundation Unleashed* is my primary recommendation to developers looking to get up to speed quickly with WPF."
—Mike Brown, Microsoft MVP, Client App Development, and president of KharaSoft, Inc.

In addition to the `IsChecked` property, `ToggleButton` defines a separate event for each value of `IsChecked`: `Checked` for `true`, `Unchecked` for `false`, and `Indeterminate` for `null`. It might seem odd that `ToggleButton` doesn't have a single `IsCheckedChanged` event, but the three separate events are handy for declarative scenarios.

As with `RepeatButton`, `ToggleButton` is in the `System.Windows.Controls.Primitives` namespace, which essentially means that the WPF designers don't expect people to use `ToggleButtons` directly or without additional customizations. It is quite natural, however, to use `ToggleButtons` directly inside a `ToolBar` control, as described in Chapter 10.

CheckBox

`CheckBox`, shown in Figure 9.2, is a familiar control. But wait a minute…isn't this section supposed to be about buttons? Yes, but consider the characteristics of a WPF `CheckBox`:

- It has a single piece of *externally supplied* content (so the standard check box doesn't count).
- It has a notion of being clicked by mouse or keyboard.
- It retains a state of being checked or unchecked when clicked.
- It supports a three-state mode, where the state toggles from checked to indeterminate to unchecked.

Does this sound familiar? It should, because a `CheckBox` is nothing more than a `ToggleButton` with a different appearance! `CheckBox` is a simple class deriving from `ToggleButton` that does little more than override its default style to the visuals shown in Figure 9.2.

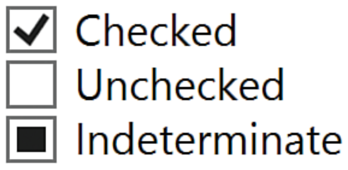


FIGURE 9.2 The WPF `CheckBox` control, with all three `IsChecked` states shown.

DIGGING DEEPER

`CheckBox` Keyboard Support

`CheckBox` supports one additional behavior that `ToggleButton` does not, for parity with a little-known feature of Win32 check boxes. When a `CheckBox` has focus, pressing the plus (+) key checks the control and pressing the minus (–) key unchecks the control! Note that this works only if `IsThreeState` hasn't been set to `true`.

RadioButton

`RadioButton` is another control that derives from `ToggleButton`, but it is unique because it has built-in support for mutual exclusion. When multiple `RadioButton` controls are grouped together, only one can be checked at a time. Checking one `RadioButton`—even programmatically—automatically unchecks all others in the same group. In fact, users can't even directly uncheck a `RadioButton` by clicking it; unchecking can only be done programmatically. Therefore, `RadioButton` is designed for multiple-choice questions. Figure 9.3 shows the default appearance of a `RadioButton`.

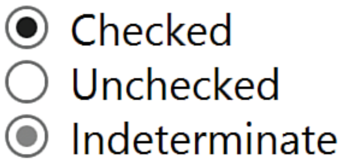


FIGURE 9.3 The WPF `RadioButton`, with all three `IsChecked` states shown.

The rarely used indeterminate state of a `RadioButton` control (`IsThreeState=true` and `IsChecked=null`) is similar to the unchecked state in that a user cannot enable this state by clicking on it; it must be set programmatically. If the `RadioButton` is clicked, it changes to the checked state, but if another `RadioButton` in the same group becomes checked, any indeterminate `RadioButtons` remain in the indeterminate state.

Placing several WPF `RadioButtons` in the same group is very straightforward. By default, any `RadioButtons` that share the same direct logical parent are automatically grouped together. For example, only one of the following `RadioButtons` can be checked at any point in time:

```
<StackPanel>
  <RadioButton>Option 1</RadioButton>
  <RadioButton>Option 2</RadioButton>
  <RadioButton>Option 3</RadioButton>
</StackPanel>
```

If you need to group `RadioButtons` in a custom manner, however, you can use the `GroupName` property, which is a simple string. Any `RadioButtons` with the same `GroupName` value get grouped together (as long as they have the same logical root). Therefore, you can group them across different parents, as shown here:

```
<StackPanel>
  <StackPanel>                                        Different
    <RadioButton GroupName="A">Option 1</RadioButton>  parents
    <RadioButton GroupName="A">Option 2</RadioButton>
  </StackPanel>
  <StackPanel>
    <RadioButton GroupName="A">Option 3</RadioButton>
  </StackPanel>
</StackPanel>
```

Or you can even create subgroups inside the same parent:

```
<StackPanel>
  <RadioButton GroupName="A">Option 1</RadioButton>             Different
  <RadioButton GroupName="A">Option 2</RadioButton>               groups
  <RadioButton GroupName="B">A Different Option 1</RadioButton>
  <RadioButton GroupName="B">A Different Option 2</RadioButton>
</StackPanel>
```

Of course, the last example would be a confusing piece of user interface without an extra visual element separating the two subgroups!

