Programming Windows Presentation Foundation
By Ian Griffiths, Chris Sells
.....
Publisher: **O'Reilly**
Pub Date: **September 2005**
ISBN: **0-596-10113-9**
Pages: **447**
Slots: **1.0**

Table of Contents | Index

Overview

Windows Presentation Foundation (WPF) (formerly known by its code name "Avalon") is a brand-new presentation framework for Windows XP and Windows Vista, the next version of the Windows client operating system. For developers, WPF is a cornucopia of new technologies, including a new graphics engine that supports 3-D graphics, animation, and more; an XML-based markup language (XAML) for declaring the structure of your Windows UI; and a radical new model for controls.

Programming Windows Presentation Foundation is the book you need to get up to speed on WPF. By page two, you'll have written your first WPF application, and by the end of Chapter 1, "Hello WPF," you'll have completed a rapid tour of the framework and its major elements. These include the XAML markup language and the mapping of XAML markup to WinFX code; the WPF content model; layout; controls, styles, and templates; graphics and animation; and, finally, deployment.

Programming Windows Presentation Foundation features:

- Scores of C# and XAML examples that show you what it takes to get a WPF application up and running, from a simple "Hello, Avalon" program to a tic-tac-toe game
- Insightful discussions of the powerful new programming styles that WPF brings to Windows development, especially its new model for controls
- A color insert to better illustrate WPF support for 3-D, color, and other graphics effects
- A tutorial on XAML, the new HTML-like markup language for declaring Windows UI
- An explanation and comparison of the features that support interoperability with Windows Forms and other Windows legacy applications

The next generation of Windows applications is going to blaze a trail into the unknown. WPF represents the

best of the control-based Windows world and the content-based web world; it's an engine just itching to be taken for a spin. Inside, you'll find the keys to the ignition.

CLOSE ↑

Editorial Reviews

Book Description
Windows Presentation Foundation (WPF) (formerly known by its code name "Avalon") is a brand-new presentation framework for Windows XP and Windows Vista, the next version of the Windows client operating system. For developers, WPF is a cornucopia of new technologies, including a new graphics engine that supports 3-D graphics, animation, and more; an XML-based markup language (XAML) for declaring the structure of your Windows UI; and a radical new model for controls.

Programming Windows Presentation Foundation is the book you need to get up to speed on WPF. By page two, you'll have written your first WPF application, and by the end of Chapter 1, "Hello WPF," you'll have completed a rapid tour of the framework and its major elements. These include the XAML markup language and the mapping of XAML markup to WinFX code; the WPF content model; layout; controls, styles, and templates; graphics and animation; and, finally, deployment.

Programming Windows Presentation Foundation features:

- Scores of C# and XAML examples that show you what it takes to get a WPF application up and running, from a simple "Hello, Avalon" program to a tic-tac-toe game
- Insightful discussions of the powerful new programming styles that WPF brings to Windows development, especially its new model for controls
- A color insert to better illustrate WPF support for 3-D, color, and other graphics effects
- A tutorial on XAML, the new HTML-like markup language for declaring Windows UI
- An explanation and comparison of the features that support interoperability with Windows Forms and other Windows legacy applications

The next generation of Windows applications is going to blaze a trail into the unknown. WPF represents the best of the control-based Windows world and the content-based web world; it's an engine just itching to be taken for a spin. Inside, you'll find the keys to the ignition.

Programming Windows Presentation Foundation

by Chris Sells and Ian Griffiths

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor:	John Osborn
Development Editor:	Michael Weinhardt
Production Editor:	Sanders Kleinfeld
Cover Designer:	Ellie Volckhausen
Interior Designer:	David Futato
Printing History:	
September 2005:	First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. Programming Windows Presentation Foundation, the image of a kudu, and related trade dress are trademarks of O'Reilly Media, Inc.

Microsoft, the .NET logo, Visual Basic .NET, Visual Studio .NET, ADO.NET, Windows, and Windows 2000 are registered trademarks of Microsoft Corporation.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 0-596-10113-9

[C]

Preface

It's been a long road to Windows Presentation Foundation, better known to many as Avalon, the in-house Microsoft code name for the new Windows Vista presentation framework.

I learned to program Windows from Programming Windows 3.1 by Charles Petzold (Microsoft Press). In those days, programming for Windows was about windows, menus, dialogs and child controls. To make it all work, we had WndProcs (windows procedure functions) and messages. We dealt with the keyboard and the mouse. If we got fancy, we would do some nonclient work. Oh, and there was the stuff in the big blank space in the middle that I could fill however I wanted with the graphics device interface (GDI), but my 2-D geometry had better be strong to get it to look right, let alone perform adequately.

Later, I moved to MFC (the Microsoft Foundation Classes), where we had this thing called a "document" which was separate from the "view." The document could be any old data I wanted it to be, and the view, well, the view was the big blank space in the middle that I could fill however I wanted with the MFC wrappers around GDI.

Later still, there was this thing called DirectX, which was finally about providing tools for filling in the space with hardware-accelerated 3-D polygons, but DirectX was built for writing full-screen games, so building content visualization and management applications just made my head hurt.

Windows Forms, on the other hand, was such a huge productivity boost and I loved it so much that I wrote a book about it (as did my co-author). Windows Forms was built on top of .NET, a managed environment that took a lot of programming minutiae off my hands so that I could concentrate on the content. Plus, Windows Forms itself gave me all kinds of great tools for laying out my windows, menus, dialogs and child controls. And the inside of the windows where I showed my content? Well, if the controls weren't already there to do what I wanted, I could draw the content however I wanted using the GDI+ wrappers in `System.Drawing`, which was essentially the same drawing model Windows programmers had been using for the last 20 years, before even hardware graphics acceleration in 2-D, let alone 3-D.

In the meantime, a whole other way of interacting with content came along: HTML was great at letting me arrange my content, both text and graphics, and it would flow it and reflow it according to the preferences of the user. Further, with the recent emergence of AJAX (Asynchronous Java And XML), this environment got even more capable. Still, HTML isn't great if you want to control more of the user experience than just the content or if you want to do anything Windows-specific—both things that even Windows 3.1 programmers took for granted.

More recently, Windows Presentation Foundation (WPF) happened. Initially it felt like another way to create my windows, menus, dialogs and child controls. However, WPF shares a much deeper love for content than has yet been provided by any other Windows programming

framework.

To support content at the lowest levels, WPF merges controls and graphics into one programming model; both are placed into the same element tree in the same way. And while these primitives are built on top of DirectX to leverage the 3-D hardware acceleration that's dormant when you're not running the latest twitch game, they're also built in .NET, providing the same productivity boost to WPF programmers that Windows Forms programmers enjoy.

One level up, WPF provides its "content model," which allows any control to host any group of other controls. You don't have to build special `BitmapButton` or `IconComboBox` classes; you put as many images, shapes, videos, 3-D models or whatever into a `Button` or a `ComboBox` as suit your fancy.

To arrange the content, whether in fixed or flow layout, WPF provides container elements that implement various layout algorithms in a way that is completely independent of the content they're holding.

To visualize the content, WPF provides data binding, styles and animation. Data binding produces and synchronizes visual elements on the fly based on your content. Styles allows you to replace the complete look of a control while maintaining its behavior. Animation brings your application to life, giving your users immediate feedback as they interact with it. These features give you the power to produce data visualizations so far beyond the capabilities of the data grid, the pinnacle most applications aspire to, that even Edward Tufte would be proud.

Combine these features with ClickOnce for the deployment and update of your WPF applications, both as standalone clients and as applications blended with your web site inside the browser, and you've got the foundation of the next generation of Windows applications.

The next generation of applications is going to blaze a trail into the unknown. WPF represents the best of the control-based Windows and content-based web worlds, combined with the performance of DirectX and the deployment capabilities of ClickOnce, building for us a vehicle just itching to be taken for a spin. And, like the introduction of fonts to the PC, which produced "ransom note" office memos, and the invention of HTML, which produced blinking online brochures, WPF is going to produce its own accidents along the road. Before we learn just what we've got in WPF, we're going to see a lot of strange and wonderful sights. I can't tell you where we're going to end up, but, with this book, I hope to fill your luggage rack so that you can make the journey.

Who This Book Is For

As much as I love the designers of the world, who are going to go gaga over WPF, this book is aimed squarely at my people: developers. We're not teaching programming here, so experience with some programming environment is a must before reading this book. Programming in .NET and C# are pretty much required; Windows Forms, XML and HTML are all recommended.

How This Book Is Organized

Here's what each chapter of this book will cover:

Chapter 1, Hello, WPF

> This chapter introduces the basics of WPF. It then provides a whirlwind tour of all the features that will be covered in the following chapters, so you can see how everything fits together before we delve into the details.

Chapter 2, Layout

> WPF provides a powerful set of tools for managing the visual layout of your applications. This chapter shows how to use this toolkit, and how to extend it.

Chapter 3, Controls

> Controls are the building blocks of a user interface. This chapter describes the controls built into the WPF framework and shows how to make your application respond when the user interacts with controls.

Chapter 4, Data Binding

> All applications need to present information to the user. This chapter shows how to use WPF's data-binding features to connect the user interface to your underlying data.

Chapter 5, Styles and Control Templates

> WPF provides an astonishing level of flexibility in how you can customize the appearance of your user interface and the controls it contains. Chapter 5 examines the customization facilities and shows how the styling and template mechanisms allow you to wield this power without compromising the consistency of your application's appearance.

Chapter 6, Resources

This chapter describes WPF's resource-handling mechanisms, which are used for managing styles, themes, and binary resources such as graphics.

Chapter 7, Graphics

WPF offers a powerful set of drawing primitives. It also offers an object model for manipulating drawings once you have created them.

Chapter 8, Animation

This chapter describes WPF's animation facilities, which allow most visible aspects of a user interface—such as size, shape, color, and position—to be animated.

Chapter 9, Custom Controls

This chapter shows how to write custom controls and other custom element types. It shows how to take full advantage of the WPF framework to build controls as powerful and flexible as the built-in controls.

Chapter 10, ClickOnce Deployment

ClickOnce allows applications to take full advantage of WPF's rich visual and interactive functionality while enjoying the benefits of web deployment.

Appendix A, XAML

The eXtensible Application Markup Language, XAML, is an XML-based language that can be used to represent the structure of n WPF user interface. This appendix describes how XAML is used to create trees of objects.

Appendix B, Interoperability

WPF is able to coexist with old user-interface technologies, enabling developers to take advantage of WPF without rewriting their existing applications. This appendix describes the interoperability features that make this possible.

Appendix C, Asynchronous and Multithreaded Programming in WPF Applications

> Multithreaded code and asynchronous programming are important techniques for making sure your application remains responsive to user input at all times. This appendix explains WPF's threading model and shows how to make sure your threads coexist peacefully with a WPF UI.

That's not to say that we've covered everything there is to know about WPF in this book. As of this writing, WPF is still pre-beta, so not everything is working as well as we'd like, some things are just plain missing, and still other things would require entire other books to get their just due. In this book, you will find little or no coverage of the following topics, among others: printing, "Metro," 3-D, video, UI automation, binding to relational data, and "eDocs."

What You Need to Use This Book

This book was produced with WinFX Beta 1, which includes WPF and WCF, and Visual Studio 2005 Beta 2. WPF is supported on Windows XP, Windows Server 2003, and, eventually, Longhorn.

