THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Beginning Silverlight 2

From Novice to Professional

Learn to build state-of-the-art Silverlight applications quickly and easily

Robert Lair

Apress®

Beginning Silverlight 2

From Novice to Professional

Robert Lair

Apress®

Beginning Silverlight 2: From Novice to Professional

ISBN-13 (pbk): 978-1-59059-952-5

ISBN-10 (pbk): 1-59059-952-7

ISBN-13 (electronic): 978-1-4302-0570-8

ISBN-10 (electronic): 1-4302-0570-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Stefan Turalski
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Tracy Brown Collins
Copy Editor: Marilyn Smith
Associate Production Director: Kari Brooks-Copony
Production Editor: Elizabeth Berry
Compositor: Octal Publishing, Inc.
Proofreader: Linda Seifert
Indexer: Broccoli Information Management
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

This book is dedicated to my mother, Linda, who passed after a long fight with cancer on January 6, 2008. Your courageous battle was and will always be an encouragement to me and all who knew you. I love you and miss you. I also would like to dedicate this book to my dad, Ken, who lost the gift most precious to him. Your strength has been an inspiration to me. I love you, Dad.

Contents at a Glance

Contents

About the Author

ROBERT LAIR has been working with .NET technologies since before its alpha release, and built the original IBuySpy Store and Portal applications that were used by Microsoft to introduce ASP.NET to the development community. He is the author of *Pure ASP.NET* (Sams, 2002), a reference for web development in the .NET Framework, and portions of *ASP.NET for Developers*, as well as numerous magazine articles on the topic of .NET. Robert has also been a speaker at a number of .NET technical conferences. Technologies in which Robert specializes include Silverlight, CRM–Live service integration, mainframe modernization to .NET, ASP.NET custom application development, and SharePoint development and integration.

Currently, Robert works as an independent consultant, offering development and architect services to companies worldwide. Follow Robert on Twitter at `http://www.twitter.com/robertlair` and on the Web at `http://www.robertlair.com`.

Acknowledgments

There are a number of people to whom I would like to express my appreciation—people who have helped me in many ways. This book proved to be a much greater challenge than I had anticipated, as throughout the course of writing, a number of life events crept in, including the death of my mom. Without these people, this book would never have been possible.

First and foremost, I would like to thank my loving and inspiring wife, Debi. While I spent hours in my office writing, she picked up the slack around the house, and through it all provided me with love, support, and a lighthouse that I could always turn to and refocus on. I cannot thank her enough for the sacrifices she has endured, and for always being there. I would also like to thank my son, Max, who gave up a great deal of time with his dad. I hope this book is proof that you can accomplish anything if you put your mind to it and have the perseverance to see it through. I look forward to once again giving you the time you both deserve and to being a family again.

I would like to thank the many people at Apress who made this book happen. I would especially like to thank Tracy Brown Collins, Ewan Buckingham, Marilyn Smith, Elizabeth Berry, Dominic Shakeshaft, and Stefan Turalski. Without all of your hard work, this book would never have happened. Thank you all.

Through life's many challenges, everyone needs someone to whom they can turn. I would like to thank a great friend of mine, Paul Strozier, for all of his support and encouragement, and especially for his friendship. True friends are priceless. Thank you, Paul, for everything.

And finally, I would like to shout out from Gooblicious to Warclock, Essy, and the members of Slighted and Distant Beliefs, who provided me with friendship and an escape when I needed one.

Introduction

Welcome to *Beginning Silverlight 2: From Novice to Professional*. This book will provide you with an introduction to Silverlight: what it is, what it means to you as a developer, and how to begin developing Silverlight-enabled applications. You'll not only read about the features of the Silverlight development environment, but also work through many hands-on examples that demonstrate exactly how to use those features to create rich Internet applications (RIAs).

Who Should Read This Book

This book is written for application developers who want to get started with Silverlight 2. It assumes that you have some experience developing applications using technologies related to Microsoft's ASP.NET, and have worked with Microsoft Visual Studio. You should be familiar with the JavaScript, C#, and XML languages.

How This Book Is Organized

Each chapter focuses on a particular area of Silverlight and contains one or more "Try It Out" exercises that allow you to apply what you have learned. Here is a summary of what each chapter includes:

- Chapter 1, "Welcome to Silverlight 2," gives you an introduction to RIAs and Silverlight. You will also learn about the tools used in developing Silverlight-enabled applications.
- Chapter 2, "Introduction to Visual Studio 2008," introduces Visual Studio 2008 and the important new features offered in this version. In this chapter, you will build your first Silverlight application.
- Chapter 3, "Layout Management in Silverlight," discusses Silverlight's flexible layout management system, which lets you specify how controls will appear in your applications. It describes Silverlight 2's layout management controls in depth.
- Chapter 4, "Silverlight Form Controls," introduces the common form controls that are provided with Silverlight 2. You will continue to work with these controls throughout the book.
- Chapter 5, "Data Binding and Silverlight List Controls," looks at the Silverlight 2 controls that display lists of data and how to bind data to these controls. You'll see that these controls are flexible and can show data in unique ways.

- Chapter 6, "Data Access and Networking," describes how data access in Silverlight applications works compared with data access in traditional applications. It then explores mechanisms for accessing data in Silverlight applications, focusing on the use of web services.
- Chapter 7, "Local Storage in Silverlight," covers localized storage in Silverlight 2, which is handled by its *isolated storage* feature. You'll learn how to store user-specific data for your application and have that data persist over browser instances.
- Chapter 8, "Introduction to Expression Blend," gets you started with Microsoft Expression Blend, which lets you edit XAML documents visually.
- Chapter 9, "Styling in Silverlight," describes how you can control the styles of your Silverlight application's user interface elements. You'll learn about defining style properties inline using both Visual Studio and Expression Blend, as well as how to use Silverlight styles.
- Chapter 10, "Transformations and Animation," covers creating animations in Silverlight 2. You'll see how Expression Blend 2 helps you create complex animations and transformations.
- Chapter 11, "Custom Controls," explains the basics of creating custom controls in Silverlight 2. First, it covers when it might be appropriate to write custom controls in Silverlight 2, and then it describes how to build a custom control that has several different states.

By the time you finish this book, you will have a firm foundation in Silverlight 2, and will be able to create your own Silverlight-enabled applications.

CHAPTER 1

Welcome to Silverlight 2

This chapter introduces Silverlight, a Microsoft cross-browser, cross-platform plug-in that allows you to create rich interactive (or Internet) applications (RIAs) for the Web. It begins with a brief look at the evolution of user interfaces, and then provides an overview of Silverlight. You'll learn how Silverlight fits into RIA solutions, the benefits it brings to developers, and the tools involved in developing Silverlight-enabled applications.

The Evolution of the User Interface

Software user interfaces are constantly evolving and improving. I remember back when I was still working with an early version of Windows and looking at Mac OS with envy. Then I remember seeing Linux systems with radical new desktop interfaces. More recently, I found myself looking again at the Mac OS X Dock (see Figure 1-1) and wanting that for my Windows XP machine—to the point where I purchased a product that mimicked it. I was dedicated to Windows through it all, but I was envious of some of the user experiences the different environments offered.

Figure 1-1. *The Mac OS Dock feature*

The evolution of the user interface continues in the Windows Vista operating system. One example is the interface for switching between applications. In past versions of Windows, when you pressed Alt+Tab to switch from one program to another, you would see a rather ugly interface offering nothing but icons. Today, when you press Alt+Tab in Vista, you get a much more user-friendly interface, presenting a clipping of the content of each window as you tab through your choices, as shown in Figure 1-2.

Figure 1-2. *Windows Vista Alt+Tab user interface*

In addition, Vista offers an even cooler way to switch between applications using the Desktop Window Manager. When you press the Windows key along with Tab, Vista displays all open windows in a cascading shuffle effect, which allows you to see a large-scale version of each window (see Figure 1-3). And if there is animated content in any of the windows, it actually shows up in the view! So, if you have a video or a game playing in one of the windows, you will see that in action as you shuffle through the windows.

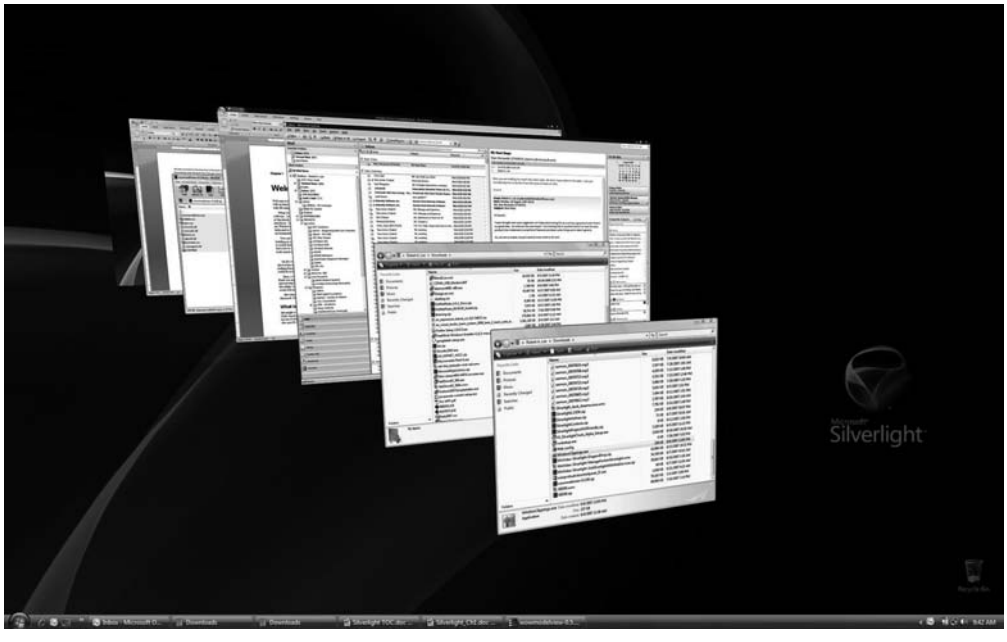

Figure 1-3. *Windows Vista Windows+Tab cascading windows shuffle effect*

These features reflect how developers have built standard desktop applications, which are meant to be installed and executed on individual client machines. Desktop applications allow for very rich and responsive user interfaces and additional features, such as offline support. Performance of the application depends on the machine on which it is installed. A challenge for desktop applications is deployment. The application needs to have a code base for each target platform, and every machine needs to have the application installed and maintained.

In contrast, we have web applications, which are HTML-focused programs designed to run within a browser and across platforms. For the Microsoft-based developer, this has recently meant developing with ASP.NET and building web services to offer services over the Internet. The focus of most of the logic and code has been placed on the server for the benefit of application performance. The price has been a poor user interface.

With recent technologies, the line between the desktop and web approaches for developing applications has started to blur. As a result, a third approach has surfaced. This new approach is termed RIA, which is defined as a web application that has the features and functionality found in traditional desktop applications.

Rich Internet Application Solutions

The concept of RIA has been around for quite some time, but the term *rich Internet application* was first used in 2002 in a Macromedia white paper. Before then, the terms *remote scripting* and *X Internet* were used to describe the concept.

Today, many different solutions fit the description of RIAs, but there is one consistent characteristic: all RIA solutions involve a runtime that runs on the client machine and architecturally sits between the user and the server.

In recent years, the technology that is most commonly used in RIAs is Flash. When Flash was introduced, it brought to the Web rich user experiences never seen before. However, due to the lack of tools allowing Microsoft .NET developers to integrate Flash into their applications, to those developers, Flash just seemed like a tool for adding some pretty effects to a web page, but nothing functional.

Then a wonderful thing happened when Adobe purchased Macromedia. All of the sudden, Flash was married to some of the development tools offered by Adobe. Microsoft retaliated by announcing Silverlight, formerly known as Windows Presentation Foundation Everywhere (WPF/E). Silverlight is the technology that many .NET developers have been waiting for.

But what exactly is Silverlight? And what impact does Silverlight actually have on us as .NET developers? Well, I'm glad you asked.

What Is Silverlight?

As I stated in the previous section, all RIAs have one characteristic in common: a client runtime that sits between the user and the server. In the case of Microsoft's RIA solution, Silverlight is this client runtime. Specifically, Silverlight is a cross-platform, cross-browser plug-in that renders user interfaces and graphical assets on a canvas that can be inserted into an HTML page.

The markup used to define a Silverlight canvas is called Extensible Application Markup Language (XAML, pronounced "zammel"). XAML is an XML-based language that is similar to HTML in some ways. Like HTML, XAML defines which elements appear, as well as the layout of those elements. However, unlike HTML, XAML goes far beyond simple element definition and layout. Using XAML, you can also specify timelines, transformations, animations, and events.

The following is an example of a Silverlight canvas defined in XAML:

```
<Canvas
   xmlns="http://schemas.microsoft.com/client/2007"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Width="640" Height="480"
   Background="White"
   x:Name="Page">
   <Rectangle
      RenderTransformOrigin="0.5,0.5"
      x:Name="rectangle"
      Width="292"
      Height="86"
      Fill="#FFFF0000"
      Stroke="#FF000000"
      StrokeThickness="3"
      Canvas.Left="115"
      Canvas.Top="70">
   </Rectangle>
</Canvas>
```

Figure 1-4 shows this canvas in Microsoft Expression Blend, the design tool used to edit and create XAML for Silverlight applications. You can see that this XAML simply defines a rectangle on a canvas, as well as the properties associated with that rectangle, including its name, location, size, color, and border.

This simple example is just intended to give you an idea of what XAML looks like. You'll learn more about XAML in upcoming chapters. For now, let's continue by looking at the benefits of Silverlight.

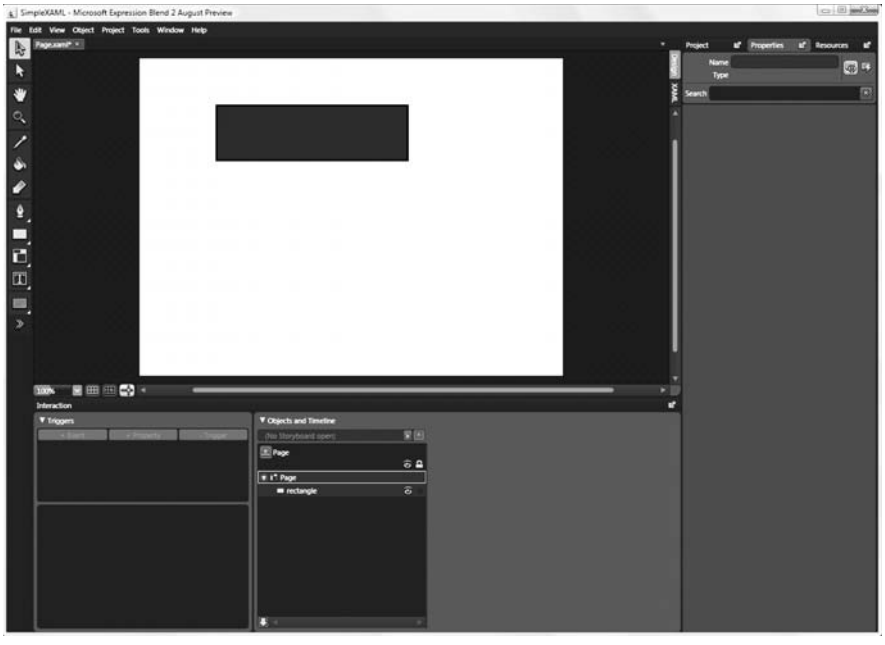

Figure 1-4. *A basic XAML canvas in Microsoft Expression Blend*

Benefits of Silverlight

Naturally, Silverlight offers all of the same benefits of RIAs, but there are a few features that set it apart from other RIA solutions, including the following:

- It offers cross-platform/cross-browser support.
- It provides a cross-platform version of the .NET Framework.
- XAML is a text-based markup language.
- Silverlight uses familiar technologies.
- It's easy to deploy the Silverlight runtime to clients.

Let's take a closer look at each of these benefits.

Cross-Platform/Cross-Browser Support

When ASP.NET was released a number of years ago, one of the benefits touted was cross-browser support. Developers would need to have only one code base, and that code base would work in all modern browsers. For the most part, this is true. No matter which browser you are using, the application will function. However, in order to receive all of the bells and whistles offered by the ASP.NET controls, you must use the latest version of Internet Explorer. If you are using any other browser, you actually get a downgraded version of the web site, which contains fewer features.

Validation controls are a prime example. If you are using a browser that ASP.NET recognizes as an "upscale" browser, you can take advantage of client-side validation. If you are using any other browser, the validation controls still function, but require a post-back to the server to do the validation. So, although ASP.NET is cross-browser, users can get different experiences, depending on which browser they are using.

With Silverlight, this changes. Microsoft is once again pulling out the term *cross-browser*, and also adding *cross-platform*, and this time they mean it. As a developer, you can create a Silverlight application and rest assured that it will run exactly the same on all supported platforms and browsers.

Currently, two platforms are supported. Naturally, the first is Windows-based platforms, and the second is Mac OS platforms. As for browser support, Internet Explorer and Firefox are currently covered. Microsoft has committed support for Safari as well, so it may be on the list by the time you're reading this book.

This leaves one large platform unsupported: Linux. Although Microsoft does not have plans to support Linux, others do. The Mono project, which is sponsored by Novell, is an open source initiative to develop and run .NET client and server applications on Linux, Solaris, Mac OS X, Windows, and Unix. The Mono team has indicated that it will soon have a Silverlight implementation, currently called the Moonlight runtime. With this addition, developers will be able to develop Silverlight applications for Windows, Macintosh, and Linux systems with one code base. Furthermore, the user experience will be identical, no matter which platform you are using.

Cross-Platform Version of the .NET Framework

Silverlight 1.0 was released by Microsoft in the summer of 2007, but this version supported only Ecma languages that are interpreted in the client. And although Silverlight 1.0 works well for developers who are already familiar with client-side scripting, many developers have their eyes on the second release of Silverlight, version 2. Silverlight 1.0 is more or less in direct competition with Flash—some have called it Microsoft's "Flash killer." However, things really get exciting with Silverlight 2.

Silverlight 2 contains its own cross-platform version of the .NET Framework, which means it has its own version of the common language runtime (CLR), the full type system, and a .NET Framework programming library that you can use in Visual Studio 2008 to build rich user experiences in the browser.

XAML, a Text-Based Markup Language

Another advantage to Silverlight is that its foundation is based on a text-based markup language. For other RIA solutions such as Flash, the base is a compiled file. This is not nearly as friendly to developers as a text-based format, for obvious reasons.

XAML is very easy to write and modify. As an example, let's say you want to change the opacity of an object. If you were using Flash to do this, you would need to open the Flash project file, find the right layer and object, and then make the adjustment there. You then would need to recompile and republish the file. In contrast, with Silverlight, you simply open the XAML file, change the opacity property of the object, and save the file.

Another advantage of XAML is that it can be created dynamically at runtime. If you think about it, the implications of this are huge. Consider the similarities between HTML and XAML. Both are text-based markup languages that have a decent similarity to XML. HTML is the base foundation of files published on the Internet. Since HTML was introduced, a number of technologies have been built on top of it. In the Microsoft camp, for example, Active Server Pages (ASP) was first introduced to allow developers to dynamically modify HTML at runtime. Today, we have ASP.NET. XAML has the same potential, since it is a text-based markup language on which developers can expand.

Use of Familiar Technologies

Microsoft is very good at creating tools that make application development easy. The Visual Studio integrated development environment (IDE) has been around for quite some time, and although new features are continually added to the tool, the environment itself has remained remarkably consistent.

Silverlight development is no different. At the core of developing Silverlight 2 applications is Visual Studio 2008, the latest version in Visual Studio's long history. This gives Silverlight a distinct advantage, as developers do not need to learn how to use a new development environment.

In addition to Visual Studio, Microsoft has released a suite of tools called Expression Studio. Included in this suite is Microsoft Expression Blend, which is used to edit and create XAML for Silverlight applications. While Expression Blend looks completely different, it still has many of the same elements as Visual Studio. In addition, Expression Blend 2 works off of the same project as Visual Studio. This means that as you make changes in each of the editors—opening a project in Visual Studio, and then opening the same project in Expression Blend to edit the XAML—the edited files will request to be refreshed when opened again in the other tool.

Small Runtime and Simple Deployment

Since Silverlight requires that a client runtime be installed on the client machine, it is vital that this runtime has a small footprint and downloads quickly. Microsoft worked very hard to get the installation size as small as possible. The developers clearly succeeded with Silverlight 1.0, as the download size is a tiny 1MB. For Silverlight 2, however, they had a harder chore ahead of them, since Silverlight 2 contains its own .NET Framework and object library. Microsoft went to each .NET Framework team and allocated it a size to fit its portion. The end result is astonishing—Silverlight 2 is approximately 4MB in size.

As for pushing the Silverlight runtime out to clients, Microsoft has provided a very easy detection mechanism. If the client does not have the proper Silverlight runtime installed, it will display a logo, as shown in Figure 1-5.

Figure 1-5. *Silverlight runtime required logo*

When users click the icon in the logo, they are taken to a web page that walks them through the process of installing the Silverlight runtime. Once the runtime is finished installing, the Silverlight application is immediately available to the user, as shown in the example in Figure 1-6.

Figure 1-6. *Silverlight application after installation of runtime*

The Silverlight Development Environment

In the past, setting up an environment to work with Microsoft's latest and greatest has been relatively straightforward, typically involving only the setup of the latest version of Visual Studio and the appropriate software development kit. However, with Silverlight, the situation is quite a bit different due to the introduction of many new tools. Let's take a look at these new tools:

Silverlight 2 Runtime: This is the Silverlight client runtime, which is required on every computer that wishes to view a Silverlight-enabled web application.

Silverlight 2 Software Development Kit (SDK): This SDK is a collection of samples, Silverlight QuickStarts, documentation, and controls that are used to develop Silverlight applications. This SDK is not required, but it is recommended that all Silverlight developers download it.

Visual Studio 2008: As noted, this is the latest version of Microsoft's IDE, the successor to Visual Studio 2005 (see Figure 1-7). Installing Visual Studio 2008 also automatically installs Microsoft .NET Framework 3.5. There are many new features in Visual Studio 2008 that make it a highly recommended upgrade in general; for serious Silverlight developers, Visual Studio 2008 is a must. Chapter 2 covers Visual Studio 2008 in more depth.

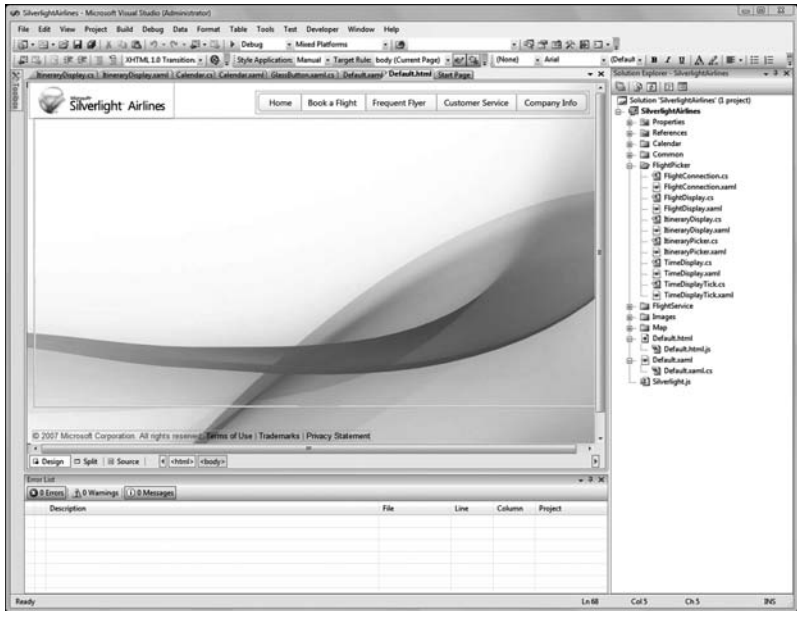

Figure 1-7. *Microsoft Visual Studio 2008*

Silverlight Tools for Visual Studio 2008: This is an add-on for Visual Studio that provides a Silverlight project system for developing Silverlight applications using C# or Visual Basic. This add-on is required if you wish to take advantage of Visual Studio to build Silverlight applications. The project system includes the following components:

- Visual Basic and C# project templates
- IntelliSense and code generators for XAML
- Debugging of Silverlight applications
- Web reference support
- Integration with Expression Blend

Microsoft Expression Blend 2: This is a "what you see is what you get" (WYSIWYG) editor for XAML (see Figure 1-8). Expression Blend is similar to Adobe's Flash MX product. It allows you to lay out the XAML canvas, add timelines, and create transformations—all in a very user-friendly and visual way. Expression Blend 2 is covered in Chapter 8.

Figure 1-8. *Microsoft Expression Blend 2*

Summary

In this chapter, we looked at the evolution of user interfaces in applications, as well as the history of RIAs. I then introduced Silverlight, and talked about the benefits it brings to developers today and how it fits into RIA solutions. Finally, you learned about the tools involved in developing Silverlight-enabled applications.

Now it is time to get your hands dirty and start building some Silverlight applications! In the next chapter, I will provide an introduction to Microsoft Visual Studio 2008, one of the primary tools used to build Silverlight applications.

CHAPTER 2

Introduction to Visual Studio 2008

The previous chapter mentioned the tools required to develop RIAs that utilize the Silverlight technology. At the core of all of these tools is Microsoft's flagship development product, Visual Studio. This chapter provides an introduction to the latest version, Visual Studio 2008. You will learn about some of the new features that are particularly helpful for developers building RIAs with Silverlight, and then work through an exercise to try out Visual Studio 2008's enhanced JavaScript IntelliSense and debugging support. Finally, you will have an opportunity to create your first Silverlight application using Visual Studio 2008. Let's get started with a brief introduction to the Visual Studio IDE.

Just What Is Visual Studio?

Any developer who has developed applications using technologies related to Microsoft's Visual Basic, ASP, or .NET has used some version of Visual Studio on a regular basis. This is because Visual Studio is Microsoft's primary development product. Whether you are developing desktop applications, web applications, mobile applications, web services, or just about any other .NET solution, Visual Studio is the environment you will be using.

Visual Studio is an IDE that allows .NET developers to implement a variety of .NET solutions within the confines of one editor. An IDE is a software application that contains comprehensive facilities to aid developers in building applications. Visual Studio fits this description for a number of reasons. First, Visual Studio offers a very rich code-editing solution. It includes features such as source code color-coding and code completion. Second, it offers an integrated debugger, which allows you to place breakpoints in your source code to stop execution at any given point, as well as step through the source line by line, analyzing the state of objects and fields at any given point in the execution. Add to these features rich support for application deployment, installation, and integration with database services, and you can understand how Visual Studio is an extremely valuable tool for developers.

■ **Note** This book assumes a basic understanding of Visual Studio. If you're new to Visual Studio, I recommend that you get started with a book devoted to the subject, such as *Beginning C# 2008, Second Edition* by Christian Gross (Apress, 2008).

THE HISTORY OF VISUAL STUDIO

Visual Studio has quite a history. The first version was called Visual Studio 97, which was most commonly known for Visual Basic 5.0. In 1998, Microsoft released Visual Studio 6.0. That version included Visual Basic 6.0, as well as Microsoft's first web-based development tool, Visual InterDev 1.0, which was used to develop ASP applications.

Next came the introduction of Microsoft .NET and ASP.NET 1.0, prompting Visual Studio.NET. As Microsoft was enhancing and releasing new versions of Microsoft .NET and ASP.NET, it also continued enhancing Visual Studio by releasing Visual Studio 2003 and then Visual Studio 2005. In addition, Microsoft has introduced a line of free development tools known as the Visual Studio Express tools, as well as the Visual Studio Team System, which can be used by large programming teams to build enterprise-level systems.

This brings us to the latest version of Visual Studio, which Microsoft developed under the code name Orcas and has now dubbed Visual Studio 2008.

What's New in Visual Studio 2008?

Microsoft has introduced a variety of new features in Visual Studio 2008, many of which are geared toward helping developers build RIAs with Silverlight and related Microsoft technologies, such as the Windows Communication Foundation (WCF), ADO.NET Data Services, and Ajax. Here we will look at some of the new features in Visual Studio 2008 that are particularly helpful to Silverlight application developers.

JavaScript IntelliSense and Debugging

Client-side scripting is a major component of developing RIAs. With the adoption of technologies like Ajax and Silverlight, developers can integrate client-side scripting into applications to enhance the user experience.

In response to the growing necessity for integrating client-side scripting into ASP.NET applications, Microsoft has implemented an extensive upgrade to Visual Studio's JavaScript IntelliSense and debugging support. Here, we'll look at the IntelliSense and debugging improvements, and then try a test run to see them in action.

IntelliSense Improvements

The first major improvement of JavaScript IntelliSense in Visual Studio 2008 is type inference. Since JavaScript is a dynamic language, a variable can be one of many different types, depending on its current state. For example, in the following code snippet, the variable x represents a different type each time it is assigned.

```
function TypeInference()
{
    var x;
    x = document.getElementById("fieldName");
    // x is now an HTML element
    alert(x.tagName);
    x = 10;
    // x is now an integer
    alert(x.toFixed());
    x = new Date();
    // x is now a date
    alert(x.getDay());
}
```

In this example, the variable x represents three different types during the execution of the function:

- First, it represents an HTML element. When the user types x followed by a period, the code-completion choices will be specific to an HTML element, as shown in Figure 2-1.

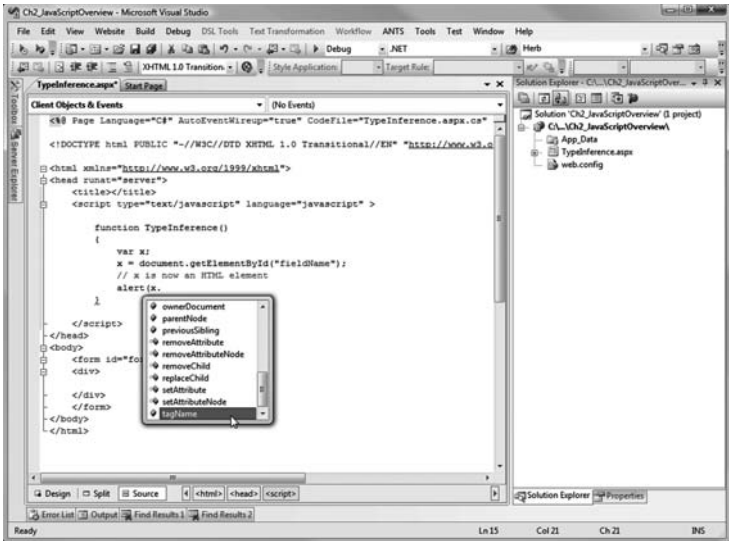

Figure 2-1. *Code completion with type inference for an HTML element*

- In the next line, x is assigned to the value 10. At this point, x has become an integer, and the code-completion choices that appear are specific to an integer, as shown in Figure 2-2.

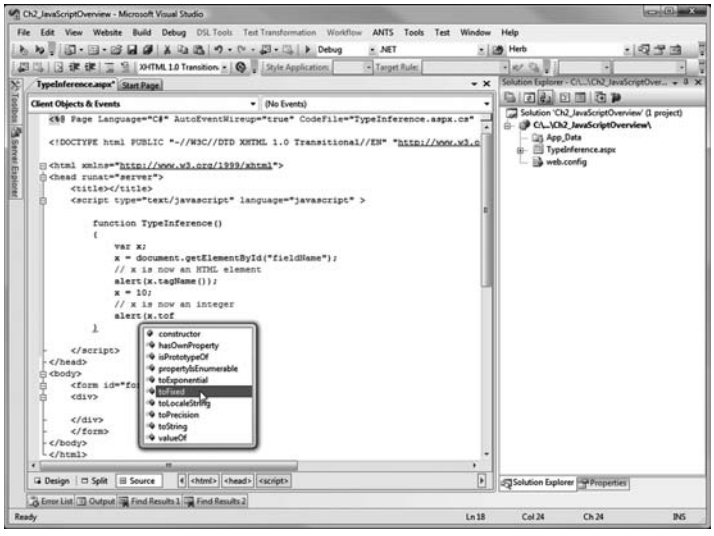

Figure 2-2. *Code completion with type inference for an integer*

- Finally, x is assigned to a date type. At this point, x represents a date type, and the code-completion choices include date-specific properties and methods.

The second notable enhancement to JavaScript IntelliSense in Visual Studio 2008 is the support for IntelliSense in external script files. In fact, there are many levels to this enhancement. First, developers will have IntelliSense while they are editing the external script files. Second, by adding a reference to other external script files, developers can get IntelliSense for functions and fields from other script files. Finally, developers will receive IntelliSense in the actual pages that reference the external script files.

Another new feature of JavaScript IntelliSense is the ability to add XML comments to your code, which will provide additional information in the IntelliSense display. These are similar to standard C# XML comments, which have been available in C# since it was initially released. The following example shows some XML comments added to a JavaScript function.

```
function HelloWorld(FirstName, LastName)
{
    /// <summary>Returns a hello message to the given name</summary>
    /// <param name="FirstName">Person's First Name</param>
    /// <param name="LastName">Person's Last Name</param>
    /// <returns>string</return>
    return ("Hello " + FirstName + " " + LastName);
}
```

This is a function called `HelloWorld`, which simply accepts a first and last name and returns a hello message customized for that person. This function is located in a file called `JScripts.js`. Notice the four XML comments added to the start of the function. These provide a summary of the function, give a description of the function's parameters, and indicate the value returned by the function. With these extra lines in place, when you add the function in your code, IntelliSense will now display this additional information. First, when you start typing `HelloWorld`, Visual Studio's JavaScript IntelliSense will help you complete the method call. After you have typed `HelloWorld` and the opening parenthesis, it will display the two parameters and their descriptions, as shown in Figure 2-3.

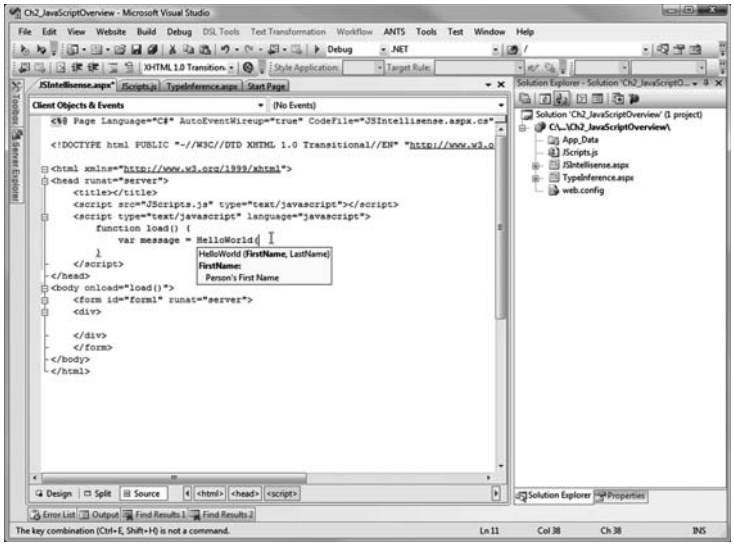

Figure 2-3. *IntelliSense for a JavaScript function with parameter tags*

Now that we have reviewed the JavaScript IntelliSense features added to Visual Studio 2008, let's take a look at the new JavaScript debugging features, which are equally as useful and long-awaited.

New Debugging Features

In previous versions of Visual Studio, ASP.NET developers were severely limited in the debugging they could do in client-side scripting. Some of the more industrious developers would find a third-party JavaScript debugging tool to assist them. However, the majority of developers would simply use hacks, such as adding alerts throughout their client-side scripting. When an alert was not hit, they could identify where the error had occurred and at least determine the basic location where attention was required.

In Visual Studio 2008, JavaScript debugging is now integrated directly into the IDE, and believe it or not, it actually works!

Figure 2-4 shows an example where a breakpoint was placed on a line of code in a local script section of an ASP.NET page. At this point, you are in Visual Studio's JavaScript debugger, and you can step through the code one line at a time. If a line of code references a function in an external script file (as is in the example), that script file will be opened, and you will be able to debug that script file as well. In addition, you can hover the mouse over code and see the current value of the objects while you are debugging your application.

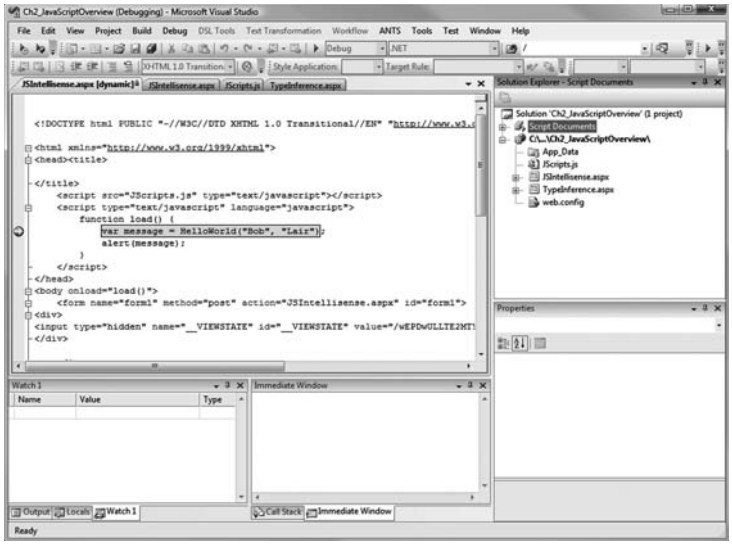

Figure 2-4. *JavaScript debugging in Visual Studio 2008*

And as if that were not enough, Visual Studio's JavaScript debugging also allows you to use the Immediate window to enter JavaScript code directly while you are debugging. This is extremely powerful, as it allows you to evaluate a line of code at any point in the process; your entries will be processed immediately.

To get started debugging JavaScript in Visual Studio, there is only one setting that you need to confirm within your browser to make certain that client-side debugging is enabled. In Internet Explorer, choose View ➤ Internet Options. This will display the Internet Options dialog box. Select the Advanced tab and find the two entries "Disable script debugging (Internet Explorer)" and "Disable script debugging (Other)." Make certain both of these options are *unchecked*, as shown in Figure 2-5, and click the OK button to close the dialog box.

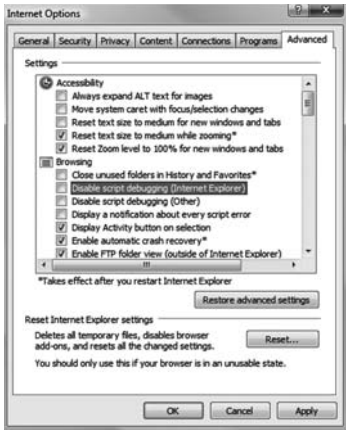

Figure 2-5. *Uncheck the "Disable script debugging" boxes in the Internet Explorer Internet Options dialog box.*

Try It Out: JavaScript IntelliSense and Debugging

Now that we have looked at some of the new JavaScript IntelliSense and debugging features in Visual Studio 2008, let's take them for a test drive.

1. Start Visual Studio 2008 and select File ➤ New ➤ Project from the main menu, as shown in Figure 2-6.

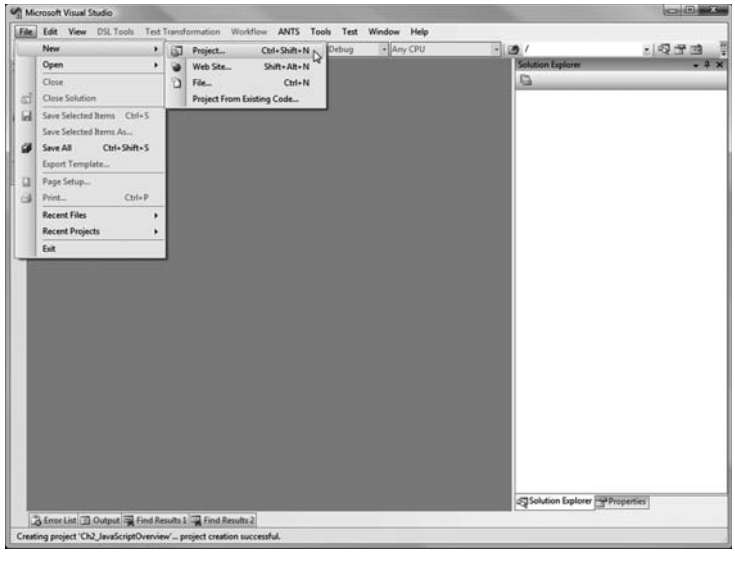

Figure 2-6. *Selecting to create a new project*

2. In the New Project dialog box, select Visual C# as the project type and ASP.NET Web Application as the template. Name the project `Ch2_JavaScriptOverview`, as shown in Figure 2-7.

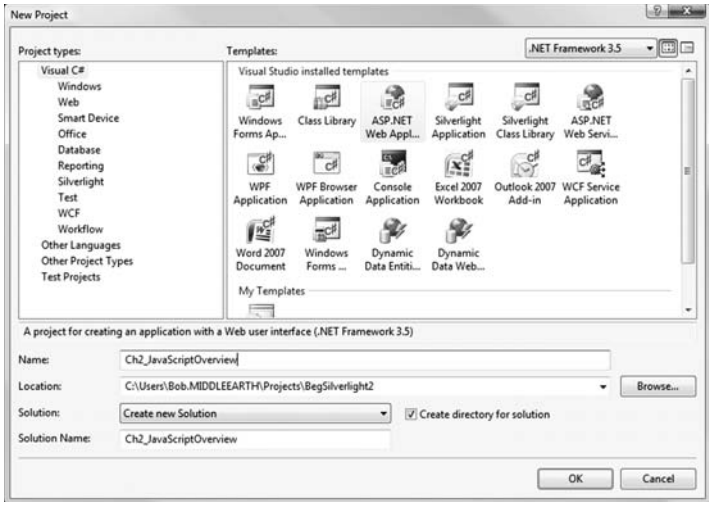

Figure 2-7. *Selecting to create an ASP.NET Web Application project*

3. A new Web Application project will now be created for you, with the `Default.aspx` file open. Select Project ➤ Add New Item from the main menu.

4. In the Add New Item dialog box, make sure that the Visual C# category is selected on the left and select JScript File in the Templates pane. Name the file `HelloWorld.js`, as shown in Figure 2-8. Then click the Add button.

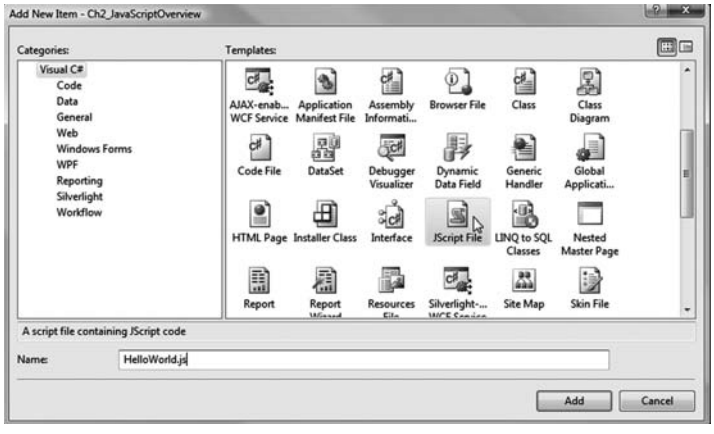

Figure 2-8. *Adding a JavaScript file to a project*

5. The JavaScript file will be added to the project and opened by default. In this file, add a new function called `HelloWorld()`, as follows:

```
function HelloWorld(FirstName, LastName)
{
    return ("Hello " + FirstName + " " + LastName);
}
```

As you typed the function, you got some IntelliSense assistance. Also notice the color-coding of the JavaScript.

6. Now insert some XML comments to display some additional IntelliSense information when the function is used. Add the following comments (shown in bold):

```
function HelloWorld(FirstName, LastName)
{
    /// <summary>Returns a hello message to the given name</summary>
    /// <param name="FirstName">Person's First Name</param>
    /// <param name="LastName">Person's Last Name</param>
    /// <returns>string</return>
    return ("Hello " + FirstName + " " + LastName);
}
```

7. Once again, select Project ➤ Add New Item. This time, select Web Form as the template and name the file `JSIntellisense.aspx`.

