

Covers Visual Studio 2008  
SPI and .NET 3.5 SPI

# WPF IN ACTION

with Visual Studio 2008

Arlen Feldman  
Maxx Daymon

 MANNING





***WPF in Action with Visual Studio 2008***

by Arlen Feldman and Maxx Daymon

**Sample Chapter 7**

Copyright 2008 Manning Publications

# *brief contents*

---

<b>PART 1</b>	<b>PAST, PRESENT, AND FUTURE .....</b>	<b>1</b>
	1 ■ The road to Avalon (WPF)	3
	2 ■ Getting started with WPF and Visual Studio 2008	22
	3 ■ WPF from 723 feet	41
<b>PART 2</b>	<b>THE BASICS .....</b>	<b>63</b>
	4 ■ Working with layouts	65
	5 ■ The Grid panel	94
	6 ■ Resources, styles, control templates, and themes	119
	7 ■ Events	147
	8 ■ Oooh, shiny!	157
<b>PART 3</b>	<b>APPLICATION DEVELOPMENT .....</b>	<b>177</b>
	9 ■ Laying out a more complex application	179
	10 ■ Commands	191
	11 ■ Data binding with WPF	209
	12 ■ Advanced data templates and binding	253

- 13 ■ Custom controls 299
- 14 ■ Drawing 315
- 15 ■ Drawing in 3D 352

**PART 4 THE LAST MILE..... 371**

- 16 ■ Building a navigation application 373
- 17 ■ WPF and browsers: XBAP, ClickOnce, and Silverlight 390
- 18 ■ Printing, documents, and XPS 406
- 19 ■ Transition effects 427
- 20 ■ Interoperability 457
- 21 ■ Threading 474

# 7 Events

---

## ***This chapter covers:***

- Bubble-up events
- Tunnel-down events
- Handling events even when they've already been handled
- Class-level events
- Clever ways to annoy your users

If you've used both MFC and Windows Forms, you'll know that the event model in Windows Forms was a major improvement over the message-map model used by MFC. Controls in Windows Forms exposed events that could be subscribed to by code that cares, and that code was called when appropriate. The classic example is a user clicking a button, resulting in the appropriate handler being called. Many other events work in the same way.

Classic Windows Forms events did have some issues. The most problematic was that the code that cared about the event either needed to have direct access to the event generator, or the event had to be manually passed up the chain. For example, picture a button on a user control on a form in an application. If the application needs to know about the event, the application either needs to know about the

button (breaking encapsulation), or the button needs to tell the user control, which needs to tell the form, which needs to tell the application—which is a pain in the neck.

WPF adds an additional complication—composition. Before WPF, a radio button was just a radio button—a control that had behavior. With WPF, you can think of a radio button as a series of shapes joined together cooperatively (a circle, a dot to indicate checked, the text, the focus rectangle, and so on). Each of these needs to know what’s going on in some manner; which “bit” of the control gets the click that eventually generates an event can vary. If you had to subscribe to an event on the circle, the dot, *and* the text in order to determine if someone had clicked the radio button, it would get seriously tedious. This nesting can be taken arbitrarily further—the circle for the radio button could be replaced with a 3D animation, itself made up of different elements.

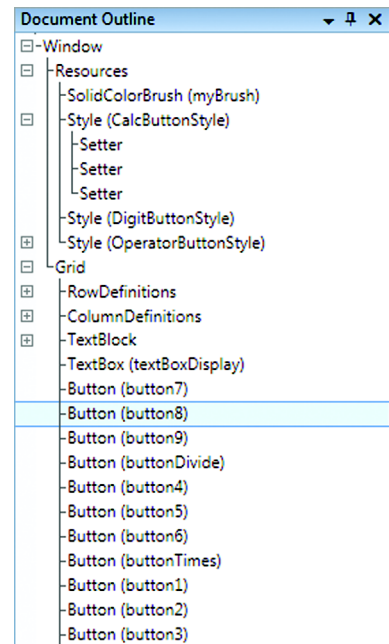
To address these and other issues, WPF has a number of new (and very cool) event-based capabilities. The most powerful and useful of these are *routed events*, the primary topic of this chapter.