By the time you read these words, Microsoft will have moved beyond these versions and provided new community technology previews of one or more of both of these technologies. However, I can say with certainty that the vast majority of the ideas and implementation details will be the same. For those that aren't, you should look at this book's web site for errata information, which we'll try to keep updated at major releases of WPF, right up until we release a new edition of this book at the release of WPF 1.0.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms and filenames.

Constant width

Indicates code, commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XML tags, HTML tags, macros, the contents of files, and the output from commands.

`Constant width bold`

Shows code or other text that should be noted by the reader.

Constant width elipses (...)

Shows code or other text not relevant to the current discussion.

This icon indicates a tip, suggestion, or general note.

This icon indicates a wrning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "Programming Windows Presentation Foundation, by Chris Sells and Ian Griffiths. Copyright 2005 O'Reilly Media, Inc., 0-596-10113-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, it means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top technology books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at http://safari.oreilly.com.

How to Contact Us

For the code samples associated with this book and errata—especially as WPF changes between the Beta 1 against which this book was written and major milestones before the release of WPF 1.0—visit the web site maintained by the authors at http://www.sellsbrothers.com/writing/avbook.

O'Reilly also maintains a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/avalon

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Ian Griffiths:

http://www.interact-sw.co.uk/iangblog/

Chris Sells:

http://sellsbrothers.com

Ian's Acknowledgments

Writing this book wouldn't have been possible without the support and feedback generously provided by a great many people. I would like to thank the following:

The readers, without whom this book would have a rather sad, lonely, and pointless existence.

My coauthor, Chris Sells, both for getting me involved in writing about WPF in the first place, and for his superb feedback and assistance.

Tim Sneath, both for his feedback and for providing me with the opportunity to meet and work with many members of the WPF team.

Microsoft employees and contractors, for producing a technology I like so much that I just had to write a book about it. And in particular, thank you to those people at Microsoft who gave their time to answer my questions or review draft chapters, including Chris Anderson, Marjan Badiei, Jeff Bogdan, Mark Boulter, Ben Carter, Dennis Cheng, Karen Corby, Beatriz de Oliveira Costa, Vivek Dalvi, Nathan Dunlap Ifeanyi Echeruo, Pablo Fernicola, Filipe Fortes, Aaron Goldfeder, John Gossman, Mark Grinols, Namita Gupta, Henry Hahn, Robert Ingebretson, Kurt Jacob, Karsten Januszewski, David Jenni, Michael Kallay, Amir Khella, Nick Kramer, Lauren Lavoie, Daniel Lehenbauer, Kevin Moore, Elizabeth Nelson, Seema Ramchandani, Rob Relyea, Chris Sano, Eli Schleifer, Adam Smith, Eric Stollnitz, Zhanbo Sun, David Teitlebaum, Stephen Turner, and Dawn Wood.

John Osborn and Caitrin McCullough at O'Reilly for their support throughout the writing process.

The technical review team: Matthew Adams, Craig Andera, Ryan Dawson, Glyn Griffiths, Adam Kinney, Drew Marsh, Dave Minter, and Brian Noyes. And particular thanks to Mike Weinhardt for his extensive and thoughtful feedback.

Finally, I especially want to thank Abi Sawyer for all her support, and for putting up with me while I wrote this book—thank you!

Chris's Acknowledgments

I'd like to thank the following people, without whom I wouldn't have been able to write this book:

First and foremost, the readers. When you've got something to say, you've got to have someone to say it to. I've been writing about WPF in various forums for more than 18 months, and you guys have always pushed and encouraged me further.

My coauthor, Ian Griffiths. Ian's extensive background in all things graphical and video-related, including technologies so deep I can't understand him half the time, plus his wonderful writing style, made him the perfect co-author on this book. I couldn't have asked for better.

Microsoft employees and contractors (in order that I found them in my WPF email folder): Lauren Lavoie, Lars Bergstrom, Amir Khella, Kevin Kennedy, David Jenni, Elizabeth Nelson, Beatriz de Oliveira Costa, Nick Kramer, Allen Wagner, Chris Sano, Tim Sneath, Steve White, Matthew Adams, Eli Schleifer, Karsten Januszewski, Rob Relyea, Mark Boulter, Namita Gupta, John Gossman, Kiran Kumar, Filipe Fortes, Guy Smith, Zhanbo Sun, Ben Carter, Joe Marini, Dwayne Need, Brad Abrams, Feng Yuan, Dawn Wood, Vivek Dalvi, Jeff Bogdan, Steve Makofsky, Kenny Lim, Dmitry Titov, Joe Laughlin, Arik Cohen, Eric Stollnitz, Pablo Fernicola, Henry Hahn, Jamie Cool, Sameer Bhangar, and Brent Rector. I regularly spammed a wide range of my Microsoft brethren, and instead of snubbing me, they answered my email questions, helped me make things work, gave me feedback on the chapters, sent me additional information without an explicit request, and, in the case of John Gossman, forwarded the chapters along to folks with special knowledge so that they could give me feedback. This is the first book I've written "inside," and with the wealth of information and conscientious people available, it'd be very, very hard to go back to writing "outside."

The external reviewers, who provide an extremely important mainstream point of view that Microsoft insiders can't: Craig Andera, Ryan Dawson, Glyn Griffiths (Ian's dad was an excellent reviewer!), Adam Kinney, Drew Marsh, Dave Minter and Brian Noyes.

Christine Morin for her work on the shared source Windows Forms version of solitaire and James Kovacs for his work extracting a UI independent engine from it so that I could build WPF Solitaire on top of it; this is the app that opened my eyes to the wonder and power of WPF. Also, to Peter Stern and Chris Mowrer who produced the faces and backs of the WPF Solitaire cards long before the technology was ready to support such a thing.

Caitrin McCullough and John Osborn from O'Reilly for supporting me in breaking a bunch of the normal ORA procedures and guidelines to publish the book I wanted to write.

Shawn Morrissey for letting me make writing a part of my first two years at Microsoft and even giving me permission to use some of that material to seed this book. Shawn put up with me,

trusting me to do my job remotely when very few Microsoft managers would. Also, Sara Williams for hiring me from my home in Oregon in spite of the overwhelming pressure to move all new employees to Washington.

Don Box for setting my initial writing quality bar and hitting me squarely between the eyes until I could clear it. Of course, thank you for the cover quote and for acting as my soundboard on this preface. You're an invaluable resource and dear friend.

Barbara Box for putting me up in the Chez Box clubhouse while I balance work and family in a way that wouldn't be possible without you.

Tim Ewald for that critical eye at the most important spots.

Michael Weinhardt as the primary developmental editor on this book. His feedback is probably the single biggest factor in whatever quality we've been able to cram into this book. As if that weren't enough, he produced many of the figures in my chapters.

Chris Anderson, architect on WPF, for a ton of illuminating conversations even after he started a competing book (although I'm convinced he'd be willing to talk to almost anyone once he'd entered the deadly "writer avoidance mode").

My family. This was the first book I've ever written while holding a full-time job and, worse than that, while I was learning a completely new job. Frankly, I neglected my family pretty thoroughly for about three solid months, but they understood and supported me, like they have all of my endeavors over the years. I am very much looking forward to getting back to them.

Chapter 1. Hello, WPF

Windows Presentation Foundation (or WPF, as we will refer to it throughout this book) is a completely new presentation framework, integrating the capabilities of those frameworks that preceded it, including User, GDI, GDI+, and HTML, and heavily influenced by toolkits targeted at the Web, such as Macromedia Flash, and popular Windows applications such as Microsoft Word. This chapter is meant to give you the basics of WPF from scratch and then take you on a whirlwind tour of the things you'll read about in detail in the chapters that follow.

I know that "Avalon" is now officially the "Windows Presentation Foundation," but that's quite a mouthful and "WPF" is something I'm still getting used to, so during this difficult transition, please don't think less of me when I use the term "Avalon."

1.1. WPF from Scratch

Example 1-1 is pretty much the smallest WPF application you can write in C#.

Example 1-1. Minimal C# WPF application

```
// MyApp.cs
using System;
using System.Windows; // the root WPF namespace

namespace MyFirstAvalonApp {
  class MyApp {
    [STAThread]
    static void Main(  ) {
      // the WPF message box
      MessageBox.Show("Hello, Avalon");
    }
  }
}
```

If you're not familiar with the `STAThread` attribute, it's a signal to .NET that when COM is initialized on the application's main thread, to make sure it's initialized to be compatible with single-threaded UI work , as required by WPF applications.

1.1.1. Building Applications

Building this application is a matter of firing off the C# compiler from a command shell with the appropriate environment variables,[*] as in Example 1-2.

[*] Start ⟶ Programs ⟶ Microsoft WinFX SDK⟶ Debug Build Environment or Release Build Environment.

Example 1-2. Building a WPF application manually

```
C:\1st>csc /target:winexe /out:.\1st.exe
  /r:System.dll
  /r:c:\WINDOWS\Microsoft.NET\Windows\v6.0.4030\WindowsBase
.dll
  /r:c:\WINDOWS\Microsoft.NET\Windows\v6.0.4030\PresentationCore
.dll
  /r:c:\WINDOWS\Microsoft.NET\Windows\v6.0.4030\PresentationFramework
.dll
  MyApp.cs

Microsoft (R) Visual C# 2005 Compiler version 8.00.50215.44
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50215
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.
```

Here, we're telling the C# compiler that we'd like to create a Windows application (instead of a Console application, which we get by default), putting the result, *1st.exe*, into the current folder, bringing in the three main WPF assemblies (`WindowsBase`, `PresentationCore` and `PresentationFramework`), along with the core .NET System assembly, and compiling the *MyApp.cs* source file.

Running the resulting *1st.exe* produces the world's lamest WPF application, as shown in Figure 1-1.

Figure 1-1. A lame WPF application

In anticipation of less lame WPF applications, refactoring the compilation command line into an *msbuild* project file is recommended, as in Example 1-3.

Example 1-3. A minimal msbuild project file

```
<!-- 1st.csproj -->
<Project
  DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild
/2003">
  <PropertyGroup>
    <OutputType>winexe</OutputType>
    <OutputPath>.\</OutputPath>
    <Assembly>1st.exe</Assembly>
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="MyApp.cs" />
```

```
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <Import Project="$(MsbuildBinPath)\Microsoft.CSharp.targets" />
</Project>
```

`Msbuild` is a .NET 2.0 command-line tool that understands XML files in the form shown in Example 1-3. The file format is shared between msbuild and Visual Studio 2005 so that you can use the same project files for both command-line and IDE builds. In this *.csproj* file (which stands for "C# Project"), we're saying the same things that we said to the C# compiler—i.e., that we'd like a Windows application, that we'd like the output to be *1st.exe* in the current folder, and that we'd like to reference the main WPF assemblies while compiling the *MyApp.cs* file. The actual smarts of how to turn these minimal settings into a compiled WPF application are contained in the .NET 2.0 *Microsoft.CSharp.targets* file that imported at the bottom of the file.

Executing *msbuild.exe* on the *1st.csproj* file looks like Example 1-4.