Simple Containers

WPF includes several built-in content controls that *don't* have a notion of being clicked like a button. Each has unique features to justify its existence. These content controls are the following:

- `Label`
- `ToolTip`
- `Frame`

Label

`Label` is a classic control that, as in previous technologies, can be used to hold some text. Because it is a WPF content control, it can hold arbitrary content in its `Content` property—a `Button`, a `Menu`, and so on—but `Label` is really useful only for text.

You can place text on the screen with WPF in several different ways, such as using a `TextBlock` element. But what makes `Label` unique is its support for access keys. You can designate a letter in a `Label`'s text that gets special treatment when the user presses the access key—the Alt key and the designated letter. You can also specify an arbitrary element that should receive focus when the user presses this access key. To designate the letter (which can appear underlined, depending on the Windows settings), you simply precede it with an underscore. To designate the target element, you set `Label`'s `Target` property (of type `UIElement`).

The classic case of using a `Label`'s access key support with another control is pairing it with a `TextBox`. For example, the following XAML snippet gives focus to the `TextBox` when Alt+U is pressed:

```
<Label Target="userNameBox">_User Name:</Label>
<TextBox x:Name="userNameBox"/>
```

Setting the value of `Target` implicitly leverages the `NameReferenceConverter` type converter described in Chapter 2. In C#, you can simply set the property to the instance of the `TextBox` control as follows (assuming that the `Label` is named `userNameLabel`):

```
userNameLabel.Target = userNameBox;
```

TIP

Controls such as `Label` and `Button` support access keys by treating an underscore before the appropriate letter specially, as in `_Open` or `Save _As`. (Win32 and Windows Forms use an ampersand [&] instead; the underscore is much more XML friendly.) If you really want an underscore to appear in your text, you need to use two consecutive underscores, as in `__Open` or `Save __As`.

`ToolTip`

The `ToolTip` control holds its content in a floating box that appears when you hover over an associated control and disappears when you move the mouse away. Figure 9.4 shows a typical `ToolTip` in action, created from the following XAML:

```
<Button>
  OK
<Button.ToolTip>
  <ToolTip>
    Clicking this will submit your request.
  </ToolTip>
</Button.ToolTip>
</Button>
```

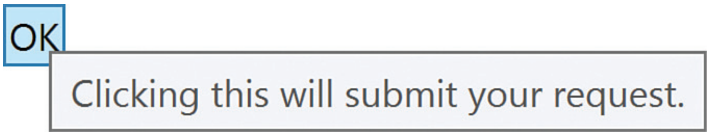


FIGURE 9.4 The WPF `ToolTip`.

The `ToolTip` class can never be placed directly in a tree of `UIElements`. Instead, it must be assigned as the value of a separate element's `ToolTip` property (defined on both `FrameworkElement` and `FrameworkContentElement`).

TIP

You don't even need to use the `ToolTip` class when setting an element's `ToolTip` property! The property is of type `Object`, and if you set it to any non-`ToolTip` object, the property's implementation automatically creates a `ToolTip` and uses the property value as the `ToolTip`'s content. Therefore, the XAML for Figure 9.4 could be simplified to the following and give the same result:

```
<Button>
  OK
<Button.ToolTip>
  Clicking this will submit your request.
</Button.ToolTip>
</Button>
```

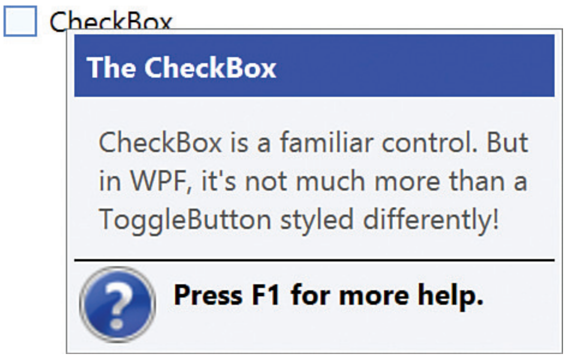
or it could be simplified further, as follows:

```
<Button Content="OK" ToolTip="Clicking this will submit your request."/>
```

Because of the flexibility of WPF's content controls, a WPF `ToolTip` can hold anything you want! Listing 9.1 shows how you might construct a Microsoft Office–style ScreenTip. The result is shown in Figure 9.5.