## 7.1 Routed events

In a classic .NET event, an interested party has to directly subscribe to an event in order to be notified. For non-UI code, this makes a lot of sense. After all, there’s no particular way for regular code to know who else might care and what legitimate rules may exist for passing events to other objects. With UI, events have a pretty clear path—below the top level, each control is owned by another control. When you look at the XAML for a Window, the natural nesting of items defines that path. In Visual Studio 2008, you can look at the document outline for our calculator (figure 7.1), for example, by selecting View > Other Windows > Document Outline from the menu.

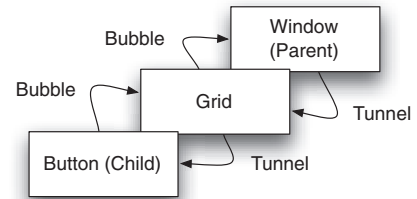
Notice how the Window holds the Grid which, in turn, holds the various text controls and all the buttons. It would be fairly natural to expect that, if a button click wasn’t handled by the button itself, perhaps the Grid or the Window might handle it.

Routed events give us this capability. An event can be defined to bubble up to its parents in the element tree. Events can also be defined to tunnel down, which is the exact opposite. If the Window doesn’t handle the event, then the Grid is given the chance, and then, finally the children are given a shot (figure 7.2).



**Figure 7.1** The Document Outline for the calculator shows the natural tree of controls.

The decision of an event's *routing strategy*—whether an event should bubble up or tunnel down—is made when the event is defined. Click, for example, is a bubble-up event, so it can be caught by elements higher up in the element tree. PreviewDragEnter is a tunnel-down event sent when a user drags something over a Window. If the highest level doesn't want to handle it, a lower level can be asked, and so on. There's also one additional routing strategy: direct. Direct events work pretty much like standard .NET events. A direct event can only be handled by subscribing to the specific element that raises it.



**Figure 7.2** Events can be set up to bubble up to their parents in the element tree or tunnel down to their children.

Routing events follow a similar pattern, and are implemented in a similar way, to properties in the WPF Property System. We associate events with objects that don't know what they're for. We'll demonstrate this in the next section.

### 7.1.1 Bubbling events

Wouldn't it be nice if, in our calculator, we didn't have to specify a handler for every single button? Well, because the Click event on a button is a bubble-up event, we can remove all the individual Click="OnClick" handlers from the buttons and, instead, put a single handler on one of the higher-level containers such as the Window or the Grid:

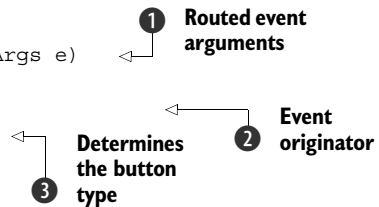
```
<Grid Button.Click="OnAnyButtonClick">
```

We have to manually add the event handler—the Properties grid list of events will only show us those events that are directly exposed by the Grid. If the Properties grid had to show all the events of all the children, it would get quite messy (although a tree that showed children and their events might not be a bad UI).

Anyway, the Button.Click handler does exactly what you'd expect—it waits to be told that a button has been clicked and then calls the OnAnyButtonClick method. It gets called if any button contained within the Grid is clicked. Just as with attached properties, we have to be more explicit in our declaration because Grid doesn't expose a Click event—we have to say Button.Click instead of just Click. Let's look at the implementation of OnAnyButtonClick (listing 7.1).

#### Listing 7.1 OnAnyButtonClick implementation

```
private void OnAnyButtonClick
    (object sender, RoutedEventArgs e)
{
    Button btn = e.OriginalSource as Button;
    if (btn.Tag is Operator)
        OnClickOperator(e.OriginalSource, e);
    else
        OnClickDigit(e.OriginalSource, e);
}
```



As we've mentioned before, instead of the old, dowdy `EventArgs` that used to be passed, routed events pass a `RoutedEventArgs` object ❶ instead. `RoutedEventArgs` have several useful properties, including the `OriginalSource` property ❷ that contains the object that originally generated the event. The original source has to be passed because the object handling the event is no longer required to be the object generating the event. If we were to look at the sender, we'd see that it's the `Grid` rather than a button.

Now we have a problem, though. Before, we simply hooked up a different handler for operators than for digits, so we knew that the proper event handler would end up being called. Now, the same handler is responsible for both types of buttons, so we need to determine if the button is an operator or a digit.