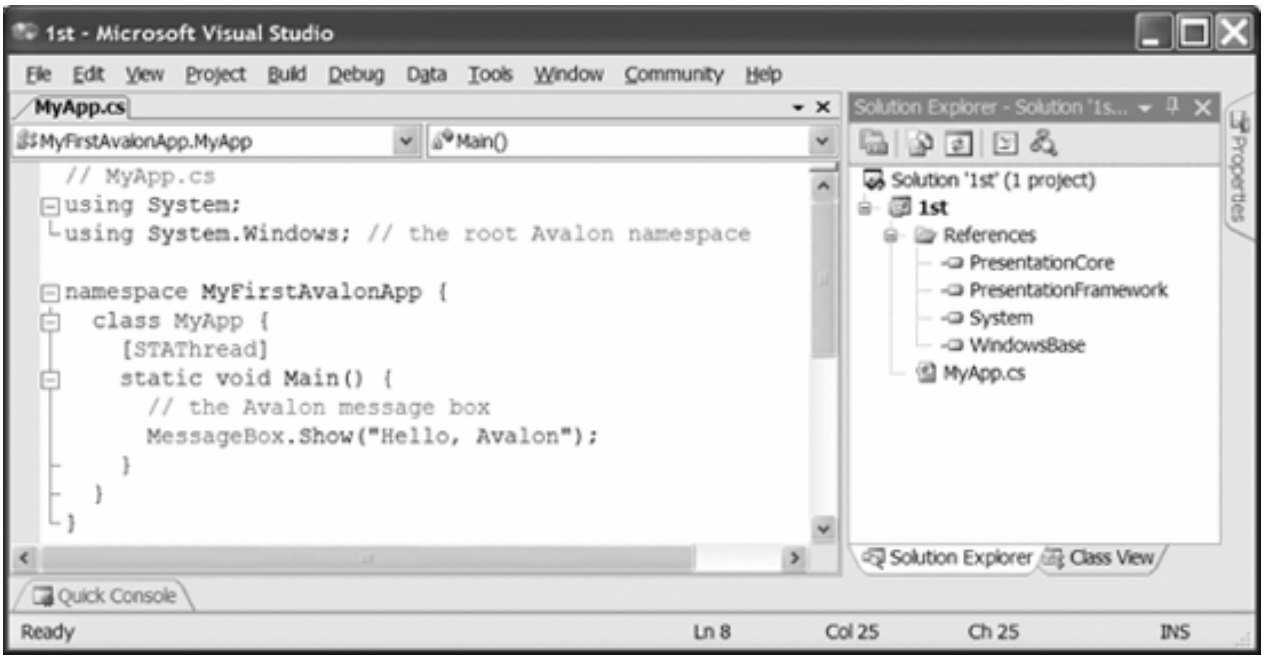
Example 1-4. Building using msbuild

```
C:\1st>msbuild 1st.csproj
Microsoft (R) Build Engine Version 2.0.50215.44
[Microsoft .NET Framework, Version 2.0.50215.44]
Copyright (C) Microsoft Corporation 2005. All rights reserved.

Build started 7/6/2005 8:20:39 PM.
__________________________________________________  _  _
Project "C:\1st\1st.csproj" (default targets):

Target PrepareForBuild:
    Creating directory "obj\Release\".
Target CompileRdlFiles:
    Skipping target "CompileRdlFiles" because it has no inputs.
Target CoreCompile:
    Csc.exe /noconfig /nowarn:"1701;1702" /reference:C:\WINDOWS\Microsoft.net\Wi
ndows\v6.0.4030\PresentationCore.dll /reference:C:\WINDOWS\Microsoft.net\Windows
\v6.0.4030\PresentationFramework.dll /reference:C:\WINDOWS\Microsoft.NET\Framewo
rk\v2.0.50215\System.dll /reference:C:\WINDOWS\Microsoft.net\Windows\v6.0.4030\W
indowsBase.dll /out:obj\Release\1st.exe /target:winexe MyApp.cs
Target CopyAppConfigFile:
    Skipping target "CopyAppConfigFile" because it has no outputs.
Target CopyFilesToOutputDirectory:
    Copying file from "obj\Release\1st.exe" to ".\1st.exe".
    1st -> C:\1st\1st.exe

Build succeeded.
    0 Warning(s)
```

```
    0 Error(s)

Time Elapsed 00:00:00.98
```

As I mentioned, msbuild and Visual Studio 2005 share a project file format, so loading the project file into VS is as easy as double-clicking on *1st.csproj*, which provides us all of the rights and privileges thereof (as shown in Figure 1-2).

Figure 1-2. Loading the minimal msbuild project file into Visual Studio



Unfortunately, as nice as the project file makes building our WPF application, the application itself is still lame.

1.1.2. The Application Object

A real WPF application is going to need more than a message box. WPF applications have an instance of the Application class from the `System.Windows` namespace. The Application class provides events like `StartingUp` and `ShuttingDown` for tracking lifetime; methods like `Run` for starting the application; and properties like `Current`, `ShutdownMode`, and `MainWindow` for finding the global application object, choosing when it shuts down, and getting the application's main window. Typically, the `Application` class serves as a base for custom application-wide behavior, as in Example 1-5.

Example 1-5. A less minimal WPF application

```
// MyApp.cs
using System;
using System.Windows;

namespace MyFirstAvalonApp {
  class MyApp : Application {
    [STAThread]
    static void Main(string[] args) {
      MyApp app = new MyApp(  );
      app.StartingUp += app.AppStartingUp;
      app.Run(args);
    }

    void AppStartingUp(object sender, StartingUpCancelEventArgs
 e) {
      // By default, when all top level windows
      // are closed, the app shuts down
      Window window = new Window(  );
      window.Text = "Hello, Avalon";
      window.Show(  );
    }
  }
}
```

Here, our `MyApp` class derives from the `Application` base class. In `Main`, we create an instance of the `MyApp` class, add a handler to the `StartingUp` event, and kick things off with a call to the `Run` method, passing the command-line arguments passed to `Main`. Those same command-line arguments are available in the `StartingUpCancelEventArgs` passed to the `StartingUp` event handler. (The `StartingUp` event handler will show its value as we move responsibility for the application's entry point to WPF later in this chapter.)

Our `StartingUp` handler creates our sample's top-level window, which is an instance of the built-in WPF `Window` class, making our sample WPF application more interesting from a developer point of view, although visually less so, as shown in Figure 1-3.

Figure 1-3. A less lame WPF application

While we can create instances of the built-in classes of WPF like `Window`, populating them and wiring them up from the application, it's much more encapsulating (not to mention abstracting) to create custom classes for such things, like the `Window1` class in Example 1-6.

Example 1-6. Window class declaring its own controls

```
// Window1.cs
using System;
using System.Windows;
using System.Windows.Controls; // Button et al

namespace MyFirstAvalonApp {
  class Window1 : Window {
    public Window1(  ) {
      this.Text = "Hello, Avalon";

      // Do something interesting (sorta...)
      Button button = new Button(  );
      button.Content = "Click me, baby, one more time!";
      button.Width = 200;
      button.Height = 25;
      button.Click += button_Click;

      this.AddChild(button);
    }

    void button_Click(object sender, RoutedEventArgs e) {
      MessageBox.Show(
        "You've done that before, haven't you...",
        "Nice!");
    }
  }
}
```

In addition to setting its caption text, an instance of our `Window1` class will include a button with its `Content`, `Width`, and `Height` properties set and its `Click` event handled. With this initialization handled in the `Window1` class itself, our app's startup code looks a bit simpler (even though the application itself has gotten "richer"), as in Example 1-7.

Example 1-7. Simplified Application instance

```
// MyApp.cs
using System;
using System.Windows;

namespace MyFirstAvalonApp {
  class MyApp : Application {
    [STAThread]
    static void Main(string[] args) {
      MyApp app = new MyApp(  );
      app.StartingUp += app.AppStartingUp;
      app.Run(args);
    }
```

```
    void AppStartingUp(object sender, StartingUpCancelEventArgs e) {
      // Let the Window1 initialize itself
      Window window = new Window1(  );
      window.Show(  );
    }
  }
}
```

The results, shown in Figure 1-4, are unlikely to surprise you much.

Figure 1-4. A slightly more interesting WPF application

As the `Window1` class gets more interesting, we're mixing two very separate kinds of code: the "look," represented by the initialization code that sets the window and child window properties, and the "behavior," represented by the event-handling code. As the look is something that you're likely to want handled by someone with artistic sensibilities (a.k.a. "turtleneck-wearing designer types"), whereas the behavior is something you'll want to leave to the coders (a.k.a. "pocket-protector-wearing engineer types"), separating the former from the latter would be a good idea. Ideally, we'd like to move the imperative "look" code into a declarative format suitable for tools to create with some drag 'n' drop magic. For WPF, that format is XAML.

1.1.3. XAML

XAML is an XML-based language for creating and initializing .NET objects. It's used in WPF as a serialization format for objects from the WPF presentation stack, although it can be used for a much larger range of objects than that. Example 1-8 shows how our Window-derived class is declared using XAML.

Example 1-8. Declaring a Window in XAML

```
<!-- Window1.xaml -->
<Window
  x:Class="MyFirstAvalonApp.Window1"
  xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
  xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
  Text="Hello, Avalon">
    <Button
      x:Name="button"
      Width="200"
      Height="25"
      Click="button_Click">Click me, baby, one more time!</Button>
</Window>
```

The root element, `Window`, is used to declare a portion of a class, the name of which is contained in the `Class` attribute from the XAML XML namespace (declared with a prefix of "x" using the "xmlns" XML namespace syntax). The two XML namespace declarations pull in two commonly used namespaces for XAML work, the one for XAML itself and the one for WPF. You can think of the XAML in Example 1-8 as creating the partial class definition[*] in Example 1-9.

[*] Partial classes are a new feature in C# 2.0 that allow you to split class definitions between multiple files.

Example 1-9. C# equivalent of XAML from Example 1-8

```
namespace MyFirstAvalonApp {
  partial class Window1 : Window {
    Button button;

    void InitializeComponent(  ) {
      // Initialize Window1
      this.Text = "Hello, Avalon";

      // Initialize button
      button = new Button(  );
      button.Width = 200;
      button.Height = 25;
      button.Click += button_Click;

      this.AddChild(button);
    }
  }
}
```

XAML was built to be as direct a mapping from XML to .NET as possible. Generally, every XAML element is a .NET class name and every XAML attribute is the name of a property or an event on that class. This makes XAML useful for more than just WPF classes; pretty much any old .NET class that exposes a default constructor can be initialized in a

XAML file.

Notice that we don't have the definition of the click event handler in this generated class. For event handlers and other initialization and helpers, a XAML file is meant to be matched with a corresponding code-behind file , which is a .NET language code file that implements behavior "behind" the look defined in the XAML. Traditionally, this file is named with a *.xaml.cs* extension and contains only the things not defined in the XAML. With the XAML from Example 1-9 in place, our single-buttoned main window code-behind file can be reduced to the code in Example 1-10.

Example 1-10. C# code-behind file

```
// Window1.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;

namespace MyFirstAvalonApp {
  public partial class Window1 : Window {
    public Window1(  ) {
      InitializeComponent(  );
    }

    void button_Click(object sender, RoutedEventArgs e) {
      MessageBox.Show(...);
    }
  }
}
```

Notice the `partial` keyword modifying the `Window1` class, which signals to the compiler that the XAML-generated class is to be paired with this human-generated class to form one complete class, each depending on the other. The partial `Window1` class defined in XAML depends on the code-behind partial class to call the `InitializeComponent` method and to handle the click event. The code-behind class depends on the partial `Window1` class defined in XAML to implement `InitializeComponent`, thereby providing the look of the main window (and related child controls).