8. In this new file, add a script reference to your `HelloWorld.js` script file. You can either drag the script file to the page header or simply edit the HTML of the form manually so that it appears as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title></title>
    <script src="HelloWorld.js" type="text/javascript"></script>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    </div>
    </form>
</body>
</html>
```

9. Next, add a local function that will run when the page loads. To do this, add a new `<SCRIPT>` section and call the function in the page body's `onload` event so that the method is called when the page is loaded, as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    <script src="HelloWorld.js" type="text/javascript"></script>
    <script type="text/javascript" language="javascript">
        function load()
        {

        }
    </script>
</head>
<body onload="load()">
    <form id="form1" runat="server">
    <div>

    </div>
    </form>
</body>
</html>
```

10. Now call the `HelloWorld()` method. Go ahead and start typing the boldfaced line of code in the `load` function:

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    <script src="HelloWorld.js" type="text/javascript"></script>
    <script type="text/javascript" language="javascript">
        function load()
        {
            var message = HelloWorld("Bob", "Lair");
            alert(message);
        }
    </script>
</head>
```

```
<body onload="load()">
    <form id="form1" runat="server">
    <div>

    </div>
    </form>
</body>
</html>
```

You will see that Visual Studio's IntelliSense tries to help you, as shown in Figure 2-9. With `HelloWorld` selected in the IntelliSense box, you can simply press the Tab key, and Visual Studio will automatically finish the function name. As you continue typing, you will also notice that the XML comments you added for the function appear (see Figure 2-3 earlier in the chapter).

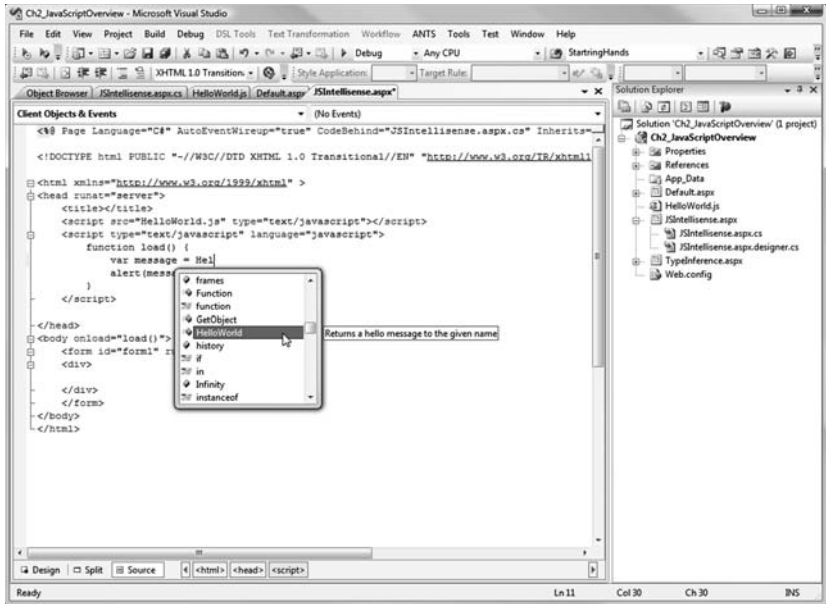

Figure 2-9. *HelloWorld appears in the JavaScript IntelliSense box.*

11. When you are finished, press F5 to start the project. If you are prompted with a Debugging Not Enabled dialog box, choose "Modify the Web.config file to enable debugging," as shown in Figure 2-10, and then click OK to continue.

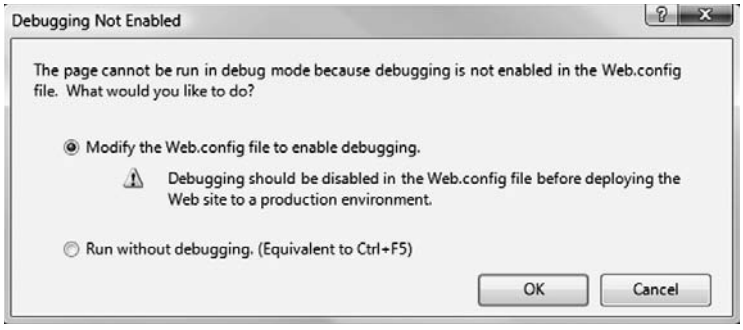

Figure 2-10. *VisualStudio will display this dialog box if debugging is not enabled.*

12. When the page is loaded, you will see an alert box appear with your message, as shown in Figure 2-11. Click OK to close the alert box.

Figure 2-11. *Customized hello message*

13. Next, let's give JavaScript debugging a try. Stop the project and return to your Visual Studio project.

14. In the `JSIntellisense.aspx` file, add a breakpoint by clicking in the gray area to the left of the line calling the `HelloWorld()` function. In design mode, the breakpoint will show up as a red dot with a white diamond, as shown in Figure 2-12.

15. Press F5 to restart the project. Visual Studio will appear in debug mode, with execution stopped on your line with the breakpoint. The breakpoint will show up as a red dot with a yellow arrow, indicating the application process has been halted at the breakpoint, as shown in Figure 2-13.

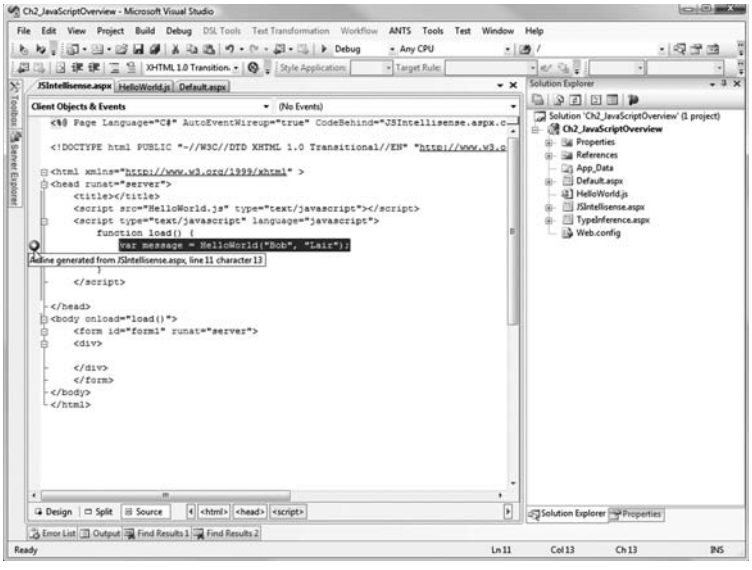

Figure 2-12. *Adding a breakpoint*

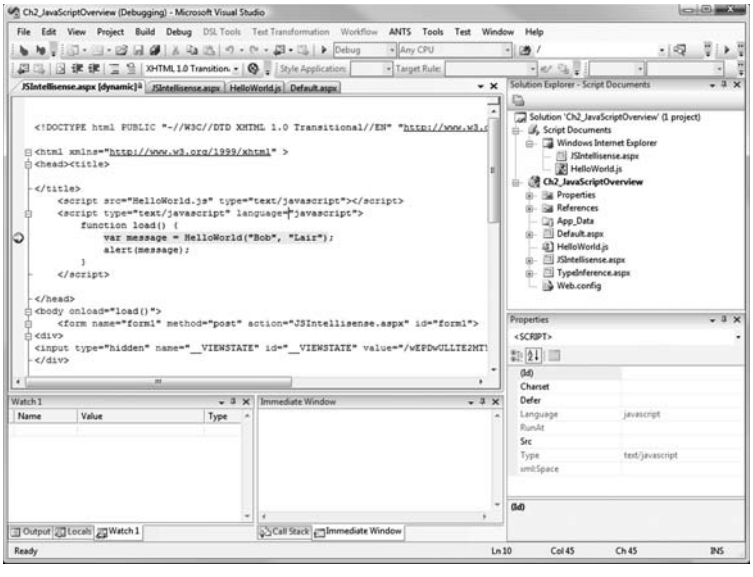

Figure 2-13. *Debugging stopped at the inserted breakpoint.*

16. Press F10 to step to the next line. If you hover your mouse over the variable message, you will see its value is currently set to `"Hello Bob Lair"`. You can also see the value of message in the Locals window.

17. Let's change the value of message. In the Immediate window, type in the following line of code and press Enter to execute it.

```
message = HelloWorld("Robert", "Lair")
```

The Immediate window will change the value of message to the output of the new call to the `HelloWorld` method, as shown in Figure 2-14.

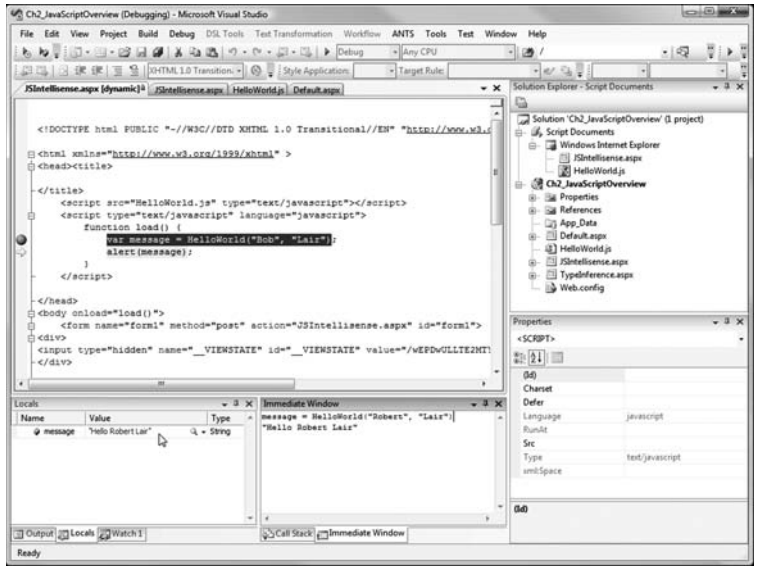

Figure 2-14. *Using the Immediate window to change a value*

This example gave you an idea of the new JavaScript IntelliSense and debugging features in Visual Studio 2008, which are far more advanced than anything ASP.NET developers have had with previous versions. These should prove to be very valuable tools in your client-side scripting tool belt.

Now, let's continue looking at other new features in the latest version of Visual Studio.

Multi-Targeting Support

My company builds ASP.NET solutions for clients, and each time a new version of the .NET Framework is released, we face a maintenance problem. Naturally, we would like to take advantage of the new features of Visual Studio and the latest .NET Framework in our new projects, but we must also be able to support the existing client base.

In the past versions of Visual Studio, projects were tied to a specific version of the .NET Framework. For example, applications written in ASP.NET 1.0 needed to be upgraded to ASP.NET 1.1 in order to take advantage of Visual Studio 2005.

An associated problem is how to handle existing systems that you only want to maintain, and have no intention of upgrading to a newer .NET Framework. For developers to support such systems, while still taking advantage of newer Visual Studio features for other projects, they would need to run different versions of Visual Studio side by side. From a personal perspective, my worst situation was when I had Visual Studio 6.0, Visual Studio .NET (2002), Visual Studio 2003, and Visual Studio 2005 installed on my laptop at the same time. What a pain!

Microsoft has helped alleviate this problem by adding *multi-targeting support* to Visual Studio 2008. This allows you to use Visual Studio 2008 for a specific targeted version of the .NET Framework. So, your Visual Studio 2005 projects that are using .NET 2.0 or .NET 3.0 can be edited with Visual Studio 2008, without being forced to upgrade to .NET 3.5. In addition, you can create new projects for a targeted platform. When you create a new project in Visual Studio, you will notice a new drop-down menu at the top-right corner of the New Project dialog box. As shown in Figure 2-15, this lists the different .NET Frameworks. If you change the selection here, the new project will be targeted to that version of the .NET Framework.

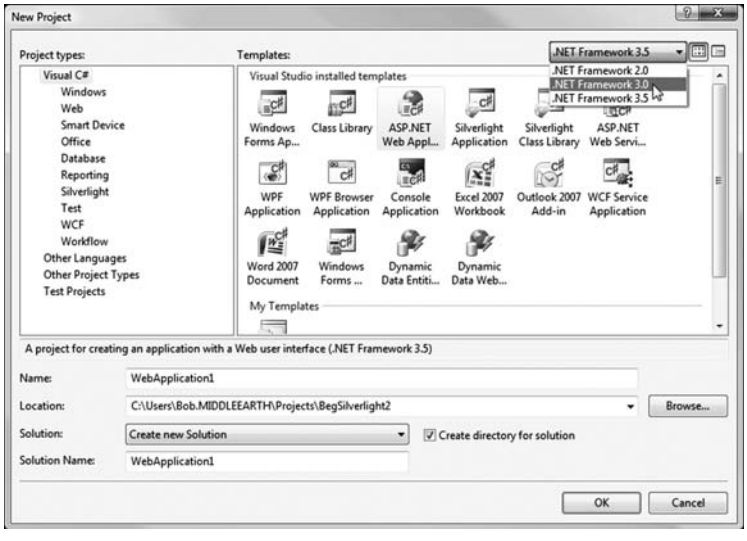

Figure 2-15. *Muti-targeting support in Visual Studio 2008*

If you open a Visual Studio 2005 project in Visual Studio 2008, you will be prompted to upgrade the project by default. If you choose not to upgrade the project, the project will be opened as a Visual Studio 2005 project within Visual Studio 2008.

■ **Note** If you open a project using a version of the .NET Framework prior to 2.0, you will be forced to upgrade. There is no support for these earlier versions in Visual Studio's 2008's multi-targeting feature. That said, Microsoft is committed to keeping this feature working for future versions of Visual Studio. So, it seems safe to say that developers will need only the latest version of Visual Studio installed from this point forward.

Transparent IntelliSense Mode

One of the problems with IntelliSense in past versions of Visual Studio was that the pop-up window hid the source code. You would need to close the pop-up window to see the source code beneath it, and then start typing again.

A new feature in Visual Studio 2008 is the semitransparent IntelliSense pop-up window. When the IntelliSense window appears, you can press the Ctrl key to make the pop-up window semitransparent, allowing you to see the source code under the window. Figures 2-16 and 2-17 illustrate this feature.

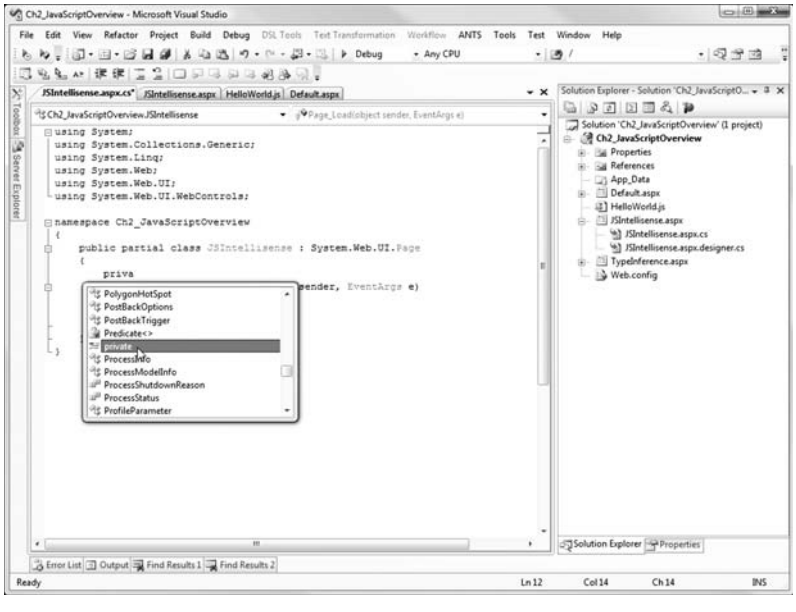

Figure 2-16. *Default IntelliSense pop-up window*

This feature works in all languages across Visual Studio, including the JavaScript IntelliSense covered earlier in this chapter.

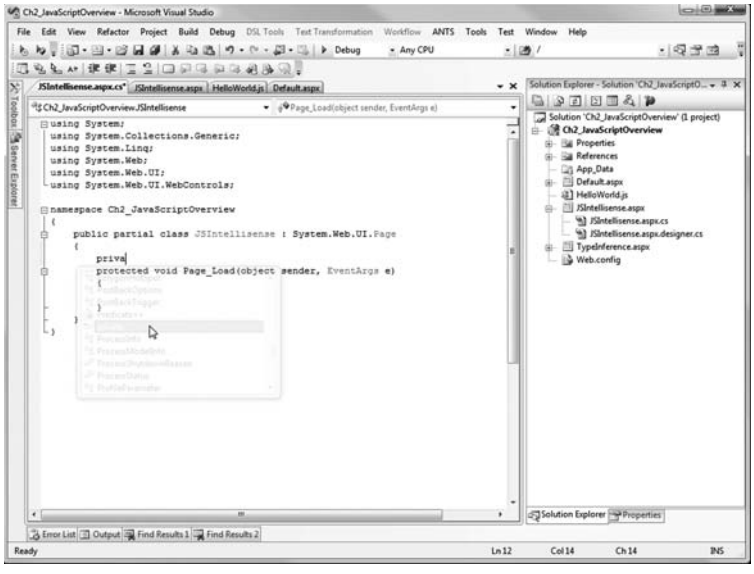

Figure 2-17. *Press the Ctrl key to make the IntelliSense pop-up window transparent.*

Building Your First Silverlight Application in Visual Studio

The best way to explore the Visual Studio IDE is to get your hands dirty and play around with it. So, it's time to build a Silverlight application.

Try It Out: Hello World in Silverlight 2

In this exercise, you'll build the Hello World Silverlight 2 application. I personally hate the Hello World sample, but it is used often because it is so simple and provides a good introduction. And who am I to break with tradition? Let's get started.

1. Start Visual Studio 2008 and Select File ➤ New ➤ Project from the main menu.

2. In the New Project dialog box, select Visual C# as the project type, and in the list under that type, choose Silverlight. Select Silverlight Application as the template and name the project `Ch2_HelloWorld`, as shown in Figure 2-18. Then click OK.

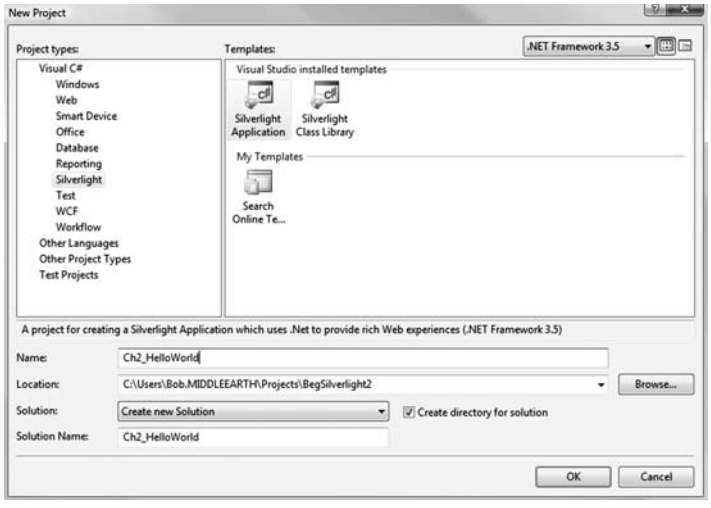

Figure 2-18. *Creating a new Silverlight project*

3. Visual Studio will display the Add Silverlight Application dialog box, informing you that your Silverlight application needs to be hosted in an HTML web page. It offers the choices of hosting the Silverlight application in a web site or within a project. For this exercise, select Web Site and stick with the default name of `Ch2_HelloWorldWeb`, as shown in Figure 2-19. Then click OK. See the next section for more information about choosing whether to use a Web Site or Web Application project for your own Silverlight applications.

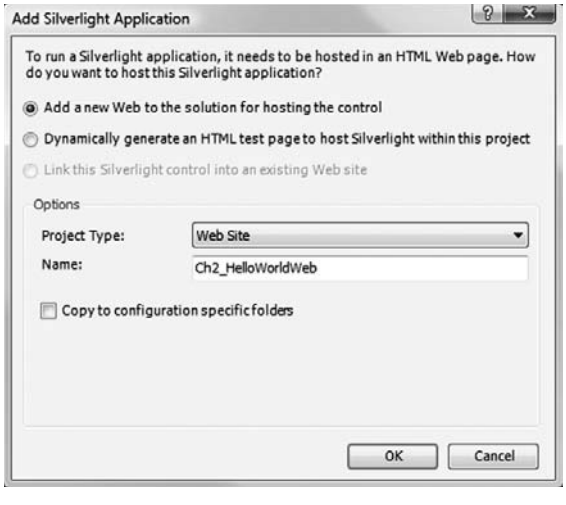

Figure 2-19. *The Add Silverlight Application dialog box*

Visual Studio will now create the base project for you. Notice that there are two projects created within your solution: one called `Ch2_HelloWorldWeb` and one called `Ch2_HelloWorld`, as shown in Figured 2-20.

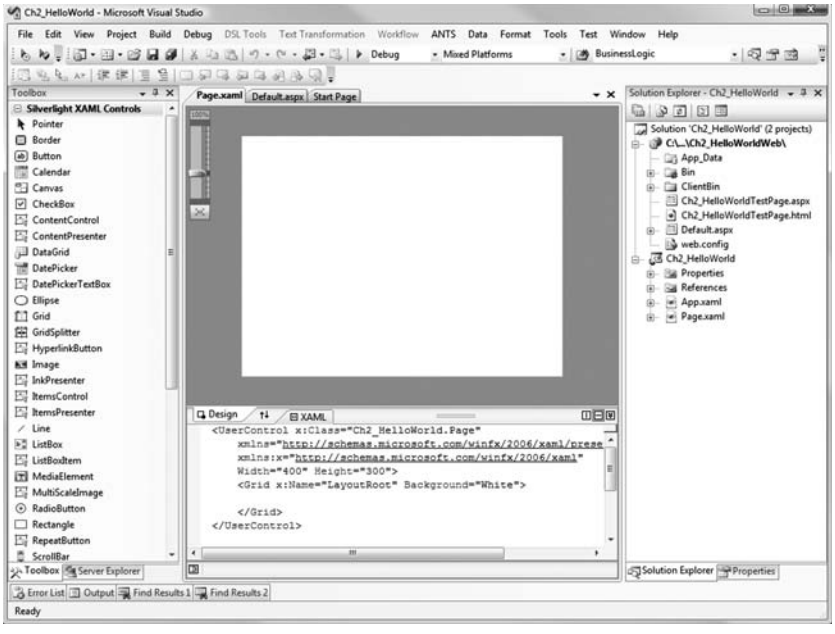

Figure 2-20. *The default Silverlight project created in Visual Studio 2008*

4. Visual Studio has already opened the `Page.xaml` file, which is where you will start working. Let's begin by adding a `TextBlock` control, which will display our "Hello World!" message. Add the `TextBlock` within your `Canvas` object, as follows:

```
<UserControl x:Class="Ch2_HelloWorld.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">
      <TextBlock x:Name="HelloMessage" Text="Hello World!" FontSize="30" />
    </Grid>
</UserControl>
```

5. Save the project and run it by pressing F5. If you see the Debugging Not Enabled dialog box (as shown in Figure 2-10, earlier in this chapter), select "Modify the Web.config to enable debugging" and click OK. The result should be as shown in Figure 2-21.

Figure 2-21. *Your first Silverlight application in Visual Studio 2008*

6. I know this isn't very interesting, so let's change things up a bit by setting the display message in the `Page.xaml.cs` code behind. In the code behind, you will notice a constructor for your `Page` class, which contains one method called `InitializeComponent()`. Under that method, change the `Text` property of your `TextBlock` as follows (the line shown in bold):

```
namespace Ch2_HelloWorld
{
    public partial class Page : UserControl
    {
        public Page()
        {
            InitializeComponent();
            this.HelloMessage.Text = "Hello Universe!";
        }
    }
}
```

7. Rebuild the application and run it again. Your result should look like Figure 2-22.

Figure 2-22. *The final result from our first Silverlight Application in Visual Studio 2008*

8. Close the application.

And there you go! You have built your first Silverlight application. Of course, this application is extremely simple, but you did get an idea of how things work in Visual Studio 2008.

Hosting Your Silverlight Application: Web Site or Web Application?

In Visual Studio 2008, should you use a Web Site project or a Web Application project to host your Silverlight application? The main difference between a Web Site and a Web Application project is how the files are compiled and deployed. Each has its advantages and disadvantages. In the end, the choice pretty much comes down to user preference. Let's take a quick look at each approach.

Using a Visual Studio Web Site

A Visual Studio Web Site is nothing more than a group of files and folders in a folder. There is no project file; instead, the site simply contains all the files under the specific folder, including all text files, images, and other file types.

A Visual Studio Web Site is compiled dynamically at runtime. An assembly will not be created, and you won't have a `bin` directory.

Here are some advantages of using a Visual Studio Web Site:

- You don't need a project file or virtual directory for the site.
- The site can easily be deployed or shared by simply copying the folder containing the site.

And here are some disadvantages of this approach:

- There is no project file that you can double-click to open the site in Visual Studio. Rather, you must browse to the folder after opening Visual Studio.
- By default, all files within the site's directory are included in the Web Site project. If there are files within the site's directory that you do not wish to be a part of the web site, you must rename the file, adding the extension `.exclude`.

Using a Visual Studio Web Application Project

A Visual Studio Web Application project is the more traditional type of web project used prior to Visual Studio 2005. When Microsoft developers introduced the "Web Site" concept, they did not take into account the many developers who were comfortable with the project-based solution approach. To accommodate those developers, Microsoft announced the Visual Studio 2005 Web Application project as an add-on to Visual Studio 2005. In Visual Studio 2008, this project type is once again a part of Visual Studio.

The following are some of the advantages of using a Web Application project:

- All of the code files are compiled into a single assembly, placed in the `bin` directory.
- You can easily exclude files from a project, since all files within the project are defined within the project file.
- It's easier to migrate from older versions of Visual Studio.

A disadvantage is that it can be more difficult to share your solution with others, if that is your intent.

In the end, both approaches have their pros and cons. You need to determine which one is more suitable for your application, depending on your specific purpose and goals. For more information about these project types, refer to the MSDN documentation.

Summary

This chapter introduced Visual Studio 2008 and some of the new features offered in this version, including the new JavaScript IntelliSense features, additional JavaScript debugging support, and multi-targeting support. In addition, you built your very first Silverlight application.

In the next chapter, we are going to start to dive into some of the Silverlight controls, beginning with the layout management controls. These controls enable you to lay out your Silverlight applications.

CHAPTER 3

Layout Management in Silverlight

The previous chapter provided an overview of Visual Studio 2008, one of the primary tools used in developing Silverlight applications. In this chapter, we are going to start to dive into some Silverlight 2 development by looking at the layout management controls.

As you have learned, Silverlight applications consist of a number of Silverlight objects that are defined by XAML. Layout management involves describing the way that these objects are arranged in your application. Silverlight 2 includes three layout management controls: `Canvas`, `StackPanel`, and `Grid`. We will take a look at each of these in depth. By the end of this chapter, you should have a good understanding of when to use which layout control.

Layout Management

Silverlight provides a very flexible layout management system that lets you specify how controls will appear in your Silverlight application. You can use a static layout as well as a liquid layout that allows your layout to automatically adjust as your Silverlight application is resized in the browser.

Each of the three layout controls provided in Silverlight 2 has its advantages and disadvantages, as summarized in Table 3-1.

Let's begin by looking at the most basic layout control: the `Canvas` panel.

Table 3-1. *Layout Control Pros and Cons*

Control	Description	Pros	Cons
`Canvas`	Based on absolute position of controls	Very simple layout.	Requires that every control have a `Canvas.Top` and `Canvas.Left` property attached to define its position on the canvas.
`StackPanel`	Based on horizontal or vertical "stacks" of controls	Allows for a quick dynamic layout. Nesting `StackPanel` controls can provide some interesting layouts.	The layout is limited to stacks of items. Spacing is limited to adding margins to the individual controls and to adjusting the alignment (with the `VerticalAlignment` and `HorizontalAlignment` properties).
`Grid`	Mimics using table elements in HTML to lay out controls	The most flexible and powerful layout control. You can define just about any type of layout using the `Grid` control.	Grid definitions can get somewhat complex at times. Nesting `Grid` components can be confusing.

The Canvas Panel

The `Canvas` panel is a basic layout control that allows you to position Silverlight objects using explicit coordinates relative to the canvas location. You can position an object within the `Canvas` panel by using two XAML attached properties: `Canvas.Left` and `Canvas.Top`. Figure 3-1 shows how the object's position is affected by these properties.

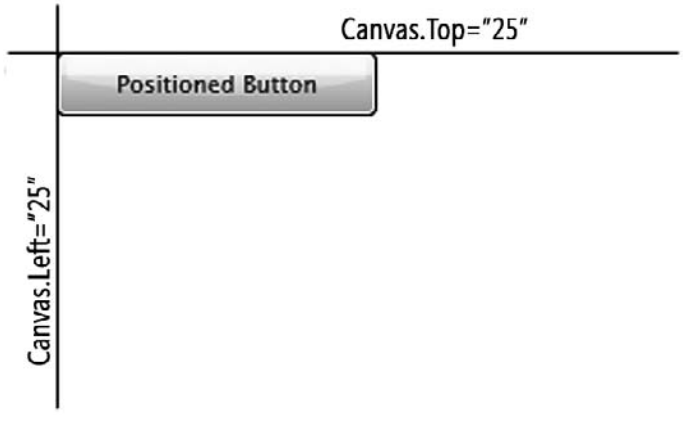

Figure 3-1. *The XML attached properties Canvas.Top and Canvas.Left allow you to position the Canvas.*

The objects within a Canvas panel have no layout policies placed on them by the layout control and will not resize automatically when your application is resized within the browser.

Try It Out: Using the Canvas Panel

Let's try out a quick example of using the Canvas panel.

1. Open Visual Studio 2008 and create a new Silverlight application called Ch3_CanvasPanel. Allow Visual Studio to create a Web Site project to host the application.

2. When the project is created, you should be looking at the Page.xaml file. If you do not see the XAML source, switch to that view so you can edit the XAML. Within the main Grid element, add a Canvas element. Assign it a Width property of 300 and a Height property of 300. In order to see the Canvas panel in the application, also set the background color to green. The following XAML adds this Canvas:

```
<UserControl x:Class="Ch3_CanvasPanel.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">

        <Canvas Background="Green" Width="300" Height="200">
        </Canvas>

    </Grid>
</UserControl>
```

   At this point, your Silverlight application doesn't look that exciting. It contains only a single green rectangle positioned at the very center of your application, as shown in Figure 3-2.

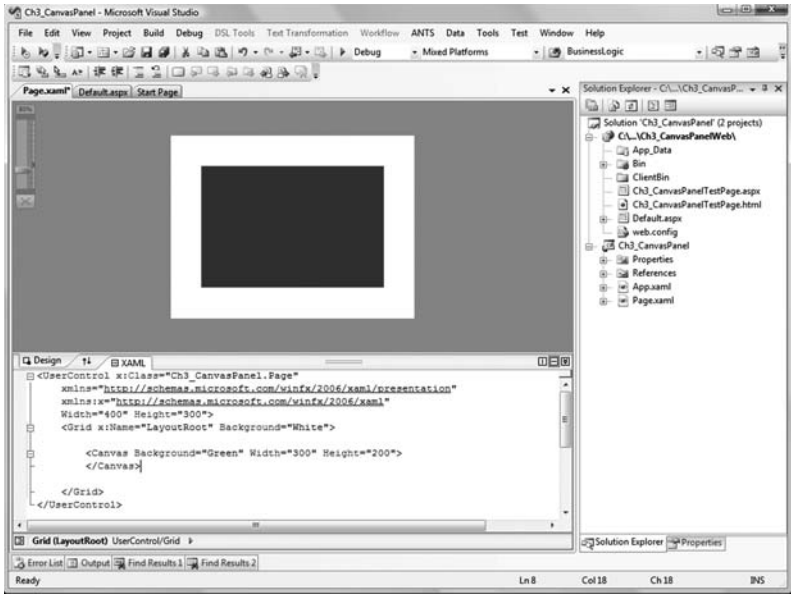

Figure 3-2. *Default Canvas with a green background*

3. Let's add a button to this `Canvas` panel. Add the following code to place the button, which has the label `Button1`, with a `Width` property of `100` and a `Height` property of `30`. (The `Button` control is covered in detail in Chapter 4.)

```
<UserControl x:Class="Ch3_CanvasPanel.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">

        <Canvas Background="Green" Width="300" Height="200">
            <Button Width="100" Height="30" Content="Button 1" />
        </Canvas>

    </Grid>
</UserControl>
```

Figure 3-3 shows the button within the canvas.

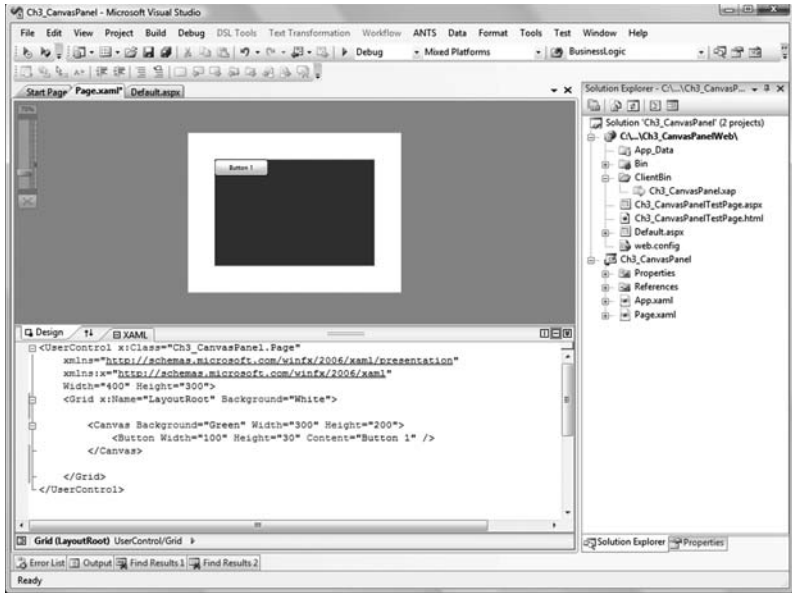

Figure 3-3. *Single button within the canvas*

4. Let's add another button to the `Canvas`, but this time, position it below and a bit to the right of the first button by setting its `Canvas.Top` and `Canvas.Left` attached properties. Give this button the label `Button 2`, as follows:

```
<Grid x:Name="LayoutRoot" Background="White">

    <Canvas Background="Green" Width="300" Height="200">
        <Button Width="100" Height="30" Content="Button 1" />
        <Button Width="100" Height="30" Content="Button 2"
            Canvas.Left="10" Canvas.Top="40" />
    </Canvas>

</Grid>
```

At this point, you now have two buttons within the canvas, but at different locations, as shown in Figure 3-4. This is still not very exciting, but this is about as cool as it gets with the `Canvas`.

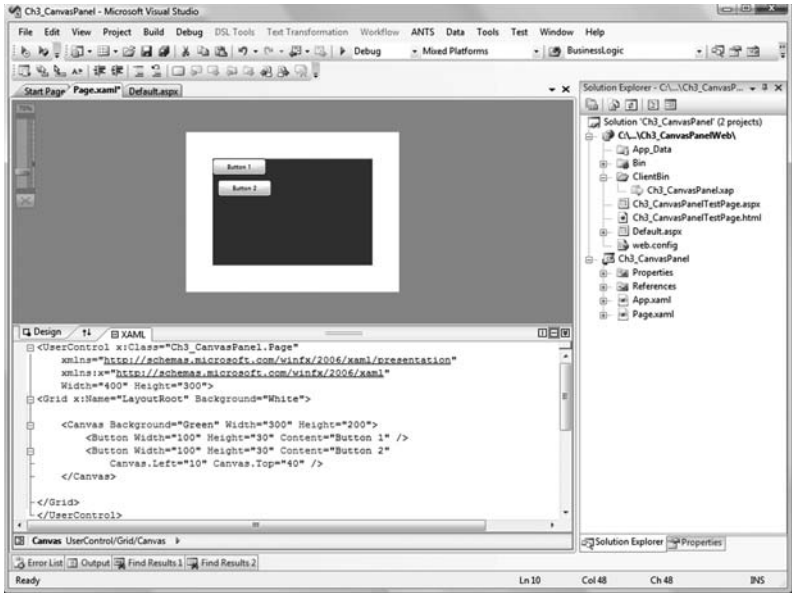

Figure 3-4. *Two buttons positioned relative to the canvas*

5. Go ahead and run the solution to see the end result as it will appear in the browser. The output is shown in Figure 3-5.

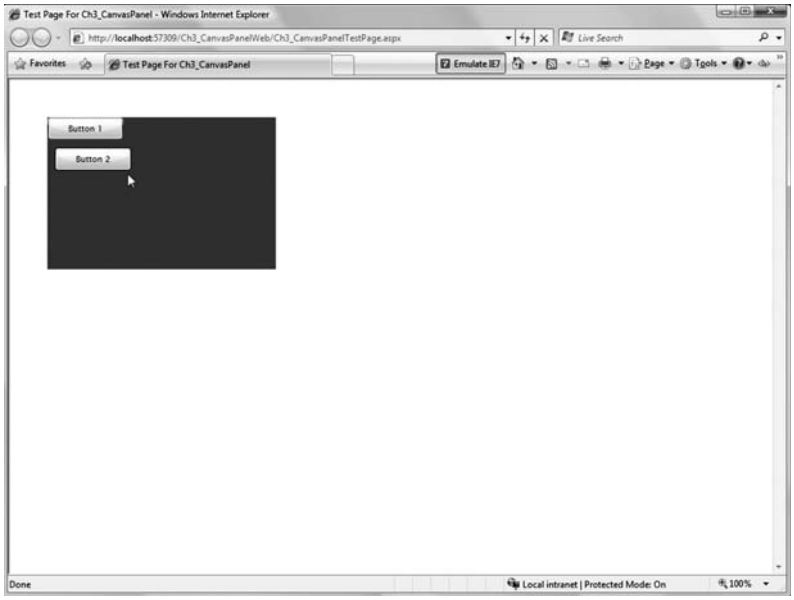

Figure 3-5. *The canvas and two buttons as seen in a browser*

Filling the Entire Browser Window with Your Application

By default, in a new Silverlight project, the root `UserControl` object is set to a width of 400 and a height of 300. In some cases, you may wish to set the width and height of your Silverlight application within the browser. At other times, however, you will want your Silverlight application to take up the entire window of your browser, and to resize as the browser is resized. This is done very easily within Silverlight. When you wish for the width and height to be set to 100%, simply omit the element's `Height` and `Width` attributes.

As an example, the following source has been adjusted for the `Canvas` panel and the Silverlight application to take up the entire browser:

```
<UserControl x:Class="Ch3_CanvasPanel.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White">

        <Canvas Background="Green">
        </Canvas>

    </Grid>
</UserControl>
```

With the omission of the `Height` and `Width` declarations for `UserControl` and `Canvas`, when you run the Silverlight application, you will see that the canvas takes up 100% of the browser window, as shown in Figure 3-6. It will resize as the browser resizes.

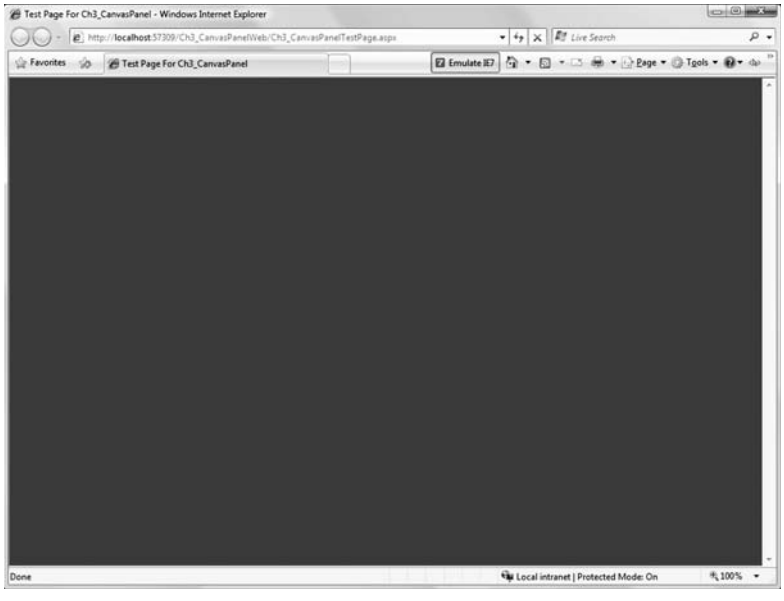

Figure 3-6. *Silverlight application taking up the entire browser*

As you've seen, the `Canvas` panel is a simple layout control. It can be used very effectively in a fixed layout. However, in most cases, you will want to use a static layout for your applications. The `StackPanel` control provides a more fluid layout control.

The StackPanel Control

The `StackPanel` panel is a new layout control that was not present in Silverlight 1.0. It provides developers with a quick layout option for positioning objects. The `StackPanel` control allows you to position Silverlight objects in more of a flow layout, stacking objects either horizontally or vertically. Figure 3-7 shows the basic concept of this layout control.

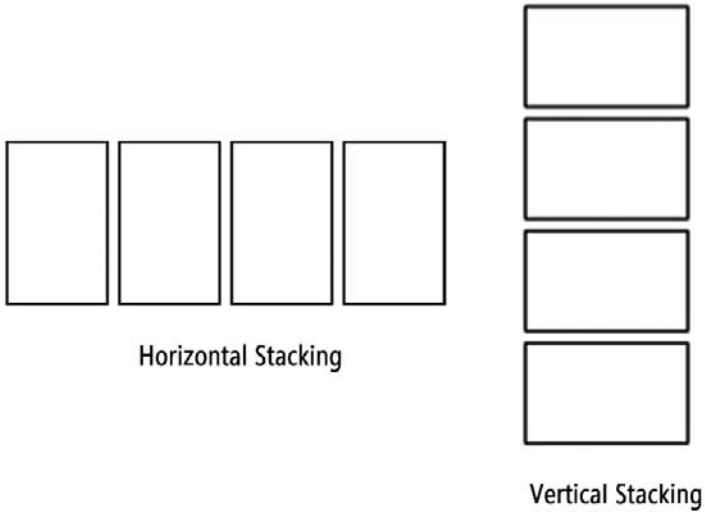

Figure 3-7. *The StackPanel control orientations*

Try It Out: Using the StackPanel Control

To better understand the `StackPanel` control, let's run through an exercise.

1. In Visual Studio 2008, create a new Silverlight application named `Ch3_StackPanel` and allow Visual Studio to create a Web Site project to host the application.

2. When the project is created you should be looking at the `Page.xaml` file. If you do not see the XAML source, switch so that you can edit the XAML. Within the main `Grid` element, add a `StackPanel` control and also three buttons with the labels `Button 1`, `Button 2`, and `Button 3`. Give all three buttons a width of 100 and a height of 30. The following XAML adds the `StackPanel` control and buttons (the new code is highlighted in bold in all the exercises):

```
<UserControl x:Class="Ch3_StackPanel.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">
        <StackPanel>
            <Button Width="100" Height="30" Content="Button 1"></Button>
            <Button Width="100" Height="30" Content="Button 2"></Button>
            <Button Width="100" Height="30" Content="Button 3"></Button>
        </StackPanel>
    </Grid>
</UserControl>
```

At this point, your application should appear as shown in Figure 3-8. Notice that the buttons are stacked vertically. This is because the default stacking orientation for the `StackPanel` control is vertical.

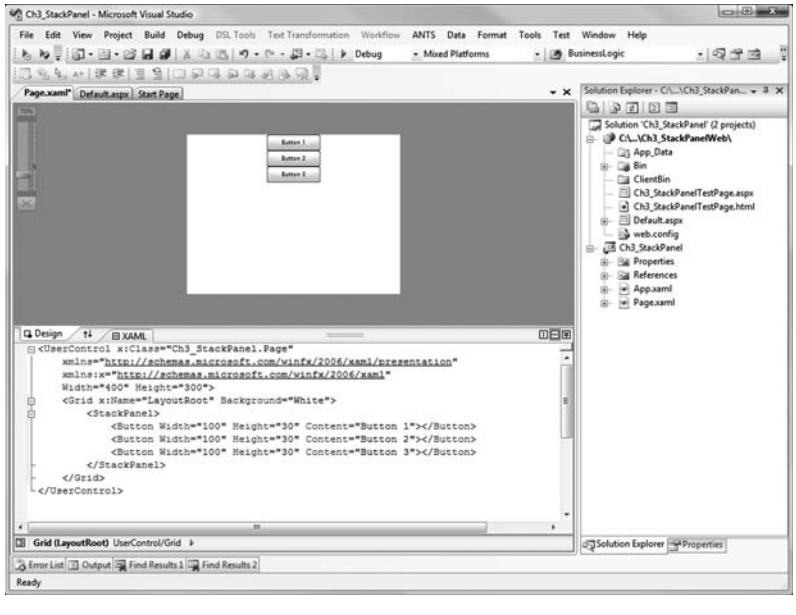

Figure 3-8. *The StackPanel control with its default orientation*

3. Change the orientation of the `StackPanel` control to be horizontal by setting the `Orientation` property to `Horizontal`, as follows:

```
<Grid x:Name="LayoutRoot" Background="White">
    <StackPanel Orientation="Horizontal" >
        <Button Width="100" Height="30" Content="Button 1"></Button>
        <Button Width="100" Height="30" Content="Button 2"></Button>
        <Button Width="100" Height="30" Content="Button 3"></Button>
    </StackPanel>
</Grid>
```

With this simple change, the buttons are now stacked horizontally, as shown in Figure 3-9.

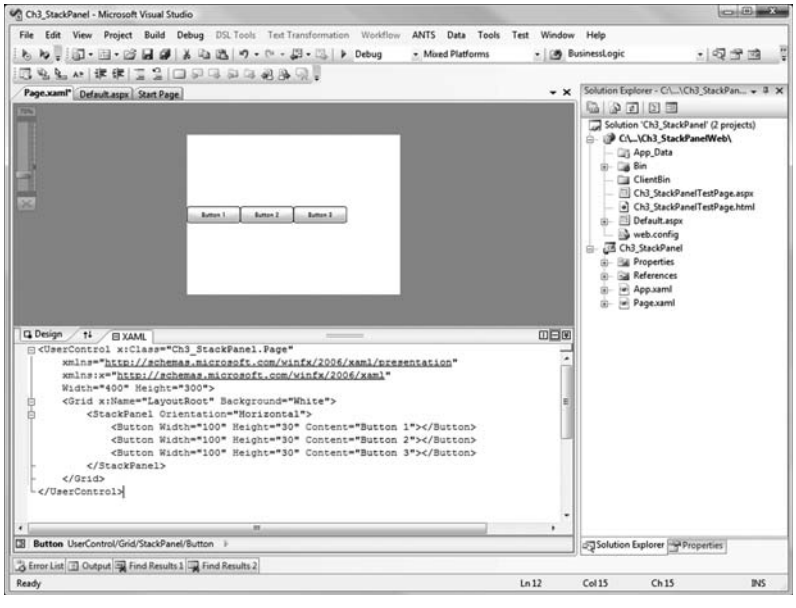

Figure 3-9. *The StackPanel control with horizontal orientation*

4. Notice that all the buttons are touching each other, which is unattractive. You can easily space them out by using their `Margin` property. In addition, you can center the buttons by setting the `StackPanel` control's `HorizontalAlignment` property to `Center`. Other options for `HorizontalAlignment` include `Left`, `Right`, and `Stretch` (which stretches the content to the left and right). Make the following changes to adjust the buttons:

```
<Grid x:Name="LayoutRoot" Background="White">
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <Button Width="100" Height="30" Content="Button 1" Margin="5"></Button>
        <Button Width="100" Height="30" Content="Button 2" Margin="5"></Button>
        <Button Width="100" Height="30" Content="Button 3" Margin="5"></Button>
    </StackPanel>
</Grid>
```

After you have made these changes, your buttons are spaced out nicely in the center of the application, as shown in Figure 3-10.

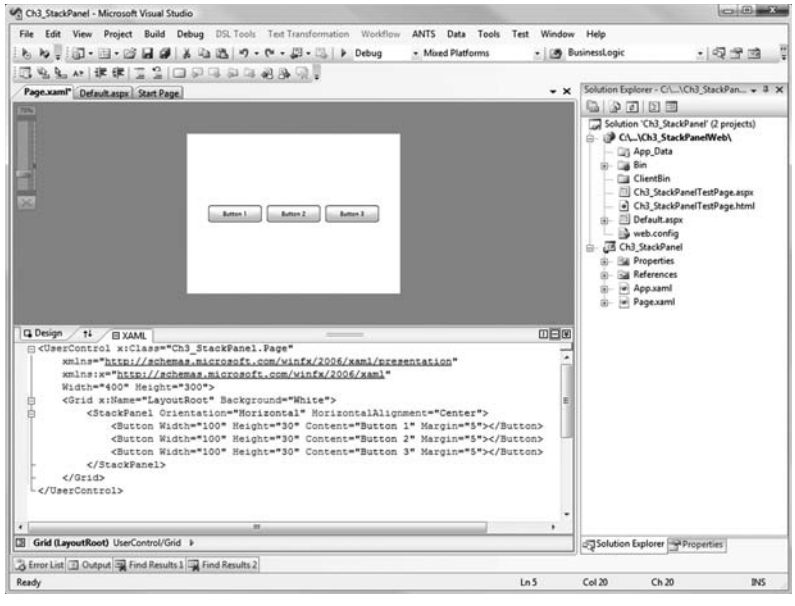

Figure 3-10. *The StackPanel control with buttons spaced apart and centered*

Try It Out: Nesting StackPanel Controls

Microsoft designed the control framework so that any object can be contained within another object. One way you can enhance your layout is by nesting a layout control within another layout control. In this example, you will nest a `StackPanel` control within another `StackPanel` control, but realize that you can nest any layout control within any other layout control to get the exact layout functionality you are seeking.

1. In Visual Studio 2008, create a new Silverlight application named `Ch3_NestedStackPanel` and allow Visual Studio to create a Web Site project to host the application.

2. In the `Page.xaml` file, add the following items:

   - A `StackPanel` control to the root `Grid` with its `Orientation` property set to `Horizontal` and the `HorizontalAlignment` property set to `Center`.
   - Within that `StackPanel`, add two buttons with the labels `Button Left` and `Button Right`.
   - In between the two buttons, add another `StackPanel` with `Orientation` set to `Vertical` and `VerticalAlignment` set to `Center`.
   - Within that nested `StackPanel`, include three buttons with the labels `Button Middle 1`, `Button Middle 2`, and `Button Middle 3`.
   - All buttons should have a `Margin` property set to `5`, and should have `Height` set to `30` and `Width` set to `100`.

   Here is what the updated source looks like:

```
<Grid x:Name="LayoutRoot" Background="White">
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <Button Width="100" Height="30" Content="Button Left" Margin="5" />
        <StackPanel VerticalAlignment="Center">
            <Button Width="100" Height="30" Content="Button Middle 1"
                     Margin="5"></Button>
            <Button Width="100" Height="30" Content="Button Middle 2"
                    Margin="5"></Button>
            <Button Width="100" Height="30" Content="Button Middle 3"
                    Margin="5"></Button>
        </StackPanel>
        <Button Width="100" Height="30" Content="Button Right"
Margin="5"></Button>
    </StackPanel>
</Grid>
```

   The cool result of this code is shown in Figure 3-11.

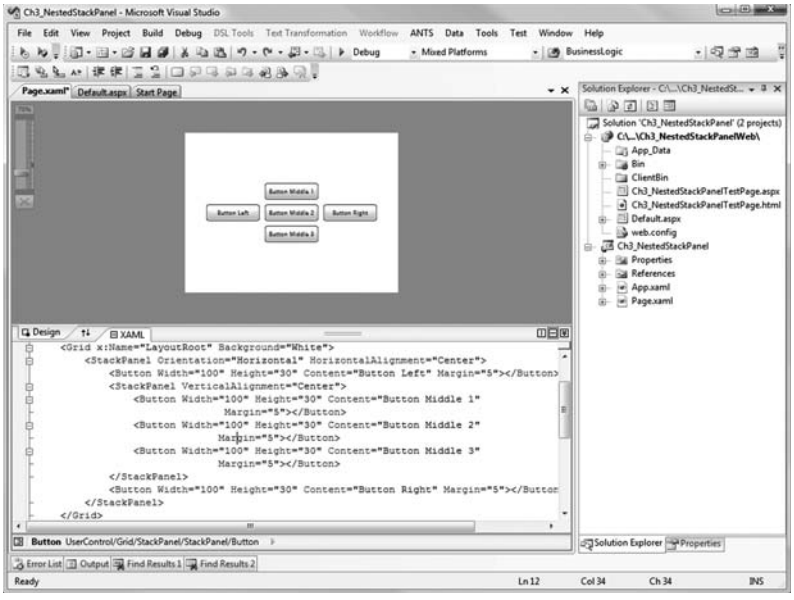

Figure 3-11. *Nested StackPanel controls*

3. Run the application to see the results.

As you can see from these two exercises, the `StackPanel` control is a very useful layout option, and you will probably use it often in your Silverlight applications. By nesting Silverlight controls, you have a lot of flexibility when designing your applications. However, in the event that you want more control of the positioning of items in your application, without needing to resort to the absolute positioning used by the `Canvas` control, the `Grid` control may be just the layout option you need.

The Grid Control

The `Grid` control provides more fine-tuned layout in Silverlight applications. As a comparison, you can think of using the `Grid` layout control as similar to using table elements to position items in HTML, only more flexible. With the `Grid` control, you can define rows and columns, thus creating grid cells, and then add objects to individual cells in the grid or to multiple cells, by using spanning.

To specify in which cell to place an object, you use the `Grid.Column` and `Grid.Row` attached properties. Note that these properties are *base zero*, so the top-left cell it is row 0 and column 0. Figure 3-12 illustrates the row and column locations for the grid.

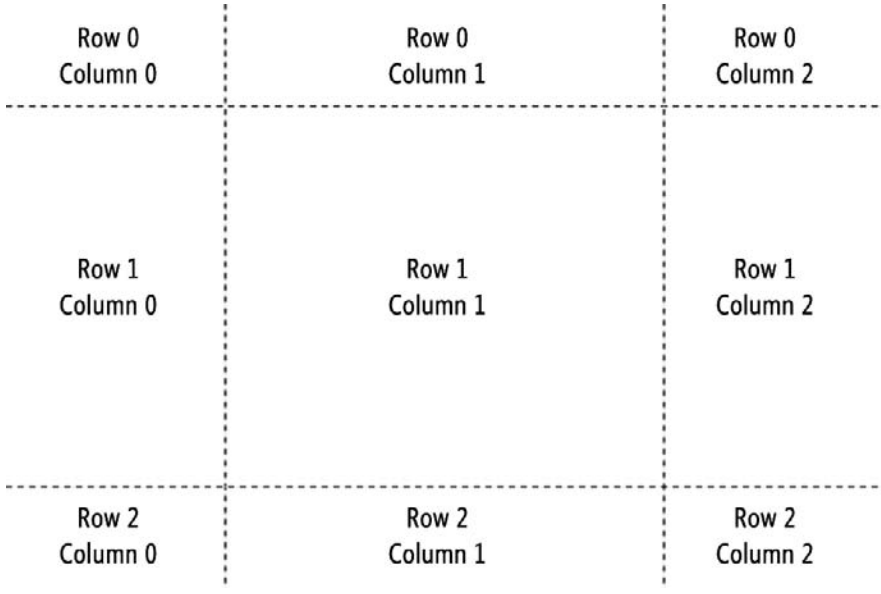

Figure 3-12. *Row and column grid cell locations in the Grid control layout*

For most developers, the `Grid` control will most likely be the layout option of choice, due to its flexibility. At the same time, the `Grid` control is significantly more complex than the others, as you'll see in the following exercises.

Try It Out: Using the Grid Control

Let's try out a simple `Grid` panel with four buttons.

1. In Visual Studio 2008, create a new Silverlight application named `Ch3_GridPanel` and allow Visual Studio to create a Web Site project to host the application.

2. For this example, you are going to need a bit more space in which to work. In the `Page.xaml` file, start out by changing the `UserControl`'s `Width` to `600` and `Height` to `400`, as follows:

```
<UserControl x:Class="Ch3_GridPanel.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="600" Height="400">
    <Grid x:Name="LayoutRoot" Background="White">

    </Grid>
</UserControl>
```

3. Add a new `Grid` control to the Silverlight application. In order to better see what is going on, turn on the display of grid lines by setting the `ShowGridLines` property to `true`. The following code shows these additions. Keep in mind that since you have not designated a size for the grid, it will automatically take up the entire size of the parent, and in this case, the entire Silverlight application.