In this case, we're relying on the fact that we're storing an `Operator` in the `Tag` of operator buttons ❸. This approach isn't super elegant, but it works. For a more complex implementation, we might have created a custom object and associated an instance with each button that made it clear which was which. Our cheap implementation looks for an `Operator` in the `Tag`. If it's an `Operator`, we assume the button is an operator. Otherwise, we assume it's a digit. We then call the appropriate handler for each. Because we're being cheap, we didn't bother creating methods that just took the button, but we could have done that too.

Those of you who are paying close attention will have noticed that there's one button we aren't handling—the decimal point, which is neither a digit nor an operator. We could put another case in the `OnAnyButtonClick()` handler, something like:

```
if (btn == buttonDecimal)
    HandleDecimal();
else if (btn.Tag is String)
    ...
```

But it seems odd to add in a single case here, when we could simply leave the handler in place for the decimal button:

```
<Button Name="buttonDecimal" Click="OnClickDecimal">.</Button>
```

The `OnClickDecimal()` method handles the decimal click as before. The only problem is that we now have the `OnAnyButtonClick()` handler in place. Unlike properties, events aren't overridden; instead, all specified handlers are called. When the user hits the decimal point, the `OnClickDecimal()` method properly handles the decimal point, and then `OnAnyButtonClick()` assumes that the decimal is a digit and calls the `OnDigit()` method, which will snort milk out of its nose and crash.

Fortunately, WPF has a simple and elegant way of handling this situation. Once a handler has handled an event, it can say so, stopping it from doing any more bubbling. We can modify the `OnClickDecimal()` handler to indicate that it has handled the event by setting the appropriate property on the `RoutedEventArgs`:

```
private void OnClickDecimal(object sender, RoutedEventArgs e)
{
    HandleDecimal();
    e.Handled = true;
}
```

Setting the `Handled` property to `true` stops the event from bubbling any further. The same property prevents additional tunneling, as you'll see in the next section. We're now handling all our buttons in a much more elegant way even if, from a user's perspective, nothing has changed. In the next section, we'll do something that adds functionality to our long-suffering utility.

### 7.1.2 Tunneling events

When you think about it, making a calculator that looks like a real, physical calculator is a little bit silly. Although it has the advantage of instant recognizability, making the user use a mouse to click buttons in rough mimicry of what he could do far more quickly with his finger on a real calculator isn't the best UI strategy—particularly if the user has a perfectly serviceable keyboard with a numeric keypad and (presumably) a perfectly serviceable finger.

If we're going to make a UI that looks like a real-world object, the least we can do is make it possible for the user to *also* use his numeric keypad or other keys on his keyboard.

To do this properly, we'll make use of a different type of event—a tunneling (or tunnel-down) event. Now, it may seem greedy to want yet another type of event, considering that, before WPF, we didn't even have bubbling (or bubble-up) events; but, if you look at how implementing this functionality works using bubble-up events, you'll see where the need to tunneling comes in.

An event called `KeyDown` is triggered when a key on the keyboard is pressed, and a matching `KeyUp` when the key is released. Conceptually, these events are familiar to you if you've used Windows Forms, MFC, or the raw Windows SDK. If you're familiar with those older technologies, you'll also now probably experience a slight twinge of pain that goes by the name of *focus*.

In Windows, only one widget at a time has focus, and that widget is the one that Windows thinks you're most likely working with—the one that, for instance, sends all keyboard events. For example, if you click the `2` button on the calculator, you notice that it gets a little dotted square around the button to let you know that the `2` button has focus. When you press a keyboard button (say, `3`), the keystroke is sent to the `2` button, which happily ignores it.

This is where WPF events can shine. We don't care what has focus, so long as it's somewhere on the calculator—we want to catch keys and act appropriately when they're hit.

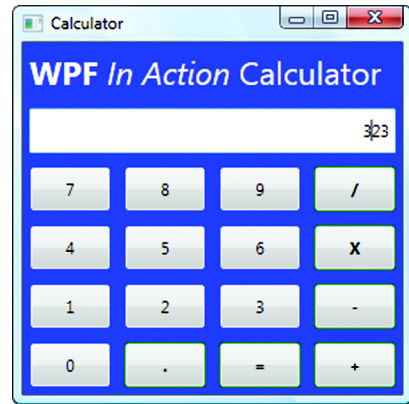
We could do exactly the same thing we did earlier with the `Click` event and catch it on the `Grid`. In that case, if the `2` button has focus and the user hits another number, the event first gives the `2` button a shot of handling it and then passes it up the chain. But what would happen if the text box where we're displaying our results has focus (figure 7.3)?