Further, as I mentioned, XAML is not just for visuals. For example, there's nothing stopping us from moving most of the definition of our custom `MyApp` class into a XAML file, as in Example 1-11.

Example 1-11. Declaring an Application in XAML

```
<!-- MyApp.xaml -->
<Application
  x:Class="MyFirstAvalonApp.MyApp"
    xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
    xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
    StartingUp="AppStartingUp">
</Application>
```

This reduces the `MyApp` code-behind file to the event handler in Example 1-12.

Example 1-12. Application code-behind file

```
// MyApp.xaml.cs
using System;
using System.Windows;

namespace MyFirstAvalonApp {
  public partial class MyApp : Application {
    void AppStartingUp(object sender, StartingUpCancelEventArgs e) {
      Window window = new Window1(  );
      window.Show(  );
    }
  }
}
```

You may have noticed that we no longer have a `Main` entry point to create the instance of the application-derived class and call its `Run` method. That's because WPF has a special project setting to specify the XAML file that defines the application class, which appears in the msbuild project file, as in Example 1-13.

Example 1-13. Specifying the application's XAML in the project file

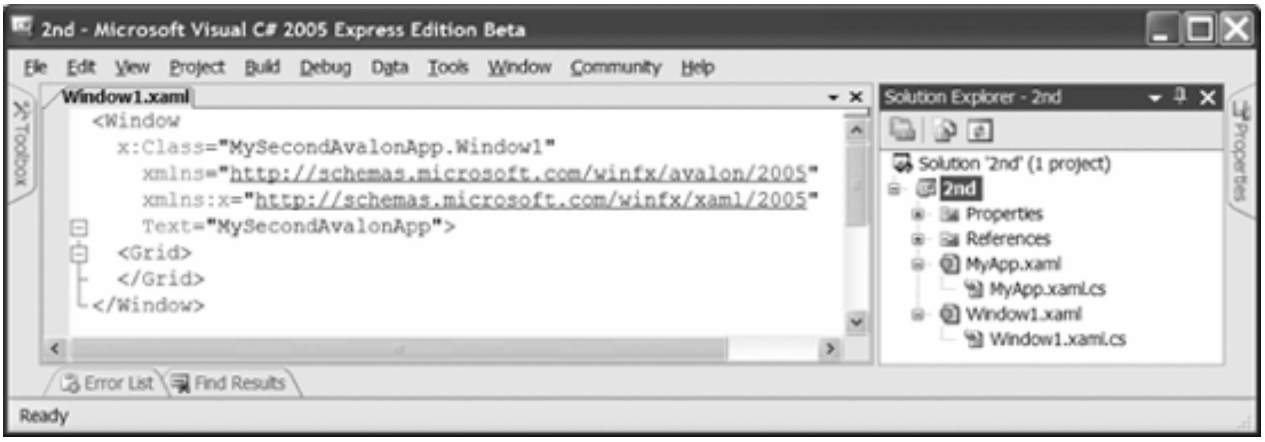
```
<!-- MyFirstAvalonApp.csproj -->
<Project ...>
  <PropertyGroup>
    <OutputType>winexe</OutputType>
    <OutputPath>.\</OutputPath>
    <Assembly>1st.exe</Assembly>
  </PropertyGroup>
  <ItemGroup>
    <ApplicationDefinition Include="MyApp.xaml" />
    <Compile Include="Window1.xaml.cs" />
    <Compile Include="MyApp.xaml.cs" />
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
```

```
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
    <Page Include="Window1.xaml" />
    <Page Include="MyApp.xaml" />
  </ItemGroup>
  <Import Project="$(MsbuildBinPath)\Microsoft.CSharp.targets" />
  <Import Project="$(MSBuildBinPath)\Microsoft.WinFX.targets" />
</Project>
```

The combination of the `ApplicationDefinition` element and the WinFX-specific *Microsoft.WinFX.targets* file produces an application entry point that will create our application for us. Also notice in Example 1-13 that we've replaced the *MyApp.cs* file with the *MyApp.xaml.cs* file, added the *Window1.xaml.c* file, and included the two corresponding XAML files as `Page` elements. The XAML files will be compiled into partial class definitions using the instructions in the *Microsoft.WinFX.targets* file.

This basic arrangement of artifacts—i.e., application and main window each split into a XAML and a code-behind file—is such a desirable starting point for a WPF application that creating a new project using the Avalon Application project template from within Visual Studio 2005 gives you just that initial configuration, as shown in Figure 1-5.

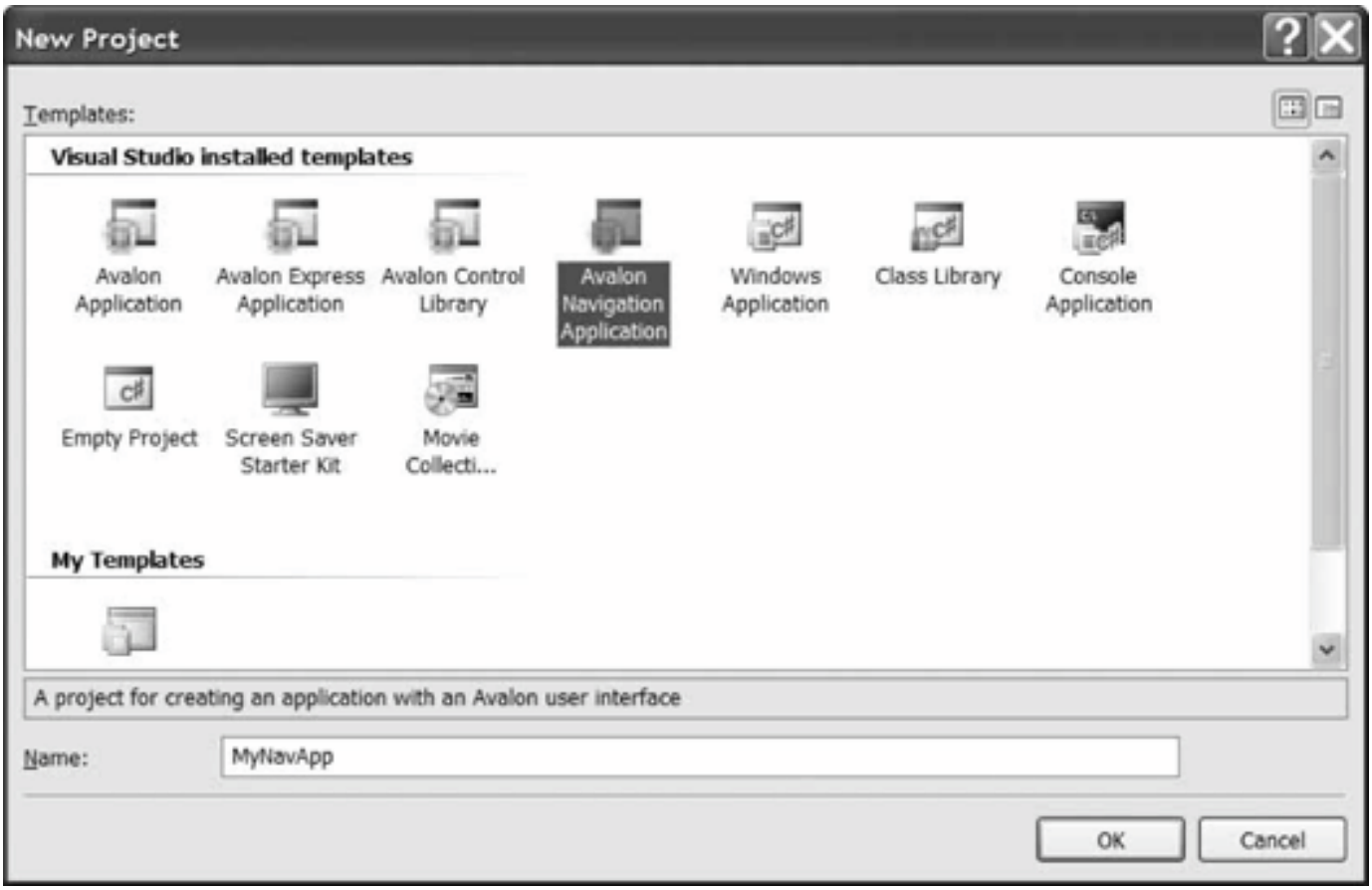
Figure 1-5. The result of running the Avalon Application project template



1.2. Navigation Applications

If you create a new WPF application using Visual Studio, you may notice that a few icons down from the Avalon Application icon is another project template called Avalon Navigation Application, as shown in Figure 1-6.

Figure 1-6. The Avalon Navigation Application project template in Visual Studio



WPF itself was created as a unified presentation framework, meant to enable building Windows applications with the best features from existing Windows application practice and existing web application practice. One of the nice things that web applications generally provide is a single window showing the user one page of content/functionality at a time, allowing for navigation between the pages. For some applications, including Internet Explorer, the Shell Explorer, Microsoft Money and a bunch of Control Panels, this is thought to be preferable to the more common Windows application practice of showing more than one window at a time.

To enable more of these kinds of applications in Windows, WPF provides the `NavigationApplication` in Example 1-14 to serve as the base of your custom application class instead of the `Application` class.

Example 1-14. The C# portion of a navigation application

```
// MyApp.xaml.cs
using System;
using System.Windows;
using System.Windows.Navigation;

namespace MyNavApp {
  public partial class MyApp : NavigationApplication {}
}
```

The `NavigationApplication` itself derives from the `Application` class and provides additional services such as navigation, history, and tracking the initial page to show when the application first starts, which is specified in the application's XAML file, as in Example 1-15.

Example 1-15. The XAML portion of a navigation application

```
<!-- MyApp.xaml.cs -->
<NavigationApplication
  x:Class="MyNavApp.MyApp"
  xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
  xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
  StartupUri="Page1.xaml">
</NavigationApplication>
```

In addition to the `StartupUri`, which specifies the first XAML page to show in our navigation application, notice that the `NavigationApplication` element doesn't have a `Text` property. In fact, if you were to set one, that would cause a compilation error, because a navigation application's main window title is set by the current page. A page in a WPF navigation application is a class that derives from the `Page` class, e.g., the XAML in Example 1-16.

Example 1-16. A sample navigation page

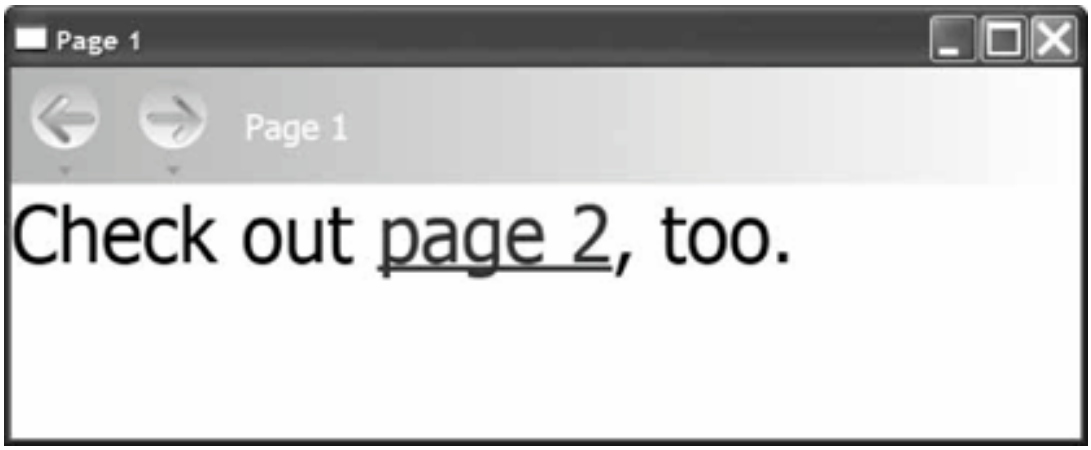
```
<!-- Page1.xaml -->
<Page
  x:Class="MyNavApp.Page1"
  xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
  xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
  Text="Page 1">
  <TextBlock FontSize="72" TextWrap="Wrap">
    Check out
    <Hyperlink NavigateUri
="page2.xaml">page 2</Hyperlink>,
    too.
  </TextBlock>
</Page>
```

Remember that the root element of a XAML file defines the base class, so this `Page` root element defines a class (`MyNavApp.Page1`) that derives from the WPF `Page` class. The `Text` property of the page will be the thing that shows in the caption as the user navigates from page to page.