```
<UserControl x:Class="Ch3_GridPanel.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="600" Height="400">
    <Grid x:Name="LayoutRoot" Background="White">
        <Grid ShowGridLines="True">

        </Grid>
    </Grid>
</UserControl>
```

4. Next, define the rows and columns in the `Grid` control. You do this using the XAML property elements `Grid.RowDefinitions` and `Grid.ColumnDefinitions`. Add the following XAML to your new grid:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid ShowGridLines="True">

        <Grid.RowDefinitions>
            <RowDefinition Height="70" />
            <RowDefinition Height="*" />
            <RowDefinition Height="70" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="150" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="150" />
        </Grid.ColumnDefinitions>

    </Grid>
</Grid>
```

Notice that for the center row and column, you are setting the `Height` and `Width` properties to "*". The asterisk tells the row and column to take up all available space. As the `Grid` control is resized with the browser window, those columns will be resized to take up all the space not consumed by the fixed-sized columns. After you have added these row and column definitions, your canvas should appear as shown in Figure 3-13.

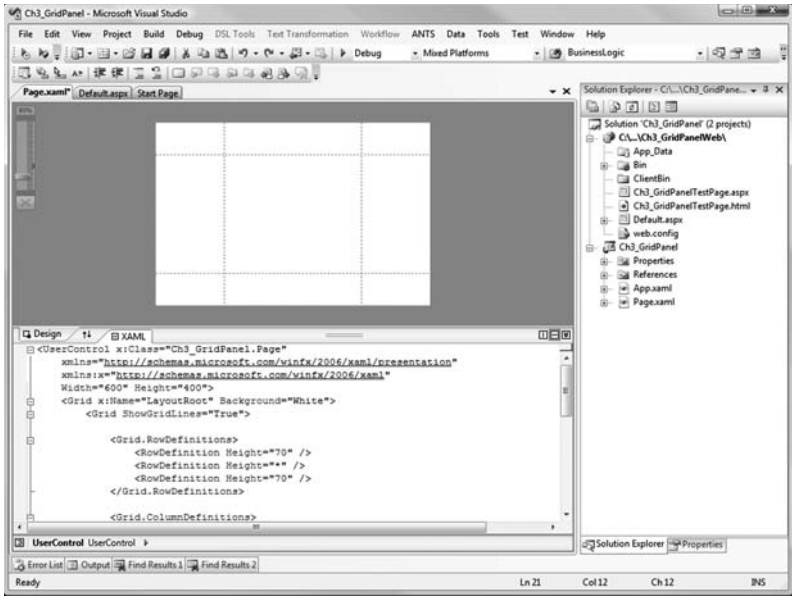

Figure 3-13. *Grid with columns and rows*

5. You can now add objects to the different grid cells. Place a button in each of the four corner cells, giving the buttons the corresponding labels `Top Left`, `Top Right`, `Bottom Left`, and `Bottom Right`. To place the buttons, add the following code:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid ShowGridLines="True">

        <Grid.RowDefinitions>
            <RowDefinition Height="70" />
            <RowDefinition Height="*" />
            <RowDefinition Height="70" />
        </Grid.RowDefinitions>
```

```
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="150" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="150" />
        </Grid.ColumnDefinitions>

        <Button Width="100" Height="30" Content="Top Left"
                Margin="5" Grid.Row="0" Grid.Column="0"></Button>
        <Button Width="100" Height="30" Content="Top Right"
                Margin="5" Grid.Row="0" Grid.Column="2"></Button>
        <Button Width="100" Height="30" Content="Bottom Left"
                Margin="5" Grid.Row="2" Grid.Column="0"></Button>
        <Button Width="100" Height="30" Content="Bottom Right"
                Margin="5" Grid.Row="2" Grid.Column="2"></Button>

    </Grid>
</Grid>
```

After the buttons are added, your application should look like Figure 3-14.

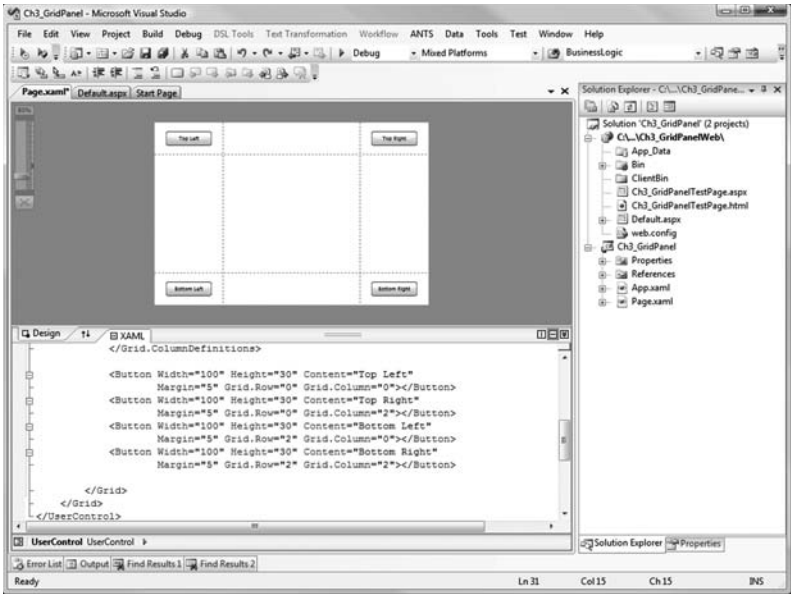

Figure 3-14. *The grid with buttons in the four corners*

Try It Out: Nesting a Grid and Spanning a Column

Next, you will nest another `Grid` control in the center cell of the `Grid` control you just added. This will make the application layout somewhat complex, but it will also serve to show how `Grid` panels are defined using XAML.

1. In the `Ch3_GridPanel` file, add the following items:

   - A `Grid` control positioned at `Grid.Column=1` and `Grid.Row=1`
   - Three `RowDefinition` and two `ColumnDefinition` elements
   - Buttons in the four corners of the new `Grid` control, as you just did in the outer `Grid` panel

   The source code should look like the following:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid ShowGridLines="True">

        <Grid.RowDefinitions>
            <RowDefinition Height="70" />
            <RowDefinition Height="*" />
            <RowDefinition Height="70" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="150" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="150" />
        </Grid.ColumnDefinitions>

        <Button Width="100" Height="30" Content="Top Left"
                Margin="5" Grid.Row="0" Grid.Column="0"></Button>
        <Button Width="100" Height="30" Content="Top Right"
                Margin="5" Grid.Row="0" Grid.Column="2"></Button>
        <Button Width="100" Height="30" Content="Bottom Left"
                Margin="5" Grid.Row="2" Grid.Column="0"></Button>
        <Button Width="100" Height="30" Content="Bottom Right"
                Margin="5" Grid.Row="2" Grid.Column="2"></Button>

        <Grid Grid.Column="1" Grid.Row="1"  ShowGridLines="True">

            <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
```

```
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Button Width="100" Height="30" Content="Nested Top Left"
            Margin="5" Grid.Row="0" Grid.Column="0"></Button>
        <Button Width="100" Height="30" Content="Nested Top Right"
            Margin="5" Grid.Row="0" Grid.Column="2"></Button>
        <Button Width="100" Height="30" Content="Nested B. Left"
            Margin="5" Grid.Row="2" Grid.Column="0"></Button>
        <Button Width="100" Height="30" Content="Nested B. Right"
            Margin="5" Grid.Row="2" Grid.Column="2"></Button>

    </Grid>

</Grid>
```

At this point, your application should look like Figure 3-15. Now, this is a pretty cool layout.

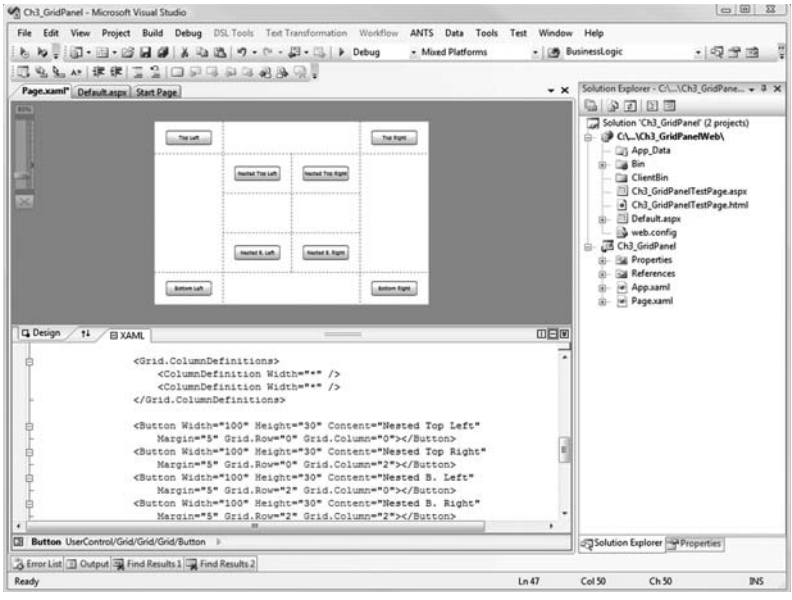

Figure 3-15. *Nested grid with buttons*

2. Notice that you have not placed anything in the two columns in the middle row of the new grid. Here, you're going to add a button that spans these two columns so the button will appear in the center of the row. In order to do this, add the new button to the `Grid` control with the `Grid.ColumnSpan` attached property set to `2`. The source changes to the innermost `Grid` control are as follows.

```
<Grid Grid.Column="1" Grid.Row="1"  ShowGridLines="True">

    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Button Width="100" Height="30" Content="Nested Top Left"
        Margin="5" Grid.Row="0" Grid.Column="0"></Button>
    <Button Width="100" Height="30" Content="Nested Top Right"
        Margin="5" Grid.Row="0" Grid.Column="2"></Button>
    <Button Width="100" Height="30" Content="Nested B. Left"
        Margin="5" Grid.Row="2" Grid.Column="0"></Button>
    <Button Width="100" Height="30" Content="Nested B. Right"
        Margin="5" Grid.Row="2" Grid.Column="2"></Button>

    <Button Width="100" Height="30" Content="Nested Center"
        Margin="5" Grid.Row="1" Grid.Column="0"
        Grid.ColumnSpan="2"></Button>

</Grid>
```

Now that you have added the button to the center column, your application should look as shown in Figure 3-16. Notice how the button spans the two columns and appears in the center. For experienced HTML developers who are used to laying out their forms with tables, this approach should be very comfortable, as it closely mimics using the `colspan` attribute for a `<TD>` tag.

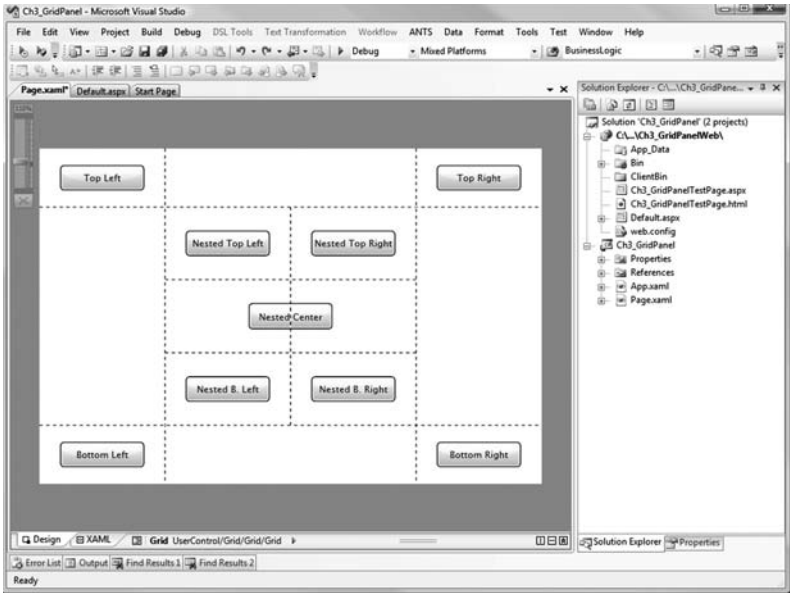

Figure 3-16. *Final application with a nested grid and buttons*

In this example, you saw how to create a relatively complex layout using the `Grid` control. As you can see, this is a very powerful and flexible layout tool for your Silverlight applications.

Summary

In this chapter, we explored the three layout controls that are available out of the box in Silverlight 2. We looked at the `Canvas`, `StackPanel`, and `Grid` controls. In the next chapter, we will take an in-depth look at the form controls that come bundled with Silverlight 2.

CHAPTER 4

Silverlight Form Controls

For those who have worked with Silverlight 1.0, one of the first observations you most likely made was the lack of common controls such as the `Button`, `TextBox`, and `ListBox`. In fact, Silverlight 1.0 provided only two basic controls: `Rectangle` and `TextBlock`. From these, the developers were expected to implement all of the rich controls they needed. As you can imagine, it was quite a bit of work to create all of the form controls using just these two base controls.

Since then, Microsoft's vision of Silverlight has gone beyond basic animations to spark up your applications and into the realm of feature-rich user interfaces (UIs). To this end, Silverlight 2 includes a strong base of controls that you can use within your Silverlight applications.

In this chapter, we will first look at the Silverlight controls in general by examining control properties and events. We will then take a brief tour of some of the more common form controls included in Silverlight 2. This chapter is meant to provide a high-level introduction to these common Silverlight controls. We will continue to work with the controls throughout the remainder of the book, so you will see more specific usage scenarios.

Setting Control Properties

The most straightforward and simple way to set a property is by using attribute syntax. However, in some cases, you will use element syntax.

Attribute Syntax

Most properties that can be represented as a simple string can be set using attribute syntax. Setting an attribute in XAML is just like setting an attribute in XML. An XML element contains a node and attributes. Silverlight controls are defined in the same way, where the control name is the node, and the properties are defined as attributes.

As an example, you can easily use attribute syntax to set the `Width`, `Height`, and `Content` properties of a `Button` control, as follows:

```
<Button Width="100" Height="30" Content="Click Me!"></Button>
```

Element Syntax

Element syntax is most commonly used when a property cannot be set using attribute syntax because the property value cannot be represented as a simple string. Again, this is very similar to using elements in XML. The following is an example of setting the background color of a button:

```
<Button Width="100" Height="30" Content="Click Me!">
    <Button.Background>
        <SolidColorBrush Color="Blue"/>
    </Button.Background>
    <Button.Foreground>
        <SolidColorBrush Color="Red"/>
    </Button.Foreground>
</Button>
```

Type-Converter–Enabled Attributes

Sometimes when defining a property via an attribute, the value cannot be represented as a simple string; rather, it is converted to a more complex type. A common usage of a type-converter–enabled attribute is `Margin`. The `Margin` property can be set as a simple string, such as in this example:

```
<Button Width="100" Content="Click Me!" Margin="15"></Button>
```

When you set the `Margin` property in this fashion, the left, right, top, and bottom margins are all set to 15 pixels. But what if you want to set the top margin to 15 pixels, but you want the other three margins to be 0? In order to do that, you would set the `Margin` property as follows:

```
<Button Width="100" Content="Click Me!" Margin="0,15,0,0"></Button>
```

In this case, Silverlight takes the string `"0,15,0,0"` and converts it into a more complex type. The string is converted to four values: left margin = 0, top margin = 15, right margin = 0, and bottom margin = 0.

This type-conversion concept is not new to Silverlight. For those of you familiar with Cascading Style Sheets (CSS), the same sort of structure exists. As an example, when you are defining a border style, within the simple string value for a border, you are actually

setting the thickness, color, and line style. The following border assignment in CSS will set the border thickness to 1 pixel, the line style to be solid, and the color to #333333 (dark gray):

```
border: 1px solid #333333;
```

Attached Properties

In the previous chapter, you learned how to set a control's position within a `Canvas` panel by using attached properties. An *attached property* is a property that is attached to parent control. In the example in the previous chapter, you specified the `Button` control's position within the `Canvas` object by setting two attached properties: `Canvas.Top` and `Canvas.Left`. These two properties reference the `Button` control's parent, which is the `Canvas`.

```
<Canvas>
    <Button Width="100" Content="Click Me!"
            Canvas.Top="10" Canvas.Left="13" />
</Canvas>
```

Nesting Controls Within Controls

When you first look at the controls included in Silverlight 2, you will probably feel pretty comfortable, as they seem to be what you would expect. However, when you dig a bit deeper into the control features, you will find that the controls are much more flexible and powerful than they first appear.

One of the key features of controls in Silverlight 2 is the ability to put just about anything within a control. A `Button` control can contain a `StackPanel`, which can contain an `Ellipse` control and a `TextBlock` control. There really are few limitations as to what the contents of a control can be. Figure 4-1 shows an example of a standard Silverlight 2 `Button` control containing a `StackPanel`, a nested `StackPanel`, an `Ellipse`, a `TextBlock`, and a `ListBox`.

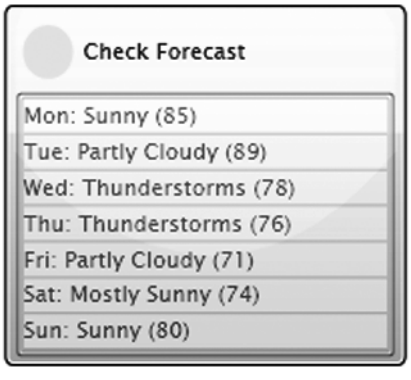

Figure 4-1. *A Button control with nested controls*

The following code was used to produce the control in Figure 4-1:

```
<Button Height="180" Width="200">
    <StackPanel Orientation="Vertical">
        <StackPanel Margin="5"
            VerticalAlignment="Center"
            Orientation="Horizontal">

            <Ellipse Fill="Yellow" Width="25" />
            <TextBlock VerticalAlignment="Center"
                Margin="5" Text="Check Forecast" />

        </StackPanel>
        <ListBox FontSize="11" Opacity="0.5"
            Margin="2" x:Name="lstForecastGlance">
            <ListBoxItem>
                <TextBlock VerticalAlignment="Center"
                    Text="Mon: Sunny (85)" />
            </ListBoxItem>
            <ListBoxItem>
                <TextBlock VerticalAlignment="Center"
                    Text="Tue: Partly Cloudy (89)" />
            </ListBoxItem>
            <ListBoxItem>
                <TextBlock VerticalAlignment="Center"
                    Text="Wed: Thunderstorms (78)" />
            </ListBoxItem>
            <ListBoxItem>
                <TextBlock VerticalAlignment="Center"
                    Text="Thu: Thunderstorms (76)" />
            </ListBoxItem>
            <ListBoxItem>
                <TextBlock VerticalAlignment="Center"
                    Text="Fri: Partly Cloudy (71)" />
            </ListBoxItem>
            <ListBoxItem>
                <TextBlock VerticalAlignment="Center"
                    Text="Sat: Mostly Sunny (74)" />
            </ListBoxItem>
            <ListBoxItem>
                <TextBlock VerticalAlignment="Center"
```

```
                Text="Sun: Sunny (80)" />
            </ListBoxItem>
        </ListBox>
    </StackPanel>
</Button>
```

As the code shows, the example simply nests additional content within the `Button` control. As you can imagine, this can be a very powerful feature.

Handling Events in Silverlight

As with other Microsoft programming frameworks, Silverlight provides an event mechanism to track actions that take place within Silverlight 2 applications. Two types of actions are tracked within Silverlight:

- Actions that are triggered based on some input from the user. Input actions are handled and "bubbled" up from the browser to the Silverlight object model.
- Actions that are triggered based on a change of state of a particular object, including the object's state in the application. These actions are handled directly from the Silverlight object model.

Event handlers are methods that are executed when a given event is triggered. You can define event handlers either in the XAML markup itself or in managed code. The following exercises will demonstrate how to define event handlers in both ways.

Try It Out: Declaring an Event in XAML

Let's get started by defining event handlers within the XAML markup.

1. Open Visual Studio 2008 and create a new Silverlight project called `Ch4_EventHandlers`. Allow Visual Studio to create a Web Site project to host the application.

2. When the project is created, you should be looking at the `Page.xaml` file. If you do not see the XAML source, switch to that view so that you can edit the XAML. Within the root `Grid` of the Silverlight page, add grid row and column definitions (as explained in Chapter 3) to define four rows and two columns, as follows:

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">

    <Grid.RowDefinitions>
        <RowDefinition Height="70" />
        <RowDefinition Height="70" />
```

```
        <RowDefinition Height="70" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
</Grid>
```

3. Next, add a `Button` control to the upper-left grid cell and a `TextBlock` control in the upper-right cell.

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">

    <Grid.RowDefinitions>
        <RowDefinition Height="70" />
        <RowDefinition Height="70" />
        <RowDefinition Height="70" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Button Width="125" Height="35" Content="XAML Event"></Button>
    <TextBlock Text="Click the XAML Event!" Grid.Column="1"
        VerticalAlignment="Center" HorizontalAlignment="Center" />
</Grid>
```

4. Add the `Click` property to the button. When you type `Click=`, Visual Studio 2008 will prompt you with the option of automatically creating a new event handler, as shown in Figure 4-2. When the <New Event Handler> option is displayed, simply press Enter, and Visual Studio will complete the `Click` property, as follows:

```
<Button Width="125" Height="35"
    Content="XAML Event" Click="Button_Click" />
```

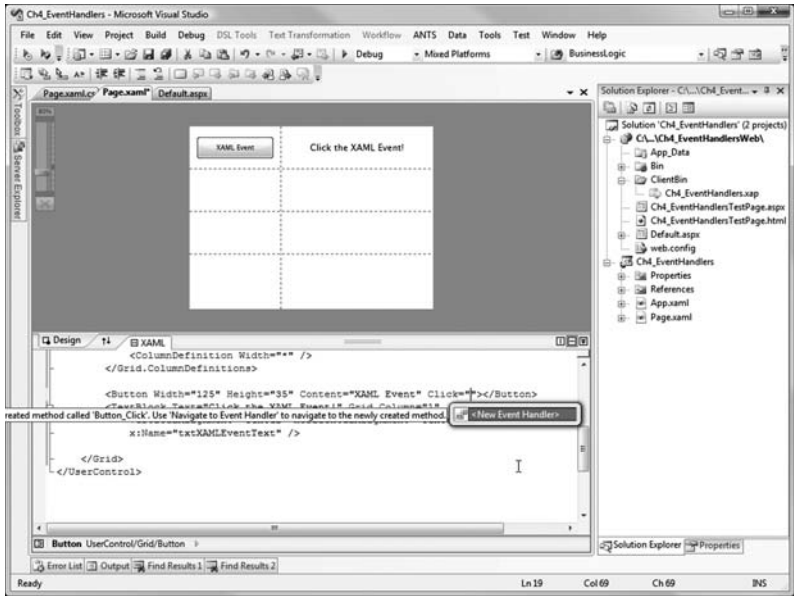

Figure 4-2. *Visual Studio's automatic creation of an event handler*

In addition, Visual Studio automatically adds an event handler called `Button_Click` to the code-behind class for the Silverlight application, as follows:

```
public partial class Page : UserControl
{
    public Page()
    {
        InitializeComponent();
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {

    }
}
```

5. For this example, you will change the `Text` property within the `TextBlock`. In order to do this, you first need to give the `TextBlock` a name so you can access it from the code behind. Add the following code.

```
<TextBlock Text="Click the XAML Event!" Grid.Column="1"
      VerticalAlignment="Center" HorizontalAlignment="Center"
      x:Name="txtXAMLEventText" />
```

6. Now change the `Text` property of the `TextBlock` within the `Button_Click` event, as follows:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    txtXAMLEventText.Text = "Thank you for clicking!";
}
```

7. Run the application and click the XAML Event button. The text to the right of the button will change to "Thank you for clicking." Figures 4-3 and 4-4 show the application before and after clicking the XAML Event button.

Figure 4-3. *The TextBlock before the button is clicked*

Now that you have seen how to define an event handler in the XAML markup, in the next exercise, you will continue by adding another event handler using managed code.

Figure 4-4. *The TextBlock after the button is clicked*

Try It Out: Declaring an Event Handler in Managed Code

Let's continue with the project named `Ch4_EventHandlers` from the previous exercise. You'll add another button and wire up its event handler using managed code.

1. Add another button and `TextBlock` in the second row of the `Grid`, as follows:

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">

    <Grid.RowDefinitions>
        <RowDefinition Height="70" />
        <RowDefinition Height="70" />
        <RowDefinition Height="70" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
```

```
    <Button Width="125" Height="35" Content="XAML Event"
        Click="Button_Click"></Button>
    <TextBlock Text="Click the XAML Event!" Grid.Column="1"
        VerticalAlignment="Center" HorizontalAlignment="Center"
        x:Name="txtXAMLEventText" />

    <Button Width="125" Height="35" Content="Managed Event"
        Grid.Row="1" ></Button>
    <TextBlock Text="Click the Managed Event!" Grid.Column="1"
        VerticalAlignment="Center" HorizontalAlignment="Center"
        Grid.Row="1" />
</Grid>
```

2. In order to reference the new `Button` control in managed code, you must give it and the `TextBlock` control a name, as shown in the following snippet:

```
    <Button Width="125" Height="35" Content="Managed Event"
        Grid.Row="1" x:Name="btnManaged" ></Button>
    <TextBlock Text="Click the Managed Event!" Grid.Column="1"
        VerticalAlignment="Center" HorizontalAlignment="Center"
        Grid.Row="1" x:Name="txtManagedEventText" />
```

Your page should now appear as shown in Figure 4-5.

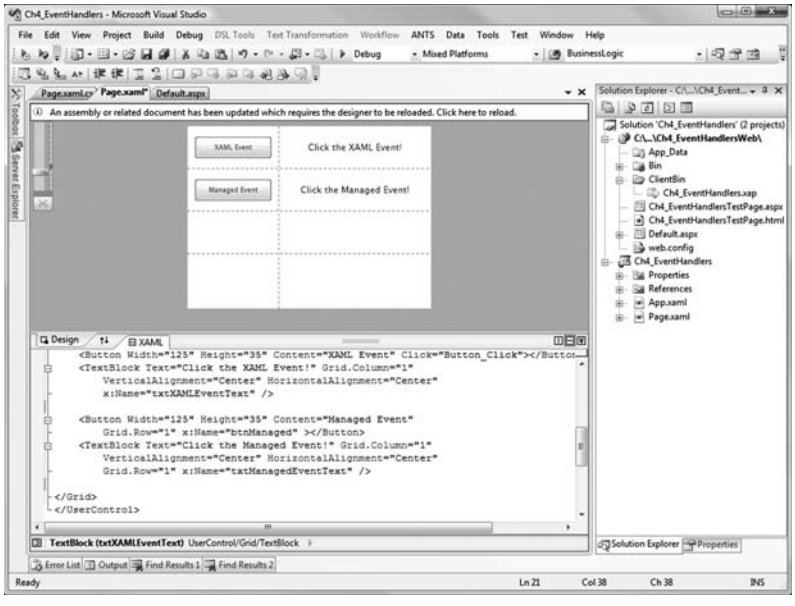

Figure 4-5. *The updated Silverlight page*

3. Next, you need to add the event handler. Right-click the Silverlight page and select View Code. This will switch to the code behind of the page.

   From here, you will use the standard CLR language-specific syntax for adding event handlers. Since you are using C#, the syntax is to use the += operator and assign it to a new `EventHandler`. Visual Studio 2008 will help you with this.

4. After the `InitializeComponent()` method call in the `Page` constructor, start typing `"this.btnManaged.Click +="`. At this point, Visual Studio will display the message "new RoutedEventHandler(bntManaged_Click); (Press TAB to insert)," as shown in Figure 4-6. Press Tab to complete the event handler definition.

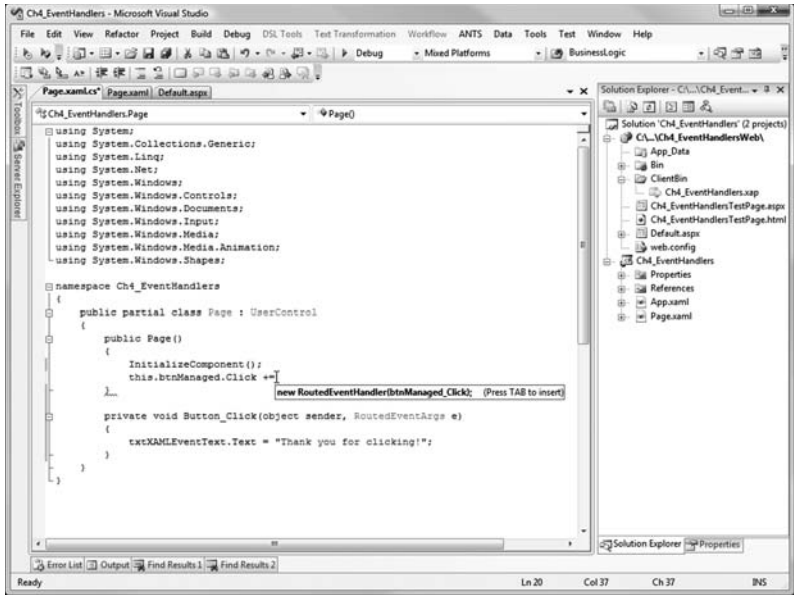

Figure 4-6. *Visual Studio assisting with wiring up an event handler in managed code*

5. Visual Studio will once again prompt you for the name of the event handler. Go ahead and press Tab again to accept the default name. At this point, your source should look like this:

```
namespace Ch4_EventHandlers
{
    public partial class Page : UserControl
    {
        public Page()
        {
            InitializeComponent();
```

```
            this.btnManaged.Click += new RoutedEventHandler(btnManaged_Click);
        }

        void btnManaged_Click(object sender, RoutedEventArgs e)
        {
            throw new NotImplementedException();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            txtXAMLEventText.Text = "Thank you for clicking!";
        }
    }
}
```

6. Now the only thing left to do is add the code to the event handler. You will notice that, by default, Visual Studio added code to automatically throw a `NotImplementedException`. Remove that line and replace it with the following line to change the `TextBlock` control's text.

```
void btnManaged_Click(object sender, RoutedEventArgs e)
{
    txtManagedEventText.Text = "Thank you for clicking";
}
```

7. Run the application and click the Managed Event button. You will see the text for the second `TextBlock` is updated to say "Thank you for clicking," as shown in Figure 4-7.

Figure 4-7. *The result of the managed code event handler*

This exercise demonstrated how to wire up an event handler using C# and managed code.

In the remainder of the chapter, we will take a tour of the more commonly used form controls in Silverlight 2. Let's start off by looking at the `Border` control.

The Border Control

The `Border` control provides a way to add a border and background to any one control in Silverlight. Even though a border is applied to only one control, you can always place a border around a `StackPanel` or `Grid`, and as a result include many controls within a border.

The syntax to add a `Border` control to your Silverlight project is very simple, as you can see from the following example:

```
<UserControl x:Class="Ch4_BorderControl.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">
        <Border BorderThickness="2" BorderBrush="Black" Margin="10">
            <StackPanel Margin="10">
                <Button Content="Sample Button" Margin="5" />
                <TextBlock Text="Sample TextBlock" Margin="5" />
                <ListBox Margin="5">
                    <ListBoxItem>
                        <TextBlock Text="ListItem 1" />
                    </ListBoxItem>
                    <ListBoxItem>
                        <TextBlock Text="ListItem 2" />
                    </ListBoxItem>
                    <ListBoxItem>
                        <TextBlock Text="ListItem 3" />
                    </ListBoxItem>
                    <ListBoxItem>
                        <TextBlock Text="ListItem 4" />
                    </ListBoxItem>
                </ListBox>
            </StackPanel>
        </Border>
    </Grid>
</UserControl>
```

Figure 4-8 shows the results.

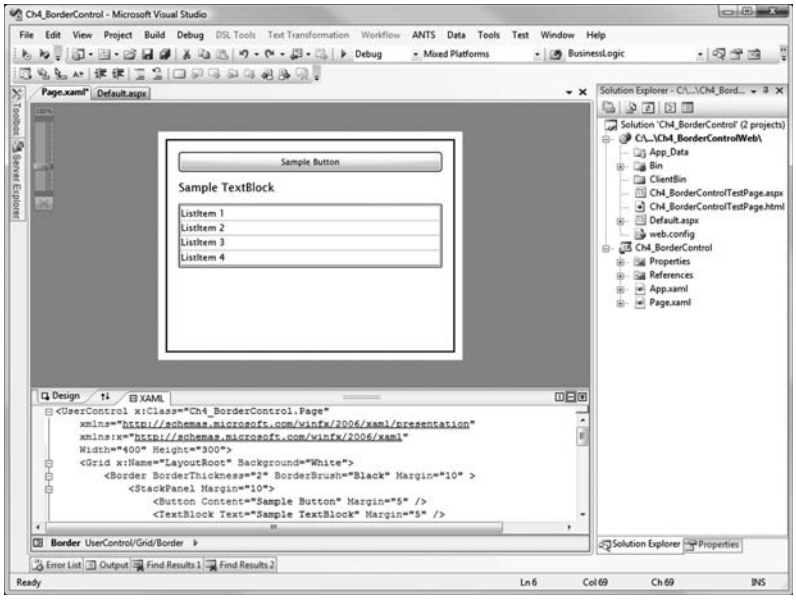

Figure 4-8. *Using the Border control*

Another feature of the `Border` control is the ability to round the corners of the border using the `CornerRadius` property. Here is how the preceding example could be modified to provide a `Border` control with a `CornerRadius` property of `10`.

```
<Border BorderThickness="2" BorderBrush="Black" Margin="10" CornerRadius="10">
 . . .
</Border>
```

The border with rounded corners is shown in Figure 4-9.

You can declare a background color for your border using the `Background` property. Like the `BorderBrush` property, the `Background` property can be set to either a color or a brush type. Here is an example of setting a border with a background color of silver:

```
<Border BorderThickness="2" BorderBrush="Black" Margin="10" CornerRadius="10"
    Background="Silver">
  . . .
</Border>
```

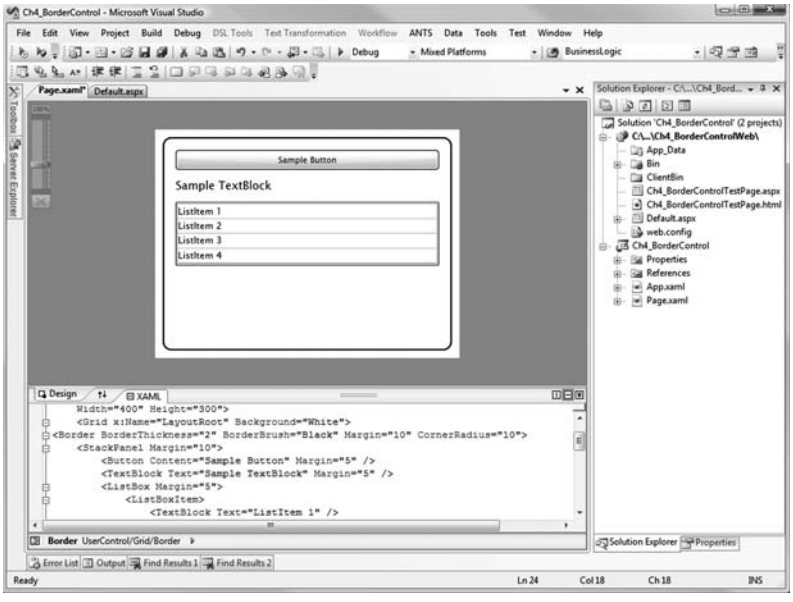

Figure 4-9. *Border control with a CornerRadius property of 10*

Figure 4-10 shows the result of adding the background color.

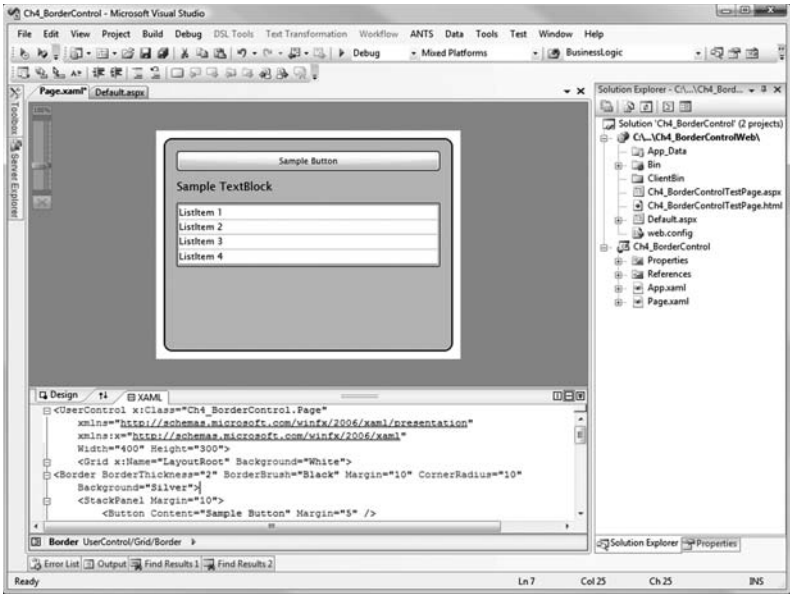

Figure 4-10. *Border control with its background set to silver*

The following is an example of a more complex `Border` control that contains a gradient for the border and background, by using a `Brush` object.

```
<Border BorderThickness="2" Margin="10" CornerRadius="10">
    <Border.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStop Color="Green" Offset="0" />
                <GradientStop Color="White" Offset="1" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Border.Background>
    <Border.BorderBrush>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStop Color="Black" Offset="0" />
                <GradientStop Color="White" Offset="1" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Border.BorderBrush>

    <StackPanel Margin="10">
        <Button Content="Sample Button" Margin="5" />
        <TextBlock Text="Sample TextBlock" Margin="5" />
        <ListBox Margin="5">
            <ListBoxItem>
                <TextBlock Text="ListItem 1" />
            </ListBoxItem>
            <ListBoxItem>
                <TextBlock Text="ListItem 2" />
            </ListBoxItem>
            <ListBoxItem>
                <TextBlock Text="ListItem 3" />
            </ListBoxItem>
            <ListBoxItem>
                <TextBlock Text="ListItem 4" />
            </ListBoxItem>
        </ListBox>
    </StackPanel>
</Border>
```

Figure 4-11 shows the border with the gradient applied.

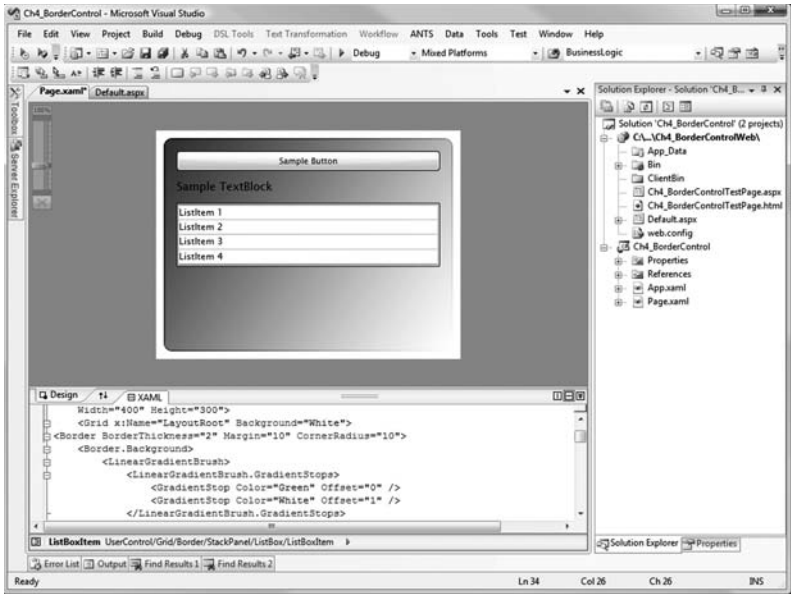

Figure 4-11. *Border control with gradient brushes for the border and background*

User Input Controls

One of the most common controls in applications is a text box, which is the standard control for collecting basic string input from the user. Also ubiquitous are check boxes and radio buttons, which allow users to select from a list of choices—more than one choice in the case of check boxes, and a single choice in the case of radio buttons. Silverlight 2 provides the `TextBox`, `CheckBox`, and `RadioButton` for these standard controls. The following exercises will also give you a chance to work with the `Ellipse` and `Rectangle` controls.

Try It Out: Working with the TextBox Control

This exercise demonstrates the use of the `TextBox` control in Silverlight 2 by creating a simple application that will request the red, green, and blue values to fill an ellipse with a given color. The resulting application will appear as shown in Figure 4-12.