Using a bubble-up event, we hit the `2` button, set focus to the `TextBox`, and then hit the number `3` one time. The `TextBox` gets the first chance at the event and handles it normally. Then our handler catches it at the `Grid` and inserts it at the end. Obviously,

we could have made the `TextBox` read-only, but in some situations, we might want to allow direct editing (for example, to allow the user to clear the value).

What we need is a chance to handle the event before the `TextBox`. So, instead of waiting for an event to bubble up from where it originated in the hopes that we're the first to handle it, we want to be the first to catch it before it gets given to some nasty control that might do something unspeakable to it before we have our chance.

With many input events (keyboard, mouse, Ink, and so on), in addition to the bubble-up event, there is also a second event that, rather than bubbling up, tunnels down, starting with the top of our visual tree and working down to the control with focus. For the keyboard, these events are called `PreviewKeyDown` and `PreviewKeyUp`. It's a convention for tunnel-down events to be prefixed with the word *Preview* and to have a matching bubble-up event without the word *Preview* to allow for complete flexibility. All the higher-level controls have a chance to preview the event and handle it if they see fit. If it isn't handled, then the regular bubbling event is fired and bubbles up until it *is* handled. Table 7.1 shows how a keystroke is routed if not handled.



**Figure 7.3** Using a bubble-up event when focus is on the text box causes a bad side effect—the text box gets the keystroke and enters the key, and then our handler gets it and handles the key properly.

**Table 7.1** Routing of a keyboard event

User hits the 3 key >	
	Window is sent the <code>PreviewKeyDown</code> event.
	Grid is sent the <code>PreviewKeyDown</code> event.
	Focused button is sent the <code>PreviewKeyDown</code> event.
	Focused button is sent the <code>KeyDown</code> event.
	Grid is sent the <code>KeyDown</code> event.
	Window is sent the <code>KeyDown</code> event.
	Window is sent the <code>PreviewKeyUp</code> event.
	Grid is sent the <code>PreviewKeyUp</code> event.
	Focused button is sent the <code>PreviewKeyUp</code> event.
	Focused button is sent the <code>KeyUp</code> event.
	Grid is sent the <code>KeyUp</code> event.
	Window is sent the <code>KeyUp</code> event.

Of course, if any of the handlers mark the event as handled, it isn't sent to any of the remaining handlers.

**NOTE** Even though the tunnel-down and bubble-up events are paired, they *are* two separate events. The `RoutedEventArgs` sent to the `PreviewKeyDown` and to `KeyDown` are two different objects. Marking `PreviewKeyDown` prevents `KeyDown` from being fired purely because of logic built into the event handler and not because of generic behavior related to paired events. Usually, this behavior is consistent, but it's possible for the behavior to be different for some events.

To handle the keyboard, we want to catch the `PreviewKeyDown` event at the Grid level:

```
<Grid Button.Click="OnAnyButtonClick" PreviewKeyDown="OnKeyDown">
```

Then we need to define the `OnKeyDown` handler (listing 7.2).

**Listing 7.2** `OnKeyDown` handler

```
private void OnKeyDown(object sender, KeyEventArgs e)
{
    if ((e.Key >= Key.D0) && (e.Key <= Key.D9))
    {
        int digit = (int)(e.Key - Key.D0);
        HandleDigit(digit);
    }
    else if ((e.Key >= Key.NumPad0) && (e.Key <= Key.NumPad9))
    {
        int digit = (int)(e.Key - Key.NumPad0);
        HandleDigit(digit);
    }
    else
    {
        switch (e.Key)
        {
            case Key.Add:
                ExecuteLastOperator(Operator.Plus);
                break;
            case Key.Subtract:
                ExecuteLastOperator(Operator.Minus);
                break;
            case Key.Divide:
                ExecuteLastOperator(Operator.Divide);
                break;
            case Key.Multiply:
                ExecuteLastOperator(Operator.Times);
                break;
            case Key.OemPlus:
            case Key.Enter:
                ExecuteLastOperator(Operator.Equals);
                break;
            case Key.Decimal:
                HandleDecimal();
                break;
        }
    }
}
```