1.2.1. Navigation

The primary way to allow the user to navigate is via the `Hyperlink` element, setting the `NavigateUri` to a relative URL of another page XAML in the project. The first page of our sample navigation application looks like Figure 1-7.

Figure 1-7. A sample navigation page in action



In Figure 1-7, the hyperlinked text is underlined in blue, and if you were to move your mouse cursor over the hyperlink, it would show up as red. Further, the page's `Text` property is set as the window caption, as well as on the toolbar across the top. This toolbar is provided for navigation applications for the sole

purpose of providing the back and forward buttons. The act of navigation through the application will selectively enable and disable these buttons, as well as fill in the history drop-down maintained by each button.

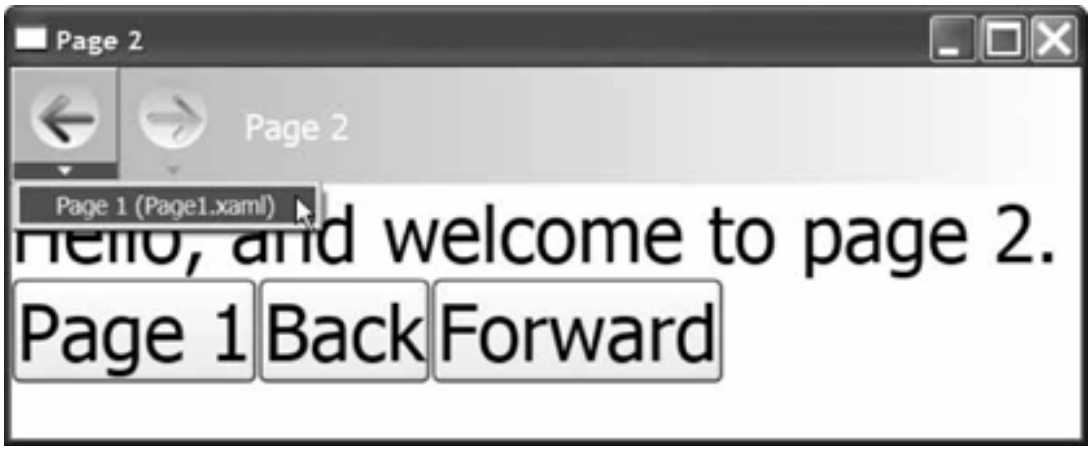
Let's define *page2.xaml*, as shown in Example 1-17.

Example 1-17. Another sample navigation page

```
<!-- Page2.xaml -->
<Page
  x:Class="MyNavApp.Page2"
  xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
  xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
  Text="Page 2">
    <TextBlock FontSize="72" TextWrap="Wrap">
    Hello, and welcome to page 2.
    <Button FontSize="72" Click="page1Button_Click">Page 1</Button>
    <Button FontSize="72" Click="backButton_Click">Back</Button>
    <Button FontSize="72" Click="forwardButton_Click">Forward</Button>
  </TextBlock>
</Page>
```

Clicking on the hyperlink on page 1 navigates to page 2, as shown in Figure 1-8.

Figure 1-8. Navigation history and custom navigation controls



Notice in Figure 1-8 that the history for the back button shows page 1, which is where we were just before going to page 2. Also notice the three buttons, which are implemented in Example 1-18 to demonstrate navigating to a specific page, navigating backward, and navigating forward.

Example 1-18. Custom navigation code

```
// Page2.xaml.cs
using System;
using System.Windows;
using System.Windows.Navigation;

namespace MyNavApp {
  public partial class Page2 : Page {
    void page1Button_Click(object sender, RoutedEventArgs e) {
      NavigationService.GetNavigationService(this).
        Navigate(new Uri("page1.xaml", UriKind.Relative));
    }

    void backButton_Click(object sender, RoutedEventArgs e) {
      NavigationService.GetNavigationService(this).GoBack(  );
    }

    void forwardButton_Click(object sender, RoutedEventArgs e) {
      NavigationService.GetNavigationService(this).GoForward(  );
    }
  }
}
```

Example 1-18 shows the use of static methods on the `NavigationService` class to navigate manually just as the hyperlink, back and forward buttons do automatically.

1.3. Content Model

While the different kinds of WPF application styles are interesting, the core of any presentation framework is in the presentation elements themselves. Fundamentally, we have "bits of content and behavior" and "containers of bits of content and behavior." We've already seen both kinds; e.g., a `Button` is a control, providing content and behavior and a `Window` is a container. There are two things that may surprise you about content containment in WPF, however.

The first is that you don't need to put a string as the content of a Button; it will take any .NET object. For example, you've already seen a string as a button's content, which looks like Figure 1-9, created with the code in Example 1-19.

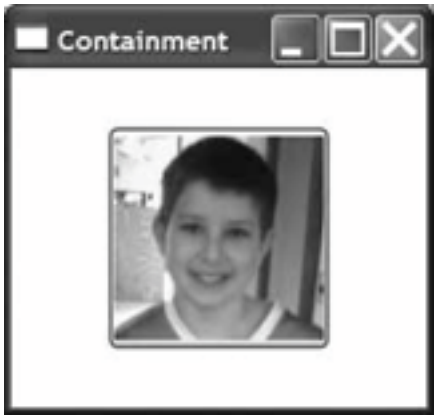
Figure 1-9. A button with string content

Example 1-19. A button with string content

```
<Window ...>
  <Button Width="100" Height="100">Hi</Button>
</Window>
```

However, you can also use an image, as in Figure 1-10 and implemented in Example 1-20.

Figure 1-10. A button with image content



Example 1-20. A button with image content

```
<Window ...>
  <Button Width="100" Height="100">
    <Image Source="tom.png" />
  </Button>
</Window>
```

You can even use an arbitrary control, like a `TextBox`, as shown in Figure 1-11 and implemented in Example 1-21.

Figure 1-11. A button with control content

Example 1-21. A button with control content

```
<Window ...>
  <Button Width="100" Height="100">
    <TextBox Width="75">edit me</TextBox>
  </Button>
</Window>
```

Further, as you'll see in Chapters 2 and 5, you can get fancy and show a collection of nested elements in a `Button` or even use nonvisual objects as the content of a `Button`. The reason that the `Button` can take any object as content is because it's derived ultimately from a class called `ContentControl`, as are many other WPF classes—e.g., `Label`, `ListBoxItem`, `ToolTip`, `CheckBox`, `RadioButton` and, in fact, `Window` itself.

A `ContentControl` knows how to hold anything that's able to be rendered, not just a string. A `ContentControl` gets its content from the `Content` property, so you could specify a `Button`'s content like so (this is the longhand version of Example 1-19):

```
<Button Width="100" Height="100" Content="Hi" />
```

`ContentControls` are especially useful, because you get all of the behavior of the "thing,"—e.g., `Button`, `Window`, or `ListBoxItem`—but you can display whatever you like in it without having to build yourself a special class—e.g., `ImageButton`, `TextBoxListBoxItem`, etc.

1.3.1. XAML Property-Element Syntax

Still, while setting the `Content` property as a string attribute in XAML works just fine for specifying a string as content, it doesn't work at all well for specifying an object as content, such as in the image example. For this reason, XAML defines the property-element syntax, which uses nested `Element.Property` elements for specifying objects as property values. Example 1-22 shows the property-element syntax to set a string as a button's content.

Example 1-22. Property element syntax with a string

```
<Button Width="100" Height="100">
  <Button.Content>Hi</Button.Content>
</Button>
```

Example 1-23 is another example using an image:

Example 1-23. Property element syntax with an Image

```
<Button Width="100" Height="100">
  <Button.Content>
    <Image Source="tom.png" />
  </Button.Content>
</Button>
```

Since XML attributes can only contain one thing, property-element syntax is especially useful when you've got more than one thing to specify. For example, you might imagine a button with a string and an image, defined in Example 1-24.

Example 1-24. Can't have multiple things in a ContentControl

```
<Button Width="100" Height="100">
  <!-- WARNING: doesn't work! -->
  <Button.Content>
    <TextBlock>Tom: </TextBlock>
    <Image Source="tom.png" />
  </Button.Content>
</Button>
```

While normally the property-element syntax would be useful for this kind of thing, in this particular case, it doesn't work at all. This brings us to the second thing that may surprise you about content containment in WPF: while `Button` can take any old thing as content, as can a `Window`, both of them can only take a single thing which, without additional instructions, WPF will center and fill up the element's entire client area. For more than one content element or a richer layout policy, you'll need a panel.

1.4. Layout

Taking another look at Example 1-24 with the `TextBlock` and the `Image` as content for the `Button`, we don't really have enough information to place them inside the area of the button. Should they be stacked left-to-right or top-to-bottom? Should one be docked on one edge and one docked to the other? How are things stretched or arranged if the button resizes? These are questions best answered with a panel.

A panel is a control that knows how to arrange its content. WPF comes with the general-purpose panel controls listed in Table 1-1.

Table 1-1. Main panel types

Panel type	Usage
`DockPanel`	Allocates an entire edge of the panel area to each child; useful for defining the rough layout of simple applications at a coarse scale.
`StackPanel`	Lays out children in a vertical or horizontal stack; extremely simple, useful for managing small scale aspects of layout.
`Grid`	Arranges children within a grid; useful for aligning items without resorting to fixed sizes and positions. The most powerful of the built-in panels.
`Canvas`	Performs no layout logic—puts children where you tell it to; allows you to take complete control of the layout process.

1.4.1. Grid Layout

The most flexible panel by far is the grid, which arranges content elements in rows and columns and includes the ability to span multiple rows and columns, as shown in Example 1-25.

Example 1-25. A sample usage of the Grid panel

```
<Window ...>
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Button Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2">A</Button>
    <Button Grid.Row="0" Grid.Column="2">C</Button>
    <Button Grid.Row="1" Grid.Column="0" Grid.RowSpan="2">D</Button>
    <Button Grid.Row="1" Grid.Column="1">E</Button>
    <Button Grid.Row="1" Grid.Column="2">F</Button>
    <Button Grid.Row="2" Grid.Column="1">H</Button>
    <Button Grid.Row="2" Grid.Column="2">I</Button>
  </Grid>
</Window>
```

Example 1-25 used the XAML property-element syntax to define a grid with three rows and three columns inside the `RowDefinition` and `ColumnDefinition` elements. In each element, we've specified the `Grid.Row` and `Grid.Column` properties so that the grid knows which elements go where (the grid can have multiple elements in the same cell). One of the elements spans two rows, and one spans two columns, as shown in Figure 1-12.