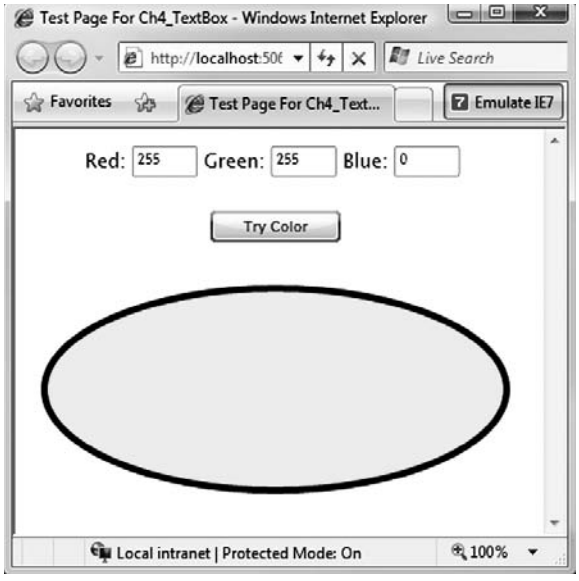

Figure 4-12. *Sample application using TextBox controls*

1. In Visual Studio 2008, create a new Silverlight application named `Ch4_TextBox`. Allow Visual Studio to create a Web Site project to host your application.

2. In the `Page.xaml` file, within the root `Grid` element, add three `RowDefinition` items, as follows:

```
<UserControl x:Class="Ch4_TextBox.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White" >

        <Grid.RowDefinitions>
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition />
        </Grid.RowDefinitions>

    </Grid>
</UserControl>
```

3. Add three TextBox and TextArea controls contained in a horizontal-oriented StackPanel to the first row, a Button control to the second row, and an Ellipse control to the third row. In addition, place a TextBlock in the third row to stack on top of the Ellipse control for error-reporting purposes. Name each of the TextBox controls, as well as the Button control and the TextBlock. These additions are shown in the following code:

```
<UserControl x:Class="Ch4_TextBox.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White" >

        <Grid.RowDefinitions>
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition />
        </Grid.RowDefinitions>

        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <TextBlock VerticalAlignment="Center" Text="Red:" />
            <TextBox x:Name="txtRed"
                Height="24" Width="50" Margin="5" />
            <TextBlock VerticalAlignment="Center" Text="Green:" />
            <TextBox x:Name="txtGreen"
                Height="24" Width="50" Margin="5" />
            <TextBlock VerticalAlignment="Center" Text="Blue:" />
            <TextBox x:Name="txtBlue"
                Height="24" Width="50" Margin="5" />
        </StackPanel>

        <Button x:Name="btnTry" Content="Try Color"
            Grid.Row="1" Width="100" Height="24" />
        <Ellipse x:Name="ellipse" Grid.Row="2"
            Stroke="Black" StrokeThickness="5" Margin="20" />
        <TextBlock x:Name="lblColor" Grid.Row="2"
            HorizontalAlignment="Center" VerticalAlignment="Center"
            FontSize="20" FontFamily="Arial" FontWeight="Bold"  />

    </Grid>
</UserControl>
```

4. Now add the `Click` event to the `Button` control. Do this in the code behind, as explained earlier in this chapter.

```
namespace Ch4_TextBox
{
    public partial class Page : UserControl
    {
        public Page()
        {
            InitializeComponent();
            this.btnTry.Click += new RoutedEventHandler(btnTry_Click);
        }

        void btnTry_Click(object sender, RoutedEventArgs e)
        {

        }
    }
}
```

5. When the button is clicked, the application will change the `Fill` property of the `Ellipse` control, which expects a `SolidColorBrush`. You can create the `SolidColorBrush` using the `Colors.FromArgb()` method, which accepts four arguments: one for opacity, and one byte each for the red, green, and blue values. You will get the red, green, and blue values from the `TextBox` controls using the `Text` property.

```
void btnTry_Click(object sender, RoutedEventArgs e)
{
        this.ellipse.Fill = new SolidColorBrush(
            Color.FromArgb(
                255,
                byte.Parse(this.txtRed.Text),
                byte.Parse(this.txtGreen.Text),
                byte.Parse(this.txtBlue.Text)
            )
        );
}
```

6. Since the values for red, green, and blue must be an integer from 0 to 255, you can either check them or take the easy way out and just wrap your code in a `try/catch` block, and then report the error using the `TextBlock`. You'll go with the latter approach here. And to keep things clean, you will make sure the error message is cleared if all works correctly. Here is the updated code:

```
void btnTry_Click(object sender, RoutedEventArgs e)
{
    try
    {
        this.ellipse.Fill = new SolidColorBrush(
            Color.FromArgb(
                255,
                byte.Parse(this.txtRed.Text),
                byte.Parse(this.txtGreen.Text),
                byte.Parse(this.txtBlue.Text)
            )
        );

        this.lblColor.Text = "";
    }
    catch
    {
        this.lblColor.Text = "Error with R,G,B Values";
    }
}
```

7. Build and run the application to see what you get. Type **255**, **0**, and **0** in the Red, Green, and Blue text boxes, respectively, and then click the Try Color button. You should see the ellipse turn red. If, just for the fun of it, you leave one of the values blank or enter a value other than 0 through 255, you will see the error message.

Now that we have taken a quick look at the `TextBox` control, let's turn our attention to two other common controls: `CheckBox` and `RadioButton`.

Try It Out: Working with the RadioButton and CheckBox Controls

The following exercise will give you a first look at the `RadioButton` and `CheckBox` controls. You will build a simple survey, as shown in Figure 4-13.

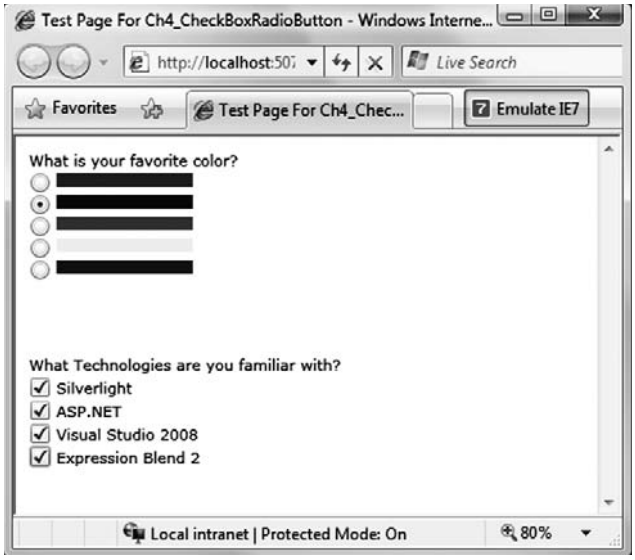

Figure 4-13. *Sample application using the RadioButton and CheckBox controls*

1. Create a new Silverlight application in Visual Studio 2008 and call it `Ch4_CheckBoxRadioButton`. Allow Visual Studio to create a Web Site project to host the application.

2. In the `Page.xaml` file, divide the root `Grid` into two rows. In each row, place a `StackPanel` with vertical orientation and a `Margin` property set to `10`.

```
<UserControl x:Class="Ch4_CheckBoxRadioButton.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">

    <Grid x:Name="LayoutRoot" Background="White">

        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>

        <StackPanel Orientation="Vertical" Grid.Row="0" Margin="10">
        </StackPanel>
```

```
        <StackPanel Orientation="Vertical" Grid.Row="1" Margin="10">
        </StackPanel>

    </Grid>

</UserControl>
```

The top row will be used to demonstrate the use of the `RadioButton` control, and the bottom row will feature the `CheckBox` control. Let's begin with the `RadioButton`.

The `RadioButton` control allows users to select only one selection out of a number of `RadioButton` controls that share the same group name. This is set using the `RadioButton`'s `Grouping` property.

Although you could simply type in each of the color choices for the radio buttons as text using the `Content` property, I thought it would be less boring to use colored rectangles instead. As we discussed earlier, one of the benefits of Silverlight 2 controls is that you can nest just about anything within the different controls. This is just another example of that flexibility.

3. Place five `RadioButton` controls in the first `StackPanel`, each with a `Rectangle` control of a different color. For the group name, use `FavoriteColor`. To make the content of the `RadioButton` controls display as left-justified, set the `HorizontalAlignment` property to `Left` for each one. Here is the code:

```
<StackPanel Orientation="Vertical" Grid.Row="0" Margin="10">

        <TextBlock Text="What is your favorite color?" />
        <RadioButton HorizontalAlignment="Left" GroupName="FavoriteColor">
            <Rectangle Width="100" Height="10" Fill="Red" />
        </RadioButton>
        <RadioButton HorizontalAlignment="Left" GroupName="FavoriteColor">
            <Rectangle Width="100" Height="10" Fill="Blue" />
        </RadioButton>
        <RadioButton HorizontalAlignment="Left" GroupName="FavoriteColor">
            <Rectangle Width="100" Height="10" Fill="Green" />
        </RadioButton>
        <RadioButton HorizontalAlignment="Left" GroupName="FavoriteColor">
            <Rectangle Width="100" Height="10" Fill="Yellow" />
        </RadioButton>
        <RadioButton HorizontalAlignment="Left" GroupName="FavoriteColor">
            <Rectangle Width="100" Height="10" Fill="Purple" />
       </RadioButton>

</StackPanel>
```

4. Next, do the same for the `CheckBox` controls in the bottom row, except here, just go the boring route and supply the choices as text. In addition, `CheckBox` controls are left-justified by default, and they do not need to be grouped. Here is the code for the `CheckBox` portion:

```
<StackPanel Orientation="Vertical" Grid.Row="1" Margin="10">

        <TextBlock Text="What Technologies are you familiar with?" />
        <CheckBox Content="Silverlight" />
        <CheckBox Content="ASP.NET" />
        <CheckBox Content="Visual Studio 2008" />
        <CheckBox Content="Expression Blend 2" />

</StackPanel>
```

5. Go ahead and run the solution to see the end result as it will appear in the browser. The output is shown in Figure 4-14. Notice that, as you would expect, you are able to select only one radio button at a time, but you can click as many check boxes as you wish.

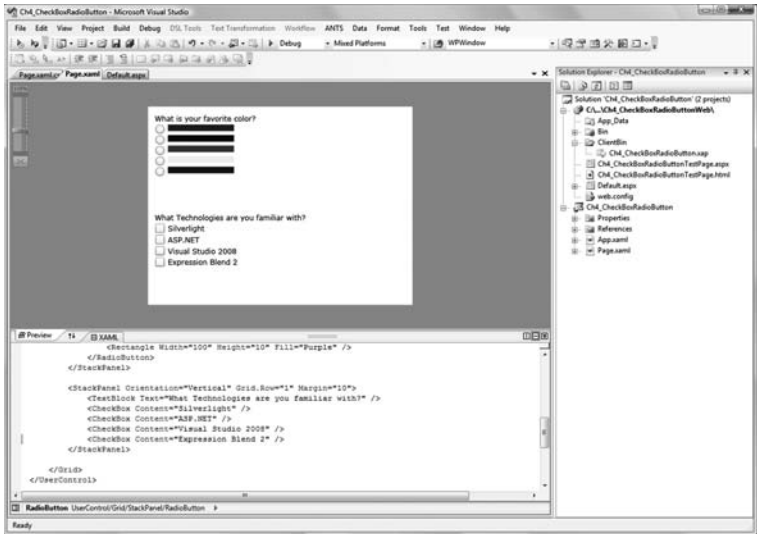

Figure 4-14. *Creating the RadioButton and CheckBox application*

Extended Controls

When a Silverlight application is deployed, it goes into an `.xap` file. This file will need to be downloaded by every client that accesses the Silverlight application.

A big benefit of Silverlight 2 is that the size of this `.xap` file is kept very small. One reason this file can be small is that the most commonly used controls are included in the Silverlight 2 Runtime, which is already present on every machine with Silverlight 2 installed.

However, Silverlight 2 provides a number of controls beyond this commonly used set of controls. These controls are included in two separate assemblies: `System.Windows.Controls.dll` and `System.Windows.Controls.Data.dll`. These dynamic link libraries (DLLs) will be included in the application `.xap` file only if the developer used a control from one of these extended control sets in that application.

Adding an Extended Control

When a developer uses a control from one of the other control libraries, an additional `xmlns` declaration will be added in the `UserControl` definition. This `xmlns` will have a prefix associated with it that will then be used to reference the individual controls.

For example, if you add a `DataGrid` to your Silverlight application in Visual Studio, your source will appear as follows:

```
<UserControl
    xmlns:data=
     "clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data"
    x:Class="SilverlightApplication1.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">
        <data:DataGrid></data:DataGrid>
    </Grid>
</UserControl>
```

Notice the additional `xmlns` declaration pointing to the `System.Windows.Controls` namespace within the `System.Windows.Controls.Data` assembly.

■ **Tip** To view which controls belong to which assemblies, first create a new Silverlight application and add a `DataGrid` and `GridSplitter` to the root `Grid`. Then select View ➤ Object Browser from the Visual Studio 2008 main menu. From the Object Browser's Browse drop-down list (in the top-left corner), select My Solution and browse the listing for three assemblies: `System.Windows`, `System.Windows.Controls.Data`, and `System.Windows.Controls`. Within each of those assemblies, drill down to the `System.Windows.Controls` namespace in order to see all of the controls that reside in that assembly.

Now we will work through an exercise using one of the controls in the `System.Windows.Controls` assembly.

Try It Out: Using the GridSplitter

One of the controls that resides in the `System.Windows.Controls` assembly is the `GridSplitter`. This control provides the ability for a user to change the width of a column or row in an application. If used properly, the `GridSplitter` can greatly improve the appearance of your application, as well as the user experience. In the following exercise, you will implement a simple `GridSplitter`.

1. Create a new Silverlight application in Visual Studio 2008 called `Ch4_GridSplitter`. Allow Visual Studio to create a Web Site project to host the application.

2. In the `Page.xaml` file, divide the root `Grid` into two columns. The first column should be 150 pixels in width, and the second should take up the remainder of the application. To be able to see what is going on in the grid, set `ShowGridLines` to `True`. Also add two `TextBlock` controls to the application: one in the first column and one in the second column. Your source should appear as follows:

```
<UserControl x:Class="Ch4_GridSplitter.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="150" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <TextBlock Text="Apress, Inc." />
        <TextBlock Grid.Column="1"
            Text="Beginning Silverlight 2 by Robert Lair" />
    </Grid>
</UserControl>
```

At this point, your Silverlight application should look like Figure 4-15.

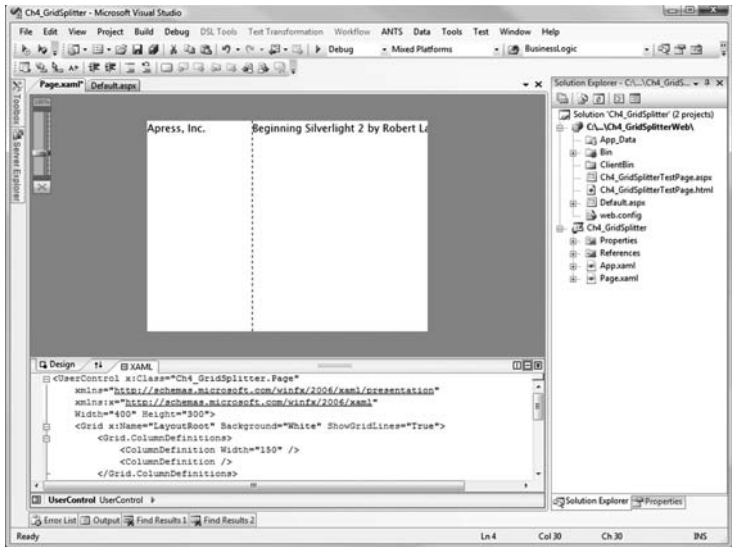

Figure 4-15. *The setup for the GridSplitter example*

Notice that you cannot see all of the text in the second column. Let's add a `GridSplitter` control to the application so users can resize the two columns to be able to view all the text in both columns.

3. Within the XAML, place the cursor just below the `TextBlock` definitions you added. Then, in the Visual Studio Toolbox, double-click the `GridSplitter` control. This will add the `xmlns` to the `System.Windows.Controls` assembly, and it will also add the `GridSplitter` to the application. Then set the Background property of the `GridSplitter` to `LightGray`. The source appears as follows:

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <TextBlock Text="Apress, Inc." />
    <TextBlock Grid.Column="1"
        Text="Beginning Silverlight 2 by Robert Lair" />

    <basics:GridSplitter Background="LightGray"></basics:GridSplitter>

</Grid>
```

4. You no longer need to see the grid lines, so remove the `ShowGridLines` property.

5. Run the application. It should look similar to Figure 4-16. Notice that you can now click and drag the `GridSplitter` to resize the two `Grid` columns.

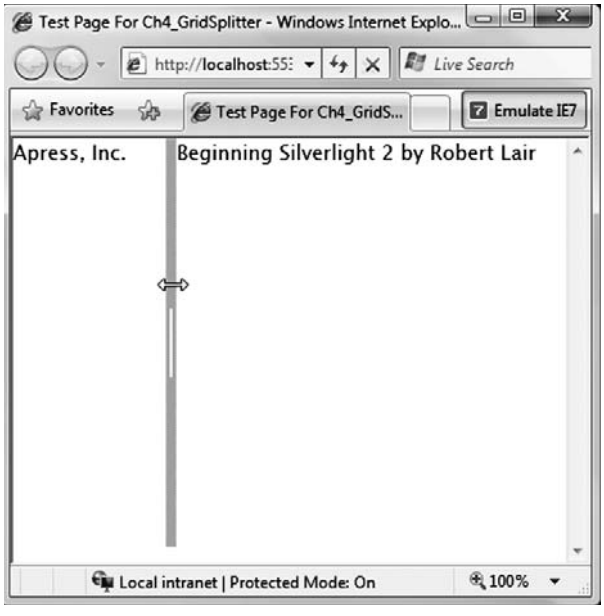

Figure 4-16. *The completed GridSplitter application*

As you can see, it's quite easy to gain the rich functionality of a grid splitter in your application with the Silverlight 2 `GridSplitter` control.

Summary

In this chapter, we took a brief look at some of the common form controls that are provided with Silverlight 2. The chapter was meant only as an introduction to the controls. We will be looking at these controls in more advanced capacities in the upcoming chapters.

In the next chapter, we will look at the Silverlight 2 list controls: `ListBox` and `DataGrid`.

CHAPTER 5

Data Binding and Silverlight List Controls

The previous chapter focused on the form controls contained in Silverlight 2. In this chapter, we will look at two controls that are made to display lists of data: the `ListBox` and `DataGrid`. These controls are typically bound to data through a technique known as *data binding*, which we'll explore first.

Data Binding

Through data binding, UI elements (called *targets*) are "bound" to data from a data source (called the *source*), as illustrated in Figure 5-1. When the data sources change, the UI elements bound to those data sources update automatically to reflect the changes. The data can come from different types of sources, and the target can be just about any UI element, including standard Silverlight 2 controls.

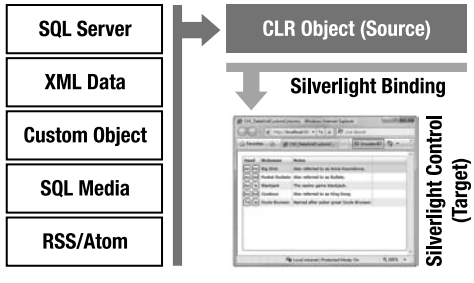

Figure 5-1. *Data binding in Silverlight*

Data binding simplifies application development. Since changes are reflected automatically, you do not need to manually update the UI elements. Also, by using data binding, you are able to separate the UI from the data in your application, which allows for a cleaner UI and easier maintenance.

The Binding Class

Data binding in Silverlight 2 is accomplished by using the `Binding` class. The `Binding` class has two components—the source and target—and a property that defines the way the two are bound, called the *binding mode*. The source is the data that is to be bound, the target is a property of the control that the data is to be bound to, and the mode defines how the data is passed between the source and the target (one-way, one-time, or two-way). You'll see how this works in the upcoming exercise.

To define the binding of a control's property, you use XAML markup extensions, such as `{Binding <path>}`. For example, to bind the `Text` property of a `TextBox` to a data source's `FirstName` element, you would use the following XAML:

```
<TextBox Text="{Binding FirstName }"  />
```

Try It Out: Simple Data Binding in Silverlight

To help explain data binding in Silverlight, let's build a very simple application. The application will include a `Book` object that contains two properties: `Title` and `ISBN`. These properties will be bound to two `TextBox` controls. Figure 5-2 shows the end result of the example.

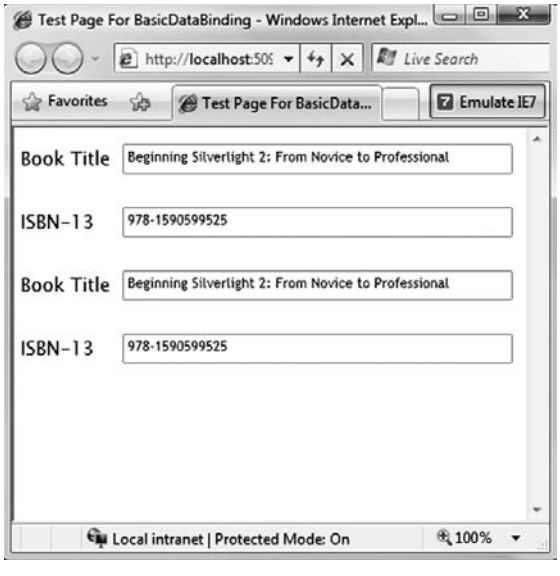

Figure 5-2. *Simple data binding example*

1. Create a new Silverlight application in Visual Studio 2008. Name the project `BasicDataBinding`, and allow Visual Studio to create a Web Site project to host your application.

2. Edit the `Page.xaml` file to define two columns and six grid rows. Place a `TextBlock` in each row in column 1 and a `TextBox` in each row in column 2. Also add some margins and some alignment assignments to improve the layout. The code for the page follows:

```
<UserControl x:Class="BasicDataBinding.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>

        <TextBlock Text="Book Title"
            VerticalAlignment="Center"
            Margin="5" />
        <TextBlock Text="ISBN-13"
            VerticalAlignment="Center"
            Margin="5"
            Grid.Row="1" />

        <TextBox Text="{Binding Title}"
            Height="24"
            Margin="5"
            Grid.Column="1" />
```

```
            <TextBox Text="{Binding ISBN}"
                Height="24"
                Margin="5"
                Grid.Column="1" Grid.Row="1" />

            <TextBlock Text="Book Title"
                VerticalAlignment="Center"
                Margin="5"
                Grid.Row="2" />
            <TextBlock Text="ISBN-13"
                VerticalAlignment="Center"
                Margin="5"
                Grid.Row="3" />

            <TextBox Text="{Binding Title}"
                Height="24"
                Margin="5"
                Grid.Column="1" Grid.Row="2" />
            <TextBox Text="{Binding ISBN}"
                Height="24"
                Margin="5"
                Grid.Column="1" Grid.Row="3" />

        </Grid>
</UserControl>
```

3. Next, edit the code behind, `page.xaml.cs`. Add a `Loaded` event handler for the application, which will fire when the application is loaded by the client. This is accomplished with the following source code:

```
public partial class Page : UserControl
{
    public Page()
    {
        InitializeComponent();
        this.Loaded += new RoutedEventHandler(Page_Loaded);
    }

    void Page_Loaded(object sender, RoutedEventArgs e)
    {

    }
}
```

4. Now you need to add a class to define a `Book` object. Below the `Page` class, add the following class definition:

```
namespace BasicDataBinding
{
    public partial class Page : UserControl
    {
        public Page()
        {
            InitializeComponent();
            this.Loaded += new RoutedEventHandler(Page_Loaded);
        }

        void Page_Loaded(object sender, RoutedEventArgs e)
        {

        }
    }

    public class Book
    {
        public string Title { get; set; }
        public string ISBN { get; set; }
    }
}
```

5. Now that you have `Book` defined, you need to create an instance of `Book` and set it to the `LayoutRoot`'s `DataContext`, as follows:

```
        void Page_Loaded(object sender, RoutedEventArgs e)
        {
            Book b = new Book() {
                Title = "Beginning Silverlight 2: From Novice to Professional",
                ISBN = "978-1590599525" };

            this.LayoutRoot.DataContext = b;
        }
```

When you set up binding definitions for different controls, the controls do not know where they are going to get their data. The `DataContext` property sets the data context for a control that is participating in data binding. The `DataContext` property can be set directly on the control. If a given control does not have a `DataContext` property specified, it will look to its parent for its data context. The nice thing about this model is that if you look above in the XAML for the page, you will see little indication of where the controls are getting their data. This provides an extreme level of code separation, allowing designers to design XAML UIs and developers to work alongside the designers, defining the specifics of how the controls are bound to their data sources.

6. At this point, you can go ahead and start debugging the application. If all goes well, you will see the four text boxes populated with the data from the `Book`'s instance (see Figure 5-2).

7. With the application running, change the book title in the first text box to just "Beginning Silverlight 2," by removing the "From Novice to Professional."

   You might expect that, since the third text box is bound to the same data, it will automatically update to reflect this change. However, a couple of things need to be done to get this type of two-way binding to work.

   One problem is that, currently, the `Book` class does not support notifying bound clients of changes to its properties. In other words, when a property changes in `Book`, the class will not notify the `TextBox` instances that are bound to the class of the change. You could take care of this by creating a change event for each property. This is far from ideal; fortunately, there is an interface that a class can implement that handles this for you. This interface is known as `INotifyPropertyChanged`. Let's use it.

8. Modify the `Book` class definition to inherit from `INotifyPropertyChanged`. Notice that when you inherit from `INotifyPropertyChanged`, you need to add `using System.ComponentModel`. Luckily, Visual Studio will help you with this, as shown in Figure 5-3.

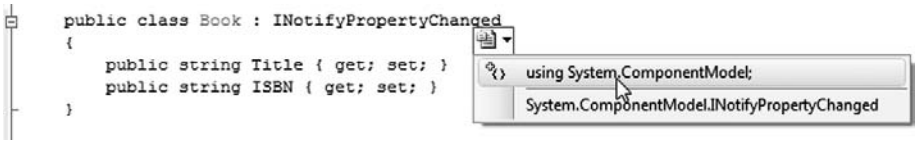

Figure 5-3. *Visual Studio assists when you need to add the System.ComponentModel namespace.*

9. Next, you can let Visual Studio do some more work for you. After adding the `using System.ComponentModel` statement, right-click `INotifyPropertyChanged` and choose the Explicitly implement interface `INotifyPropertyChanged` option, as shown in Figure 5-4.

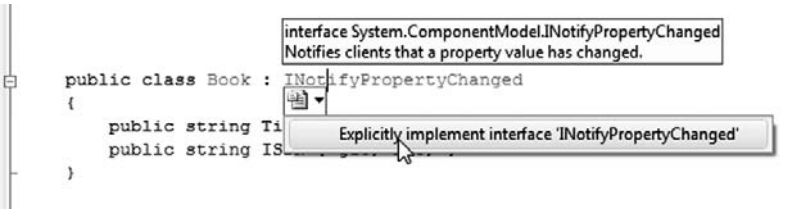

Figure 5-4. *Visual Studio also assists in implementing the INotifiyPropertyChanged interface.*

Now Visual Studio has added a new public event to your class:

```
public class Book : INotifyPropertyChanged
{
    public string Title { get; set; }
    public string ISBN { get; set; }

    #region INotifyPropertyChanged Members

    public event PropertyChangedEventHandler PropertyChanged;

    #endregion
}
```

10. Next, you need to create a convenience method that will fire the `PropertyChanged` event. Call it `FirePropertyChanged`, as shown in the following code.

```
public class Book : INotifyPropertyChanged
{
    public string Title { get; set; }
    public string ISBN { get; set; }

    #region INotifyPropertyChanged Members

    void FirePropertyChanged(string property)
    {
        if (PropertyChanged != null)
```

```
        {
            PropertyChanged(this,
                new PropertyChangedEventArgs(property));
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    #endregion
}
```

11. Now you need to extend the simplified properties by adding private members and full `get`/`set` definitions to define the `get` and `set` operations, as shown in the following code. The `get` is just like a normal `get` operation, where you simply return the internal member value. For the `set`, you first set the internal member value, and then call the `FirePropertyChanged` method, passing it the name of the property.

```
public class Book : INotifyPropertyChanged
{
    private string _title;
    private string _isbn;

    public string Title
    {
        get
        {
            return _title;
        }
        set
        {
            _title = value;
            FirePropertyChanged("Title");
        }
    }

    public string ISBN
    {
        get
        {
            return _isbn;
        }
        set
```

```
        {
            _isbn = value;
            FirePropertyChanged("ISBN");
        }
    }

    #region INotifyPropertyChanged Members

    void FirePropertyChanged(string property)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this,
                new PropertyChangedEventArgs(property));
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    #endregion
}
```

With this completed, your class is set up to notify bound clients of changes to the `Title` and `ISBN` properties. But you still need to take one more step. By default, when you bind a source to a target, the `BindingMode` is set to `OneWay` binding, which means that the source will send the data to the target, but the target will not send data changes back to the source. In order to get the target to update the source, you need to implement two-way (`TwoWay`) binding.

■ **Note** Earlier, I mentioned that there are three options for `BindingMode`. The third option is `OneTime` binding. In this mode, the values are sent to the target control property when the object is set to the `DataContext`. However, the values of the target property are not updated when the source value changes.

---

12. To change to two-way binding, add the `Mode=TwoWay` parameter when defining the `{Binding}` on a control, as follows:

```
<TextBlock Text="Book Title"
    VerticalAlignment="Center"
    Margin="5" />
```

```
<TextBlock Text="ISBN-13"
    VerticalAlignment="Center"
    Margin="5"
    Grid.Row="1" />

<TextBox Text="{Binding Title, Mode=TwoWay}"
    Height="24"
    Margin="5"
    Grid.Column="1" />
<TextBox Text="{Binding ISBN, Mode=TwoWay }"
    Height="24"
    Margin="5"
    Grid.Column="1" Grid.Row="1" />

<TextBlock Text="Book Title"
    VerticalAlignment="Center"
    Margin="5"
    Grid.Row="2" />
<TextBlock Text="ISBN-13"
    VerticalAlignment="Center"
    Margin="5"
    Grid.Row="3" />

<TextBox Text="{Binding Title, Mode=TwoWay }"
    Height="24"
    Margin="5"
    Grid.Column="1" Grid.Row="2" />
<TextBox Text="{Binding ISBN, Mode=TwoWay }"
    Height="24"
    Margin="5"
    Grid.Column="1" Grid.Row="3" />
```

13. Rebuild and run your application. Update any of the fields, and leave the focus on the control. You'll see that the two-way binding is triggered, and the corresponding field is also updated, as shown in Figure 5-5.

Figure 5-5. *Two-way binding in action*

Congratulations! You have just created a Silverlight application that allows for two-way data binding. We will now move on to look at data binding lists of data to the two list controls provided in Silverlight 2: `DataGrid` and `ListBox`.

The DataGrid Control

The data grid type of control has been around for ages and has been the primary choice for developers who need to display large amounts of data. The `DataGrid` control provided by Silverlight is not just a standard data grid, however. It contains a great deal of rich user functionality that, in the past, has been present only in third-party data grid components. For example, the Silverlight `DataGrid` handles resizing and reordering of grid columns.

Figure 5-6 shows an example of a very simple `DataGrid`, where the columns were automatically generated. Notice how the column titled Male is a check box. The `DataGrid` control has built-in intelligence to automatically show Boolean data types as check box cells.

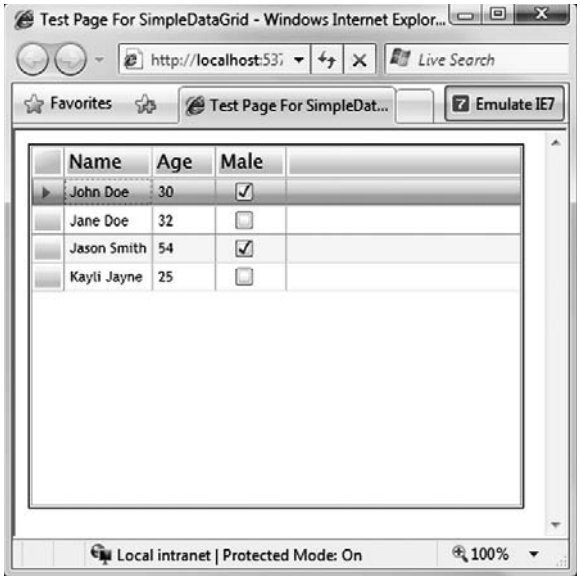

Figure 5-6. *A simple DataGrid example*

Try It Out: Building a Simple DataGrid

Let's run through a simple `DataGrid` example.

1. Create a new Silverlight application in Visual Studio 2008. Name the project `SimpleDataGrid`, and have Visual Studio create a hosting web site application for you.

2. Add the `DataGrid` to your application. To do this, simply add the `DataGrid` to the root `Grid` in your XAML, and set the `Margin` property to `10` to get some spacing around the grid. In addition, give the `DataGrid` the name `grid`. Note that, by default, the `Grid`'s `AutoGenerateColumns` property is set to `true`. If you were going to define the columns manually, you would want to set this property to `false`. However, since you want the grid to create the columns automatically, you can simply omit the property. The `DataGrid` definition follows:

```
<Grid x:Name="LayoutRoot" Background="White">
    <data:DataGrid x:Name="grid" Margin="10" />
</Grid>
```

■ **Note** So, why use `<data:DataGrid>`? As discussed in Chapter 4, the `DataGrid` is contained in an assembly called `System.Windows.Controls.Data`, which is not added to Silverlight applications by default. This way, if your application does not need any of the extended controls, the file size of your Silverlight application can be smaller. However, in order to add a `DataGrid` to your application, you need to reference the new assembly and add an `xmlns` reference to the assembly in the `UserControl` definition. As you might expect by now, Visual Studio can do all the work for you. To use this functionality in Visual Studio, drag the `DataGrid` control from the Toolbox to add it to your application. Visual Studio will add a new `xmlns` reference in the `UserControl` at the top of the `.xaml` page called `data`, which references the `System.Windows.Controls.Data` assembly. For the `DataGrid`, you will see the `xml` namespace referenced in the `DataGrid` definition `<data:DataGrid>`.

3. Next, build the class that will be bound to the `DataGrid`. Call the class `GridData` for simplicity, and give it three properties: `Name` (`string`), `Age` (`int`), and `Male` (Boolean). Also for simplicity, create a static method that will return an `ObservableCollection` containing some sample data that will be bound to the grid. In addition, define the class directly in the `page.xaml.cs` file. This is not really a good idea in the real world, but for the sake of an example, it will work just fine. Ideally, you will want to define your classes in separate files or even in completely separate projects and assemblies. The code for the `GridData` class follows:

```
namespace SimpleDataGrid
{
    public partial class Page : UserControl
    {
        public Page()
        {
            InitializeComponent();
        }
    }

    public class GridData
    {
        public string Name { get; set; }
        public int Age { get; set; }
        public bool Male { get; set; }

        public static ObservableCollection<GridData> GetData()
        {
            ObservableCollection<GridData> data =
                new ObservableCollection<GridData>();
```

```
            data.Add(new GridData() {
                Name = "John Doe",
                Age = 30,
                Male = true });

            data.Add(new GridData() {
                Name = "Jane Doe",
                Age = 32,
                Male = false});

            data.Add(new GridData() {
                Name = "Jason Smith",
                Age = 54,
                Male = true });

            data.Add(new GridData() {
                Name = "Kayli Jayne",
                Age = 25,
                Male = false });

            return data;
        }
    }
}
```

■ **Note** When you are binding a collection of data to a `DataGrid` or `ListBox`, you may be tempted to use the `List` generic class. The problem with using the `List` class is that it does not have built-in change notifications for the collection. In order to bind a `DataGrid` and `ListBox` to dynamic data that will be updated, you should use the `ObservableCollection` generic class. The `ObservableCollection` class represents a collection of dynamic data that provides built-in notification when items in the collection are added, removed, or refreshed.

4. Now that you have the XAML and the class defined, you can wire them up. To do this, first create an event handler for the `Loaded` event of the page, as follows:

```
public partial class Page : UserControl
{
    public Page()
    {
        InitializeComponent();
```

```
        this.Loaded += new RoutedEventHandler(Page_Loaded);
    }

    void Page_Loaded(object sender, RoutedEventArgs e)
    {

    }
}
```

5. When the page is loaded, you want to call `GetData()` from the `GridData` class and bind that to the `DataGrid`'s `ItemsSource` property, as follows:

```
public partial class Page : UserControl
{
    public Page()
    {
        InitializeComponent();
        this.Loaded += new RoutedEventHandler(Page_Loaded);
    }

    void Page_Loaded(object sender, RoutedEventArgs e)
    {
        this.grid.ItemsSource = GridData.GetData();
    }
}
```

6. Build and run the application. If all is well, you should see the `DataGrid` displayed (see Figure 5-6).

Let's take a few moments and play around with this `DataGrid` to explore some of its features. First of all, if you click any of the column headers, you will notice that sorting is automatically available, as shown in Figure 5-7.

Next, if you place your cursor at the edge of one of the columns, you can use the mouse to click and drag the column's edge to resize the column, as shown in Figure 5-8. Again, this functionality is provided for free with the `DataGrid`'s rich client-side functionality.

And finally, if you click and hold the mouse on one of the column headers, then drag it left or right to another column header's edge, you will see a little red triangle appear above the columns. For instance, click and drag the Name column so the little red triangle appears to the far right, as shown in Figure 5-9. When the red triangle is where you want it, release the mouse, and you will see that the Name column now appears as the last column in the `DataGrid`.

Figure 5-7. *Sorting in the DataGrid*

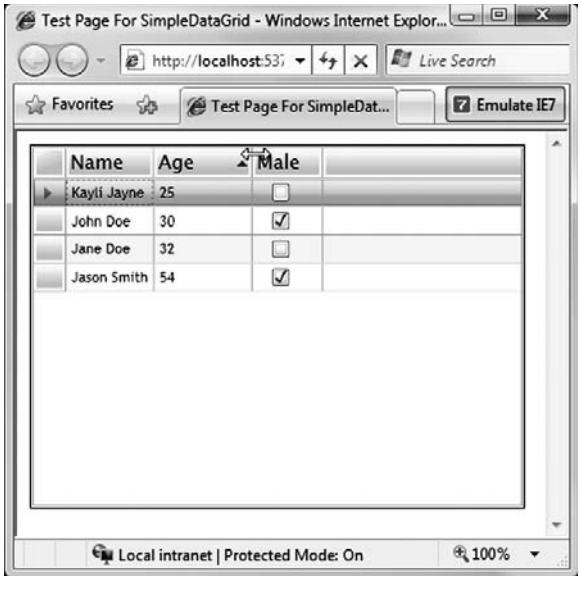

Figure 5-8. *Resizing columns in a DataGrid*

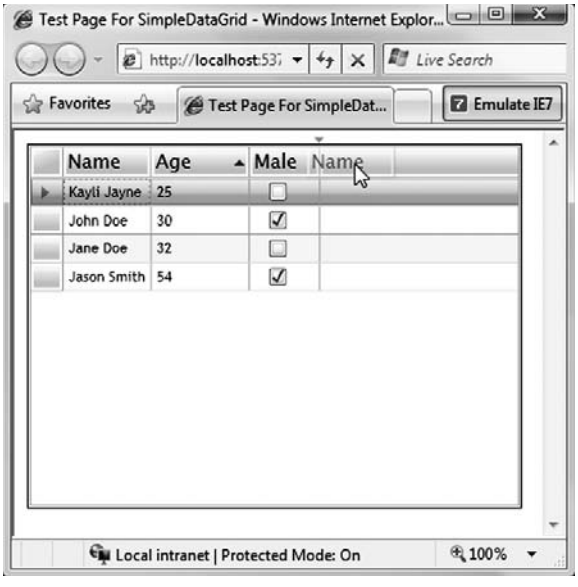

Figure 5-9. *Column reordering in action*

You'll agree that this is pretty nice out-of-the-box functionality for simply defining a `DataGrid` with this code:

```
<data:DataGrid x:Name="grid" Margin="10" />
```

Now that you have implemented a simple `DataGrid` example, let's explore some of the additional options available.

The Columns Collection

In the previous example, you allowed the `DataGrid` to automatically generate columns based on the data to which it was bound. This is not a new concept—it has been around in data grid components since the initial release of ASP.NET. But what if you want to have some additional control over the columns that are created in your `DataGrid`? What if you want to add a column that contains some more complex information, such as an image? You can do this by first setting the `AutoGenerateColumns` property on the grid to `false`. Then you need to generate the columns manually.

Columns are defined in a `DataGrid` using the `Columns` collection. The following is an example of setting the `Columns` collection in XAML. Notice that it sets the `AutogenerateColumns` property to `False`. If you neglect to do this, you will get all of the autogenerated columns in addition to the columns you define within the `Columns` collection.

```
<my:DataGrid x:Name="grid" Margin="10" AutoGenerateColumns="False">
    <my:DataGrid.Columns>

    </my:DataGrid.Columns>
</my:DataGrid>
```

You can place three types of columns within a `Columns` collection: a text column (`DataGridTextColumn`), a check box column (`DataGridCheckBoxColumn`), and a template column (`DataGridTemplateColumn`). All of the column types inherit from type `DataGridColumn`. A number of notable properties apply to all three column types, as shown in Table 5-1.

Table 5-1. *DataGridColumn Properties*

Property	Description
`CanUserReorder`	Turns on and off the ability for the user to drag columns to reorder them
`CanUserResize`	Turns on or off the ability for the user to resize the column's width with the mouse
`DisplayIndex`	Determines the order in which the column appears in the `DataGrid`
`Header`	Defines the content of the column's header
`IsReadOnly`	Determines if the column can be edited by the user
`MaxWidth`	Sets the maximum column width in pixels
`MinWidth`	Sets the minimum column width in pixels
`Visibility`	Determines whether or not the column will be visible to the user
`Width`	Sets the width of the column, or can be set to automatic sizing mode

DataGridTextColumn

The `DataGridTextColumn` defines a column in your grid for plain text. This is the equivalent to `BoundColumn` in the ASP.NET `DataGrid`. The primary properties that can be set for a `DataGridTextColumn` are the `Header`, which defines the text that will be displayed in the columns header, and the `DisplayMemberBinding` property, which defines the property in the data source bound to the column.

The following example defines a text column with the header `Name` and is bound to the data source's `Name` property.

```
<my:DataGrid x:Name="grid" Margin="10" AutoGenerateColumns="False">
    <my:DataGrid.Columns>
        <my:DataGridTextColumn
                Header="Name"
                DisplayMemberBinding="{Binding Name}" />
            </my:DataGrid.Columns>
</my:DataGrid>
```

DataGridCheckBoxColumn

As you would expect, the DataGridCheckBoxColumn contains a check box. If you have data that you want to display as a check box in your grid, this is the control to use. Here is an example of the DataGridCheckBoxColumn that contains the header Male? and is bound to the data source's Male property:

```
<my:DataGrid x:Name="grid" Margin="10" AutoGenerateColumns="False">
    <my:DataGrid.Columns>
        <my:DataGridCheckBoxColumn
                Header="Male?"
                DisplayMemberBinding="{Binding Male}" />
            </my:DataGrid.Columns>
</my:DataGrid>
```

DataGridTemplateColumn

If you want data in your grid column that is not plain text and is not a check box, the DataGridTemplateColumn provides a way for you to define the content for your column. The DataGridTemplateColumn contains a CellTemplate and CellEditingTemplate, which determine what content is displayed, depending on whether the grid is in normal view mode or in editing mode.

Note that while you get features such as automatic sorting in the other types of DataGrid columns, that is not true of the DataGridTemplateColumn. These columns will need to have additional logic in place to allow for sorting.

Let's consider an example that has two fields: FirstName and LastName. Suppose that when you are in normal view mode, you want the data to be displayed side by side in TextBlock controls. However, when the user is editing the column, you want to display two TextBox controls that allow the user to edit the FirstName and LastName columns independently.