1 Key event arguments

2 Regular keyboard digit

3 Numeric keypad digit

4 Handles other keys

```

    }
  }
  e.Handled = true;
}

```

5 Marks event as handled

We aren't going to go into a huge amount of detail about the method itself, but there are a few things worth noting. First, we're getting a `KeyEventArgs` ❶ instead of a `RoutedEventArgs`. `KeyEventArgs` is derived from `RoutedEventArgs` but adds a few additional details (like the key that's hit). We check to see if the key hit is a digit ❷ or a numeric keypad digit ❸ based on the enum value, and then convert to a digit and call the digit handler. It's kind of cool that we can easily tell digits and operators apart without worrying about scan codes, but it's also a pain that there's no easy way to ask: "Is this a digit?" Then we look for other keys ❹—operator, decimal, and so on—and handle them appropriately.

Finally, we mark the event as handled ❺—no matter what. For our calculator, we're saying that we want the final word on all keystrokes and don't want anything else to handle keys. We could be a bit more flexible and only mark the event as handled if we, you know, handle it, but this way we don't allow any extraneous, unplanned keyboard behavior.

This is a pretty low-level way of handling keystrokes. There is another mechanism in WPF that we could use—we could associate keystrokes with `Commands`. Whereas events tend to be more low level (mouse moved, key was hit), `Commands` are more like the options you see on a menu or toolbar, such as `Save` or `Print`. Often with `Commands`, you don't care whether a command came from a menu, toolbar, hot-key, or somewhere else, and the command mechanism in WPF is built to handle these scenarios. We'll demonstrate that mechanism in chapter 10. But there are still many situations where you'll want to do things at the lowly event level.

One thing that we've left out is making the buttons provide feedback when the associated key is detected. This would be a nice affordance, but because the appearance of the digit in the output provides some feedback, we laxed out on that.

So far we've defined all our events via XAML; but, in the real world, there are often situations where you want or need to subscribe to events via code, such as when you're dynamically creating controls.

## 7.2 *Events from code*

As with properties, and WPF in general, anything you can do in XAML, you can also do in code, although the reverse isn't always true. If you want to subscribe to a routed event on the object that exposes it, you can use the traditional event subscription mechanism:

```
button1.Click += new RoutedEventHandler(OnButton1Clicked);
```

This is fine if you want to directly subscribe to the object that contains the event, but it won't work if you want to catch the event at a higher level. `Grid`, for example, doesn't have a `Click` event exposed, and it wouldn't make much sense for it to do so because

it isn't a button. Nor could Grid, Window, or any of the containing classes practically expose all the possible events of all possible children.

Instead, you can call a method called `AddHandler` to indicate your interest in an event. This method takes a `RoutedEventArgs` which is generally available as a static member on the class that exposes the event. This parallels the Property System mechanism.

For example, let's add a handler to our top-level Window to catch the `Click` event, as well as the beep every time a user clicks a button. You might want to do this, say, if you really hate your users. A good place to do this would be in the `Window_Loaded()` handler. (In your code, make sure you've subscribed to the `Loaded` event on the Window.)

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    AddHandler(Button.ClickEvent, new RoutedEventArgsHandler(OnAnyClickOnForm));
}
```

`Button.ClickEvent` is the static `RoutedEventArgs` for the `Click` event that lets the system hook into the appropriate event. The second argument is the handler for the method we want to call, which plays a beep:

```
private void OnAnyClickOnForm(object sender, RoutedEventArgs e)
{
    System.Media.SystemSounds.Beep.Play();
}
```

Go ahead and run the calculator. As you click the buttons, if everything is working, you should get really annoyed. Note that using the keyboard doesn't cause the beep. We're explicitly dealing with the button clicks. Something odd that you may notice: You get a beep for *almost* all the buttons, but not the decimal point. Strange, no?

Actually, it isn't that strange. We already have a handler for the decimal point, and that handler marks the event as `Handled`. It needs to do this to stop the generic grid-button handler from getting confused. Fortunately, there's a nice, simple fix.

### 7.2.1 *handledEventsToo*

Sometimes a property or parameter has a name that pretty much tells you everything you need to know. `handledEventsToo` is a flag you can specify if you want to handle the event too, even if it has been marked as handled. The flag is a parameter on the `AddHandler` method—we pass `true` as a third argument to have the handler be called even if another handler has marked the event as handled:

```
AddHandler(Button.ClickEvent,
            new RoutedEventArgsHandler(OnAnyClickOnForm), true);
```

Now, when we run the code, we get a nice, irritating beep even when the decimal point is clicked. By the way, we have to set this flag via the `AddHandler` call; there's no way to set it via XAML.

This code catches all buttons clicks that belong to the object and objects below on the visual tree. Sometimes, you want to catch an event for all instances of that object. There's a way of doing that too.

### 7.2.2 Class events

WPF allows you to register for an event for all instances of a particular class. For example, you could catch the `Click` event for all buttons, no matter where they are. There are several advantages to this approach versus putting a handler at the top-level control. One is that this handler is called *first* before all the specific handlers, so you have the first crack at dealing with the event. Another is that it avoids cluttering up the top-level object and lets you encapsulate the handlers more appropriately—particularly useful with your own custom controls. Also, it's a little bit faster because it doesn't have to navigate the tree.

You have to register for class events in a static constructor. The following code registers for the `ClickEvent` on all buttons:

```
static Window1()
{
    EventManager.RegisterClassHandler(typeof(Button), Button.ClickEvent,
        new RoutedEventHandler(ClassButtonHandler));
}
```

Here, we specify the type of the class for which we're registering, the specific event and the method to call. The method (`ClassButtonHandler`) looks much like any other routed event handler, except that it has to be static:

```
private static void ClassButtonHandler(object sender, RoutedEventArgs e)
{
    System.Media.SystemSounds.Beep.Play();
}
```

If you're following along, make sure that you remove the `OnAnyClickOnForm` registration and handler before you run this code or clicking buttons might lead to temporary two-beep insanity.

By the way, you could also mark the event as handled here, in which case none of the other handlers will be called, unless they've set `handledEventsToo` to `true`.

## 7.3 Summary

When we start talking about custom controls, we'll need to look into how events are implemented and new events are defined. Overall, the event system is fairly nice, and the ability to bubble-up and tunnel-down is extremely handy.

We've improved the calculator by adding keyboard support and made it more annoying by adding beeps when you click keys. But the calculator is still pretty plain vanilla. Given all the hype about WPF, we should be able to make the calculator a lot cooler—and that's the subject of the next chapter.

# WPF IN ACTION with Visual Studio 2008

Arlen Feldman and Maxx Daymon

Free ebook  
SEE INSERT

**W**indows Presentation Foundation is the .NET subsystem for building graphical interfaces. It's a replacement for WinForms that provides a wide range of features for 2D and 3D drawing, typography, animation, and data binding. Now that it's integrated into Visual Studio 2008, you can easily build WPF applications with the tools you already use for your .NET development.

Requiring no prior exposure to the subject, **WPF in Action** takes you smoothly from zero WPF knowledge to cruising speed. Following the crystal-clear writing and practical C# examples, you'll master XAML, WPF's declarative language, and core components like layouts, styles, and the new event model. As the book progresses, you'll tackle more sophisticated examples involving data access, custom controls, 3D, and animation. This book is written for developers with a background in .NET and C#.

## What's Inside

- Learn WPF and XAML using VS 2008
- Styles, layout, and command-handling
- Data binding, printing, and threading
- 2D and 3D drawing, animation
- Interoperate with WinForms

## About the Authors

**Arlen Feldman** specializes in meta-data driven applications and usability. He is Chief Architect for Cherwell Software. **Maxx Daymon** is an architect at Configuresoft. His focus is metaprogramming and agile development.

For online access to the authors, code samples, and a free ebook for owners of this book, go to [www.manning.com/WPFinAction](http://www.manning.com/WPFinAction)

"... essential for anyone learning WPF."

—Curt Christianson, DF-Software

"Excellent real-world code examples."

—Nishant Sivakumar  
Microsoft MVP

"A thorough step-by-step guide for Visual Studio users."

—Jeff Maurone, MSNBC.com

"Required reading for all WPF designers and developers."

—Donald Burnett, Microsoft  
Phizzpop.com Editor/Moderator

"The definitive source on WPF."

—Aleksey Nudelman  
C# Computing, LLC.

"A great, detailed jumpstart into practical WPF development."

—Joe Stagner, Microsoft

ISBN-10: 1933988223  
ISBN-13: 978-1933988221



9 781933 988221



MANNING

\$44.99 / Can \$44.99 [INCLUDING eBook]