Figure 1-12. A sample Grid panel in action

Using the grid, we can be explicit about how we want to arrange an image with a text caption, as in

Example 1-26.

Example 1-26. Arranging an image and text in a grid

```
<Button Width="100" Height="100">
  <Button.Content>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <Image Grid.Row="0" Source="tom.png" />
      <TextBlock
        Grid.Row="1"
        HorizontalAlignment="Center">Tom</TextBlock>
    </Grid>
  </Button.Content>
</Button>
```

Figure 1-13 shows how the grid arranges the image and text for us.

Figure 1-13. A grid arranging an image and a text block

Since we're just stacking one element on top of another, we could've used the stack panel, but the grid is so general-purpose that many WPF programmers find themselves using it for most layout configurations.

1.4.2. XAML Attached Property Syntax

You may have noticed that in setting up the `Grid.Row` and `Grid.Panel` attributes of the `Button` elements, we used another dotted syntax, similar to the property-element syntax, but this time on the attribute instead of on the element. This is the attached-property syntax and is used to set a property as

associated with a particular element— e.g., a `Button`—but as defined by another element—e.g., a `Grid`.

The attached-property syntax is used in WPF as an extensibility mechanism. We don't want the `Button` class to have to know that it's being arranged in a `Grid`, but we do want to specify `Grid`-specific attributes on it. If the `Button` were being hosted in a `Canvas`, the `Grid` properties wouldn't make any sense, so building `Row` and `Column` properties into the `Button` class isn't such a great idea. Further, when we define our own custom panel that the WPF team never considered—e.g., `HandOfCards`—we want to be able to apply the `HandOfCards` attached properties to arbitrary elements it contains.

This kind of extensibility is what the attached-property syntax was designed for, and it is common when arranging content on a panel.

For the nitty-gritty of layout, including the other panels and text composition that I didn't show, you'll want to read Chapter 2.

1.5. Controls

While the layout panels provide the container, the controls are the important things you'll be arranging. So far, you've already seen examples of creating instances of controls, setting properties, and handling events. You've also seen the basics of the content model that makes controls in WPF special. However, for the details of event routing, command handling, mouse/keyboard input and an enumeration of the controls in WPF, you'll want to check out Chapter 3. Further, for information about packaging up custom UI and behavior, as well as the techniques discussed in the rest of this chapter and the rest of this book, you'll want to read Chapter 9.

1.6. Data Binding

Once we've got a set of controls and a way to lay them out, we still need to fill them with data and keep that data in sync with wherever the data actually lives. (Controls are a great way to show data but a poor place to keep it.)

For example, imagine that we'd like to build an actual WPF application for keeping track of people's nicknames. Something like Figure 1-14 would do the trick.

Figure 1-14. Data binding to a collection of custom types

In Figure 1-14, we've got two `TextBox` controls, one for the name and one for the nickname; the actual nickname entries in a `ListBox` in the middle; and a `Button` to add new entries. The core data of such an application could easily be built with a class, as shown in Example 1-27.

Example 1-27. A custom type with data binding support

```
public class Nickname : INotifyPropertyChanged {
  // INotifyPropertyChanged Member
  public event PropertyChangedEventHandler PropertyChanged;
  protected void OnPropertyChanged(string propName) {
    if( PropertyChanged != null ) {
      PropertyChanged(this, new PropertyChangedEventArgs(propName));
    }
  }

  string name;
  public string Name {
    get { return name; }
    set {
      name = value;
      OnPropertyChanged("Name"); // notify consumers
```

```
    }
  }

  private string nick;
  public string Nick {
    get { return nick; }
    set {
      nick = value;
      OnPropertyChanged("Nick"); // notify consumers
    }
  }

  public Nickname(  ) : this("name", "nick") { }
  public Nickname(string name, string nick) {
    this.name = name;
    this.nick = nick;
  }
}
```

This class knows nothing about data binding, but it does have two public properties that expose the data, and it implements the standard `INotifyPropertyChanged` interface to let consumers of this data know when it has changed.

In the same way that we have a standard interface for notifying consumers of objects when they change, we also have a standard way to notify consumers of collections of changes called `INotifyCollectionChanged`. WPF provides an implementation of this interface called `ObservableCollection`, which we'll use to fire the appropriate event when `Nickname` objects are added or removed, as in Example 1-28.

Example 1-28. A custom collection type with data binding support

```
  // Notify consumers
  public class Nicknames : ObservableCollection<Nickname> { }
```

Around these classes, we could build nickname-management logic that looks like Example 1-29.

Example 1-29. Making ready for data binding

```
// Window1.xaml.cs
...
namespace DataBindingDemo {
  public class Nickname : INotifyPropertyChanged {...}
  public class Nicknames : ObservableCollection<Nickname> { }

  public partial class Window1 : Window {
    Nicknames names;

    public Window1(  ) {
      InitializeComponent(  );
      this.addButton.Click += addButton_Click;

      // create a nickname collection
      this.names = new Nicknames(  );

      // make data available for binding
      dockPanel.DataContext = this.names;
    }

    void addButton_Click(object sender, RoutedEventArgs e) {
      this.names.Add(new Nickname(  ));
    }
  }
}
```

Notice the window's class constructor provides a click event handler to add a new nickname and creates the initial collection of nicknames. However, the most useful thing that the `Window1` constructor does is set its `DataContext` property so as to make the nickname data available for data binding.

Data binding is about keeping object properties and collections of objects synchronized with one or more controls' view of the data. The goal of data binding is to save you the pain and suffering associated with writing the code to update the controls when the data in the objects change and with writing the code to update the data when the user edits the data in the controls. The synchronization of the data to the controls depends on the `INotifyPropertyChanged` and `INotifyCollectionChanged` interfaces that we've been careful to use in our data and data-collection implementations.

For example, because the collection of our sample nickname data and the nickname data itself both

notify consumers when there are changes, we can hook up controls using WPF data binding, as in Example 1-30.

Example 1-30. An example of data binding

```
<!-- Window1.xaml -->
<Window x:Class="DataBindingDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
    xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
    Text="Nicknames">
  <DockPanel x:Name="dockPanel">
    <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
      <TextBlock VerticalAlignment="Center">Name: </TextBlock>
      <TextBox Text="{Binding Path=Name}" />
      <TextBlock VerticalAlignment="Center">Nick: </TextBlock>
      <TextBox Text="{Binding Path=Nick}" />
    </StackPanel>
    <Button DockPanel.Dock="Bottom" x:Name="addButton">Add</Button>
    <ListBox
      ItemsSource="{Binding}"
      IsSynchronizedWithCurrentItem="True" />
  </DockPanel>
</Window>
```

This XAML lays out the controls as shown in Figure 1-14, using a dock panel to arrange things top-to-bottom and a stack panel to arrange the editing controls. The secret sauce that takes advantage of data binding is the `{Binding}` values in the control attributes instead of hardcoded values. By setting the `Text` property of the `TextBox` to `{Binding Path=Name}`, we're telling the `TextBox` to use data binding to peek at the `Name` property out of the current `Nickname` object. Further, if the data changes in the `Name TextBox`, the `Path` is used to poke the new value back in.

The current `Nickname` object is determined by the `ListBox` because of the `IsSynchronizedWithCurrentItem` property, which keeps the `TextBox` controls showing the same `Nickname` object as the one that's currently selected in the `ListBox`. The `ListBox` is bound to its data by setting the `ItemsSource` attribute to `{Binding}` without a `Path` statement. In the `ListBox`, we're not interested in showing a single property on a single object, but rather all of the objects at once.

But how do we know that both the `ListBox` and the `TextBox` controls are sharing the same data? That's where setting the dock panel's `DataContext` comes in. In the absence of other instructions,

when a control's property is set using data binding, it looks at its own `DataContext` property for data. If it doesn't find any, it looks at its parent and then that parent's parent, and so on, all the way up the tree. Because the `ListBox` and the `TextBox` controls have a common parent that has a `DataContext` property set (the `DockPanel`), all of the data-bound controls will share the same data.

1.6.1. XAML Markup-Extension Syntax

Before we take a look at the results of our data binding, let's take a moment to discuss the XAML markup-extension syntax, which is what you're using when you set an attribute to something inside of curly braces—e.g., `Text="{Binding Path=Name}"`. The markup-extension syntax adds special processing to XAML attribute values. For example, the `BindingExtension` class creates an instance of the `Binding` class, populating its properties with the parsed string that comes afterward. Logically, the following:

```
<TextBox Text="{Binding Path=Name}" />
```

turns into the following:

```
Binding binding = new Binding(  );
binding.Path = "Name";
textbox1.Text =
  binding.ProvideValue(textbox1, TextBox.TextProperty);
```

In fact, the binding-extension syntax is just a shortcut for the following (which you'll recognize as the property-element syntax):

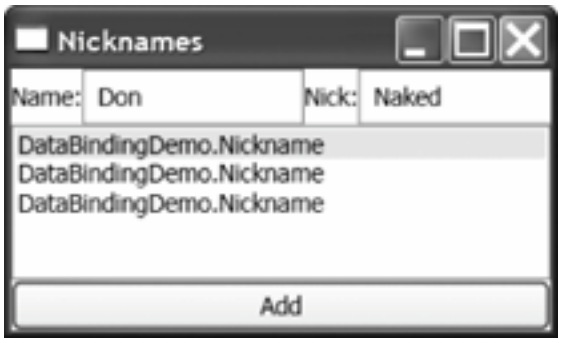
```
<TextBox.Text>
  <Binding Path="Name" />
</TextBox.Text>
```

For a complete discussion of markup extensions, as well as the rest of the XAML syntax, you'll want to read Appendix A.

1.6.2. Data Templates

With the data-binding markup syntax explained, let's turn back to our sample data-binding application, which so far doesn't look quite like what we had in mind, as seen in Figure 1-15.

Figure 1-15. ListBox showing objects of a custom type without special instructions



It's clear that the data is making its way into the application, since the currently selected name and nickname are shown for editing. The problem is that, unlike the `TextBox` controls which were each given a specific field of the `Nickname` object to show, the `ListBox` is expected to show the whole thing. Lacking special instructions, the `ListBox` calling the `ToString` method of each object, which only results in the name of the type. To show the data, we need to compose a data template, as shown in Example 1-31.

Example 1-31. Using a data template

```
<ListBox
  ItemsSource="{Binding}"
  IsSynchronizedWithCurrentItem="True">

  <ListBox.ItemTemplate>
    <DataTemplate>
      <TextBlock>
        <TextBlock TextContent="{Binding Path=Name}" />:
        <TextBlock TextContent="{Binding Path=Nick}" />
      </TextBlock>
    </DataTemplate>
  </ListBox.ItemTemplate>

</ListBox>
```

The `ListBox` control has an `ItemTemplate` property that expects a data template: a template of elements that should be inserted for each listbox item, instead of the results of the call to `ToString`. In Example 1-31, we've composed a data template from a text block that flows together two other text blocks, each bound to a property on a `Nickname` object separated by a colon, as shown in Figure 1-16.

Figure 1-16. How a ListBox shows objects of a custom type with a data template

At this point, we've got a completely data-bound application. As data in the collection or the individual objects changes, the UI will be updated and vice versa. However, there is a great deal more to say on this topic, not least of which is pulling in XML as well as object data, which are covered in Chapter 4.