```
<my:DataGridTemplateColumn Header="Name">
    <my:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
           <StackPanel Orientation="Horizontal">
               <TextBlock Padding="5,0,5,0"
                   Text="{Binding FirstName}"/>
               <TextBlock Text="{Binding LastName}"/>
           </StackPanel>
        </DataTemplate>
    </my:DataGridTemplateColumn.CellTemplate>
    <my:DataGridTemplateColumn.CellEditingTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBox Padding="5,0,5,0"
                    Text="{Binding FirstName}"/>
```

```
                <TextBox Text="{Binding LastName}"/>
            </StackPanel>
        </DataTemplate>
    </my:DataGridTemplateColumn.CellEditingTemplate>
</my:DataGridTemplateColumn>
```

Now that we have covered the basics of manually defining the grids in a `DataGrid`, let's try it out.

Try It Out: Building a DataGrid with Custom Columns

I thought it would be fun to build a `DataGrid` that contains a list of starting hands in poker. If you have ever watched poker on TV, you most likely heard the players refer to things like "pocket rockets" and "cowboys." These are simply nicknames they have given to starting hands. The `DataGrid` you are going to build in this example will look like Figure 5-10.

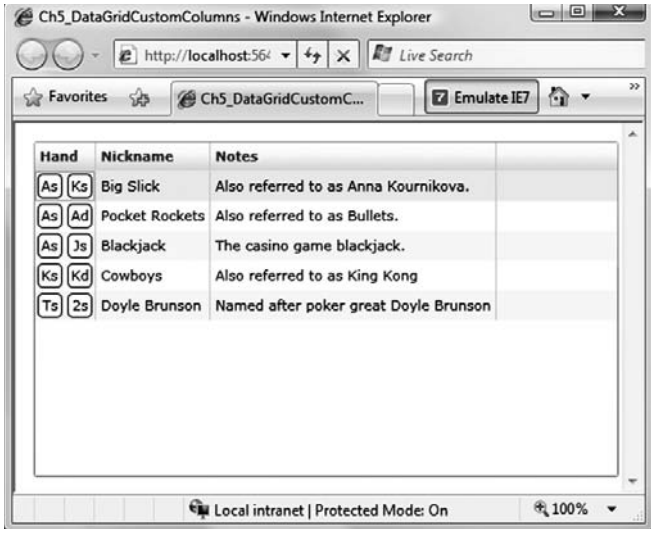

Figure 5-10. *DataGrid with custom columns*

1. Create a new Silverlight application called `Ch5_DataGridCustomColumns`. Allow Visual Studio to create a Web Site project to host the application.

2. After the project is loaded, right-click the `Ch5_DataGridCustomColumns` project and select Add New Item. In the Add New Item dialog box, select Class for the template, and name the class `StartingHands.cs`, as shown in Figure 5-11. Click the Add button to add the class to the project.

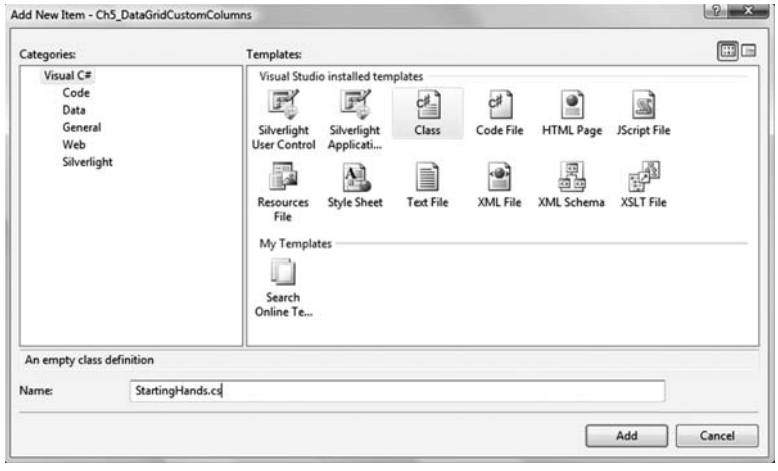

Figure 5-11. *Adding a new class to the Silverlight project*

3. Now define the `StartingHands` class. The class will contain four properties: `Nickname` (`string`), `Notes` (`string`), `Card1` (`string`), and `Card2` (`string`). Also create a static method in the class called `GetHands()`, which returns an `ObservableCollection` of `StartingHands` instances. The code follows:

```
using System;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using System.Collections.ObjectModel;

namespace Ch5_DataGridCustomColumns
{
    public class StartingHands
    {
        public string Nickname { get; set; }
        public string Notes { get; set; }
        public string Card1 { get; set; }
        public string Card2 { get; set; }
```

```
public static ObservableCollection<StartingHands> GetHands()
{
    ObservableCollection<StartingHands> hands =
    new ObservableCollection<StartingHands>();

    hands.Add(
        new StartingHands()
        {
            Nickname = "Big Slick",
            Notes = "Also referred to as Anna Kournikova.",
            Card1 = "As",
            Card2 = "Ks"
        });

    hands.Add(
    new StartingHands()
    {
        Nickname = "Pocket Rockets",
        Notes = "Also referred to as Bullets.",
        Card1 = "As",
        Card2 = "Ad"
    });

    hands.Add(
        new StartingHands()
        {
            Nickname = "Blackjack",
            Notes = "The casino game blackjack.",
            Card1 = "As",
            Card2 = "Js"
        });

    hands.Add(
        new StartingHands()
        {
            Nickname = "Cowboys",
            Notes = "Also referred to as King Kong",
            Card1 = "Ks",
            Card2 = "Kd"
        });
```

```
            hands.Add(
                new StartingHands()
                {
                    Nickname = "Doyle Brunson",
                    Notes = "Named after poker great Doyle Brunson",
                    Card1 = "Ts",
                    Card2 = "2s"
                });


            return hands;
        }
    }
}
```

4. Now that the class is built, in the `Page.xaml` file, change the width of the `UserControl` to be `500` and add a `DataGrid` named `grdData` to the root `Grid` by double-clicking the `DataGrid` control in the Toolbox. Add a 15-pixel margin around the `DataGrid` for some spacing, and set the `AutoGenerateColumns` property to `False`. The code follows:

```
<UserControl
    xmlns:data="clr-namespace:System.Windows.Controls; ➥
assembly=System.Windows.Controls.Data"
    x:Class="Ch5_DataGridCustomColumns.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="500" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">
        <data:DataGrid Margin="15" AutoGenerateColumns="False"></data:DataGrid>
    </Grid>
</UserControl>
```

5. Next, define the columns in the `DataGrid`. To do this, add the `DataGrid.Columns` collection, as follows:

```
<data:DataGrid x:Name="grdData" Margin="15" AutoGenerateColumns="False">
    <data:DataGrid.Columns>

    </data:DataGrid.Columns>
</data:DataGrid>
```

6. Referring back to Figure 5-10, the first column in the `Grid` contains the two cards in the hand. To build this, you use a `DataGridTemplateColumn`. Within the `DataGridTemplateColumn`, add a `CellTemplate` containing a `Grid` with two columns, each containing a `Border`, `Rectangle`, and `TextBlock`, which will overlap each other. Bind the two `TextBlock` controls to the `Card1` and `Card2` properties from the data source. Enter the following code:

```
<data:DataGrid x:Name="grdData" Margin="15" AutoGenerateColumns="False">
    <data:DataGrid.Columns>
        <data:DataGridTemplateColumn Header="Hand">
            <data:DataGridTemplateColumn.CellTemplate>
                <DataTemplate>
                    <Grid>

                        <Grid.ColumnDefinitions>
                            <ColumnDefinition />
                            <ColumnDefinition />
                        </Grid.ColumnDefinitions>

                        <Border
                            Margin="2" CornerRadius="4"
                            BorderBrush="Black" BorderThickness="1" />
                        <Rectangle
                            Margin="4" Fill="White" Grid.Column="0" />
                        <Border
                            Margin="2" CornerRadius="4" BorderBrush="Black"
                            BorderThickness="1" Grid.Column="1" />
                        <Rectangle
                            Margin="4" Fill="White" Grid.Column="1" />
                        <TextBlock
                            Text="{Binding Card1}" HorizontalAlignment="Center"
                            VerticalAlignment="Center" Grid.Column="0" />
                        <TextBlock
                            Text="{Binding Card2}" HorizontalAlignment="Center"
                            VerticalAlignment="Center" Grid.Column="1" />

                    </Grid>
                </DataTemplate>
            </data:DataGridTemplateColumn.CellTemplate>
        </data:DataGridTemplateColumn>
    </data:DataGrid.Columns>
</data:DataGrid>
```

7. Again, referring back to Figure 5-10, the next two columns contain the nickname of the starting hand and notes about the starting hand. To implement this, use two `DataGridTextColumn` columns. Set the `Headers` of the columns to `Nickname` and `Notes` accordingly.

```
<data:DataGrid x:Name="grdData" Margin="15" AutoGenerateColumns="False">
    <data:DataGrid.Columns>
        <data:DataGridTemplateColumn Header="Hand">
            <data:DataGridTemplateColumn.CellTemplate>
                <DataTemplate>
                    <Grid>

                        <Grid.ColumnDefinitions>
                            <ColumnDefinition />
                            <ColumnDefinition />
                        </Grid.ColumnDefinitions>

                        <Border
                            Margin="2" CornerRadius="4"
                            BorderBrush="Black" BorderThickness="1" />
                        <Rectangle
                            Margin="4" Fill="White" Grid.Column="0" />
                        <Border
                            Margin="2" CornerRadius="4" BorderBrush="Black"
                            BorderThickness="1" Grid.Column="1" />
                        <Rectangle
                            Margin="4" Fill="White" Grid.Column="1" />
                        <TextBlock
                            Text="{Binding Card1}" HorizontalAlignment="Center"
                            VerticalAlignment="Center" Grid.Column="0" />
                        <TextBlock
                            Text="{Binding Card2}" HorizontalAlignment="Center"
                            VerticalAlignment="Center" Grid.Column="1" />

                    </Grid>
                </DataTemplate>
            </data:DataGridTemplateColumn.CellTemplate>
        </data:DataGridTemplateColumn>

        <data:DataGridTextColumn
            Header="Nickname"
            Binding="{Binding Nickname}"  />
```

```
            <data:DataGridTextColumn
                Header="Notes"
                Binding="{Binding Notes}" />

        </data:DataGrid.Columns>
    </data:DataGrid>
```

8. Finally, wire up the controls to the data source. To do this, navigate to the `page.xaml.cs` file and add an event handler to the `Page Loaded` event. Within that `Loaded` event, simply set the `DataGrid`'s `ItemsSource` property equal to the return value of the `StartingHands.GetHands()` static method. Here's the code:

```
public partial class Page : UserControl
{
    public Page()
    {
        InitializeComponent();
        this.Loaded += new RoutedEventHandler(Page_Loaded);
    }

    void Page_Loaded(object sender, RoutedEventArgs e)
    {
        this.grdData.ItemsSource = StartingHands.GetHands();
    }
}
```

9. Compile and run your application. If all goes well, your application should appear as shown earlier in Figure 5-10.

This completes our `DataGrid` with custom columns example. Naturally, in a real-world application, you would be getting the data for these hands from an external data source, such as a web service or an XML file. We will be looking at that in Chapter 6. Now, let's take a look at the `ListBox` control.

The ListBox Control

In the past, the list box type of control has been considered one of the common controls in programming—no more special than a drop-down list. However, in Silverlight 2, this has all changed. The `ListBox` is perhaps one of the most flexible controls used to display lists of data. In fact, referring back to ASP.NET controls, the Silverlight 2 `ListBox` is more a cousin to the `DataList` control than the ASP.NET `ListBox` control. Let's take a peek at this powerful control.

Default and Custom ListBox Items

If we wire up the `ListBox` to our `Person` data from our earlier `DataGrid` example, you will see that, by default, the `ListBox` really is just a standard `ListBox`.

```
<ListBox Margin="10" x:Name="list" DisplayMemberPath="Name" />
```

One additional property you may have noticed in this `ListBox` definition is `DisplayMemberPath`. If you are defining a simple text-based `ListBox`, the `ListBox` needs to know which data member to display. Since the `Person` class contains three properties (`Name`, `Age`, and `Male`), we need to tell it that we want the `Name` to be displayed. Figure 5-12 shows the results.

Figure 5-12. *A simple default ListBox*

However, the `ListBox` control can contain much more than plain text. In fact, if you define a custom `ItemTemplate` for the `ListBox`, you can present the items in a more interesting way. Here's an example using the same `Person` data:

```
<ListBox Margin="10" x:Name="list" DisplayMemberPath="Name">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Margin="5" Orientation="Vertical">
                <TextBlock
                    FontSize="17"
                    FontWeight="Bold"
                    Text="{Binding Name}" />
```

```
                <StackPanel Margin="5,0,0,0" Orientation="Horizontal">
                    <TextBlock Text="Age: " />
                    <TextBlock Text="{Binding Age}" />
                    <TextBlock Text=", Male: " />
                    <TextBlock Text="{Binding Male}" />
                </StackPanel>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Figure 5-13 shows how this custom `ListBox` appears in a browser.

Figure 5-13. *A customized ListBox example*

Try It Out: Building a ListBox with Custom Content

Let's take the same data that displayed poker starting hands from the previous exercise and see what type of cool `ListBox` you can build with it. Figure 5-14 shows the custom `ListBox` you'll create in this exercise.

Figure 5-14. *The customized ListBox application*

1. Start out by creating a new Silverlight application called `Ch5_ListBoxCustom` and allow Visual Studio to create a hosting web site.

2. You will use the same class that you built in the earlier `DataGrid` exercise. Right-click the Silverlight project, choose Add Existing Item, and browse to `StartingHands.cs` to add that class to the project.

3. When you add the existing `StartingHands.cs` class, it is in a different namespace than your current project. You can reference that namespace by adding a `using` statement at the top of your Silverlight application, or you can just change the namespace, as follows:

```
namespace Ch5_ListBoxCustom
{
    public class StartingHands
    {
        public string Nickname { get; set; }
        public string Notes { get; set; }
```

```
        public string Card1 { get; set; }
        public string Card2 { get; set; }

        ...
    }
}
```

4. Next, you need to define the `ListBox`'s `ItemTemplate`. The `ItemTemplate` will contain a horizontal-oriented `StackPanel` including the grid to display the two cards. It will also include a nested vertical-oriented `StackPanel` that will contain two `TextBlock` controls to display the `Nickname` and `Notes` data. Here is the code:

```
<Grid x:Name="LayoutRoot" Background="White">
    <ListBox Margin="10" x:Name="list">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel Margin="5" Orientation="Horizontal">
                    <Grid>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition />
                            <ColumnDefinition />
                        </Grid.ColumnDefinitions>

                        <Border
                            Margin="2" CornerRadius="4"
                            BorderBrush="Black" BorderThickness="1" />
                        <Rectangle Margin="4" Fill="White"
                            Grid.Column="0" Width="20" />
                        <Border
                            Margin="2" CornerRadius="4" BorderBrush="Black"
                            BorderThickness="1" Grid.Column="1" />
                        <Rectangle Margin="4" Fill="White"
                            Grid.Column="1" Width="20" />
                        <TextBlock
                            Text="{Binding Card1}" HorizontalAlignment="Center"
                            VerticalAlignment="Center" Grid.Column="0" />
                        <TextBlock
                            Text="{Binding Card2}" HorizontalAlignment="Center"
                            VerticalAlignment="Center" Grid.Column="1" />
                    </Grid>
```

```
                    <StackPanel Orientation="Vertical">
                        <TextBlock
                            Text="{Binding Nickname}"
                            FontSize="16"
                            FontWeight="Bold" />
                        <TextBlock
                            Text="{Binding Notes}" />
                    </StackPanel>
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</Grid>
```

5. The only thing left to do is to wire up the `ListBox` to the data source. To do this, navigate to the `page.xaml.cs` code behind, and add an event handler for the `Page Loaded` event. Then, within that `Loaded` event handler, add the following code to set the `ListBox`'s `ItemsSource` to the return value from the `StartingHands.GetHands()` method, as you did earlier in the `DataGrid` example.

```
namespace Ch5_ListBoxCustom
{
    public partial class Page : UserControl
    {
        public Page()
        {
            InitializeComponent();
            this.Loaded += new RoutedEventHandler(Page_Loaded);
        }

        void Page_Loaded(object sender, RoutedEventArgs e)
        {
            list.ItemsSource = StartingHands.GetHands();
        }
    }
}
```

6. Run the application. If all goes well, you will see the `ListBox` shown in Figure 5-14.

As you can see, the `ListBox` control's flexibility lets developers display lists of data in some very cool ways.

Summary

In this chapter, we looked at how to bind lists of data to Silverlight controls. Then we focused on two controls typically bound to data: the `DataGrid` control and the `ListBox` control. You saw how these controls are flexible and can show data in unique ways. However, in all of these examples, the classes contained static data. In real-world examples, the data that you will normally list in a `DataGrid` or `ListBox` will be coming from some external data source, such as an XML file or a web service. In the next chapter, we will look at how to get data from these external data sources and how to use that data to bind to your Silverlight applications.

CHAPTER 6

Data Access and Networking

Data access in Silverlight applications works differently than it does in traditional applications. You'll need to be aware of how it works and the limitations. In this chapter, we will look at what makes data access different, and then explore mechanisms for accessing data in a Silverlight application.

Data Access in Silverlight Applications

As discussed in Chapter 1, RIAs bridge the gap between Windows-based smart clients and web-based applications. When moving to this type of environment, data access and networking can be confusing.

In a Windows-based smart client, the application has access to the database at all times. It can create a connection to the database, maintain state with the database, and remain connected.

On the other hand, a web application is what is known as a *pseudo-conversational* environment, which is, for the most part, a completely stateless and disconnected environment. When a client makes a request to the web server, the web server processes the request and returns a response to the client. After that response has been sent, the connection between the client and the server is disconnected, and the server moves on to the next client request. No connection or state is maintained between the two.

In Silverlight applications, we have one additional layer of complexity. The application runs from the client's machine; however, it is still a disconnected environment, because it is hosted within a web browser. There is no concept of posting back for each request or creating a round-trip to the server for data processing. Therefore, data access is limited to a small number of options.

In addition, a Silverlight application has a number of security restrictions placed on it to protect the users from the application gaining too much control over their machine. For instance, the Silverlight application has access to only an isolated storage space to store its disconnected data. It has no access whatsoever to the client's hard disk outside its "sandbox." Silverlight's isolated storage is discussed in more detail in Chapter 7.

So what are your options for accessing data in a Silverlight application? The following main mechanisms are available:

- The most common mechanism to access data from a Silverlight application is through web services, typically a WCF service.
- Silverlight applications can access data using ADO.NET Data Services, which provides access to data through a URI syntax.
- Silverlight 2 also has built-in socket support, which allows applications to connect directly to a server through TCP sockets.
- Silverlight 2 has out-of-the-box support for JavaScript Object Notation (JSON), as well as RSS 2.0 and Atom 1.0 syndication feed formats.

Of these mechanisms, we'll explore accessing WCF services from Silverlight 2 in depth, and then have a high-level look at using sockets. For examples and more information on accessing other data services, refer to *Pro Silverlight 2 in C# 2008* by Matthew MacDonald (Apress, 2008).

Accessing Data Through Web Services

One of the ways that a Silverlight application can access data is through web services. These can be ASP.NET Web Services (ASMX), Windows Communication Foundation (WCF) services, or representational state transfer (REST) services. Here, we will concentrate on using a WCF service, which is the preferred way of accessing data in a Silverlight application through web services.

Try It Out: Accessing Data Through a WCF Service

To demonstrate accessing data from a WCF service, we will build the same application that we built in Chapter 5 to try out the `DataGrid`. (For more information about any part of this exercise regarding the `DataGrid`, refer back to Chapter 5.) The difference will be that the application will get the data through a web service.

As you'll recall, this application displays common starting hands in poker and the nicknames that have been given to those starting hands. The UI will have three columns: the first column will display two images of the cards in the hand, the second column will display the nickname, and the third column will contain notes about the hand. The completed application is shown in Figure 6-1.

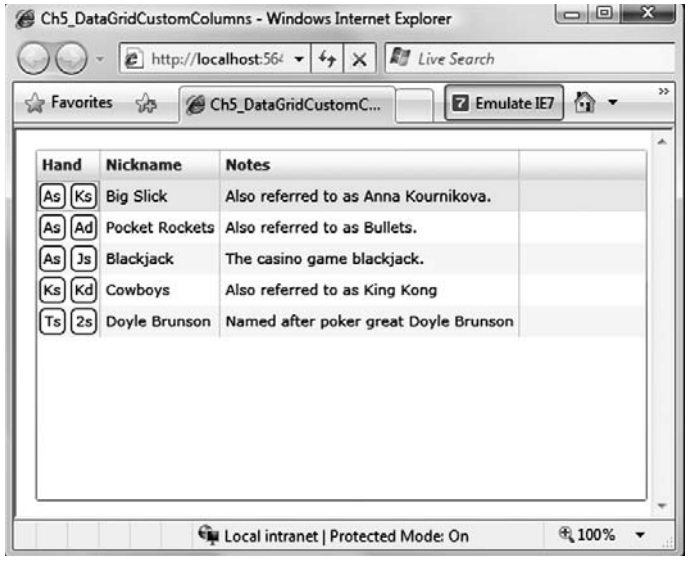

Figure 6-1. *The poker starting hands application*

1. Create a new Silverlight application in Visual Studio 2008. Call the application `Ch6_WCFService`, and allow Visual Studio to create a Web Application project named `Ch6_WCFService.Web` to host your application, as shown in Figure 6-2.

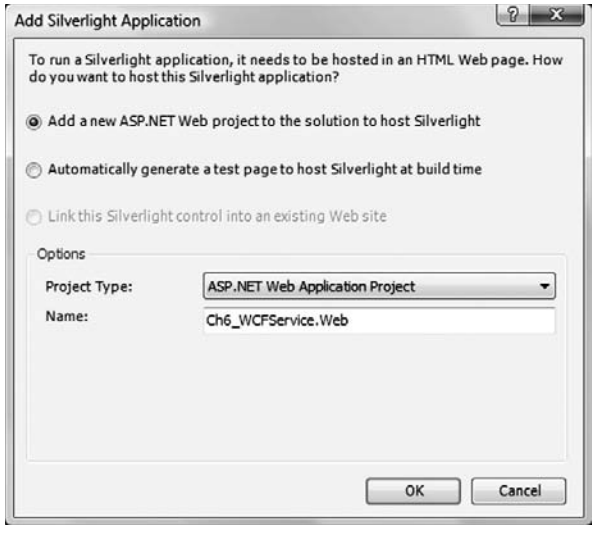

Figure 6-2. *Adding the Silverlight application hosting project*

2. Right-click the `Ch6_WCFService.Web` project and select Add ➤ Class. Name the new class `StartingHands.cs`, as shown in Figure 6-3.

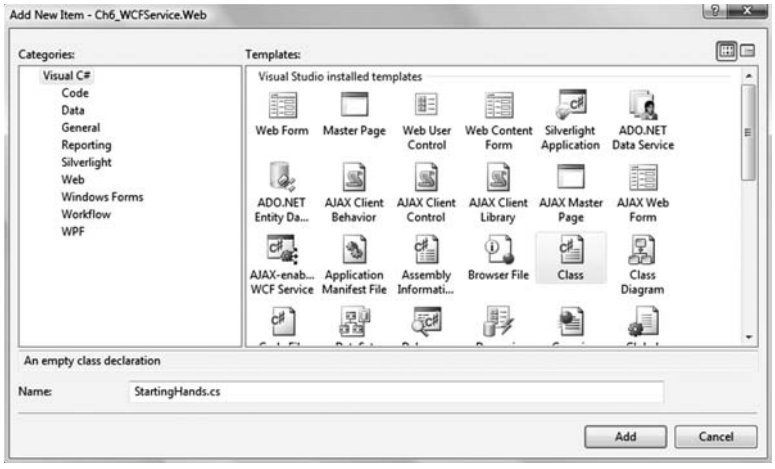

Figure 6-3. *Adding the StartingHands.cs class to the project*

3. Now you need to implement the `StartingHands.cs` class. It is very similar to the class used in Chapter 5's `DataGrid` example. To save yourself some typing, you can copy the code from that project. As shown in bold in the following code, the only differences are the namespace and the return type of the `GetHands()` method. Instead of using an `ObservableCollection`, it will return a simple `List<StartingHands>`.

■ **Note** In a real-world example, the `StartingHands.cs` class would be doing something like retrieving data from a SQL Server database and executing some business logic rules on the data. For simplicity, this example just returns a static collection.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace Ch6_WCFService.Web
{
    public class StartingHands
    {
        public string Nickname { get; set; }
        public string Notes { get; set; }
```

```
public string Card1 { get; set; }
public string Card2 { get; set; }

public static List<StartingHands> GetHands()
{
    List<StartingHands> hands = new List<StartingHands>();

    hands.Add(
        new StartingHands()
        {
            Nickname = "Big Slick",
            Notes = "Also referred to as Anna Kournikova.",
            Card1 = "As",
            Card2 = "Ks"
        });

    hands.Add(
    new StartingHands()
    {
        Nickname = "Pocket Rockets",
        Notes = "Also referred to as Bullets.",
        Card1 = "As",
        Card2 = "Ad"
    });

    hands.Add(
        new StartingHands()
        {
            Nickname = "Blackjack",
            Notes = "The casino game blackjack.",
            Card1 = "As",
            Card2 = "Js"
        });

    hands.Add(
        new StartingHands()
        {
            Nickname = "Cowboys",
            Notes = "Also referred to as King Kong",
            Card1 = "Ks",
            Card2 = "Kd"
        });
```

```
            hands.Add(
                new StartingHands()
                {
                    Nickname = "Doyle Brunson",
                    Notes = "Named after poker great Doyle Brunson",
                    Card1 = "Ts",
                    Card2 = "2s"
                });


            return hands;
        }
    }
}
```

4. Next, you need to add the WCF service that will call the `StartingHands.GetHands()` method. Right-click the `Ch6_WCFService.Web` project and select Add ➤ New Item. In the Add New Item dialog box, select the template named "Silverlight-enabled WCF Service" and name it `StartingHandService.svc`, as shown in Figure 6-4. Then click the Add button.

Figure 6-4. *Adding the Silverlight-enabled WCF service*

5. This will add a service named `StartingHandService.svc` to the project with an attached code-behind file named `StartingHandService.svc.cs`. View that code behind. You will see that Visual Studio has already created the base WCF service, including a sample method called `DoWork()`, as follows:

```
namespace Ch6_WCFService.Web
{
    [ServiceContract(Namespace = "")]
    [AspNetCompatibilityRequirements(RequirementsMode =
        AspNetCompatibilityRequirementsMode.Allowed)]
    public class StartingHandService
    {
        [OperationContract]
        public void DoWork()
        {
            // Add your operation implementation here
            return;
        }

        // Add more operations here and mark them
        // with [OperationContract]
    }
}
```

6. Replace the `DoWork()` method with a `GetHands()` method that returns a `List<StartingHands>` collection, as follows:

```
namespace Ch6_WCFService.Web
{
    [ServiceContract(Namespace = "")]
    [AspNetCompatibilityRequirements(RequirementsMode =
        AspNetCompatibilityRequirementsMode.Allowed)]
    public class StartingHandService
    {
        [OperationContract]
        public List<StartingHands> GetHands();

        // Add more operations here and mark them
        // with [OperationContract]
    }
}
```

This method simply returns the results from calling the `StartingHands.GetHands()` method.

7. Now that you have a Silverlight-enabled WCF service, you need to add a reference in your Silverlight project so that your Silverlight application can access the service. To do this, right-click References within the `Ch6_WCFService` in Solution Explorer and select Add Service Reference, as shown in Figure 6-5. This brings up the Add Service Reference dialog box.

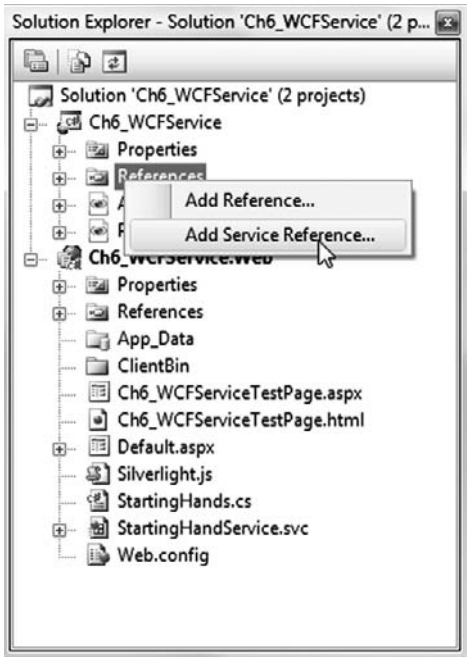

Figure 6-5. *Choosing to add a service reference*

8. In the Add Service Reference dialog box, click the down arrow next to the Discover button and select Services in Solution, as shown in Figure 6-6.

9. Visual Studio will find the `StartingHandService.svc` and will populate the Services list in the Add Service Reference dialog box. Expand the `StartingHandService.svc` node to show the `StartingHandService`. Click `StartingHandService` to see the `GetHands()` web method in the Operations listing, as shown in Figure 6-7. Enter `StartingHandServiceReference` as the Namespace field, and then click OK to continue.

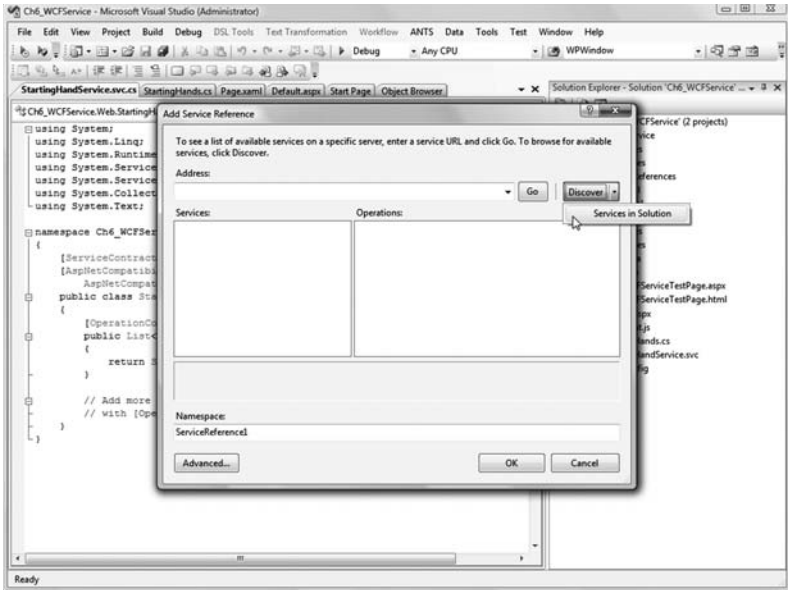

Figure 6-6. *Finding the services in the solution*

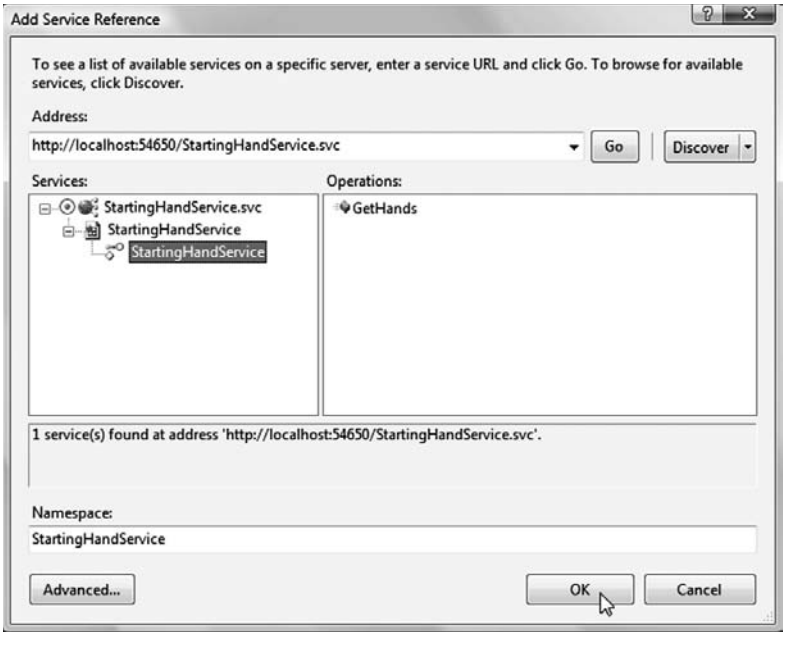

Figure 6-7. *Adding a service reference for StartingHandService*

10. Open the Visual Studio Object Browser by selecting View ➤ Object Browser from the main menu. Navigate to the `Ch6_WCFService` entry and expand the tree. You will find `Ch6_WCFService.StartingHandService` under your project. Within that, you will see an object named `StartingHandServiceClient`. Select this object to examine it, as shown in Figure 6-8.

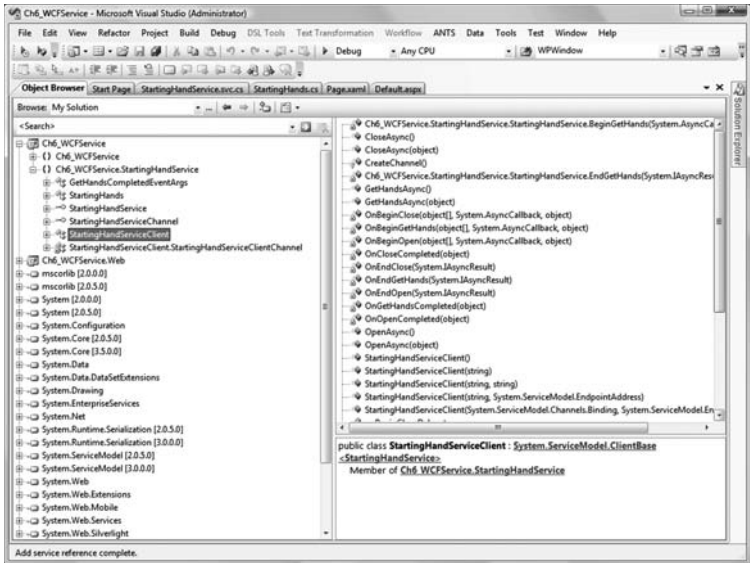

Figure 6-8. *Object Browser for StartingHandService*

11. Look at the members listed on the right side of the Object Browser. There are a number of items that are added, but take specific note of the method named `GetHandsAsync()` and the event named `GetHandsCompleted`. You will need to use both of these in order to call your web service from Silverlight.

12. Now it's time to create the Silverlight application's UI. Open the `Page.xaml` file in Visual Studio. Place the cursor within the root `Grid` and double-click the `DataGrid` control in the Toolbox. This adds the following XAML:

```
<UserControl
    xmlns:data="clr-namespace:System.Windows.Controls; ➥
assembly=System.Windows.Controls.Data"
    x:Class="Ch6_WCFService.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
```

```
    <Grid x:Name="LayoutRoot" Background="White">
        <data:DataGrid></data:DataGrid>
    </Grid>
</UserControl>
```

13. Highlight the `DataGrid` definition in the solution and replace it with the following `DataGrid` definition, which is from the previous `DataGrid` exercise in Chapter 5. The `DataGrid` contains three columns: one template column containing the two cards in the hand, and two text columns containing the nickname and notes about the hand.

```
<data:DataGrid x:Name="grdData" Margin="15" AutoGenerateColumns="False">
    <data:DataGrid.Columns>
        <data:DataGridTemplateColumn Header="Hand">
            <data:DataGridTemplateColumn.CellTemplate>
                <DataTemplate>
                    <Grid>

                        <Grid.ColumnDefinitions>
                            <ColumnDefinition />
                            <ColumnDefinition />
                        </Grid.ColumnDefinitions>

                        <Border
                            Margin="2" CornerRadius="4"
                            BorderBrush="Black" BorderThickness="1" />
                        <Rectangle
                            Margin="4" Fill="White" Grid.Column="0" />
                        <Border
                            Margin="2" CornerRadius="4" BorderBrush="Black"
                            BorderThickness="1" Grid.Column="1" />
                        <Rectangle
                            Margin="4" Fill="White" Grid.Column="1" />
                        <TextBlock
                            Text="{Binding Card1}" HorizontalAlignment="Center"
                            VerticalAlignment="Center" Grid.Column="0" />
                        <TextBlock
                            Text="{Binding Card2}" HorizontalAlignment="Center"
                            VerticalAlignment="Center" Grid.Column="1" />

                    </Grid>
                </DataTemplate>
```

```
            </data:DataGridTemplateColumn.CellTemplate>
        </data:DataGridTemplateColumn>

        <data:DataGridTextColumn
            Header="Nickname"
            Binding="{Binding Nickname}"  />
        <data:DataGridTextColumn
            Header="Notes"
            Binding="{Binding Notes}" />

    </data:DataGrid.Columns>
</data:DataGrid>
```

14. Save the `Page.xaml` file and navigate to the code behind for the application, located in the `page.xaml.cs` file. Wire up the `Loaded` event handler for the page, as follows:

```
namespace Ch6_WCFService
{
    public partial class Page : UserControl
    {
        public Page()
        {
            InitializeComponent();
            this.Loaded += new RoutedEventHandler(Page_Loaded);
        }

        void Page_Loaded(object sender, RoutedEventArgs e)
        {
            throw new NotImplementedException();
        }
    }
}
```

Next, you need to call the WCF service. In Silverlight 2, web services can be called only asynchronously, so the browser's execution is not blocked by the transaction. In order to do this, you need to get an instance of the service reference (commonly referred to as the *web service proxy class*) named `StartingHandService`, which you added earlier. You will then wire up an event handler for the service's `GetHandsCompleted` event, which you examined in the Object Browser (in step 11). This is the event handler that will be called when the service has completed execution. Finally, you will execute the `GetHandsAsync()` method.

■ **Tip** In a real-world scenario, you will want to present the user with a progress bar or animation while the service is being called, since the duration of a web service call can be lengthy.

15. Within the `Page_Loaded` event handler, first obtain an instance of `StartingHandService`. Then, in the `GetHandsCompleted` event handler, bind the `ItemsSource` of the `DataGrid` to the result returned from the service call, as shown in the following code. Note that normally you will want to check the result to make certain that the web service call was successful, and alert the user accordingly in case of failure.

```
using Ch6_WCFService.StartingHandService;

namespace Ch6_WCFService
{
    public partial class Page : UserControl
    {
        public Page()
        {
            InitializeComponent();
            this.Loaded += new RoutedEventHandler(Page_Loaded);
        }

        void Page_Loaded(object sender, RoutedEventArgs e)
        {
            StartingHandServiceClient service = new StartingHandServiceClient();
            service.GetHandsCompleted += new
             EventHandler<GetHandsCompletedEventArgs>(service_GetHandsCompleted);
            service.GetHandsAsync();
        }

       void service_GetHandsCompleted(object sender, GetHandsCompletedEventArgs e)
        {
            this.grdData.ItemsSource = e.Result;
        }
    }
}
```

16. Test your application. If all goes well, you should see the populated `DataGrid`, as shown earlier in Figure 6-1.

This example demonstrated how to use the Silverlight-enabled WCF service provided in Visual Studio to allow your Silverlight application to access data remotely. As noted earlier, this is one of the most common approaches to data access with Silverlight.

Using a Standard WCF Service with Silverlight

It is very possible to use a standard WCF service with Silverlight, instead of the provided Silverlight-enabled WCF service. Selecting the WCF Service project type in Visual Studio 2008 will add an interface and web service to the solution that you need to implement. This is perfectly valid for Silverlight 2 applications, but you must be careful with the binding.

WCF uses `wsHttpBinding` as its default binding. In the `web.config` file, you will notice that the service endpoint element has an attribute `binding="wsHttpBinding"`. Silverlight 2, on the other hand, supports only basic binding. Therefore, in order for your WCF service to work in your application, you will need to modify the binding attribute to be `binding="basicHttpBinding"`.

Accessing Services from Other Domains

In the previous example, the web service was on the same domain as your Silverlight application. What if you want to call a service that is on a different domain?

In fact, as a best practice, it is preferred to have your web services stored on a domain separate from your web application. So even for applications where you control both the web service and the Silverlight application, you may be dealing with different domains.

If you attempt to access a service from a different domain in Silverlight, you will notice that it fails. This is because, by default, a Silverlight application cannot call services that are on a different domain, unless it is permitted to do so by the service host. In order for Silverlight to determine if it has permission to access a service on a certain domain, it will look for one of two files in the root of the target domain: `clientaccesspolicy.xml` or `crossdomain.xml`.

First, Silverlight will look for a file named `clientaccesspolicy.xml` in the domain's root. This is Silverlight's client-access policy file. If you are publishing your own services that you want to be accessible by Silverlight applications, this is the file that you want to use, as it provides the most options for Silverlight application policy permissions. The following is a sample `clientaccesspolicy.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="*"/>
```

```
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

The important elements are `<allow-from>` and `<grant-to>`. The `<allow-from>` element defines which domains are permitted to access the resources specified in the `<grant-to>` element.

If Silverlight cannot find a `clientaccesspolicy.xml` file at the root of the domain from which you are attempting to access a service, it will then look for a file named `crossdomain.xml` in the root. This is the XML policy file that has been used to provide access for Flash applications to access cross-domain services, and Silverlight supports this file as well. The following is an example of a `crossdomain.xml` file:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
   SYSTEM "http://www.macromedia.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-http-request-headers-from domain="*" headers="*"/>
</cross-domain-policy>
```

Again, even though Silverlight supports `crossdomain.xml`, using `clientaccesspolicy.xml` for Silverlight applications is the preferred and best practice.

Accessing Data Through Sockets

In the majority of cases, your Silverlight applications will access data through web services. However, Silverlight provides another mechanism that, though rarely used, can be quite powerful. This mechanism is socket communications. In this section, we will look at a greatly simplified example of communicating with a server via sockets and TCP. The main purpose here is to give you a taste of using sockets in Silverlight so you have a basic understanding of the process and can consider whether you would like to take this approach. If so, you can refer to a more advanced resource, such as *Pro Silverlight 2 in C# 2008* by Matthew MacDonald (Apress 2008).

For our example, let's assume that we have a socket server running at the IP address 192.168.1.100 on port 4500. The socket server simply accepts text inputs and does something with them. In Silverlight, we want to connect to that socket server and send it text from a `TextBox` control.

First, we make a connection to the socket server. To do this, we create an instance of a `System.Net.Sockets.Socket` object for IP version 4 (`AddressFamily.InterNetwork`). The type will be `Stream`, meaning it will accept a stream of bytes, and the protocol will be TCP.

```
Socket socket;
socket = new Socket(
    AddressFamily.InterNetwork,
    SocketType.Stream,
    ProtocolType.Tcp);
```

We need to execute the socket's `ConnectAsync()` method, but first we must create an instance of `SocketAsyncEventArgs` to pass to the method, using a statement similar to the following:

```
SocketAsyncEventArgs socketArgs = new SocketAsyncEventArgs()
{
    RemoteEndPoint = new IPEndPoint(
        IPAddress.Parse("192.168.1.100"),
        4500)
};
```

This statement sets the target for the socket connection as 192.168.1.100 on port 4500.

In addition, since this is an asynchronous connection, we need to have notification when the connection has been established. To get this notification, we wire up an event handler to be triggered on the `SocketAsyncEventArgs.Completed` event. Once we have that wired up, we simply call the `ConnectAsync()` method, passing it our `SocketAsyncEventArgs` instance.

```
socketArgs.Completed += new
    EventHandler<SocketAsyncEventArgs>(socketArgs_Completed);
socket.ConnectAsync(socketArgs);
```

The method for this event handler will first remove the event handler, and then it will examine the response from the socket server. If it is successful, it will send a stream of bytes from our `TextBox` control to the socket server through our established connection.

```
void socketArgs_Completed(object sender, SocketAsyncEventArgs e)
{
    e.Completed -= socketArgs_Completed;

    if (e.SocketError == SocketError.Success)
    {
        SocketAsyncEventArgs args = new SocketAsyncEventArgs();
        args.SetBuffer(bytes, 0, bytes.Length);
        args.Completed += new EventHandler<SocketAsyncEventArgs>(OnSendCompleted);
        socket.SendAsync(args);
    }
}
```

Once again, since the calls to the socket are asynchronous, we wire up another event handler called `OnSendCompleted`, which will fire when our `SendAsync()` method is completed. This event handler will do nothing more than close the socket.

```
void OnSendCompleted(object sender, SocketAsyncEventArgs e)
{
    socket.Close();
}
```

Although this seems pretty simple, it is complicated by client-access policy permissions. In the same way that a Silverlight application can call a web service on a separate domain only if it has the proper client-access policy permissions, a Silverlight application can call a socket server only if that server contains the proper client-access policy permissions. The following is an example of a client-access policy for a socket server:

```
<?xml version="1.0" encoding ="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from>
        <domain uri="*" />
      </allow-from>
      <grant-to>
        <socket-resource port="4500-4550" protocol="tcp" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

Recall that when you're using a web service, the client-access policy is contained in a file named `clientaccesspolicy.xml`, which is placed in the domain's root. In a socket access situation, things are a bit more complex.

Before Silverlight will make a socket request to a server on whatever port is requested by the application, it will first make a socket request of its own to the server on port 943, requesting a policy file. Therefore, your server must have a socket service set up to listen to requests on port 943 and serve up the contents of the client-access policy in order for Silverlight applications to be able to make a socket connection.

Summary

In this chapter, we focused on accessing data from your Silverlight applications through WCF services. We also discussed accessing data from different domains and cross-domain policy files. In addition, we looked at using sockets in Silverlight 2 from a high level.

In the next chapter we will look at local storage within Silverlight.

CHAPTER 7

Local Storage in Silverlight

Localized storage in Silverlight 2 is handled by its *isolated storage* feature, which is a virtual file system that can be used to store application data on the client's machine. As just a few examples, you might use local storage in your application to store user settings, undo information, shopping cart contents, or a local cache for your commonly used objects. Implementations of this feature are really limited only by your imagination.

In this chapter, we will explore Silverlight's isolated storage. We walk through building a virtual storage explorer to view the directories and files contained within isolated storage for an application. In addition, we will look at the isolated storage quota and how to increase the quota size for your Silverlight 2 applications.

Working with Isolated Storage

Storing application information has always been a challenge for developers of traditional web applications. Often, implementing such storage means storing information in cookies or on the server, which requires using a postback to retrieve the data. In the case of desktop applications, implementing storage for application information is significantly easier, as developers have more access to the user's hard drive. Once again, Silverlight 2 bridges the gap between desktop applications and web applications by offering isolated storage.

Using the Silverlight classes for working with isolated storage, you can not only store settings locally, but also create files and directories, as well as read and write files within isolated storage.

Using the Isolated Storage API

The classes for accessing isolated storage are contained within the `System.IO.IsolatedStorage` namespace. This namespace contains the following three classes:

- `IsolatedStorageFile`
- `IsolatedStorageFileStream`
- `IsolatedStorageSettings`

We'll look at each class to see what it represents.

IsolatedStorageFile

The `IsolatedStorageFile` class represents the isolated storage area, and the files and directories contained within it. This class provides the majority of the properties and methods used when working with isolated storage in Silverlight 2. As an example, in order to get an instance of the user's isolated storage for a given application, use the static method `GetUserStoreForApplication()`, as follows:

```
using (var store = IsolatedStorageFile.GetUserStoreForApplication())
{
    //...
}
```

Once the storage instance has been retrieved, a number of operations are available, including `CreateDirectory()`, `CreateFile()`, `GetDirectoryNames()`, and `GetFileNames()`. Also, the class has properties such as `Quota` and `AvailableFreeSpace`. The following example creates a directory in isolated storage called `Directory1`, and then it retrieves the total and available free space in isolated storage:

```
using (var store = IsolatedStorageFile.GetUserStoreForApplication())
{
    store.CreateDirectory("Directory1");
    long quota = store.Quota;
    long availableSpace = store.AvailableFreeSpace;
}
```

IsolatedStorageFileStream

The `IsolatedStorageFileStream` class represents a given file. It is used to read, write, and create files within isolated storage. The class extends the `FileStream` class, and in most cases, developers will use a `StreamReader` and `StreamWriter` to work with the stream. As an

example, the following code creates a new file named `TextFile.txt` and writes a string to the file:

```
using (var store = IsolatedStorageFile.GetUserStoreForApplication())
{
    IsolatedStorageFileStream stream = store.CreateFile("TextFile.txt");
    StreamWriter sw = new StreamWriter(stream);
    sw.Write("Contents of the File);
    sw.Close();
}
```

IsolatedStorageSettings

The `IsolatedStorageSettings` class allows developers to store key/value pairs in isolated storage. The key/value pairs are user-specific and provide a very convenient way to store settings locally. The following example demonstrates storing the user's name in `IsolatedStorageSettings`.