LISTING 9.1 A Complex `ToolTip`, Similar to a Microsoft Office ScreenTip

```
<CheckBox>
  CheckBox
<CheckBox.ToolTip>
  <StackPanel>
    <Label FontWeight="Bold" Background="Blue" Foreground="White">
      The CheckBox
    </Label>
    <TextBlock Padding="10" TextWrapping="WrapWithOverflow" Width="200">
      CheckBox is a familiar control. But in WPF, it's not much
      more than a ToggleButton styled differently!
    </TextBlock>
    <Line Stroke="Black" StrokeThickness="1" X2="200"/>
    <StackPanel Orientation="Horizontal">
      <Image Margin="2" Source="help.gif"/>
      <Label FontWeight="Bold">Press F1 for more help.</Label>
    </StackPanel>
  </StackPanel>
</CheckBox.ToolTip>
</CheckBox>
```



CheckBox is a familiar control. But in WPF, it's not much more than a ToggleButton styled differently!

FIGURE 9.5 A tooltip like the ScreenTips in Microsoft Office is easy to create in WPF.

Although a `ToolTip` can contain interactive controls such as `Button`s, those controls never get focus, and you can't click or otherwise interact with them.

`ToolTip` defines `Open` and `Closed` events in case you want to act on its appearance and disappearance. It also defines several properties for tweaking its behavior, such as its placement, whether it should stay open until explicitly closed, or even whether a drop shadow should be rendered. Sometimes you might want to apply the same `ToolTip` on multiple controls, yet you might want the `ToolTip` to behave differently depending on the control to which it is attached. For such cases, a separate `ToolTipService` static class can meet your needs.

`ToolTipService` defines a handful of attached properties that can be set on any element using the `ToolTip` (rather than on the `ToolTip` itself). It has several of the same properties as `ToolTip` (which have a higher precedence in case the `ToolTip` in question has conflicting values), but it also adds several more. For example, `ShowDuration` controls how long the `ToolTip` should be displayed while the mouse pointer is paused over an element, and `InitialShowDelay` controls the length of time between the pause occurring and the

ToolTip first being shown. You can add ShowDuration to the first ToolTip example as follows:

```
<Button ToolTipService.ShowDuration="3000">
  …
</Button>
```

FAQ

How do I get a ToolTip to appear when hovering over a disabled element?

Simply use the ShowOnDisabled attached property of the ToolTipService class. From XAML, this would look as follows on a Button:

```
<Button ToolTipService.ShowOnDisabled="True">
  …
</Button>
```

Or from C# code, you can call the static method corresponding to the attached property:

```
ToolTipService.SetShowOnDisabled(myButton, true);
```

Frame

The Frame control holds arbitrary content, just like all other content controls, but it isolates the content from the rest of the user interface. For example, properties that would normally be inherited down the element tree stop when they reach the Frame. In many respects, WPF Frames act like frames in HTML.

FAQ

How can I forcibly close a ToolTip that is currently showing?

Set its IsOpen property to false.

Speaking of HTML, Frame's claim to fame is that it can render HTML content in addition to WPF content. Frame has a Source property of type System.Uri that can be set to any HTML (or XAML) page. Here's an example:

```
<Frame Source="http://www.adamnathan.net"/>
```

TIP

When using Frame to navigate between web pages, be sure to handle its NavigationFailed event to perform any error logic and set NavigationFailedEventArgs.Handled to true. Otherwise, an unhandled exception (such as a WebException) gets raised on a different thread. The NavigationFailedEventArgs object passed to the handler provides access to the exception among other details.

As explained in Chapter 7, Frame is a navigation container with built-in tracking that applies to both HTML and XAML content. So, you can think of the Frame control as a more flexible version of the Microsoft Web Browser ActiveX control or the WPF WebBrowser control that wraps this ActiveX control.