1.7. Dependency Properties

While our data-source `Nickname` object made its data available via standard .NET properties, we need something special to support data binding on the target element. While the `TextContent` property of the `TextBlock` element is exposed with a standard property wrapper, for it to integrate with WPF services such as data binding, styling and animation, it also needs to be a dependency property . A dependency property provides several features not present in .NET properties, including the ability to inherit its value from a container element, support externally set defaults, provide for object-independent storage (providing a potentially huge memory savings), and change tracking.

Most of the time, you won't have to worry about dependency properties versus .NET properties, but when you need the details, you can read about them in the Chapter 9.

1.8. Resources

Resources are named chunks of data defined separately from code and bundled with your application or component. .NET provides a great deal of support for resources , a bit of which we already used when we referenced *tom.png* from our XAML button earlier in this chapter. WPF also provides special support for resources scoped to elements defined in the tree.

As an example, let's declare some default instances of our custom `Nickname` objects in XAML in Example 1-32.

Example 1-32. Declaring objects in XAML

```
<!-- Window1.xaml -->
<?Mapping XmlNamespace="local" ClrNamespace="DataBindingDemo" ?>
<Window
    x:Class="DataBindingDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
    xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
    xmlns:local="local"
    Text="Nicknames">
  <Window.Resources>
    <local:Nicknames x:Key="names">
      <local:Nickname Name="Don" Nick="Naked" />
      <local:Nickname Name="Martin" Nick="Gudge" />
      <local:Nickname Name="Tim" Nick="Stinky" />
    </local:Nicknames>
  </Window.Resources>
  <DockPanel DataContext="{StaticResource names}">
    <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
      <TextBlock VerticalAlignment="Center">Name: </TextBlock>
      <TextBox Text="{Binding Path=Name}" />
      <TextBlock VerticalAlignment="Center">Nick: </TextBlock>
      <TextBox Text="{Binding Path=Nick}" />
    </StackPanel>
    ...
  </DockPanel>
</Window>
```

Notice the `Window.Resources`, which is property-element syntax to set the `Resources`

property of the `Window1` class. Here, we can add as many named objects as we like, with the name coming from the `Key` attribute and the object coming from the XAML elements (remember that XAML elements are just a mapping to .NET class names). In this example, we're creating a Nicknames collection named `names` to hold three `Nickname` objects, each constructed with the default constructor, and then setting each of the `Name` and `Nick` properties.

Also notice the use of the `StaticResource` markup extension to reference the `names` resource as the collection to use for data binding. With this XAML in place, our window construction reduces to the code in Example 1-33.

Example 1-33. Finding a resource in code

```
public partial class Window1 : Window {
  Nicknames names;

  public Window1(  ) {
    InitializeComponent(  );
    this.addButton.Click += addButton_Click;

    // get names collection from resources
    this.names = (Nicknames)this.FindResource("names");

    // no need to make data available for binding here
    //dockPanel.DataContext = this.names;
  }

  void addButton_Click(object sender, RoutedEventArgs e) {
    this.names.Add(new Nickname(  ));
  }
}
```

Now instead of creating the collection of names, we can pull it from the resources with the `FindResource` method. Just because this collection was created in XAML doesn't mean that we need to treat it any differently than we treated it before, which is why the `Add` button event handler is the exact same code. Also, there's no need to set the data context on the dock panel, because that property was set in the XAML.

1.8.1. XAML Mapping Syntax

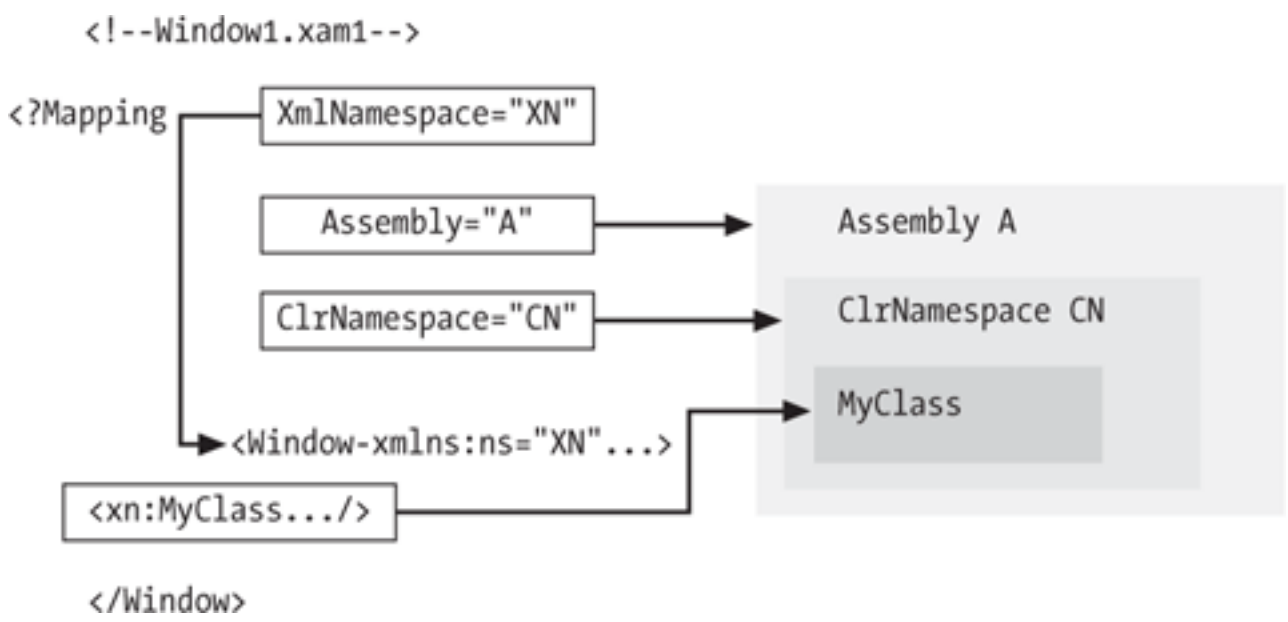
Before we go on with resources, we need to discuss a new XAML syntax that's come up, the mapping syntax . The XAML mapping syntax provides the ability to bring in types not already known by the XAML compiler. Our use of the mapping syntax looks like Example 1-34.

Example 1-34. XAML mapping syntax

```
<?Mapping XmlNamespace="local" ClrNamespace="DataBindingDemo" ?>
<Window
    x:Class="DataBindingDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
    xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
    xmlns:local="local"
    Text="Nicknames">
  <Window.Resources>
   <local:Nicknames x:Key="names">
    ...
  </Window.Resources>
  ...
</Window>
```

When bringing in a new type in XAML, we have to do a two-step mapping. The first step is to map a CLR namespace to an XML namespace. That's what the `<?Mapping ... ?>` line does. The second step is to map the XML namespace to a namespace prefix, which is what the `xmlns:local` attribute establishes. I've used `local` for both the XML namespace and prefix. I've chosen `local` because the CLR namespace to which I'm referring must be part of the assembly being compiled along with the XAML in question. You can import CLR namespaces for another assembly by specifying the optional `Assembly` attribute as part of the mapping, as Figure 1-17 shows.

Figure 1-17. XAML mapping syntax summary



As of the current build at the time of writing, when you're using a mapping directive in a XAML file, you must also specify a UI culture in the .csproj file, which you can do by adding a `UICulture` property to a `PropertyGroup` element, e.g.:

```
<!-- DataBindingDemo.csproj -->
<Project ...>
  <PropertyGroup>
    <UICulture>en-US</UICulture>
    ...
  </PropertyGroup>
 ...
</Project>
```

With the mapping completed, you're able to use the XML namespace prefix in front of any class with a default constructor to create an instance of it in a XAML file.

For the full scoop on resources, including resource scoping and lookup, as well as using them for theming and skinning, read Chapter 6.

1.9. Styles and Control Templates

One of the major uses for Resources is for styles . A style is a set of property/value pairs to be applied to one or more elements. For example, recall the two `TextBlock` controls from our `Nickname` sample, each of which was set to the same `VerticalAlignment` (see Example 1-35).

Example 1-35. Multiple TextBlock controls with the same settings

```
<!-- Window1.xaml -->
<Window ...>
  <DockPanel ...>
    <StackPanel ...>
      <TextBlock VerticalAlignment="Center">Name: </TextBlock>
      <TextBox Text="{Binding Path=Name}" />
      <TextBlock VerticalAlignment="Center">Nick: </TextBlock>
      <TextBox Text="{Binding Path=Nick}" />
    </StackPanel>
    ...
  </DockPanel>
</Window>
```

If we wanted to bundle the `VerticalAlignment` setting into a style, we could do this with a `Style` element in a `Resources` block, as shown in Example 1-36.

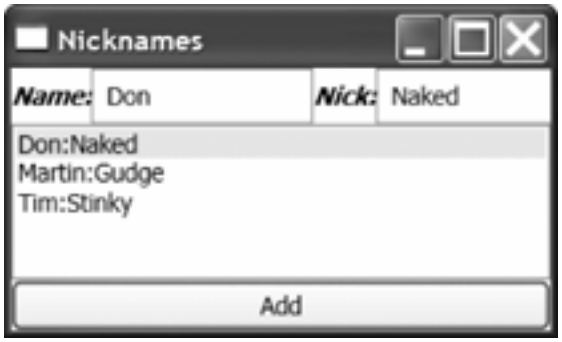
Example 1-36. An example TextBlock style

```
<Window ...>
  <Window.Resources>
    <Style x:Key="myStyle" TargetType="{x:Type TextBlock}">
      <Setter Property="VerticalAlignment" Value="Center" />
      <Setter Property="FontWeight" Value="Bold" />
      <Setter Property="FontStyle" Value="Italic" />
    </Style>
  </Window.Resources>
  <DockPanel ...>
    <StackPanel ...>
      <TextBlock Style="{StaticResource myStyle}">Name: </TextBlock>
      <TextBox Text="{Binding Path=Name}" />
      <TextBlock Style="{StaticResource myStyle}">Nick: </TextBlock>
```

```
      <TextBox Text="{Binding Path=Nick}" />
    </StackPanel>
    ...
  </DockPanel>
</Window>
```

The style element is really just a named collection of `Setter` elements for a specific class (specified with the `Type` markup extension). The `TextBlock myStyle` style centers the vertical alignment property and, just for fun, sets the text to bold italic as well. With the style in place, it can be used to set the `Style` property of any `TextBlock` that references the style resource. Applying this style as in Example 1-36 yields Figure 1-18.

Figure 1-18. Named style in action on two TextBlock controls



Notice that only two of the `TextBlock` controls we used in this example use our style, because we only applied it to two `TextBlock` controls. If we want to take the next step and apply this style to all `TextBlock` controls defined in the scope of the style definition, we can do so by specifying the target type without specifying a key, as in Example 1-37.

Example 1-37. Styles assigned based on type

```
<Window ...>
  <Window.Resources>
    <Style TargetType="{x:Type TextBlock}">
      <Setter Property="VerticalAlignment" Value="Center" />
      <Setter Property="FontWeight" Value="Bold" />
      <Setter Property="FontStyle" Value="Italic" />
    </Style>
  </Window.Resources>
  <DockPanel ...>
    <StackPanel ...>
      <TextBlock>Name: </TextBlock>
      <TextBox Text="{Binding Path=Name}" />
      <TextBlock>Nick: </TextBlock>
      <TextBox Text="{Binding Path=Nick}" />
    </StackPanel>
    ...
  </DockPanel>
</Window>
```

In Example 1-37, we've dropped the `x:Key` attribute while leaving the `TargetType` property set. The use of just `TargetType` applies the style to all elements of that type, which means that we can also drop any mention of style on the individual `TextBlock` elements. Figure 1-19 shows the results.