```
public partial class Page : UserControl
{
    private IsolatedStorageSettings isSettings =
        IsolatedStorageSettings.ApplicationSettings;

    public Page()
    {
        InitializeComponent();
        this.Loaded += new RoutedEventHandler(Page_Loaded);
        this.cmdSave.Click += new RoutedEventHandler(cmdSave_Click);
    }

    void cmdSave_Click(object sender, RoutedEventArgs e)
    {
        isSettings["name"] = this.txtName.Text;
        SetWelcomeMessage();
    }

    void Page_Loaded(object sender, RoutedEventArgs e)
    {
        SetWelcomeMessage();
    }
```

```
    private void SetWelcomeMessage()
    {
        if (isSettings.Contains("name"))
        {
            string name = (string)isSettings["name"];
            this.txtWelcome.Text = "Welcome " + name;
        }
        else
        {
            txtWelcome.Text =
                "Welcome! Enter Your Name and Press Save.";
        }
    }
}
```

The first time users access the application, they will see the message "Welcome! Enter Your Name and Press Save." They can then enter their name and click the Save Name button. The name will be saved in local storage under the key/value pair called `name`. The next time the user accesses the application, his name will still be stored in local storage, and he will see the friendly welcome message, as shown in Figure 7-1.

Figure 7-1. *Saving a user's name with IsolatedStorageSettings*

Now that we have briefly looked at some of the key classes associated with Silverlight 2's isolated storage, let's try building an application that uses this storage.

Try It Out: Creating a File Explorer for Isolated Storage

In this example, you will create a file explorer that will allow a user to navigate through an application's virtual storage within Silverlight's isolated storage. The file explorer will allow users to view, modify, and create new files within the given directories. Keep in mind that a Silverlight application has its own isolated storage, so the file explorer will be unique to the application. The end result will appear as shown in Figure 7-2.

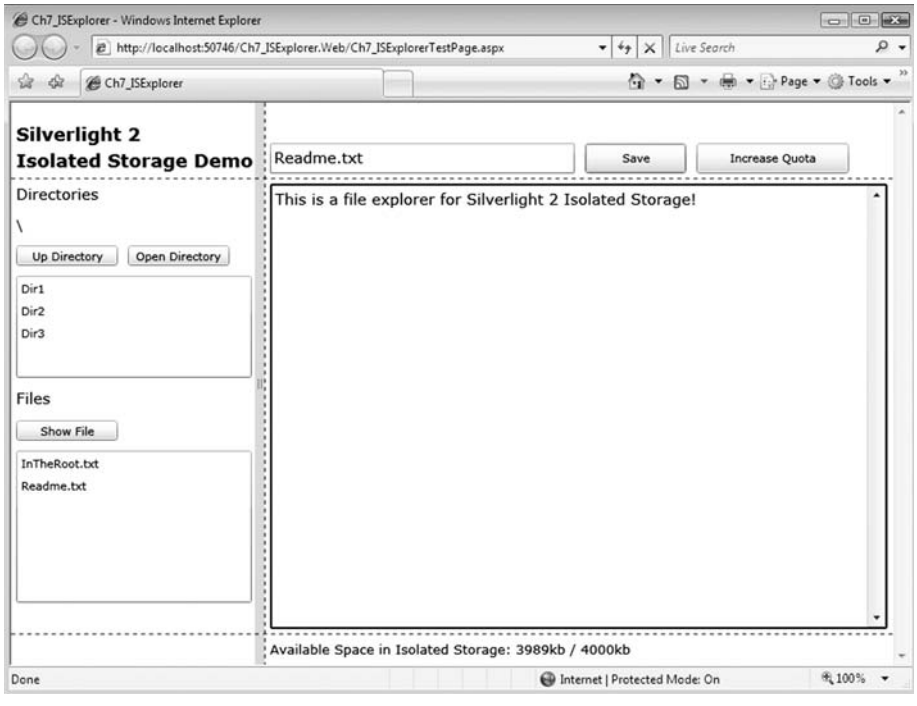

Figure 7-2. *The isolated storage file explorer demo*

Creating the Application Layout

Let's get started by setting up the application layout.

1. Create a new Silverlight application in Visual Studio 2008. Name it `Ch7_ISExplorer` and allow Visual Studio to create an ASP.NET web site called `Ch7_ISExplorer.Web` to host your application.

2. When the project is created, you should be looking at the `Page.xaml` file. If you do not see the XAML source, switch to that view so that you can edit the XAML. The application should take up the entire browser window, so begin by removing the `Width` and `Height` properties from your base `UserControl`.

```
<UserControl x:Class="Ch7_ISExplorer.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White">

    </Grid>
</UserControl>
```

3. Next, define a `Grid` for the form layout. Add two columns and three rows to the `Grid`. Set the `Width` property of the first column to `250`. Set the `Height` property of the top row to `75` and the bottom row to `30`. Also, in order to better see your `Grid` layout, set the `ShowGridLines` property to `True`.

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="250" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="75" />
        <RowDefinition />
        <RowDefinition Height="30" />
    </Grid.RowDefinitions>
</Grid>
```

4. Run your application. It should look like Figure 7-3.

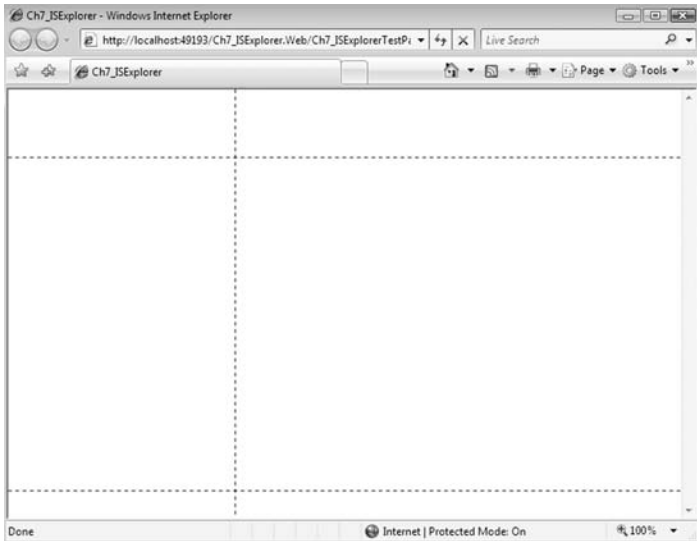

Figure 7-3. *The grid layout of the file explorer application*

5. Next, add a `GridSplitter` to allow the user to resize the left and right columns. Set the `Grid.RowSpan` to `3` and `HorizontalAlignment` to `Right`.

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="250" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="75" />
        <RowDefinition />
        <RowDefinition Height="30" />
    </Grid.RowDefinitions>

    <basics:GridSplitter
        Grid.RowSpan="3"
        HorizontalAlignment="Right" />
</Grid>
```

Now you will start filling the `Grid` cells with controls. You will add quite a few controls, using nested `StackPanel` components to assist in getting the desired layout. These controls have been discussed in detail in Chapters 4 and 5, and you can refer back to those chapters for more information about any of the controls used here.

6. In `Grid.Row` and `Grid.Column (0,0)`, place a `StackPanel` that contains a couple cosmetic `TextBlock` controls that will serve as your application title, as follows (with some of the existing code omitted for brevity):

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">
    ...
    <basics:GridSplitter …

    <StackPanel
        VerticalAlignment="Bottom"
        Orientation="Vertical"
        Margin="5">

        <TextBlock
            FontSize="18"
            FontWeight="Bold"
            Text="Silverlight 2">
        </TextBlock>
        <TextBlock
```

```
            FontSize="18"
            FontWeight="Bold"
            Text="Isolated Storage Demo">
        </TextBlock>

    </StackPanel>
</Grid>
```

Referring to Figure 7-2, you will notice that the content is divided into two sections: one for directories (top) and one for files (bottom). Let's first take care of the section for directories.

7. In `Grid.Row` and `Grid.Column (1,0)`, place another `StackPanel`, which spans two rows, with a couple `TextBlock` controls, three `Button` controls, and two `ListBox` controls. The XAML should appear as follows (again, with some of the source code omitted, but the changes are shown):

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">
    ...
    <basics:GridSplitter ...

    <StackPanel
        VerticalAlignment="Bottom"
        Orientation="Vertical"
        Margin="5">

        <TextBlock
            FontSize="18"
            FontWeight="Bold"
            Text="Silverlight 2">
        </TextBlock>
        <TextBlock
            FontSize="18"
            FontWeight="Bold"
            Text="Isolated Storage Demo">
        </TextBlock>

    </StackPanel>

    <StackPanel
        Grid.Row="1"
        Grid.RowSpan="2"
        Orientation="Vertical">
```

```
        <TextBlock
            FontSize="15"
            Text="Directories"
            Margin="5">
        </TextBlock>

        <TextBlock
            x:Name="lblCurrentDirectory"
            FontSize="13"
            Text="Selected Directory"
            Margin="5">
        </TextBlock>

        <StackPanel Orientation="Horizontal">
            <Button
                x:Name="btnUpDir"
                Margin="5"
                Click="btnUpDir_Click"
                Content="Up Directory"
                Width="100"
                Height="20" />
            <Button
                x:Name="btnOpenDir"
                Margin="5"
                Click="btnOpenDir_Click"
                Content="Open Directory"
                Width="100"
                Height="20" />
        </StackPanel>

        <ListBox Height="150"
            x:Name="lstDirectoryListing"
            Margin="5,5,13,5">
        </ListBox>
    </StackPanel>
</Grid>
```

First is a simple cosmetic `TextBlock` for the section title. This is followed by the `TextBlock` named `lblCurrentDirectory`, which will be filled with the current directory. As the users navigate through the directories, it will be important to inform them which directory they are in.

Next are two `Button` controls (`btnUpDir` and `btnOpenDir`), which will be used for navigating through the directories. This is simplified into two basic tasks: moving up a directory and opening the currently selected directory. To get the buttons to appear visually as desired, they are contained in a `StackPanel` with horizontal orientation.

The final `ListBox` will be populated with directories named `lstDirectoryListing`. As the users navigate through the directories using the `btnUpDir` and `btnOpenDir` buttons, this `ListBox` will be repopulated automatically with the directories contained in the user's current location.

8. Next, still within `Grid.Row` and `Grid.Column (1,0)`, add the files section, as follows:

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">

    ...
        <ListBox Height="100"
            x:Name="lstDirectoryListing"
            Margin="5,5,13,5">
        </ListBox>

        <TextBlock
            FontSize="15"
            Text="Files"
            Margin="5">
        </TextBlock>

        <StackPanel Orientation="Horizontal">
            <Button
                x:Name="btnOpenFile"
                Margin="5"
                Click="btnOpenFile_Click"
                Content="Show File"
                Width="100"
                Height="20" />
        </StackPanel>

        <ListBox Height="150"
            x:Name="lstFileListing"
            Margin="5,5,13,5">
        </ListBox>

    </StackPanel>
</Grid>
```

As with the previous section, the first `TextBlock` holds the section title. Next is a `Button` control called `btnOpenFile`. Notice that even though there is only one button, it is still placed within a `StackPanel` for consistency. In the future, if you want to extend this application—for example, to add file deletion functionality—you may want to add buttons to this `StackPanel`. This is purely user preference; the `StackPanel` really was not required in this instance.

Finally, you have the `ListBox` that will be filled with the files in the current directory, in the same way that the directories `ListBox` will be filled in the top section.

9. To see what you have so far, press F5 (or choose Debug ➤ Start Debugging from the menu bar) to start your Silverlight application.

   Notice that Visual Studio will compile successfully and will open the browser instance. However, just when you think everything is going great and you are excited to see your beautiful form coming to life, you get an `XamlParseException` with a cryptic message:

   ```
   AG_E_PARSER_BAD_PROPERTY_VALUE [Line: 66 Position: 34].
   ```

   This is caused by the fact that, within the code behind, you have not declared the delegates that are referred to in your XAML.

■ **Note** The line and position noted in the error message you see may be slightly different from those shown here, depending on the spacing you included when adding the controls to the code.

---

10. Stop debugging by clicking the Stop button. Press F7 or select View ➤ View Code. Sure enough, there are no event handlers.

    At this point, you could go through and manually add the handlers in the code. But I think you've done enough typing already, so let's have Visual Studio do it for you.

11. Return to your XAML by clicking the `Page.xaml` file in the Files tab. Look at the controls you have added. You will notice that the code refers to three event handlers, one for each of the buttons: `btnUpDir_Click`, `btnOpenDir_Click`, and `btnOpenFile_Click`.

12. Find the first reference, `btnUpDir_Click`. Right-click it and select the Navigate to Event Handler option, as shown in Figure 7-4. Visual Studio will automatically create the event handler in the code behind, as follows:

    ```
    public partial class Page : UserControl
    {
        public Page()
    ```

```
    {
        InitializeComponent();
    }

    private void btnUpDir_Click(object sender, RoutedEventArgs e)
    {

    }
}
```

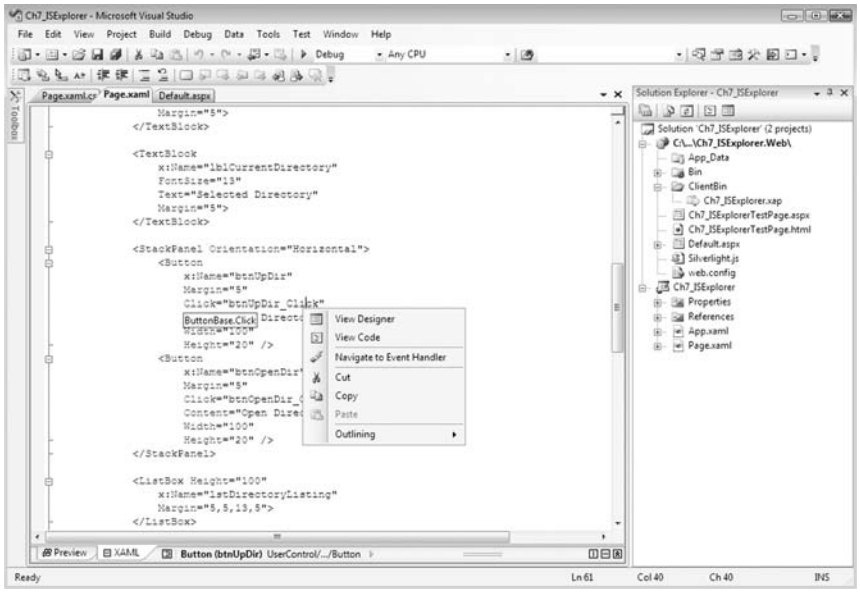

Figure 7-4. *Choosing the Navigate to Event Handler option in Visual Studio*

13. Repeat step 12 for the other two event handlers. At this point, your code behind should look as follows:

```
public partial class Page : UserControl
{
    public Page()
    {
        InitializeComponent();
    }

    private void btnUpDir_Click(object sender, RoutedEventArgs e)
    {
```

```
    }

    private void btnOpenDir_Click(object sender, RoutedEventArgs e)
    {

    }

    private void btnOpenFile_Click(object sender, RoutedEventArgs e)
    {

    }
}
```

14. Run the application. Once again, press F5 to start debugging. Barring any typos, the Silverlight application should appear as shown in Figure 7-5.

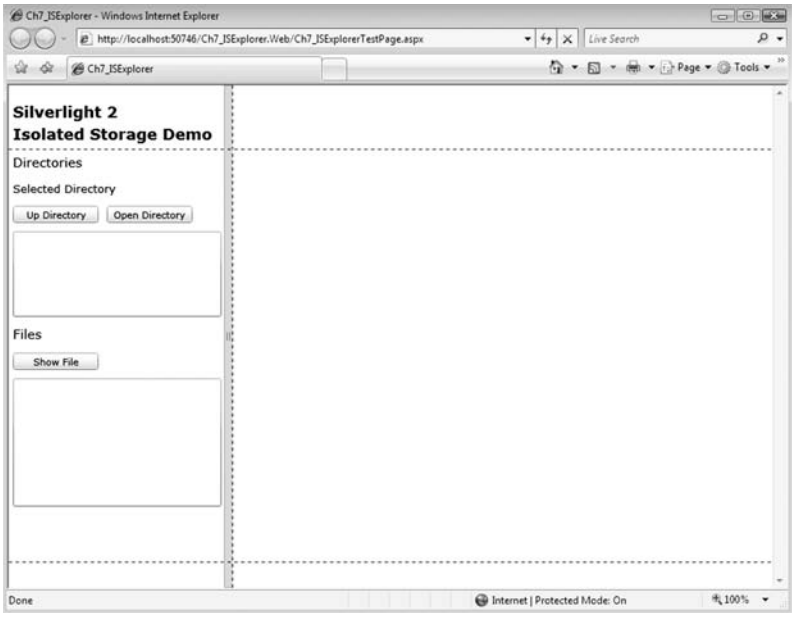

Figure 7-5. *Application with left portion layout*

It's looking good so far! You are almost finished with the application layout. Now, let's move on to the right column and add the final controls.

15. At the bottom of your `Grid` definition within `Grid.Row` and `Grid.Column (0,1)`, place another `StackPanel`. Within it, add a `TextBox` named `txtFileName` that will contain the name of the file being edited, along with a `Button` control named `btnSave`, which will save the file referred to in `txtFileName`. Your XAML should look as follows:

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">

    ...

    </StackPanel>

    <StackPanel
        VerticalAlignment="Bottom"
        Orientation="Horizontal"
        Grid.Row="0"
        Grid.Column="1">

        <TextBox
            x:Name="txtFileName"
            Text="File1.txt"
            Margin="5"
            Width="300"
            Height="30"
            FontSize="15">
        </TextBox>
        <Button
            x:Name="btnSave"
            Margin="5"
            Content="Save"
            Width="100"
            Height="30"
            Click="btnSave_Click">
        </Button>

    </StackPanel>

</Grid>
```

16. While you are at it, go ahead and have Visual Studio create the event handler for `btnSave_Click`. Right-click it and choose the Navigate to Event Handler option to add the following handler:

```
public partial class Page : UserControl
{
```

```
    ...

    private void btnSave_Click(object sender, RoutedEventArgs e)
    {

    }
}
```

17. Navigate back to the XAML. Within `Grid.Row` and `Grid.Column (1,1)`, add a `TextBox` named `txtContents`, which will display the contents of the opened file, as follows:

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">

    ...

    </StackPanel>

    <TextBox
        x:Name="txtContents"
        VerticalScrollBarVisibility="Visible"
        HorizontalScrollBarVisibility="Auto"
        AcceptsReturn="True"
        BorderBrush="Black" BorderThickness="2"
        Margin="5" Grid.Column="1" Grid.Row="1"
        FontSize="15" FontFamily="Courier">
    </TextBox>

</Grid>
```

Since this should be a multiline `TextBox`, you set the `AcceptsReturn` property to `True`. You also set the `VerticalScrollBarVisibility` property to `Visible`, which makes it always appear, and the `HorizontalScrollBarVisibility` property to `Auto`, which makes it appear only when there is enough text to require left and right scrolling.

18. Within `Grid.Row` and `Grid.Column (1,2)`, place a `StackPanel` that contains five `TextBlock` controls, some that are simply cosmetic, and some that will be populated in the application's code, as follows:

```
<Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True">

    ...

    </StackPanel>
```

```
        <TextBox
            x:Name="txtContents"
            VerticalScrollBarVisibility="Visible"
            HorizontalScrollBarVisibility="Auto"
            AcceptsReturn="True"
            BorderBrush="Black" BorderThickness="2"
            Margin="5" Grid.Column="1" Grid.Row="1"
            FontSize="15" FontFamily="Courier">
        </TextBox>

        <StackPanel
            VerticalAlignment="Bottom" Orientation="Horizontal"
            Margin="5" Grid.Column="1" Grid.Row="2">

            <TextBlock FontSize="13"
                Text="Available Space in Isolated Storage: " />
            <TextBlock x:Name="txtAvalSpace" FontSize="13" Text="123" />
            <TextBlock FontSize="13" Text="kb / " />
            <TextBlock x:Name="txtQuota" FontSize="13" Text="123" />
            <TextBlock FontSize="13" Text="kb" />

        </StackPanel>

    </Grid>
```

With this, you are finished creating the application layout! We can now turn our attention to the code behind.

Coding the File Explorer

Now let's add the functionality that demonstrates accessing Silverlight's isolated storage.

1. When the file explorer is started, it will do two things. First, it will load some sample directories and files in isolated storage. Second, it will populate the directories and files `ListBox` controls, as well as update the informative `TextBlock` controls. You will encapulate these tasks into two methods: `LoadFilesAndDirs()` and `GetStorageData()`. Create a `Loaded` event handler and add these two method calls to the event.

```
public partial class Page : UserControl
{
    public Page()
    {
        InitializeComponent();
        this.Loaded += new RoutedEventHandler(Page_Loaded);
```

```
    }

    void Page_Loaded(object sender, RoutedEventArgs e)
    {
        LoadFilesAndDirs();
        GetStorageData();
    }

    private void LoadFilesAndDirs()
    {

    }

    private void GetStorageData()
    {

    }

    private void btnUpDir_Click(object sender, RoutedEventArgs e)
    {

    }

    private void btnOpenDir_Click(object sender, RoutedEventArgs e)
    {

    }

    private void btnOpenFile_Click(object sender, RoutedEventArgs e)
    {

    }

    private void btnSave_Click(object sender, RoutedEventArgs e)
    {

    }
}
```

2. Next, add references to two namespaces for your application. Also create a global string variable called `currentDir`, which will store the current directory.

```
using ...
using System.IO;
using System.IO.IsolatedStorage;

namespace Ch7_ISExplorer
{
    public partial class Page : UserControl
    {
        private string currentDir = "";

        public Page()
        {
            InitializeComponent();
            this.Loaded += new RoutedEventHandler(Page_Loaded);
        }

        ...
    }
}
```

3. Let's implement the `LoadFilesAndDirs()` method. The first step is to get an instance of the user's isolated storage for the application using the `IsolatedStorageFile` class's `GetUserStoreForApplication()` method. You will do this within a C# `using` statement so the instance is disposed of automatically.

```
private void LoadFilesAndDirs()
{
    using (var store =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
    }
}
```

4. Now that you have an instance of the isolated storage, create three root-level directories and three subdirectories, one in each of the root-level directories. Use the `CreateDirectory()` method to create the directories, as follows:

```
private void LoadFilesAndDirs()
{
    using (var store =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        // Create three directories in the root.
```

```
            store.CreateDirectory("Dir1");
            store.CreateDirectory("Dir2");
            store.CreateDirectory("Dir3");

            // Create three subdirectories under Dir1.
            string subdir1 = System.IO.Path.Combine("Dir1", "SubDir1");
            string subdir2 = System.IO.Path.Combine("Dir2", "SubDir2");
            string subdir3 = System.IO.Path.Combine("Dir3", "SubDir3");
            store.CreateDirectory(subdir1);
            store.CreateDirectory(subdir2);
            store.CreateDirectory(subdir3);
        }
    }
```

5. Next, create two files: one in the root and one in a subdirectory. To do this, use the `CreateFile()` method, which returns an `IsolatedStorageFileStream` object. For now, you will leave the files empty, so after creating the files, simply close the stream.

```
private void LoadFilesAndDirs()
{
    using (var store =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        // Create three directories in the root.
        store.CreateDirectory("Dir1");
        store.CreateDirectory("Dir2");
        store.CreateDirectory("Dir3");

        // Create three subdirectories under Dir1.
        string subdir1 = System.IO.Path.Combine("Dir1", "SubDir1");
        string subdir2 = System.IO.Path.Combine("Dir2", "SubDir2");
        string subdir3 = System.IO.Path.Combine("Dir3", "SubDir3");
        store.CreateDirectory(subdir1);
        store.CreateDirectory(subdir2);
        store.CreateDirectory(subdir3);

        // Create a file in the root.
        IsolatedStorageFileStream rootFile =
            store.CreateFile("InTheRoot.txt");
        rootFile.Close();

        // Create a file in a subdirectory.
        IsolatedStorageFileStream subDirFile =
```

```
            store.CreateFile(
                System.IO.Path.Combine(subdir1, "SubDir1.txt"));
        subDirFile.Close();
    }
}
```

■ **Caution** Notice the `Path.Combine()` method call here is fully qualified (specified with the namespace). This is because there is another `Path` class in `System.Windows.Shapes`. If you don't fully qualify `Path`, the ambigous name will cause an error.

That completes the `LoadFilesAndDirs()` method. Next, you will implement the `GetStorageData()` method, which will display the storage information in the application.

6. Since you will be populating the directories and files `ListBox` controls, you need to make sure you clear them each time the `GetStorageData()` method is called. You will do this by calling the `Items.Clear()` method on the two `ListBox` controls. Then you will get an instance of the user's isolated storage, in the same way as you did in the `LoadFilesAndDirs()` method.

```
private void GetStorageData()
{
    this.lstDirectoryListing.Items.Clear();
    this.lstFileListing.Items.Clear();

    using (var store =
        IsolatedStorageFile.GetUserStoreForApplication())
    {

    }
}
```

7. Next, you want to list all of the directories that are contained in the directory passed to the method. In order to do this, you will construct a search string using the `System.IO.Path.Combine()` method. You will then call the `GetDirectoryNames()` method along with the search string. This will return a string array, which you can then step through to manually populate the directories `ListBox`.

```
private void GetStorageData()
{
```

```
    this.lstDirectoryListing.Items.Clear();
    this.lstFileListing.Items.Clear();

    using (var store =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        string searchString =
            System.IO.Path.Combine(currentDir, "*.*");

        string[] directories =
            store.GetDirectoryNames(searchString);

        foreach (string sDir in directories)
        {
            this.lstDirectoryListing.Items.Add(sDir);
        }
    }
}
```

8. Now populate the files `ListBox`. You do this in the same way that you populated the directories `ListBox`, except this time, use the `GetFileNames()` method, which similiarly returns a string array.

```
private void GetStorageData()
{
    this.lstDirectoryListing.Items.Clear();
    this.lstFileListing.Items.Clear();

    using (var store =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        string searchString =
            System.IO.Path.Combine(currentDir, "*.*");

        string[] directories =
            store.GetDirectoryNames(searchString);

        foreach (string sDir in directories)
        {
            this.lstDirectoryListing.Items.Add(sDir);
        }
```

```
            string[] files = store.GetFileNames(searchString);

            foreach (string sFile in files)
            {
                this.lstFileListing.Items.Add(sFile);
            }
        }
    }
```

9. Now that the two `ListBox` controls are populated, you want to populate three additional `TextBlock` controls. One will show the current directory. The other two will display the amount of free space remaining in isolated storage and the available quota for the application. You get this information by using the `Quota` and `AvailableFreeSpace` properties, which return the total and free space in bytes, respectively.

```
private void GetStorageData()
{
    this.lstDirectoryListing.Items.Clear();
    this.lstFileListing.Items.Clear();

    using (var store =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        string searchString =
            System.IO.Path.Combine(currentDir, "*.*");

        string[] directories =
            store.GetDirectoryNames(searchString);

        foreach (string sDir in directories)
        {
            this.lstDirectoryListing.Items.Add(sDir);
        }

        string[] files = store.GetFileNames(searchString);

        foreach (string sFile in files)
        {
            this.lstFileListing.Items.Add(sFile);
        }
```

```
            long space = store.AvailableFreeSpace;
            txtAvalSpace.Text = (space / 1000).ToString();

            long quota = store.Quota;
            txtQuota.Text = (quota / 1000).ToString();

            this.lblCurrentDirectory.Text =
                String.Concat("\\", currentDir);
        }
    }
```

■ **Note** For simplicity, we are dividing by 1000 instead of 1024. Therefore, the calculation will not be exact, but close enough for the purposes of our example.

---

10. Run the application. You will see that the current directory is set to `\`, and that the three directories and the file you created at the root level are displayed in the `ListBox` controls, as shown in Figure 7-6.

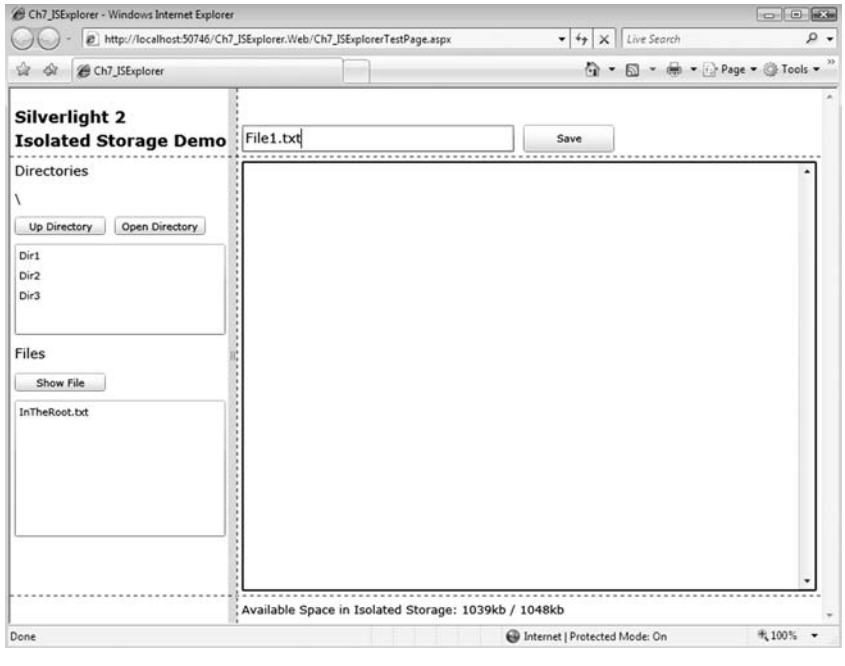

Figure 7-6. *The file explorer application showing the root*

Now you can implement the `Button` events, starting with the Up Directory and Open Directory buttons.

11. When the user clicks the Up Directory button, the system will find the current directory's parent directory using `System.IO.Path.GetDirectoryName()`, set the current directory to be that parent directory, and reexecute the `GetStorageData()` method.

```
private void btnUpDir_Click(object sender, RoutedEventArgs e)
{
    if (currentDir != "")
    {
        currentDir =
            System.IO.Path.GetDirectoryName(currentDir);
    }

    GetStorageData();
}
```

12. When the user clicks the Open Directory button, you will combine the current directory with the selected directory from the directory `ListBox` using the `System.IO.Path.Combine()` method, set the current directory to that new directory, and once again reexecute the `GetStorageData()` method.

```
private void btnOpenDir_Click(object sender, RoutedEventArgs e)
{
    if (this.lstDirectoryListing.SelectedItem != null)
    {
        currentDir =
            System.IO.Path.Combine(
                currentDir,
                this.lstDirectoryListing.SelectedItem.ToString());
    }
    GetStorageData();
}
```

13. Next, implement the Show File button's `Click` event, as follows:

```
private void btnOpenFile_Click(object sender, RoutedEventArgs e)
{
    if (this.lstFileListing.SelectedItem != null)
```

```
    {
        this.txtFileName.Text =
            this.lstFileListing.SelectedItem.ToString();

        using (var store =
            IsolatedStorageFile.GetUserStoreForApplication())
        {
            string filePath =
                System.IO.Path.Combine(
                    currentDir,
                    this.lstFileListing.SelectedItem.ToString());

            IsolatedStorageFileStream stream =
                store.OpenFile(filePath, FileMode.Open);
            StreamReader sr = new StreamReader(stream);

            this.txtContents.Text = sr.ReadToEnd();
            sr.Close();
        }
    }
}
```

When a user clicks the Show File button, the file from isolated storage opens, and its contents are displayed in `txtContents`. You achieve this by first getting an instance of the user's isolated storage, and then generating the path to the file by combining the current directory with the file name provided in `txtFileName`. After you have constructed the full file path, you open the file using `OpenFile()`, which returns a `Stream` containing the file contents. You attach a `StreamReader` to the `Stream` to assist in working with the stream, and then display the contents of the `Stream` using the `StreamReader`'s `ReadToEnd()` method.

14. Finally, wire up the Save button, which will save the contents of `txtContents` to the file name specified in `txtFileName`. You want to make it so that if the user enters a file name that doesn't exist, the application will create a new file. If the user enters one that does exist, the application will override the contents of that file. Although this is not perfect for use in the real world, it serves as a fine demo for using isolated storage.

```
private void btnSave_Click(object sender, RoutedEventArgs e)
{
    string fileContents = this.txtContents.Text;
```

```
        using (var store =
            IsolatedStorageFile.GetUserStoreForApplication())
        {
            IsolatedStorageFileStream stream =
                store.OpenFile(
                    System.IO.Path.Combine(
                        currentDir,
                        this.txtFileName.Text),
                    FileMode.OpenOrCreate);

            StreamWriter sw = new StreamWriter(stream);
            sw.Write(fileContents);
            sw.Close();
            stream.Close();
        }

        GetStorageData();
    }
```

This method is similar to the `ShowFile()` method. Basically, you get the isolated storage instance, and open the file using the `OpenFile()` method, passing it the full file path. However, this time, you pass the `OpenFile()` method `FileMode.OpenOrCreate`. This way, if the file doesn't exist, the application will create it. You then attach the returned stream to a `StreamWriter`, and write the contents to the `Stream` using the `StreamWriter`'s `Write()` method.

After writing the file, you clean up the objects and call the `GetStorageData()` method, which will cause the newly created file to appear in the files `ListBox` (in the event a new file was created).

At this point, you're ready to test your completed application.

Testing the File Explorer

Now let's try out your new file explorer.

1. Fire up the application by pressing F5. If all goes well, you should see the application.

2. Highlight Dir1 in the Directories list box and click the Open Directory button. The application will navigate to that directory and refresh the list boxes to show the directories and files contained within that file.

3. Enter the file name `SampleTextFile.txt` in the `txtFileName` text box. For the contents, enter some arbitrary data. If you have Microsoft Word, you can generate a ton of random text using `=Rand(10,20)` and paste the content into the text box.

4. After you enter the contents, click the Save button. You will see the file appear in the Files list box, as shown in Figure 7-7.

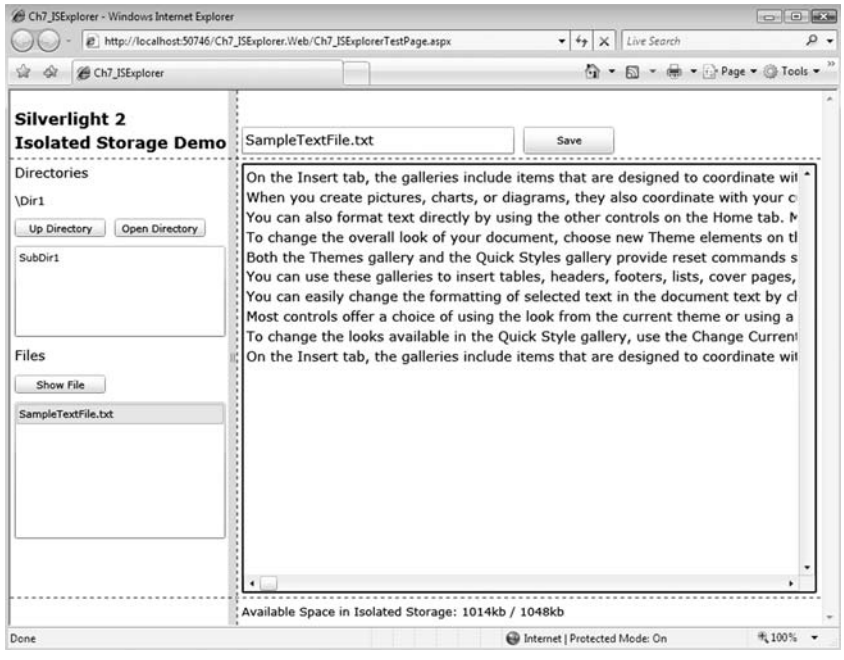

Figure 7-7. *Testing the completed file explorer*

5. Click the Up Directory button to navigate back to the root. You will notice that the current directory changes, as do the contents of the list boxes. For kicks, click Save again. This time, the application will save the same file in the root directory.

6. Highlight the `InTheRoot.txt` file and click the Show File button. Since you left the file empty, nothing will appear in the `txtContents` box. You can enter some text in the text box and click Save.

7. Highlight `SampleTextFile.txt` and click Show File. The contents of your file are still there. It really works!

8. Try adding some files (preferrably with a large amount of text). Take a look at the display of the current free space and quota of the isolated storage at the bottom of the application. You should see the amount of free space decrease.

9. Stop debugging. Now restart debugging. Notice anything? Your files are still there! That is because isolated storage is persistent data, and it will remain until the user clears the isolated storage, as explained in the next section.

This exercise demonstrated how Silverlight's isolated storage works and how you can access it. In the following section, you will learn how to manage isolated storage, including changing its quota.

Managing Isolated Storage

By default, the amount of isolated storage space available for a Silverlight application is 1MB. You can view the available storage, clear it, and increase its size.

Viewing and Clearing Isolated Storage

In order to view the isolated storage saved on your machine, simply right-click any Silverlight application and select Silverlight Configuration from the pop-up menu. This will display the Microsoft Silverlight Configuration window. Navigate to the Application Storage tab, as shown in Figure 7-8. There, you can see your test application in the listing, and depending on what other Silverlight applications you have accessed, you may see other web sites listed.

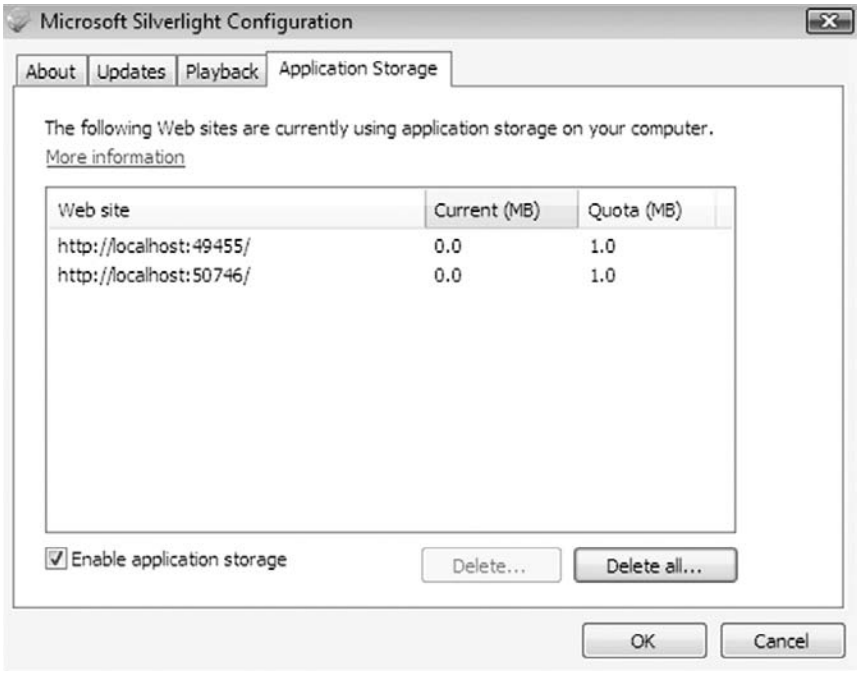

Figure 7-8. *Viewing application storage information in the Microsoft Silverlight Configuration window*

If users want to clear the storage space, they simply need to highlight the site they want to clear data for and click Delete. This will display a confirmation dialog box, as shown in Figure 7-9.

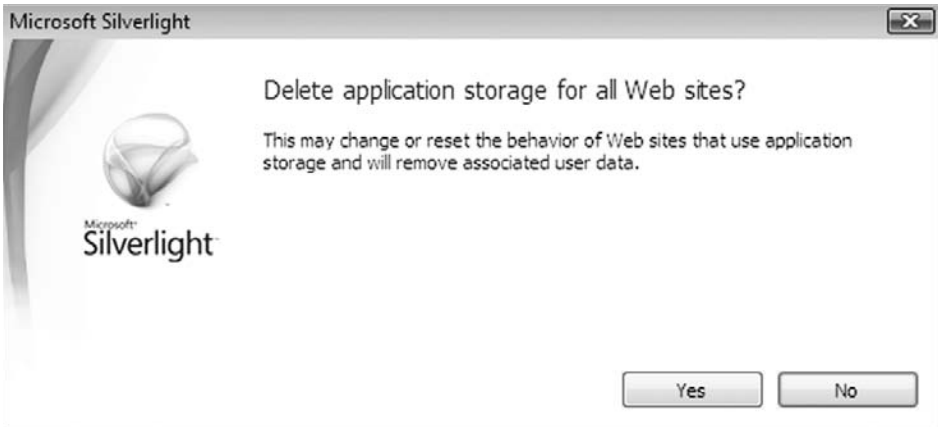

Figure 7-9. *Deleting an application's isolated storage*

So what if you want more storage space for your application? Developers can request additional storage space by using the `TryIncreaseQuotaTo()` method. A restriction placed on this task is that it can be executed only in a user-triggered event, such as a `Button` control's `Click` event. This restriction is in place to prevent the application from increasing the quota without the user's knowledge.

Try It Out: Increasing the Isolated Storage Quota

To demonstrate how to increase the isolated storage quota, let's add a button to the file explorer demo to increase the quota to 4MB.

1. Open the `IsolatedStorageExplorer` project that you created in the previous exercise.

2. In the `Page.xaml` file, locate the definition of the Save button and add a new `Button` control called `btnIncreaseQuota`, with the caption `Increase Quota`, as follows:

```
<StackPanel
    VerticalAlignment="Bottom"
    Orientation="Horizontal"
    Grid.Row="0"
    Grid.Column="1">
```

```
        <TextBox
            x:Name="txtFileName"
            Text="File1.txt"
            Margin="5"
            Width="300"
            Height="30"
            FontSize="15">
        </TextBox>
        <Button
            x:Name="btnSave"
            Margin="5"
            Content="Save"
            Width="100"
            Height="30"
            Click="btnSave_Click">
        </Button>
        <Button
            x:Name="btnIncreaseQuota"
            Margin="5"
            Content="Increase Quota"
            Width="150"
            Height="30"
            Click="btnIncreaseQuota_Click">
        </Button>

    </StackPanel>
```

3. You have wired up the `Click` event to a new event handler created by Visual Studio. Navigate to the code behind's definition of that event handler.

```
private void btnIncreaseQuota_Click(object sender, RoutedEventArgs e)
{
}
```

4. Next, you want to get an instance of the user's isolated storage, just as you did numerous times in creating the file explorer. Then call the `IncreaseQuotaTo()` method, passing it `4000000`, which is roughly 4MB. Add the following to event handler:

```
private void btnIncreaseQuota_Click(object sender, RoutedEventArgs e)
{
    using (var store =
        IsolatedStorageFile.GetUserStoreForApplication())
```

```
    {
        if (store.IncreaseQuotaTo(4000000))
        {
            GetStorageData();
        }
        else
        {
            // The user rejected the request to increase the quota size
        }
    }
}
```

■ **Note** As I mentioned earlier, these numbers are not exact, which is fine for the demonstration here. You can increase the quota to 4MB exactly by multiplying 1024 by 4.

Notice that the `IncreaseQuotaTo()` method returns a Boolean value. Depending on whether the user accepted the application's request to increase the quota size, `true` or `false` will be returned. If the user accepted the request, you will want to redisplay the information displayed for the quota. The easiest way to do this is to simply call the `GetStorageData()` method, as you did in the event handler here.

5. Try out your new addition by running your application and clicking the new Increase Quota button. You will see the dialog box shown in Figure 7-10.

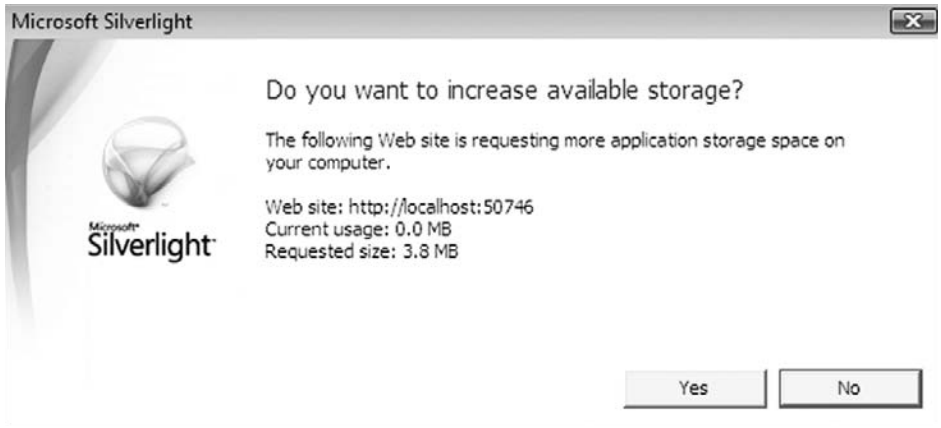

Figure 7-10. *Dialog box to request to increase available storage*

6. Click Yes, You will notice that the available quota is now increased in your application, as shown in Figure 7-11.

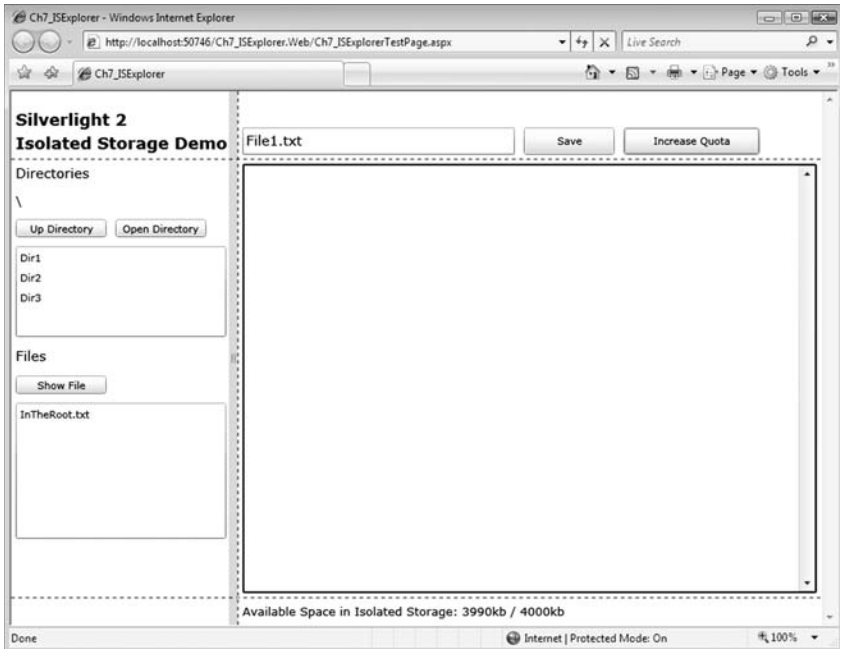

Figure 7-11. *File explorer showing additional storage space*

This completes the file explorer. Now you can apply these concepts to your own persistent storage implementations in your Silverlight 2 applications.

Summary

In this chapter, we looked at Silverlight's isolated storage feature. As you saw, it is very straightforward to store user-specific data for your application and have that data persist over browser instances. This provides a very convenient way for developers to add offline content or save user settings.

In the next chapter, we will look at Microsoft Expression Blend 2, an application created for the sole purpose of visually editing XAML.

CHAPTER 8

Introduction to Expression Blend

So far in this book, the primary focus has been on using Visual Studio 2008 to create Silverlight applications. Visual Studio provides developers with a strong IDE for developing RIAs. However, you may want your Silverlight applications to contain some complicated design elements, and in these cases, it's not much fun to edit the XAML manually. To address this problem, Microsoft has created Expression Blend, a product built to edit XAML documents visually.

Whereas Visual Studio has been designed to cater to the developer, Expression Blend has been built for the designer. As you've seen, Silverlight does a fantastic job of separating the appearance and logic of an application, so developers and designers can work side by side. ASP.NET took a few strides to achieve this separation, but still fell short in many ways. I think you will find that Silverlight has reached a new layer in this separation, making it much more practical for designers and developers to truly work in parallel in designing applications.

The first reaction most ASP.NET software developers will have when opening Expression Blend is shock. "Wow, this looks like no Microsoft development product I have ever seen!" And it is true that Expression Blend is quite different from the standard Visual Studio IDE type of product. The Microsoft developers have finally provided a product for the graphic designer audience, and they have attempted to make it very similar to the tools designers are accustomed to using. As software developers, we may need to play around a bit in Expression Blend to get the feel of it. Personally, I have found it quite cool to learn and use, and I think you will, too.

This chapter will get you started with Expression Blend. You'll learn about its key features and its workspace. Finally, we'll walk through creating a grid layout with Expression Blend.

Key Features in Expression Blend 2

In this section, we will look at some of the notable features in Expression Blend 2, including the following:

- Visual XAML editor
- Visual Studio 2008 integration
- Split-view mode
- Visual State Manager and template editing support
- Timeline

■ **Note** One of the things that Microsoft has done better and better over the past few years is documentation. Expression Blend's documentation is quite comprehensive. For additional information about any of the items discussed in this chapter, refer to the User Guide provided with Expression Blend.

Visual XAML Editor

Clearly, the biggest feature of Expression Blend is that it provides a WYSIWYG editor for XAML. XAML is a very clean language, but it can also get quite complex quickly when you are working with your applications. This is especially true when you start to add animations and transformations, which are covered in Chapter 10.

Although it is possible to edit your XAML files completely in Visual Studio using IntelliSense, there is no substitute for a visual editor. In addition, the XAML that Expression Blend creates is very clean and developer-friendly. This should make developers happy, considering the terrible memories of earlier versions of FrontPage, where every change you made would result in your code being mangled beyond recognition.

In addition, when you start working with styles (covered in Chapter 9), IntelliSense support in Visual Studio becomes limited, so the XML is very difficult to edit manually. Expression Blend provides an extremely quick and easy way to edit and create styles, which is another reason it is an invaluable tool for editing your Silverlight applications.

Visual Studio 2008 Integration

Due to the strong push for developers and designers to work in parallel, and given the fact that XAML files are included directly within Visual Studio 2008 projects, a valid concern

would be how well Expression Blend and Visual Studio work together. If there were conflicts between the two IDEs, there could be conflicts between the developers and designers, resulting in resistance to working in parallel.

The good news is that Expression Blend integrates with Visual Studio. Visual Studio 2008 projects can be opened directly in Expression Blend and vice versa. In addition, while Expression Blend creates Visual Studio 2008 projects by default, it is also capable of opening Visual Studio 2005 projects.

Split-View Mode

As is shown in Figure 8-1, Expression Blend allows you to work in design and source (XAML) mode simultaneously. For example, you can draw an object at the top in design mode, and the XAML in the source window will be updated automatically. In addition, you can just as easily edit the XAML, and the change will be reflected automatically in the design window.

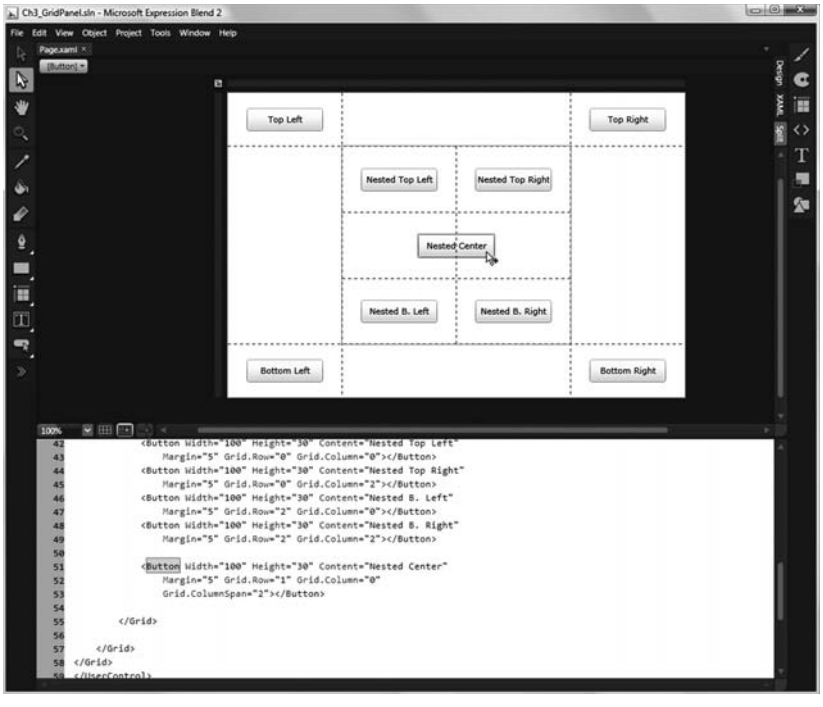

Figure 8-1. *Expression Blend 2's split-view mode*

Visual State Manager and Template Editing Support

One of the cool features of Silverlight 2 is the fact that all controls released with it support the new Parts and State model, which requires strict separation between a control's logic and appearance. Microsoft recommends that all custom controls also support this model.

By separating the logic from the appearance of a control, a developer or designer can completely change the appearance of a control without affecting its behavior. This process is known as creating a template, or *skinning*, and is regulated by Visual State Manager (VSM). Expression Blend provides a very clean way to create and edit these parts and states, which makes skinning your applications a relatively simple task. You'll learn more about VSM and skinning in *Pro Silverlight 2 in C# 2008* by Matthew MacDonald (Apress, 2008).

World-Class Timeline

In Silverlight 2, animations are based on keyframes within a storyboard. These keyframes are set on a timeline, and they define the start and end points of a smooth visual transition. Figure 8-2 shows the Expression Blend timeline, which is located in the Objects and Timeline panel.

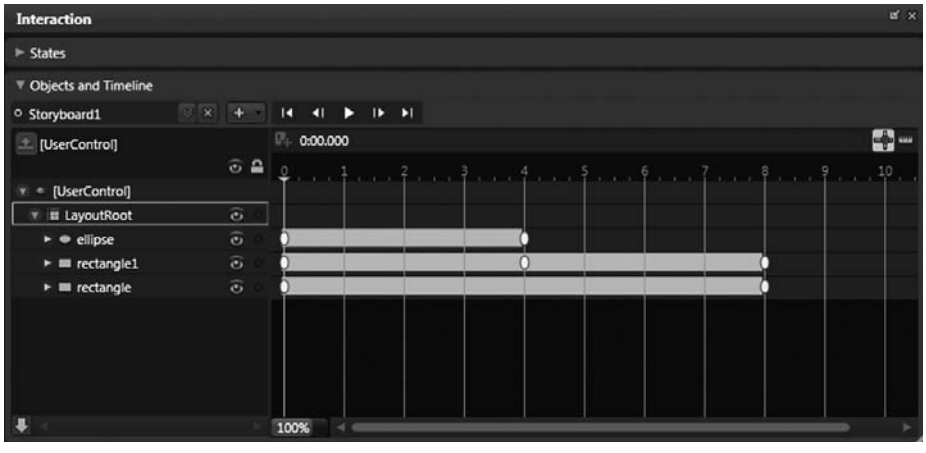

Figure 8-2. *The Expression Blend timeline*

The timeline provides you with structure for all of the animation sequences in your Silverlight application. Instead of the timeline being based on abstract frames, it is based on time, which makes it very straightforward and easy to understand. Also, as you develop your animations, you can quickly navigate to any given time on the timeline to check the appearance of your application at that point.

Try It Out: Working with Projects in Expression Blend 2

As you've learned, one of the key features of Expression Blend 2 is that it integrates directly with Visual Studio 2008 projects. This exercise demonstrates how you can use the two products side by side while creating and editing projects.

1. Open Expression Blend 2. By default, when you open Expression Blend, you will see the splash screen shown in Figure 8-3. If you do not want this screen to appear when you start Expression Blend, you can simply uncheck the Run at startup check box at the bottom left. For now, if this screen appears, click Close to continue with the example.

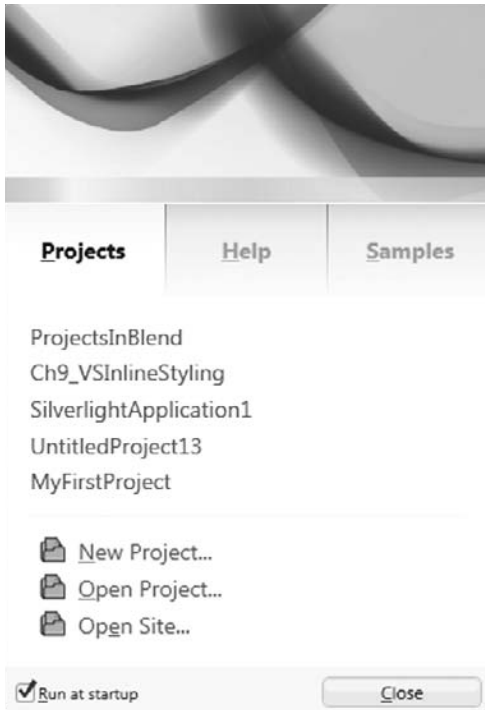

Figure 8-3. *Startup screen for Expression Blend 2*

2. You should now have an empty Expression Blend workspace. From the main menu, click File ➤ New Project. This will display the New Project dialog box.

3. In the New Project dialog box, select Silverlight 2 Application for the project type, and then enter `Ch8_BlendProjects` for the project name, as shown in Figure 8-4. Click OK to create the new project.

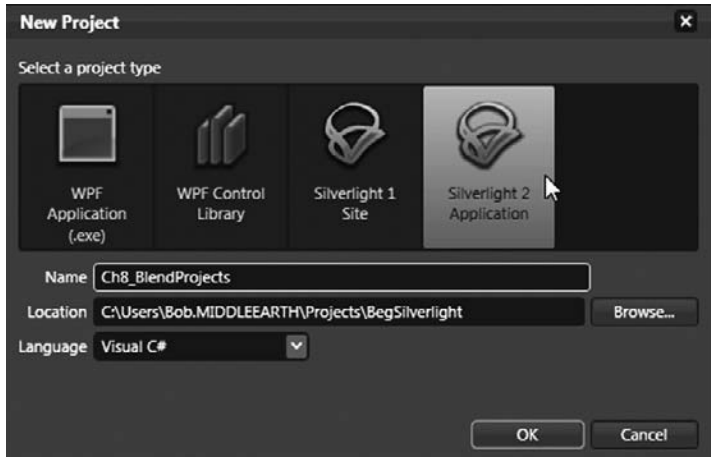

Figure 8-4. *Creating a new project in Expression Blend*

4. By default, Expression Blend will open the `Page.xaml` file for editing. In the upper-right portion of the artboard (which contains the XML) are options to switch between design, XAML, and split-mode view. Click Split to see both the XAML and the design view at the same time, as shown in Figure 8-5.

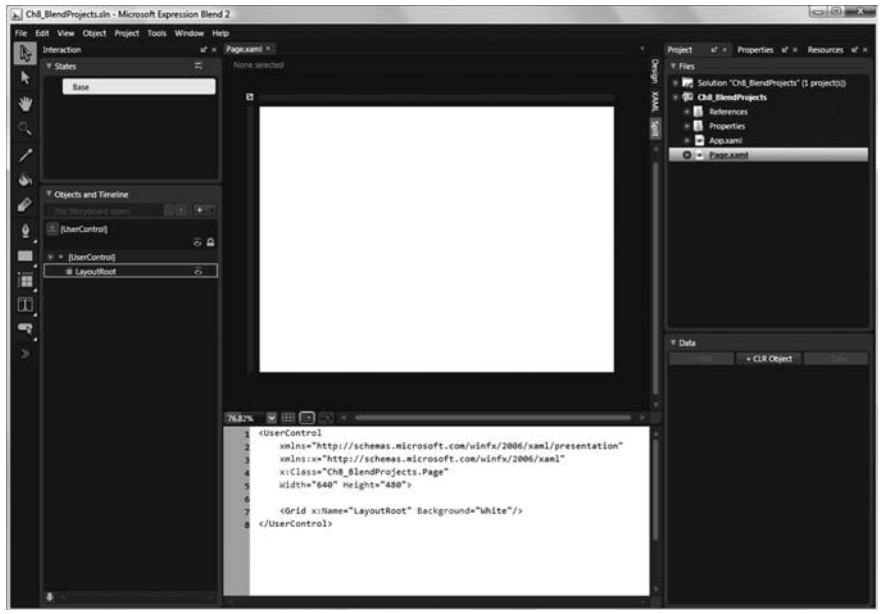

Figure 8-5. *Split-view mode in Expression Blend*

5. Now edit this project in Visual Studio. In the Project panel, right-click the `Ch8_BlendProjects` project and select Edit in Visual Studio, as shown in Figure 8-6. This will automatically start Visual Studio 2008 and open your project.

■ **Note** Step 5 assumes that you have already installed Visual Studio 2008. If not, you will need to install that to continue.

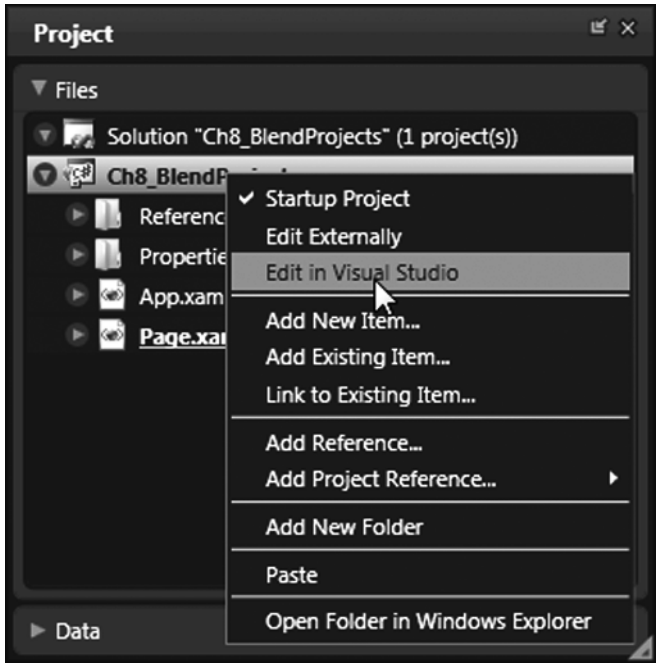

Figure 8-6. *Editing a Expression Blend project in Visual Studio*

6. In Visual Studio 2008, double-click `Page.xaml` in Solution Explorer. Let's make a simple change to the application in Visual Studio.

7. Modify the root `Grid` to add the following code shown in bold, to define a `StackPanel` with a `TextBlock`, `TextBox`, and `Button`.

```
<UserControl
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ProjectsInBlend.Page"
    Width="640" Height="480">
```

```
    <Grid x:Name="LayoutRoot" Background="White">
        <StackPanel Margin="20" Orientation="Vertical">
            <TextBlock Margin="5" Text="Enter Your Name:" />
            <TextBox Margin="5" x:Name="txtName" />
            <Button Margin="5" Content="Click Me!" />
        </StackPanel>
    </Grid>
</UserControl>
```

With this change, your application should now look as shown in Figure 8-7.

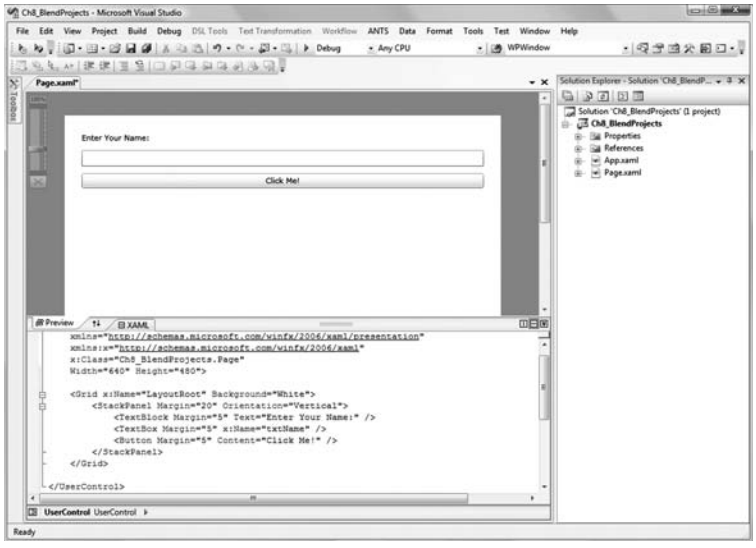

Figure 8-7. *The modified Expression Blend project in Visual Studio*

8. From the main menu, click File ➤ Save All, just to make sure everything is saved.

9. Switch back to Expression Blend. It will prompt you with the File Modified dialog box, as shown in Figure 8-8. Click Yes. You will see Expression Blend refresh the project so that it reflects the changes you made in Visual Studio 2008.

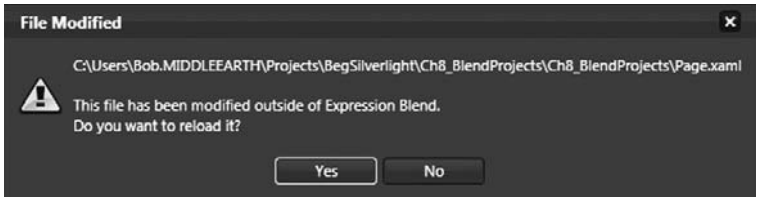

Figure 8-8. *File modification notification in Expression Blend 2*

Pretty nifty, right? The same file modification is offered when you do the reverse: make a change in Expression Blend and then go back into Visual Studio. Feel free to try this out yourself.

As this exercise demonstrated, Expression Blend and Visual Studio work together seamlessly. You can switch back and forth between the two products without fear of data loss or conflicts.

■ **Note** Although usually Expression Blend will be used together with Visual Studio, Expression Blend will actually pick up on changes to open files caused by edits in any editor.

Exploring the Workspace

Now that we have briefly discussed some of the key features of Expression Blend 2, we will take a look at the different elements of its workspace. Despite its radically new appearance, developers will find many similarities between Visual Studio and Expression Blend.

Let's start out by looking at Expression Blend in Animation workspace mode. You enter this mode by selecting Window ➤ Active Workspace ➤ Animation Workspace from the main menu. Starting at the left, you will see the Toolbox and the artboard, which contains the application and the XAML source. On the right is the Properties panel. Docked with the Properties panel are the Project and Resources panels. At the bottom of the workspace, you will see the VSM panel and Objects and Timeline panel. Let's take a closer look at some of these workspace elements.

Toolbox

The Expression Blend Toolbox provides the tools for adding and manipulating objects within your application. As shown in Figure 8-9, it is divided into five primary sections: selection tools, view tools, brush tools, object tools, and asset tools. The object tool group includes six submenus, which contain path tools, shape tools, layout tools, text controls, and common controls.

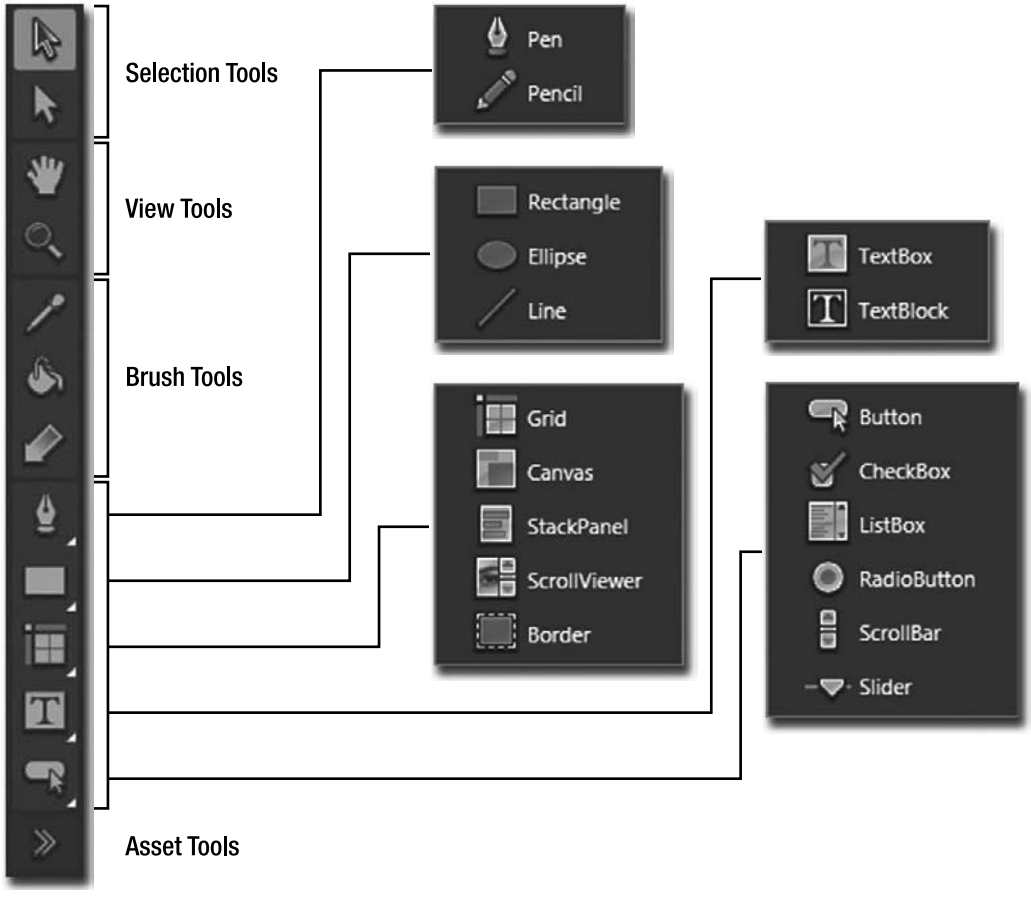

Figure 8-9. *The Expression Blend Toolbox*

Clicking the Asset Tools icon at the very bottom of the Toolbox opens the Asset Library window, which lists the Silverlight system controls, as shown in Figure 8-10.

You may notice that a number of controls that you might expect to see in the system controls listing are not shown. As mentioned in Chapter 4, two control libraries are not included by default in Silverlight 2 projects: `System.Windows.Controls.dll` and `System.Windows.Controls.Data.dll`. These two libraries contain controls such as `GridSplitter` and `DataGrid`. To get these other controls to show up in the Asset Library window, from the Project panel, right-click the `References` folder in your project and choose Add Reference, as shown in Figure 8-11.

Figure 8-10. *The Asset Library window*

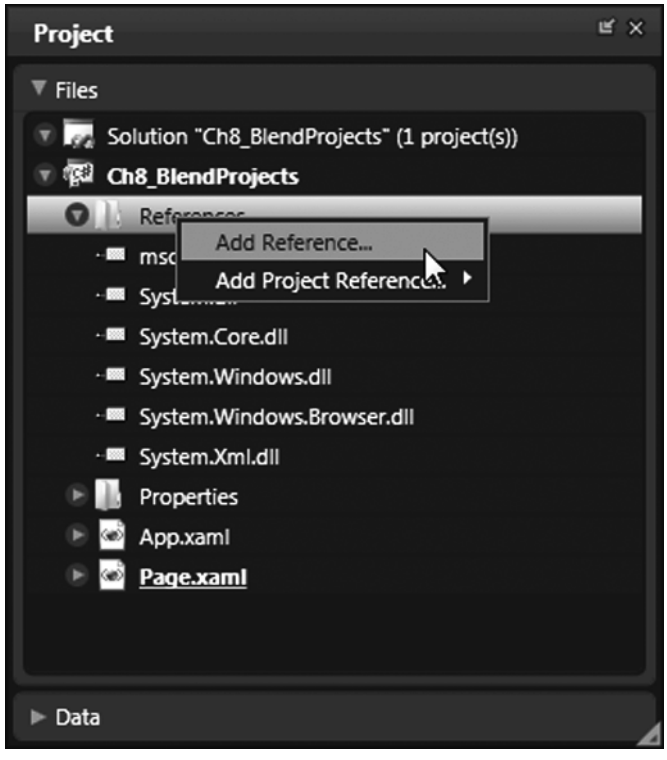

Figure 8-11. *Adding a new reference to a project in Expression Blend 2*

This will bring up the Open dialog box. Browse to the `C:\Program Files\Microsoft SDKs\Silverlight\v2.0\Libraries\Client` folder and select the `System.Windows.Controls.dll` file. Then repeat the process for `System.Windows.Controls.Data.dll`, which is in the same directory. After you have added references to both of these libraries in your project, when you view the Asset Library window, it should appear as shown in Figure 8-12.

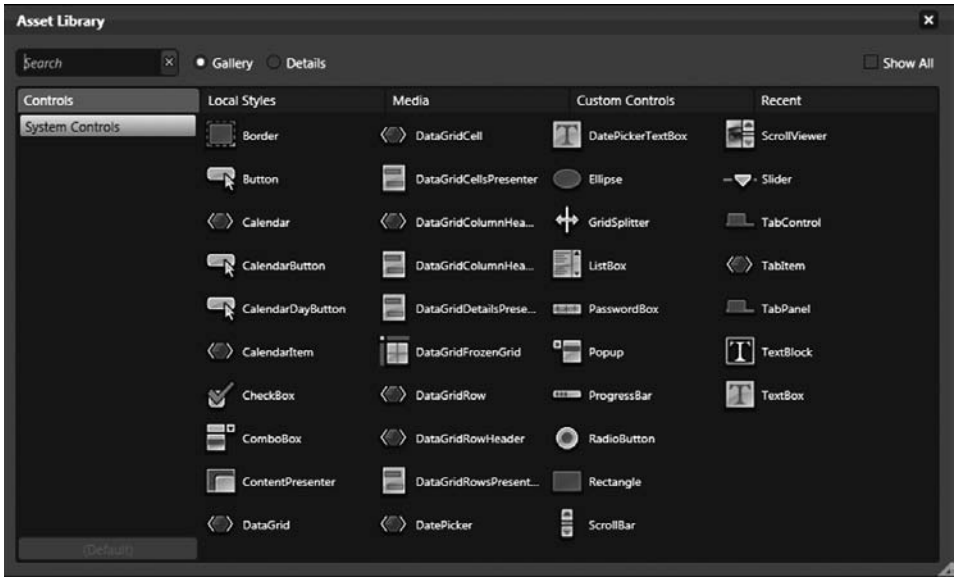

Figure 8-12. *The Asset Library window listing controls in the Extended and Data libraries*

Project Panel

The Project panel is very similar to Solution Explorer in Visual Studio. It lists all the files associated with the project.

The Project panel also displays project references and properties. See Figure 8-11 for an example of the Project panel.

Properties Panel

The Properties panel allows you to view and modify the properties of objects on the artboard. Figure 8-13 shows an example of the Properties panel when an `Ellipse` control is selected.

The Properties panel is divided into a number of sections to help you easily find specific properties. The sections displayed depend on the object you have selected. In addition, the Search box at the top of the Properties panel allows you to filter the listing by typing in the property name. Figure 8-14 shows an example of the Properties panel after searching for the `Margin` property.

Figure 8-13. *The Properties panel*

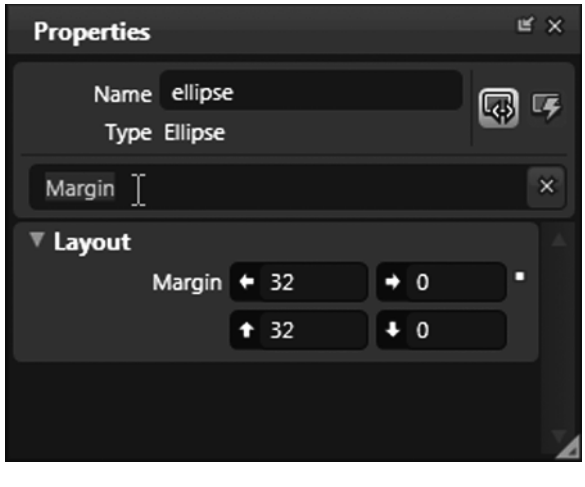

Figure 8-14. *Filtering the Properties panel*

Objects and Timeline Panel

All objects that are added to your Silverlight application are represented in the Objects and Timeline panel. Since items can be nested within other objects, a type of layering takes place. For objects that contain additional objects, an arrow will appear to the left of the item. Click this arrow to expand and collapse the display of the nested objects.

When animation is added to your Silverlight application, storyboards are created. Storyboards are represented in the timeline, as shown earlier in Figure 8-2. You'll learn more about the timeline in Chapter 10.

Laying Out an Application with Expression Blend

As discussed in Chapter 3, you have a number of options when it comes to laying out your Silverlight application. Although these layout controls can be added manually, Expression Blend 2 offers a visual option. In this section we will look at how Expression Blend can be used to easily work with the `Grid` layout control.

Working with the Grid Control in Expression Blend

In Expression Blend, you place dividers to create columns and rows in the grid. When a `Grid` control is defined, Expression Blend will show blue rulers above and to the left of the grid. When you move your cursor over the blue rulers, a row divider will appear. Clicking the blue ruler will place the divider, and dragging a placed divider will move it. You will have a chance to try this out in a moment.

In the top-left corner of the window is an icon that determines the grid's edit mode. There are two layout editing modes for a grid within Expression Blend:

Canvas layout mode: In canvas layout mode, when column and row dividers are moved, elements inside those rows and columns stay in place.

Grid layout mode: In grid layout mode, the elements move with the column and row dividers.

Try It Out: Editing a Layout Grid with Expression Blend

Let's give layout in Expression Blend a try. In this exercise, you will create a simple grid layout with three rows and two columns. Then you will nest a secondary grid within the right-center cell, and place two more rows within that grid. The end product will look like Figure 8-15.

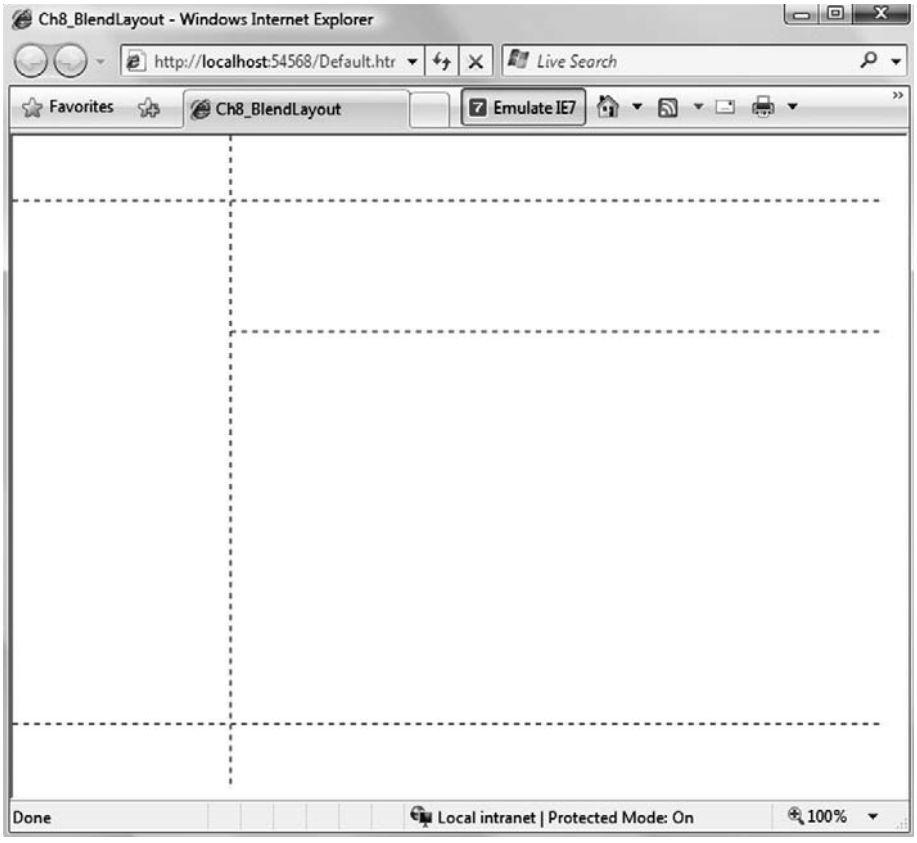

Figure 8-15. *The completed grid layout*

1. In Expression Blend, create a new Silverlight 2 Application project named `Ch8_BlendLayout`. The `Page.xaml` file will be opened automatically, and as usual, a root `Grid` named `LayoutRoot` will be present.

2. First, create the column definitions. To do this, at about 25% from the left of the top blue grid ruler, click the ruler to place a grid divider, as shown in Figure 8-16. If you examine the XAML, you will notice that the `<Grid.ColumnDefinitions>` element has been added, along with two `<ColumnDefinition>` elements, as follows (note that your percentages do not need to be exact):

```
<UserControl
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    x:Class="GridsInBlend.Page"
    Width="640" Height="480">

    <Grid x:Name="LayoutRoot" Background="White" >
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.25*"/>
            <ColumnDefinition Width="0.75*"/>
        </Grid.ColumnDefinitions>
    </Grid>
</UserControl>
```

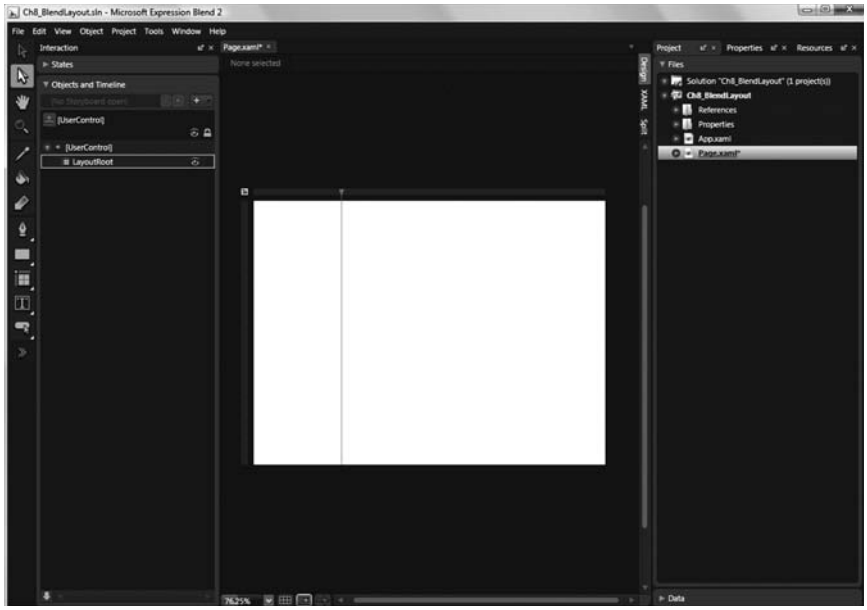

Figure 8-16. *Adding column defintions*

3. Next, create the rows. In the blue grid ruler on the left, click at about 10% from the top and 10% from the bottom to place two dividers. Your grid should now look like the one shown in Figure 8-17.

   The source for the `Page.xaml` file should be very similar to the following (the actual heights and widths do not need to be exact):

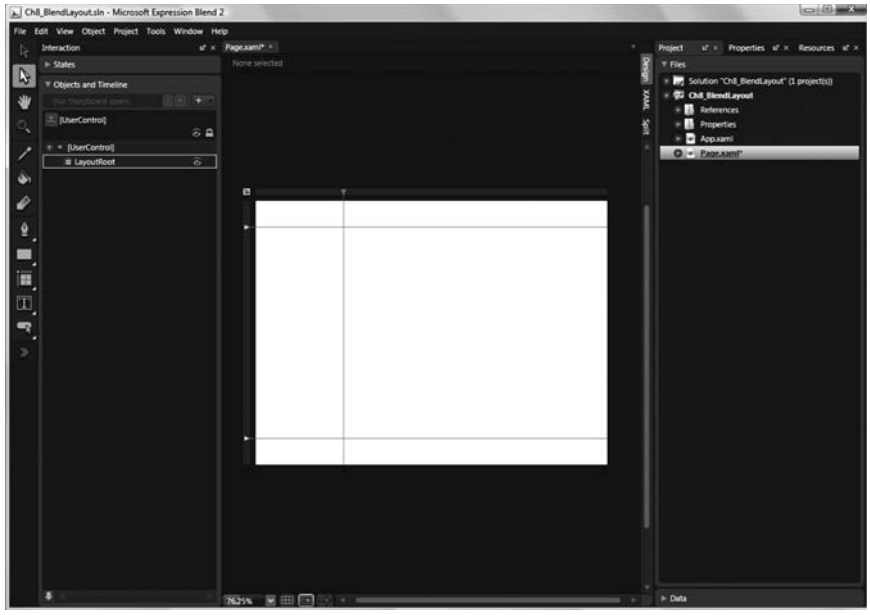

Figure 8-17. *Adding row definitions*

```
<UserControl
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="GridsInBlend.Page"
    Width="640" Height="480">

    <Grid x:Name="LayoutRoot" Background="White" >
        <Grid.RowDefinitions>
            <RowDefinition Height="0.1*"/>
            <RowDefinition Height="0.8*"/>
            <RowDefinition Height="0.1*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.25*"/>
            <ColumnDefinition Width="0.75*"/>
        </Grid.ColumnDefinitions>
    </Grid>
</UserControl>
```

4. At this point, you have created a number of cells. Now, let's create a nested grid within the right-center cell. To do this, make certain that the `LayoutRoot` is selected in the Objects and Timeline panel, and then double-click the `Grid` control in the Toolbox. This will add a `Grid` of the default size to your application, as shown in Figure 8-18.

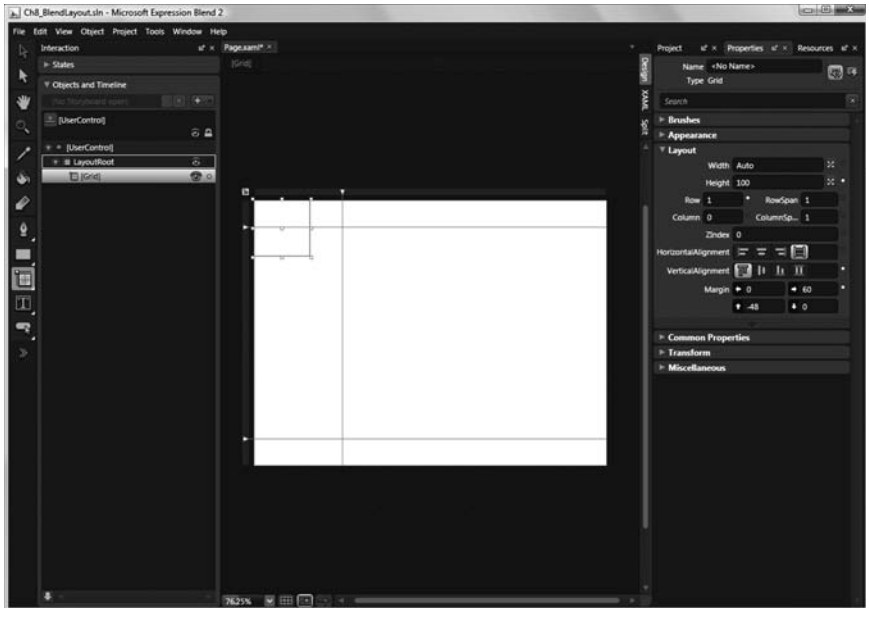

Figure 8-18. *Adding a nested grid*

5. With this new grid selected, edit its properties. In the Properties panel, set the properties as shown in Figure 8-19.

6. The nested grid should now take up the entire right-center cell. In the Objects and Timeline panel, double-click the `innerGrid` object you just added. The top and left grid rulers will now appear for the inner grid, as shown in Figure 8-20.

   At this point, you could easily add rows and columns using the rulers, as you did with the `LayoutRoot`, but let's try a different method.

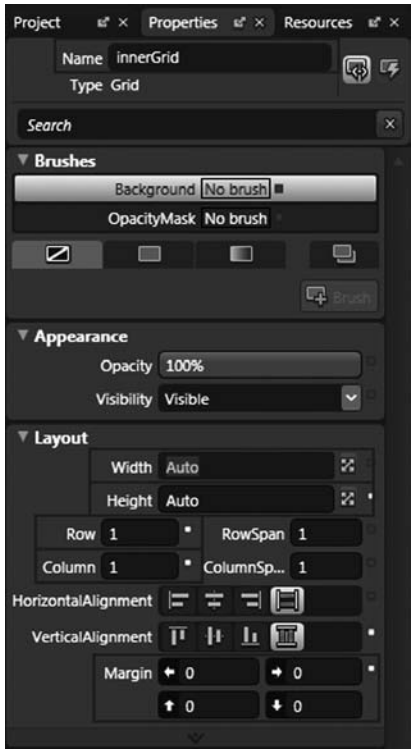

Figure 8-19. *Setting the nested grid properties*

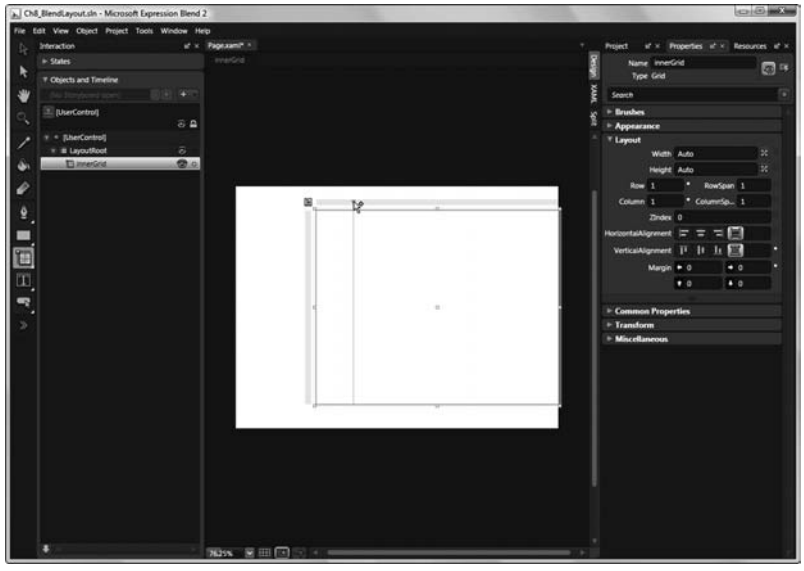

Figure 8-20. *Nested grid with row and column rulers*

7. With `innerGrid` selected, in the Properties panel's Search box, type **Definitions**. This will display the `RowDefinitions` and `ColumnDefinitions` properties, as shown in Figure 8-21.

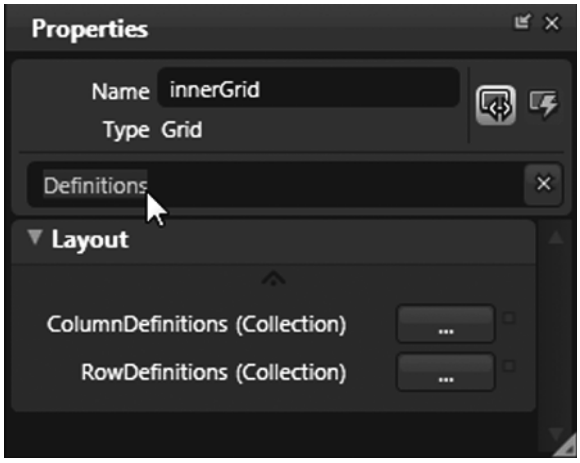

Figure 8-21. *RowDefinition and ColumnDefinition property collections*

8. Click the button to the right of RowDefinitions (Collection) to bring up the RowDefinition Collection Editor dialog box.

9. Click the "Add another item" button near the bottom of the RowDefinition Collection Editor dialog box and add two `RowDefinition` items. Set the `Height` property for the first `RowDefinition` to be `.25` and the `Height` property for the second `RowDefinition` to `.75`, as shown in Figure 8-22. Then click OK to close the editor.

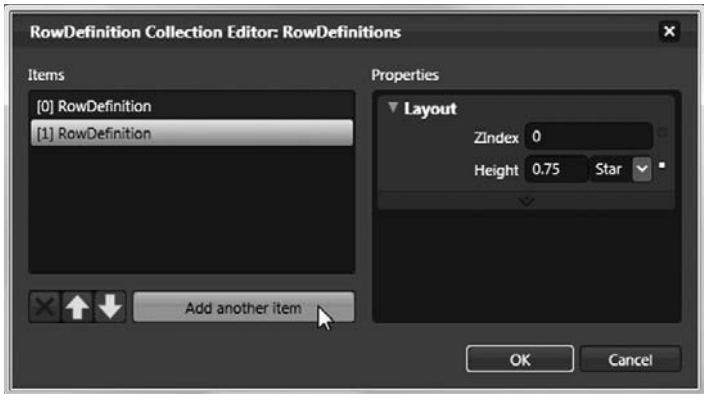

Figure 8-22. *Adding RowDefinition items in the RowDefinition Collection Editor*

10. In the Properties panel, set the `ShowGridLines` property for both `Grids` to `True`.

    The final XAML should look like the following (again, the heights and widths only need to be close):

```
<UserControl
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="GridsInBlend.Page"
    Width="640" Height="480">

    <Grid x:Name="LayoutRoot" Background="White" ShowGridLines="True" >
        <Grid.RowDefinitions>
            <RowDefinition Height="0.1*"/>
            <RowDefinition Height="0.8*"/>
            <RowDefinition Height="0.1*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.25*"/>
            <ColumnDefinition Width="0.75*"/>
        </Grid.ColumnDefinitions>
        <Grid Height="Auto"
            Margin="0,0,0,0"
            VerticalAlignment="Stretch"
            Grid.Row="1"
            x:Name="innerGrid"
            Grid.Column="1"
            ShowGridLines="True">
            <Grid.RowDefinitions>
                <RowDefinition Height="0.25*"/>
                <RowDefinition Height="0.75*"/>
            </Grid.RowDefinitions>
        </Grid>
    </Grid>
</UserControl>
```

11. Press F5 to test your application. The result should appear as shown earlier in Figure 8-15.

As you can see, once you get used to working with Expression Blend, it can save you quite a bit of typing. This will make laying out your applications a much faster and easier task.

Summary

In this chapter, we took a first look at Expression Blend 2 and how it can be used alongside Visual Studio 2008 to help you design your Silverlight applications. We also looked at working with the `Grid` layout control to create complex layouts for your applications.

The upcoming chapters explain how to use Expression Blend to style your Silverlight applications, as well as add transformations and animations to your applications.

CHAPTER 9

Styling in Silverlight

Of course you will want to create a rich appearance for your Silverlight application. You'll make choices about your design. What font size and family will you use? How much space will you place between your objects? What size of text boxes and buttons will you use?

As you'll learn in this chapter, you can control the styles of your Silverlight application's UI elements in several ways. The first approach we will explore is the straightforward use of inline properties. Then we will look at how to define and apply Silverlight styles.

Inline Properties

You can simply define style properties directly in the object definitions. As an example, the following code snippet sets the `FontFamily`, `FontSize`, `FontWeight`, and `Margin` properties within the `TextBlock` itself.

```
<TextBlock
    Grid.Row="0"
    Grid.Column="0"
    Text="First Name"
    FontFamily="Verdana"
    FontSize="16"
    FontWeight="Bold"
    Margin="5" />
```

You can set inline properties using either Visual Studio or Expression Blend. Let's try out both.

Try It Out: Setting Inline Properties with Visual Studio

The following exercise demonstrates how to use Visual Studio 2008 to define the appearance of your Silverlight applications with inline properties. In this exercise, you will create the UI for a simple data-input application. You will not add any logic to the application, since the focus is on the appearance of the controls.

1. Open Visual Studio 2008 and create a new Silverlight application named `Ch9_VSInlineStyling`. Allow Visual Studio to create a Web Site project to host the application.

2. When the project is created, you should be looking at the `Page.xaml` file. If you do not see the XAML source, switch to that view. Start by adjusting the size of the `UserControl` to get some additional space in which to work. Set `Height` to `400` and `Width` to `600`, as follows:

```
<UserControl x:Class="Ch9_VSInlineStyling.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="600" Height="400">
    <Grid x:Name="LayoutRoot" Background="White">

    </Grid>
</UserControl>
```

3. Add four rows and two columns to the root `Grid`. Set the width of the left column to `150`, leaving the rest of the row and column definitions unspecified, as follows:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
</Grid>
```

4. Next, add `TextBlock` controls in the three top-left columns and `TextBox` controls in the top-right columns, with the text `First Name`, `Last Name`, and `Age`. Then add three `Button` controls within a horizontal `StackPanel` in the bottom-right column. Give these buttons the labels `Save`, `Next`, and `Delete`. (Again, you won't be adding any logic to these controls; you will simply be modifying their appearance.) The code for this layout follows:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <TextBlock Grid.Row="0" Grid.Column="0" Text="First Name" />
    <TextBlock Grid.Row="1" Grid.Column="0" Text="Last Name" />
    <TextBlock Grid.Row="2" Grid.Column="0" Text="Age" />

    <TextBox Grid.Row="0" Grid.Column="1" />
    <TextBox Grid.Row="1" Grid.Column="1" />
    <TextBox Grid.Row="2" Grid.Column="1"  />

    <StackPanel Grid.Row="3" Grid.Column="2" Orientation="Horizontal">
        <Button Content="Save" />
        <Button Content="Next" />
        <Button Content="Delete" />
    </StackPanel>
</Grid>
```

5. Press F5 to start the application. You will see that the UI you have created is far from attractive, as shown in Figure 9-1. So let's make this ugly UI look a bit nicer by adding some styling.