Figure 1-19. A type-based style applied to TextBlock controls

In addition to setting styles on controls, you can set them on arbitrary types (like the `Nickname` object we defined earlier) or replace a control's entire look, both of which you can read about in Chapter 5.

Further, if you'd like to apply property changes over time, you can do so with styles that include animation information, which is discussed in Chapter 8 (although Figure 1-20 is a small taste of what

WPF animations can produce).

Figure 1-20. Buttons with animated glow (Color Plate 1)

1.10. Graphics

If no element or set of elements provides you with the look you want in your application, you can build it up from the set of graphics primitives that WPF provides, including rectangles, polygons, lines, ellipses, etc. WPF also lets you affect the way it renders graphics in any element, offering facilities that include bordering, rotating, or scaling another shape or control. WPF's support for graphics is engineered to fit right into the content model we're already familiar with, as shown in Example 1-38. from Chapter 7.

Example 1-38. Adding graphics to a Button

```
<Button LayoutTransform="scale 3 3">
  <StackPanel Orientation="Horizontal">
    <Canvas Width="20" Height="18" VerticalAlignment="Center">
      <Ellipse Canvas.Left="1" Canvas.Top="1" Width="16" Height="16"
        Fill="Yellow" Stroke="Black" />
      <Ellipse Canvas.Left="4.5" Canvas.Top="5" Width="2.5" Height="3"
        Fill="Black" />
      <Ellipse Canvas.Left="11" Canvas.Top="5" Width="2.5" Height="3"
        Fill="Black" />
      <Path Data="M 5,10 A 3,3 0 0 0 13,10" Stroke="Black" />
    </Canvas>
    <TextBlock VerticalAlignment="Center">Click!</TextBlock>
  </StackPanel>
</Button>
```

Here we've got three ellipses and a path composed inside a canvas, which is hosted inside a stack panel with a text block that, when scaled via the `LayoutTransform` property on the button, produces Figure 1-21.

Figure 1-21. A scaled button with a collection of graphic primitives

Notice that there's nothing special about the graphic primitives in XAML; they're declared and integrated as content just like any of the other WPF elements we've discussed. The graphics and the transformation are integrated into the same presentation stack as the rest of WPF, which is a bit of a difference for User/GDI programmers of old.

Further, graphics in WPF are not limited to 2-D; Figure 1-22 shows an example of a simple 3-D figure that was defined declaratively just like a 2-D graphic.

Figure 1-22. A very simple 3-D model (Color Plate 2)

For a complete discussion of how graphics primitives, retained drawings, color, lines, brushes, and transformations happen in WPF, both declaratively and in code, as well as an introduction to 3-D and video, you'll want to read Chapter 7.

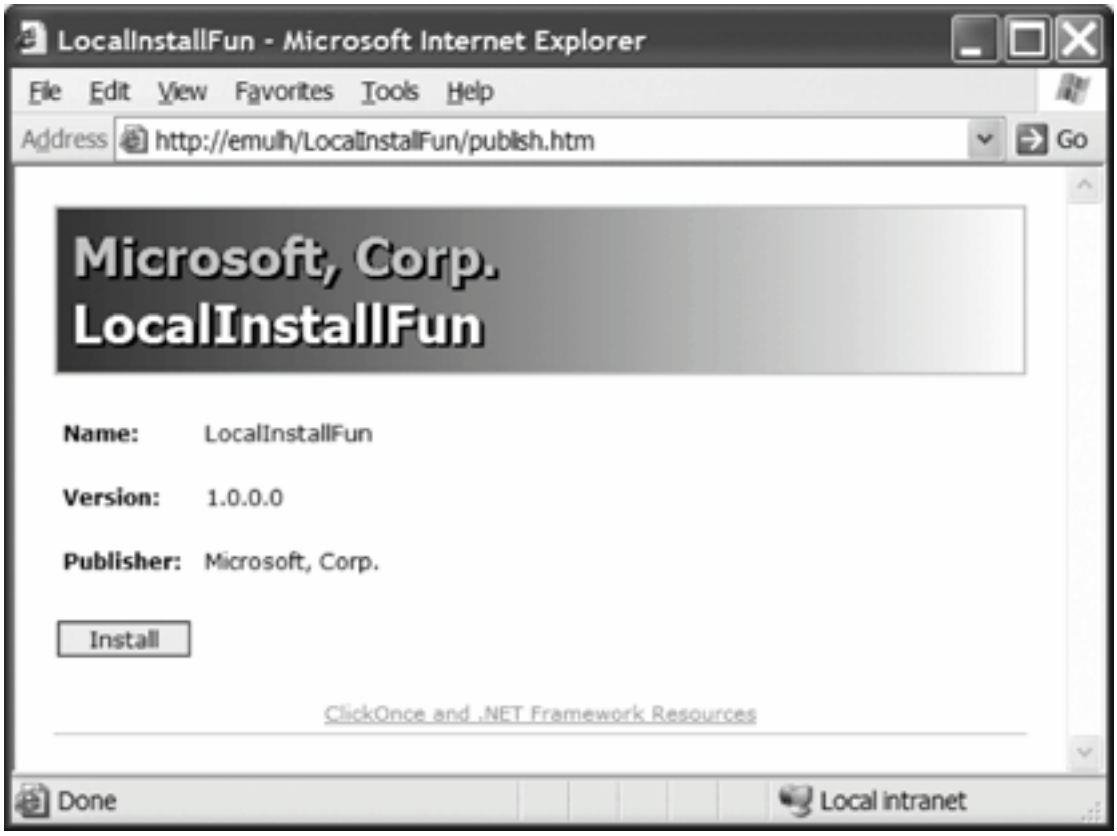
1.11. Application Deployment

Once you've packed all of the features that WPF supports into your application, you still have the challenge of getting it to your users. As part of its effort to merge the best of Windows and the Web, WPF leverages the ClickOnce application-deployment support built into .NET 2.0 to enable WPF applications to be deployed over the Web.

You can deploy your WPF application to a web server for their enjoyment by right-clicking on the project in the Solution Explorer and choosing the Publish option, which brings up the Publish Wizard, which asks you where and how you would like to publish the output of your project.

If you choose the defaults, you'll have chosen to publish a ClickOnce "local install" application to the local machine's web server. A successful publication will bring up a Visual Studio-generated HTML file for you to test your application, as shown in Figure 1-23.

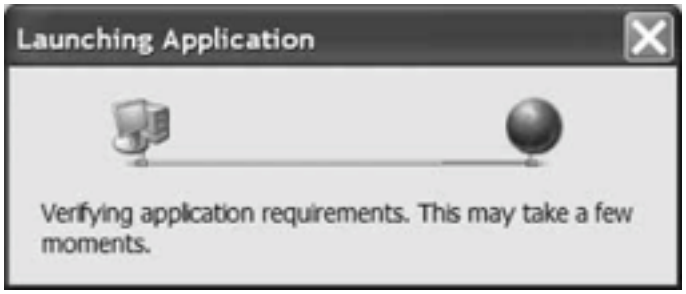
Figure 1-23. Visual Studio-generated publish.htm



This version of *publish.htm* uses the default settings, including the name of the company of the author who built this sample (which is only coincidently the name of the company that developed WPF), the project name of the application, and the version number for the initial publication of this application.

Clicking on this link deploys the application. The first step is downloading the files that make up the application, as shown in Figure 1-24.

Figure 1-24. Launching a ClickOnce application



Once the files are downloaded, they're checked for a certificate signature from a known publisher. Since this requires nondefault settings, what you'll see is Figure 1-25.

Figure 1-25. Launching a ClickOnce application from an unknown publisher

The Security Warning dialog is shown when the user hasn't already awarded a publisher the permissions to take certain liberties on their computer or when the publisher isn't known at all. If the user presses the Cancel button at this point, no potentially "evil" code has been downloaded or executed. If the user presses the Install button, the application's files are put into the right place, a Start menu folder is established and the application itself is executed, just like a normal WPF application.

If the user can establish a connection to the web server, subsequent runs will detect application updates and offer to upgrade the user's copy of the application, providing them a way to roll back

to previous versions (or even uninstall completely) using the Add or Remove Control Panel. Also, even if there is no network connection, since the application is locally installed, no connection is needed to launch the application from the Start menu.

Those of you with trusting souls will now be thinking to yourself, "Wow! What an easy way to get bits out to my users and keep them up to date with the latest features and bug fixes!" and you'd be absolutely right; that's exactly what ClickOnce was designed for and why WPF was specially engineered to work with it.

Those of you with suspicion in your hearts will now be thinking, "Wow! What a great way for naughty men to format my hard drive, send incriminating email to my boss in my name, or reset my Minesweeper high scores!" and you'd be right, too— except that ClickOnce applications run in the .NET Code Access Security sandbox.

As of this writing, WPF locally installed ClickOnce applications require "full trust" (which is .NET-speak for "I have always depended on the kindness of strangers"), but this will not be the case for the release of WPF 1.0.

This has been the quickest possible overview of ClickOnce deployment of WPF applications. For much more detail on WPF application deployment, including express applications that are deployed over the Web and hosted in the browser, please read Chapter 10.

1.12. Where Are We?

The Application object forms the initial piece on which to build your WPF applications. The Application definition, along with any `Window` or `Page` objects you may have, are most often split between a declarative XAML file for the look and an imperative code file for the behavior. Your applications can be normal, like a standard Windows application, or navigation-based, like the browser. In fact, the latter can be integrated into the browser, and both can be deployed and kept up to date over the Web using ClickOnce.

Building your application is a matter of grouping controls in containers: either single content containers, such as windows or buttons, or multiple content containers that provide layout capabilities, such as the canvas and the grid.

When bringing your controls together, you'll want to populate them with data that's synchronized with the in-memory home of the data, which is what data binding is for, and keep them pretty, which is what styles are for. If you want to declare data or styles in your XAML, you can do so using resources, which are just arbitrary named objects that aren't used to render WPF UI directly.

If no amount of data or style property settings makes you satisfied with the look of your control, you can replace it completely with control templates, which can be made up of other controls or graphics primitives. In addition, you can apply graphic operations—such as rotating, scaling, or animation—to graphic primitives or controls in WPF's integrated way.

Chapter 2. Layout

All applications need to present information to users. For this information to be conveyed effectively, it should be arranged onscreen in a clear and logical way. WPF provides a powerful and flexible array of tools for controlling the layout of the user interface.

There is a fine line between giving the developer or designer enough control over the user interface's layout, and leaving them to do all the work. A good layout system should be able to automate common scenarios such as resizing, scaling, and adaptation to localization but should allow manual intervention where necessary.

WPF provides a set of *panels* : elements that handle layout. Each individual panel type offers a straightforward and easily understood layout mechanism. As with all WPF elements, layout objects can be composed in any number of different ways, so while each individual element type is fairly simple, the flexible way in which they can be combined makes for a very powerful layout system. And you can even create your own layout element types should the built-in ones not meet your needs.

In this chapter, we will look at where each of the basic layout panels fits into a typical UI design. We will also examine some of the text-layout features of WPF.