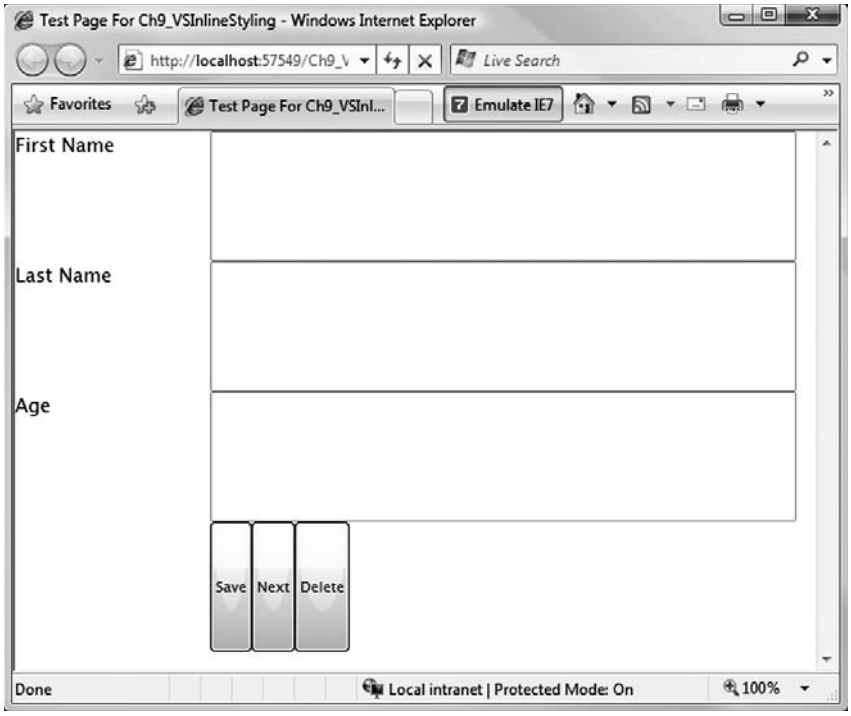

Figure 9-1. *Default input form without styling*

6. Start with the three `TextBlock` controls. Within Visual Studio, set the `FontFamily`, `FontSize`, `FontWeight`, and `Margin` properties directly within each `TextBlock` definition, as shown in the following code snippet. As you type the property names, you will notice that IntelliSense makes this task a bit less tedious. Once you have set the four properties on the `First Name TextBlock`, copy and paste the properties to the other two `TextBlock` controls.

```
<TextBlock Grid.Row="0" Grid.Column="0" Text="First Name"
    FontFamily="Verdana"
    FontSize="16"
    FontWeight="Bold"
    Margin="5" />
<TextBlock Grid.Row="1" Grid.Column="0" Text="Last Name"
    FontFamily="Verdana"
    FontSize="16"
    FontWeight="Bold"
    Margin="5" />
```

```
<TextBlock Grid.Row="2" Grid.Column="0" Text="Age"
    FontFamily="Verdana"
    FontSize="16"
    FontWeight="Bold"
    Margin="5" />
```

7. Run the application again. You can see the changes that have been made to the `TextBlock` labels, as shown in Figure 9-2.

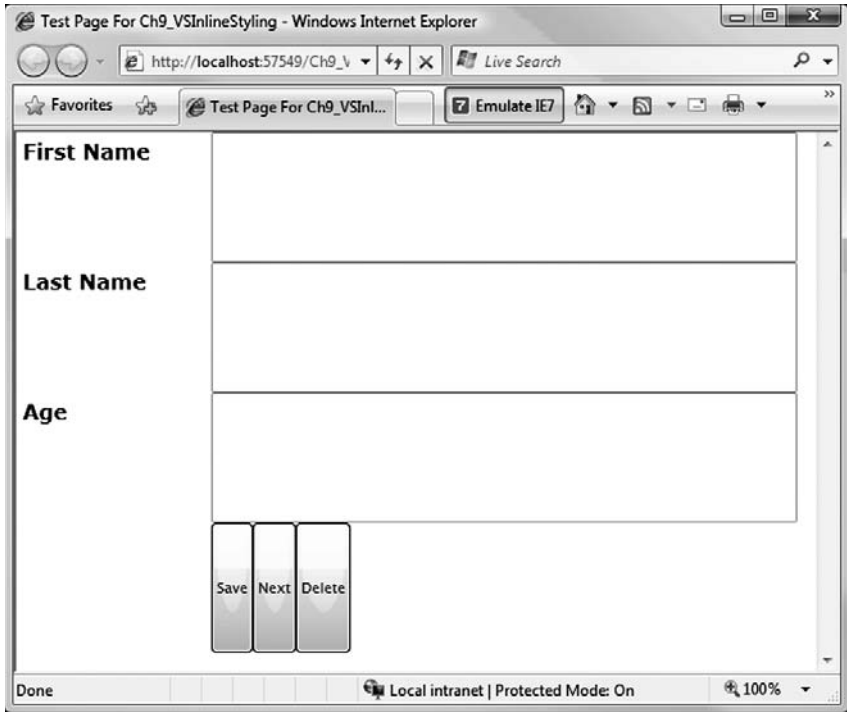

Figure 9-2. *Input form with styled TextBlock labels*

8. Now let's focus on the `TextBox` controls. Add the following style attributes to these controls.

```
<TextBox Grid.Row="0" Grid.Column="1"
    VerticalAlignment="Top"
    Height="24"
    Margin="5"
    FontSize="14"
    FontFamily="Verdana"
    Foreground="Blue"
    Background="Wheat" />
```

```
<TextBox Grid.Row="1" Grid.Column="1"
    VerticalAlignment="Top"
    Height="24"
    Margin="5"
    FontSize="14"
    FontFamily="Verdana"
    Foreground="Blue"
    Background="Wheat" />

<TextBox Grid.Row="2" Grid.Column="1"
    VerticalAlignment="Top"
    Height="24"
    Margin="5"
    FontSize="14"
    FontFamily="Verdana"
    Foreground="Blue"
    Background="Wheat" />
```

9. Run the application to see the effect. It should look like Figure 9-3.

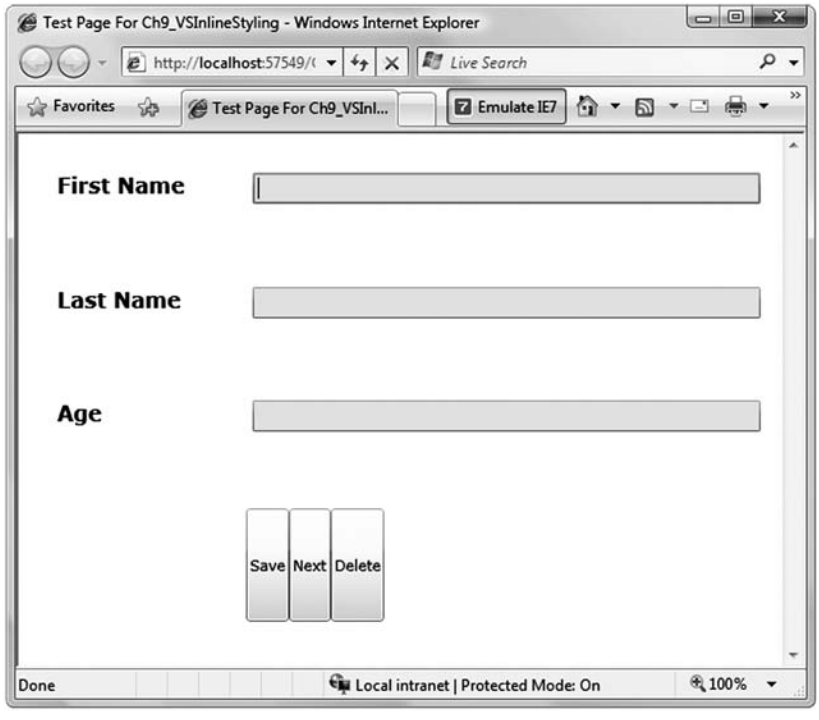

Figure 9-3. *Input form with styled TextBox controls*

10. Notice that the spacing between the rows is too large. Ideally, the spaces should only be large enough to allow the margins of the controls to provide the separation. To adjust this spacing, on each `RowDefinition`, change the `Height` property to `Auto`, as follows:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="150" />
    <ColumnDefinition />
</Grid.ColumnDefinitions>
```

11. Once more, run the application to see how it looks at this point. Figure 9-4 shows the results of the automatic height settings.

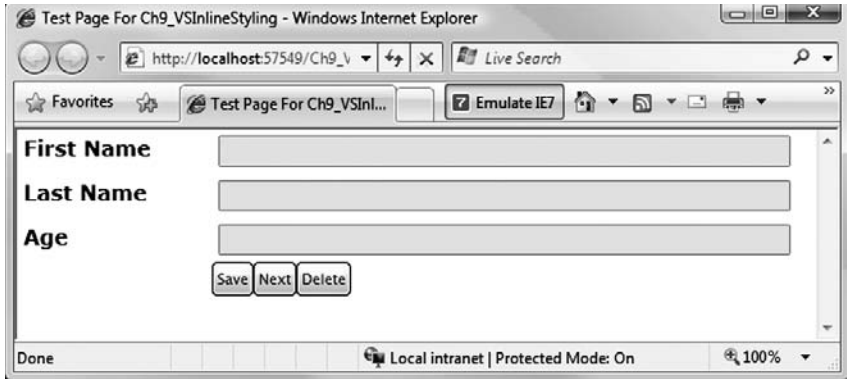

Figure 9-4. *Input form with styled RowDefinitions*

12. The next elements to tackle are the `Button` controls. Add the following style attributes to these three controls:

```
<Button Content="Save"
    FontFamily="Verdana"
    FontSize="11"
    Width="75"
    Margin="5" />
```

```
<Button Content="Next"
    FontFamily="Verdana"
    FontSize="11"
    Width="75"
    Margin="5" />

<Button Content="Delete"
    FontFamily="Verdana"
    FontSize="11"
    Width="75"
    Margin="5" />
```

13. Run the application to see the effect. It should look like Figure 9-5.

Figure 9-5. *Input form with styled buttons*

14. Finally, it would be nice to add a margin around the entire application. To do this, simply add a `Margin` property definition to the root `Grid`, as follows:

```
<Grid x:Name="LayoutRoot" Background="White" Margin="25">
```

15. Press F5. The final product is a UI that looks pretty nice, as shown in Figure 9-6.

As you saw in this exercise, the process of setting inline properties in Visual Studio is simple and straightforward. However, the sample application contained only nine controls. We will look at some better options later in this chapter, in the "Silverlight Styles" section. Next, let's see how to set inline properties within Expression Blend.

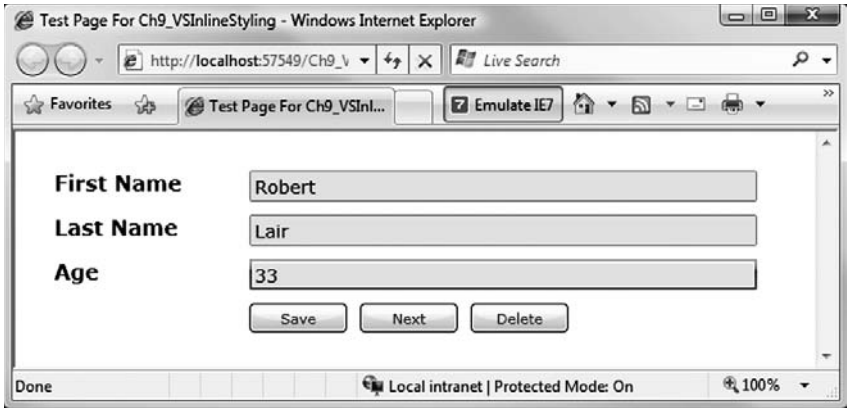

Figure 9-6. *Final input form styled with inline properties*

Try It Out: Setting Inline Properties with Expression Blend

The previous example used Visual Studio to set the inline properties of an application's controls. For those of you who are not a big fan of a lot of typing, you may find that Expression Blend is a better place to set these properties. In this next exercise, you will perform the same styling as in previous exercise, but using Expression Blend 2 to set the properties, rather than Visual Studio 2008. Let's give it a try!

1. Open Expression Blend 2 and create a new Silverlight 2 application named `Ch9_BlendStyling`.

2. The `UserControl` is 640 by 480 by default when created in Expression Blend, so you can leave that size. The first thing to do is add the column and row definitions. You can copy and paste the grid definitions from the previous exercise, or you can add the columns and rows using Expression Blend's grid editor, as described in Chapter 8. The end result should look like Figure 9-7.

3. Next, add the controls to the form. In the Toolbox, double-click the `TextBlock` control three times to add three `TextBlock` controls to the grid. Then double-click the `TextBox` control three times, which will add three `TextBox` controls below the `TextBlock` controls.

4. Double-click the `StackPanel` layout control. Once the `StackPanel` is added, double-click it in the Objects and Timeline panel so that it has a yellow border, as shown in Figure 9-8.

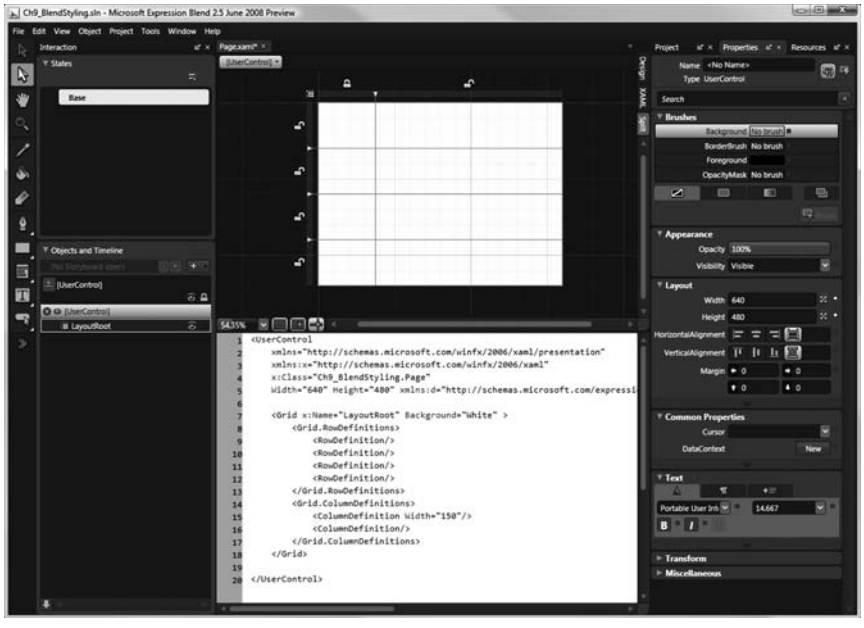

Figure 9-7. *Completed grid layout*

Figure 9-8. *Selecting the StackPanel in the Objects and Timeline panel*

5. With the `StackPanel` selected, double-click the `Button` control three times. The three `Button` controls will appear within the `StackPanel`, as shown in Figure 9-9.

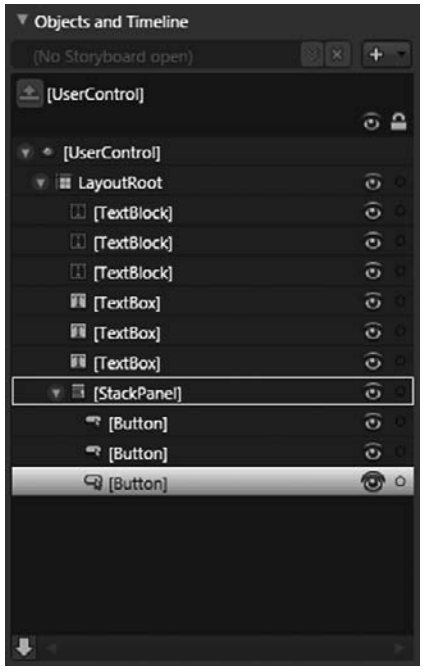

Figure 9-9. *The Button controls added to the StackPanel*

By default, Expression Blend adds a number of properties that we don't want. In the next steps, you'll remove the properties shown in bold in the following XAML:

```
<Grid x:Name="LayoutRoot" Background="White" >
   <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
   </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
      <ColumnDefinition Width="150"/>
      <ColumnDefinition/>
   </Grid.ColumnDefinitions>
   <TextBlock HorizontalAlignment="Left"
      VerticalAlignment="Top" Text="TextBlock" TextWrapping="Wrap"/>
   <TextBlock HorizontalAlignment="Left"
      VerticalAlignment="Top" Text="TextBlock" TextWrapping="Wrap"/>
```

```
    <TextBlock HorizontalAlignment="Left"
       VerticalAlignment="Top" Text="TextBlock" TextWrapping="Wrap"/>
    <TextBox HorizontalAlignment="Left"
       VerticalAlignment="Top" Text="TextBox" TextWrapping="Wrap"/>
    <TextBox HorizontalAlignment="Left"
       VerticalAlignment="Top" Text="TextBox" TextWrapping="Wrap"/>
    <TextBox HorizontalAlignment="Left"
       VerticalAlignment="Top" Text="TextBox" TextWrapping="Wrap"/>
    <StackPanel Margin="0,0,50,20">
       <Button Content="Button"/>
       <Button Content="Button"/>
       <Button Content="Button"/>
    </StackPanel>
</Grid>
```

6. In the Objects and Timeline panel, highlight all of the `TextBlock` and `TextBox` controls, as shown in Figure 9-10. You can highlight multiple items in the Objects and Timeline panel by holding down the Shift or Ctrl key as you click.

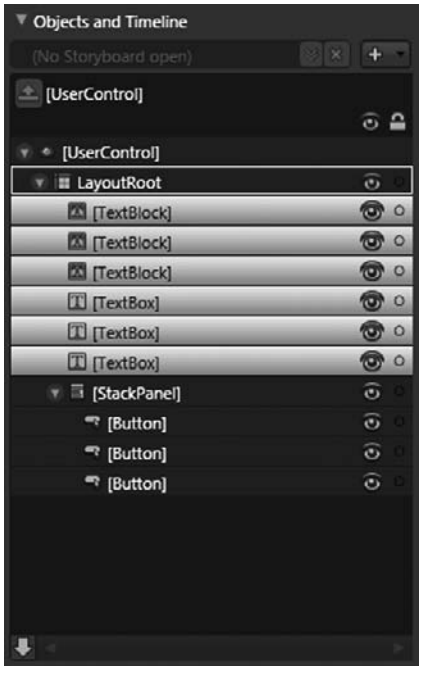

Figure 9-10. *Selecting multiple objects in the Objects and Timeline panel*

7. With these six controls selected, look in the Properties panel. Notice that any property that is set in the XAML has a white dot to its right. (Properties you cannot edit have a gray dot.) You can easily remove these properties from the XAML and "reset" the code by clicking the white dot and selecting Reset. Start out by resetting the `HorizontalAlignment` property located in the Layout section of the Properties panel, as shown in Figure 9-11. Then reset the `VerticalAlignment` property. This will remove the `HorizontalAlignment` and `VerticalAlignment` property definitions in the XAML.

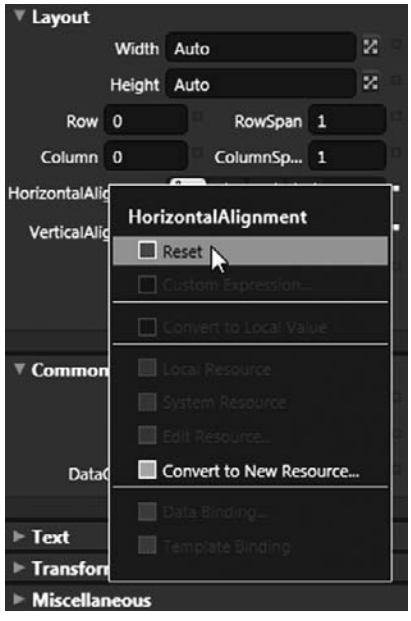

Figure 9-11. *Resetting the HorizontalAlignment property*

8. The `TextWrapping` property is located in the Text Section of the Properties panel, but you must extend the section to see it. I figured that this would be a good opportunity to show you another feature of the Properties panel. At the top of the Properties panel, type **TextWrapping** into the Search box. That will filter the Properties panel to show only the `TextWrapping` property. Click and reset that property as well.

9. Next, highlight the `StackPanel` and reset its `Margin` property in the same way. When you have finished all of these steps, the XAML should contain the following source code:

```
<Grid x:Name="LayoutRoot" Background="White" >
   <Grid.RowDefinitions>
     <RowDefinition/>
```

```
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock Text="TextBlock"/>
    <TextBlock Text="TextBlock"/>
    <TextBlock Text="TextBlock"/>
    <TextBox Text="TextBox"/>
    <TextBox Text="TextBox"/>
    <TextBox Text="TextBox"/>
    <StackPanel>
        <Button Content="Button"/>
        <Button Content="Button"/>
        <Button Content="Button"/>
    </StackPanel>
</Grid>
```

10. Now you need to place these controls in the proper cells in your grid. Click to highlight the control in the Objects and Timeline panel. In the Layout section of the Properties panel, you will see `Row` and `Column` properties. Set their values so that you have the following result:

```
<Grid x:Name="LayoutRoot" Background="White" >
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock Text="TextBlock"/>
    <TextBlock Text="TextBlock" Grid.Row="1"/>
    <TextBlock Text="TextBlock" Grid.Row="2"/>
    <TextBox Text="TextBox" Grid.Column="1"/>
    <TextBox Text="TextBox" Grid.Column="1" Grid.Row="1"/>
```

```
    <TextBox Text="TextBox" Grid.Row="2" Grid.Column="1"/>
    <StackPanel Grid.Column="1" Grid.Row="3">
        <Button Content="Button"/>
        <Button Content="Button"/>
        <Button Content="Button"/>
    </StackPanel>
</Grid>
```

11. Go through each of the TextBlock controls to set the `Text` properties to `First Name`, `Last Name`, and `Age`. Next, set the `Text` property of the `TextBox` controls to blank (or just reset the property). Then set the `Orientation` property for the `StackPanel` to `Horizontal`. Finally, set the `Content` property for the `Button` controls to `Save`, `Next`, and `Delete`. The final result should be the following:

```
<Grid x:Name="LayoutRoot" Background="White" >
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock Text="First Name"/>
    <TextBlock Text="Last Name" Grid.Row="1"/>
    <TextBlock Text="Age" Grid.Row="2"/>
    <TextBox Grid.Column="1"/>
    <TextBox Grid.Column="1" Grid.Row="1"/>
    <TextBox Grid.Row="2" Grid.Column="1"/>
    <StackPanel Grid.Column="1" Grid.Row="3" Orientation="Horizontal">
        <Button Content="Save"/>
        <Button Content="Next"/>
        <Button Content="Delete"/>
    </StackPanel>
</Grid>
```

12. Run the solution, and you will see the initial layout, which should look the same as what you started with in the previous exercise (Figure 9-1). The next thing to do is set the style properties for your controls.

13. Highlight all three `TextBlock` controls. In the Properties panel, set the following properties:

    - `FontFamily`: `Verdana`
    - `FontSize`: `16`
    - `FontWeight`: `Bold`
    - `Margin`: `5,5,5,5`

14. Select the three `TextBox` controls and set the following properties:

    - `FontFamily`: `Verdana`
    - `FontSize`: `14`
    - `FontWeight`: `Bold`
    - `Foreground`: `#FF0008FF`
    - `Background`: `#FFF9F57D`
    - `VerticalAlignment`: `Top`
    - `Margin`: `5,5,5,5`

15. Highlight the three `Button` controls and set the following properties:

    - `FontFamily`: `Verdana`
    - `FontSize`: `11`
    - `Width`: `75`
    - `Margin`: `5,5,5,5`

16. Switch to split-view mode. Within the XAML, place your cursor within one of the `RowDefinition` items. Then, in the Properties panel, set the `Height` property to `Auto`. Repeat this for all of the `RowDefinition` items in the `Grid`. When you are finished setting the `Height` properties on the `RowDefinition` items, the XAML for the application should be as follows:

```
<Grid x:Name="LayoutRoot" Background="White" >
   <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
```

```
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="150"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock Text="First Name" FontFamily="Verdana"
        FontSize="16" FontWeight="Bold" Margin="5,5,5,5"/>
    <TextBlock Text="Last Name" Grid.Row="1" FontFamily="Verdana"
        FontSize="16" FontWeight="Bold" Margin="5,5,5,5"/>
    <TextBlock Text="Age" Grid.Row="2" FontFamily="Verdana"
        FontSize="16" FontWeight="Bold" Margin="5,5,5,5"/>
    <TextBox Text="" Grid.Row="0" Grid.Column="1"
        FontFamily="Verdana" FontSize="14" FontWeight="Bold"
        Foreground="#FF0008FF" Background="#FFF9F57D"
        VerticalAlignment="Top" Margin="5,5,5,5"/>
    <TextBox Text="" Grid.Row="1" Grid.Column="1"
        FontFamily="Verdana" FontSize="14" FontWeight="Bold"
        Foreground="#FF0008FF" Background="#FFF9F57D"
        VerticalAlignment="Top" Margin="5,5,5,5"/>
    <TextBox Text="" Grid.Row="2" Grid.Column="1"
        FontFamily="Verdana" FontSize="14" FontWeight="Bold"
        Foreground="#FF0008FF" Background="#FFF9F57D"
        VerticalAlignment="Top" Margin="5,5,5,5"/>
    <StackPanel Grid.Row="3" Grid.Column="1" Orientation="Horizontal">
        <Button Content="Save" Margin="5,5,5,5"
            Width="75" FontFamily="Verdana"/>
        <Button Content="Next" Margin="5,5,5,5"
            Width="75" FontFamily="Verdana"/>
        <Button Content="Delete" Margin="5,5,5,5"
            Width="75" FontFamily="Verdana"/>
        </StackPanel>
</Grid>
```

17. Your application will appear something like what is shown in Figure 9-12. When you run the application, it should look very similar to the application at the end of the previous exercise (Figure 9-6).

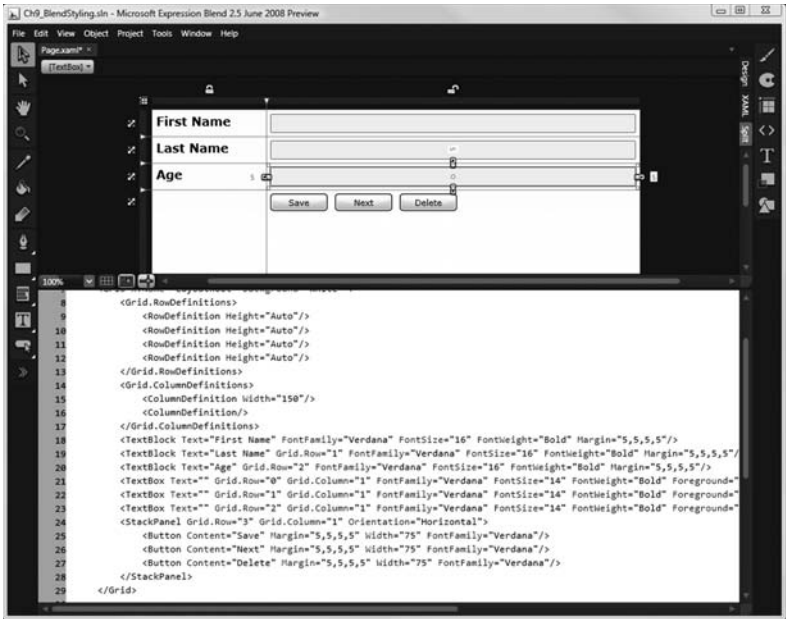

Figure 9-12. *Final project in Expression Blend*

Getting the code perfect is not the point of this exercise. It's OK if your application doesn't look exactly like my screenshot. The main objective was to get you familiar with setting and resetting inline properties in Expression Blend.

In these two exercises, you saw how to change the appearance of your Silverlight applications using inline properties in Visual Studio 2008 and Expression Blend. Although this method is very straightforward, in a normal application with a lot of controls, setting all of the properties can become tedious. And if you need to change the appearance of some elements throughout the application, it will not be an easy task. This is where Silverlight styles come in.

Silverlight Styles

In the previous section, you saw how you can change the appearance of a Silverlight application by setting inline properties. This works perfectly fine, but it presents maintenance problems. From a maintenance perspective, it's better to separate the style properties from the control definitions. For example, consider the following `TextBlock` definition:

```
<TextBlock
    Grid.Row="0"
    Grid.Column="0"
    Text="First Name"
```

```
    FontFamily="Verdana"
    FontSize="16"
    FontWeight="Bold"
    Margin="5" />
```

Suppose you defined all your `TextBlock` controls this way, throughout your application. Then, if you wanted to update the look of your application's text boxes, you would need to modify the `TextBox` definitions one by one. To save time and avoid errors, it's preferable to be able to make updates to properties related to the control's appearance in one central location, rather than in each instance of the control.

This problem is certainly not new to Silverlight. Developers and designers have faced this challenge for years with HTML-based pages. HTML solves the problem with a technology known as Cascading Style Sheets (CSS). Instead of specifying the different attributes of HTML controls directly, developers can simply specify a style for the control that corresponds to a style in a style sheet. The style sheet, not the HTML, defines all of the different appearance attributes for all controls. This way, if developers want to adjust an attribute of a control in an application, they can change it in the style sheet one time, and that change will be automatically reflected in every control in the application that references that style.

Silverlight offers a similar solution. Silverlight allows you to create style resources, in much the same way you would define styles in a CSS style sheet. In Silverlight, style resources are hierarchical, and can be defined at either the page level or the application level. If defined at the page level, the styles will be available only to controls on that page. Styles defined at the application level can be utilized by controls on all pages across the entire application. The "Silverlight Style Hierarchy" section later in this chapter provides more information about the style hierarchy.

A Silverlight style is defined using the `<Style>` element, which requires two attributes: the `Key` attribute represents the name of the style, and the `TargetType` attribute tells Silverlight which type of control gets the style. Within the `<Style>` element, the style is made up of one or more `<Setter>` elements, which define a `Property` attribute and a `Value` attribute. As an example, the preceding `TextBlock` control's appearance properties could be defined in the following Silverlight style definition:

```
<Style x:Key="FormLabel" TargetType="TextBlock">
   <Setter Property="FontFamily" Value="Verdana"/>
   <Setter Property="FontSize" Value="16"/>
   <Setter Property="FontWeight" Value="Bold"/>
   <Setter Property="Margin" Value="5,5,5,5"/>
</Style>
```

In HTML, to reference a style from a control, you simply set the style attribute. In Silverlight, this syntax looks a little different. Silverlight styles are referenced in a control using an XAML markup extension. You saw markup extensions in use in Chapter 5—when

working with data binding in Silverlight, you set a control's property using the form `{Binding, <path>`. To reference the sample `FormLabel` style from your `TextBlock`, the syntax would look as follows:

```
<TextBlock Text="Age" Grid.Row="2" Style="{StaticResource FormLabel}"/>
```

Let's give styles a try, starting with defining styles at the page level.

Try It Out: Using Styles As Static Resources

In this exercise, you will define the styles as a static resource at the page level, using Expression Blend. The application will have a very simple UI, so we can focus on styles.

1. In Expression Blend 2, create a new Silverlight 2 application named `Ch9_Styles`.

2. Double-click the `StackPanel` control in the Toolbox to add a `StackPanel`. With the `StackPanel` selected, reset the `Width` and `Height` property so the `StackPanel` will automatically resize. Next, double-click the `StackPanel` in the Objects and Timeline panel so it is selected (you should see the yellow border around the `StackPanel`). With the `StackPanel` selected, add two `TextBox` and two `Button` controls to the `StackPanel`. The Objects and Timeline panel should appear as shown in Figure 9-13.

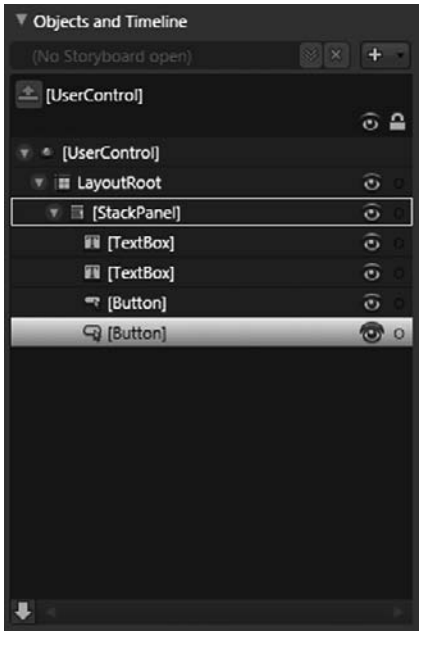

Figure 9-13. *The controls for the application in the Objects and Timeline panel*

The XAML at this point should appear as follows:

```
<Grid x:Name="LayoutRoot" Background="White" >
   <StackPanel HorizontalAlignment="Left" VerticalAlignment="Top">
      <TextBox Text="TextBox" TextWrapping="Wrap"/>
      <TextBox Text="TextBox" TextWrapping="Wrap"/>
      <Button Content="Button"/>
      <Button Content="Button"/>
   </StackPanel>
</Grid>
```

3. Run the application. As shown in Figure 9-14, at this point, it really is nothing special. Now you'll use Silverlight styles to spice up its appearance.

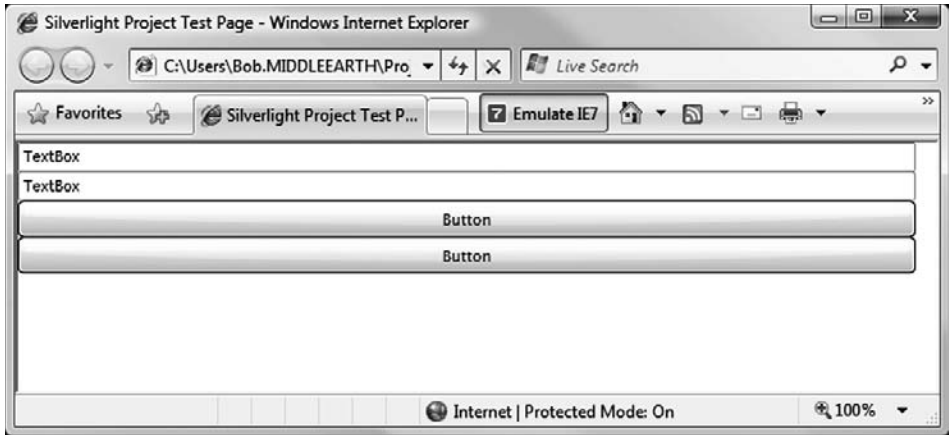

Figure 9-14. *Initial Silverlight application without styles*

4. First, you need to build your Silverlight styles. Select the first `TextBox` in the Objects and Timeline panel and select Object ➤ Edit Style ➤ Create Empty from the main menu. This will bring up the Create Style Resource dialog box. Enter `TextBoxStyle` in the Name text box, and stick with the default "Define in" option, which is to define the style in the current document. Your dialog box should look like Figure 9-15. Click OK.

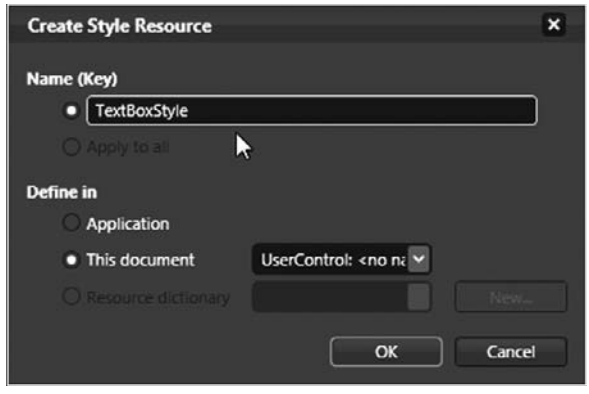

Figure 9-15. *The Create Style Resource dialog box*

At this point, you may notice a few changes:

- The Objects and Timeline panel now contains the style object, but all of the form objects are no longer visible. At the top of the Objects and Timeline panel, you will see an up arrow with the text TextBoxStyle (TextBox Style) to its right. If you hover the mouse over the arrow, you will see a message that reads "Return scope to [UserControl]," as shown in Figure 9-16. Clicking this arrow will return you to the Objects and Timeline panel that you have grown used to, with the different form objects showing.

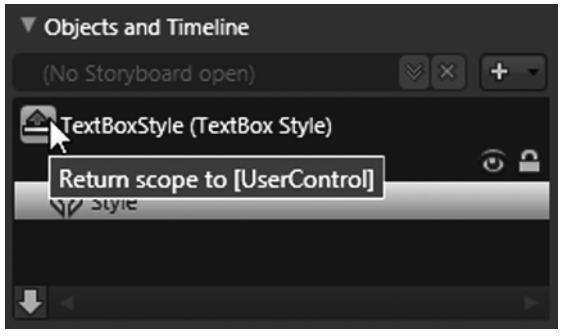

Figure 9-16. *Click the arrow next to the style name to see the controls in the UserControl's scope listed in the Objects and Timeline panel.*

- A new breadcrumb appears at the top of the artboard, as shown in Figure 9-17. The breadcrumb provides another way for you to navigate back to normal design mode.

Figure 9-17. *A new breadcrumb allows you to navigate back to normal design mode.*

- The XAML has changed. A new `<UserControl.Resources>` section has been added, and the first `TextBox` has an added `Style="{StaticResource TextBoxStyle}"` attribute, as follows:

```
<UserControl.Resources>
   <Style x:Key="TextBoxStyle" TargetType="TextBox"/>
</UserControl.Resources>

<Grid x:Name="LayoutRoot" Background="White" >
   <StackPanel HorizontalAlignment="Left" VerticalAlignment="Top">
      <TextBox Text="TextBox" TextWrapping="Wrap"
         Style="{StaticResource TextBoxStyle}"/>
      <TextBox Text="TextBox" TextWrapping="Wrap"/>
      <Button Content="Button"/>
      <Button Content="Button"/>
   </StackPanel>
</Grid>
```

5. Next, you will set the different style attributes for your `TextBoxStyle`. Make certain that the `TextBoxStyle` is still in the Objects and Timeline panel, and from the Properties panel, set the following properties:

   - `FontSize`: `22`
   - `FontFamily`: `Trebuchet MS`
   - `Foreground`: `#FFFF0000`
   - `Margin`: `5,5,5,5`

   If you now examine the XAML, you will see that Expression Blend has added a number of `Setter` elements to the `TextBoxStyle`, as follows:

```
<UserControl.Resources>
   <Style x:Key="TextBoxStyle" TargetType="TextBox">
      <Setter Property="FontSize" Value="22"/>
      <Setter Property="FontFamily" Value="Trebuchet MS"/>
```

```
        <Setter Property="Foreground" Value="#FFFF0000"/>
        <Setter Property="Margin" Value="5,5,5,5"/>
    </Style>
</UserControl.Resources>
```

6. Click the up arrow in the Objects and Timeline panel to return to the `UserControl`, and highlight the first `Button` control you added. With it selected, choose Object ➤ Edit Style ➤ Create Empty from the main menu. Name the style `ButtonStyle` and leave it as defined in this document.

7. This will create the new style `ButtonStyle` of `TargetType` `Button` and will add the `Style` attribute to the first button on your form. With the `ButtonStyle` selected, set the following properties:

   - `FontSize`: `20`
   - `FontFamily`: `Trebuchet MS`
   - `FontWeight`: `Bold`
   - `Width`: `200`
   - `Margin`: `5,5,5,5`
   - `Foreground`: `#FF0000FF`

   With these properties set, your XAML will be updated to add the new `Setter` elements to the `ButtonStyle` style, as follows:

```
<UserControl.Resources>
    <Style x:Key="TextBoxStyle" TargetType="TextBox">
        <Setter Property="FontSize" Value="22"/>
        <Setter Property="FontFamily" Value="Trebuchet MS"/>
        <Setter Property="Foreground" Value="#FFFF0000"/>
        <Setter Property="Margin" Value="5,5,5,5"/>
    </Style>
    <Style x:Key="ButtonStyle" TargetType="Button">
        <Setter Property="FontSize" Value="20"/>
        <Setter Property="FontFamily" Value="Trebuchet MS"/>
        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="Width" Value="200"/>
        <Setter Property="Foreground" Value="#FF0000FF"/>
        <Setter Property="Margin" Value="5,5,5,5"/>
    </Style>
</UserControl.Resources>
```

Now you have two styles defined, and two of your controls are set to these styles. Next, you need to set the style for your other controls.

8. Return to the `UserControl` in the Objects and Timeline panel and select the second `TextBox` control. Select Object ➤ Edit Style ➤ Apply a Resource ➤ TextBoxStyle from the main menu. This will add the `Style="{StaticResource TextBoxStyle}"` attribute to the second `TextBox`.

9. Select the second `Button` control and select Object ➤ Edit Style ➤ Apply a Resource ➤ ButtonStyle.

   Your XAML should now look as follows:

```
<UserControl.Resources>
   <Style x:Key="TextBoxStyle" TargetType="TextBox">
      <Setter Property="FontSize" Value="22"/>
      <Setter Property="FontFamily" Value="Trebuchet MS"/>
      <Setter Property="Foreground" Value="#FFFF0000"/>
      <Setter Property="Margin" Value="5,5,5,5"/>
   </Style>
   <Style x:Key="ButtonStyle" TargetType="Button">
      <Setter Property="FontSize" Value="20"/>
      <Setter Property="FontFamily" Value="Trebuchet MS"/>
      <Setter Property="FontWeight" Value="Bold"/>
      <Setter Property="Width" Value="200"/>
      <Setter Property="Foreground" Value="#FF0000FF"/>
      <Setter Property="Margin" Value="5,5,5,5"/>
   </Style>
</UserControl.Resources>

<Grid x:Name="LayoutRoot" Background="White" >
   <StackPanel HorizontalAlignment="Left" VerticalAlignment="Top">
      <TextBox Text="TextBox" TextWrapping="Wrap"
         Style="{StaticResource TextBoxStyle}"/>
      <TextBox Text="TextBox" TextWrapping="Wrap"
         Style="{StaticResource TextBoxStyle}"/>
      <Button Content="Button" Style="{StaticResource ButtonStyle}"/>
      <Button Content="Button" Style="{StaticResource ButtonStyle}"/>
   </StackPanel>
</Grid>
```

10. Run the application. The form now appears as shown in Figure 9-18.

Figure 9-18. *Silverlight application with styles Applied*

Now, let's say that you want to change the width of the text boxes in your application. Currently, their width is automatically set, but you would like to change them to a fixed width of 400 pixels. If you were using inline properties, as in the first two exercises in this chapter, you would need to set the property for each `TextBox` control in your application. However, since you are using Silverlight styles, you can simply change the `TextBoxStyle`, and all `TextBox` controls assigned to that style will be updated automatically. Let's see how this works.

11. To modify the `TextBoxStyle` property from Expression Blend, click the Resources panel. When you expand the `UserControl` item, you will see your two styles listed. To the right of `TextBoxStyle`, you will see an Edit Resource button, as shown in Figure 9-19. Click this button, and you will see that you have returned to the `TextBoxStyle`'s design scope.

Figure 9-19. *Resources panel showing the TextBoxStyle*

12. In the Properties panel, set the `Width` property of the `TextBoxStyle` to `400`. Then click the up arrow in the Objects and Timeline panel to return to the `UserControls` scope.

    Your XAML should now look as follows:

```
<Style x:Key="TextBoxStyle" TargetType="TextBox">
   <Setter Property="FontSize" Value="22"/>
   <Setter Property="FontFamily" Value="Trebuchet MS"/>
   <Setter Property="Foreground" Value="#FFFF0000"/>
   <Setter Property="Margin" Value="5,5,5,5"/>
   <Setter Property="Width" Value="400"/>
</Style>
```

13. Run the application to confirm that the width of both text boxes has been updated, as shown in Figure 9-20.

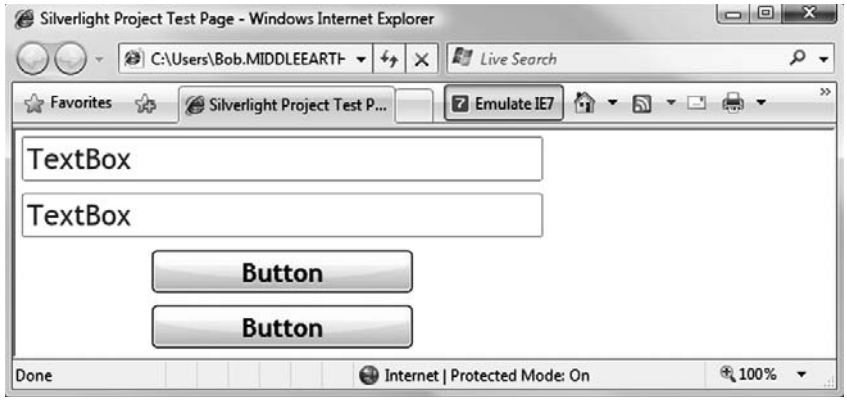

Figure 9-20. *The application with the updated TextBoxStyle*

This exercise showed how Silverlight styles can be used as an alternative to defining styles inline. As you can see, this approach provides for much cleaner XAML and also greatly improves the ease of maintaining your application.

Defining Styles at the Application Level

In the previous example, you defined the styles locally, within your `UserControl`. If you have multiple `UserControl` components that you would like to share styles, you can define the styles at the application level. As far as the controls are concerned, there is absolutely no difference. You still indicate the style for the control using the `Style="{StaticResource StyleName}"` extended attribute. What does change is where the styles are defined.

In the preceding example, your styles were defined within the `<UserControl.Resources>` element on the `UserControl` itself, as follows:

```
<UserControl.Resources>
   <Style x:Key="TextBoxStyle" TargetType="TextBox">
      <Setter Property="FontSize" Value="22"/>
      <Setter Property="FontFamily" Value="Trebuchet MS"/>
      <Setter Property="Foreground" Value="#FFFF0000"/>
      <Setter Property="Margin" Value="5,5,5,5"/>
      <Setter Property="Width" Value="400"/>
   </Style>
   <Style x:Key="ButtonStyle" TargetType="Button">
      <Setter Property="FontSize" Value="20"/>
      <Setter Property="FontFamily" Value="Trebuchet MS"/>
      <Setter Property="FontWeight" Value="Bold"/>
      <Setter Property="Width" Value="200"/>
      <Setter Property="Foreground" Value="#FF0000FF"/>
      <Setter Property="Margin" Value="5,5,5,5"/>
   </Style>
</UserControl.Resources>

<Grid x:Name="LayoutRoot" Background="White" >
   <StackPanel HorizontalAlignment="Left" VerticalAlignment="Top">
      <TextBox Text="TextBox" TextWrapping="Wrap"
         Style="{StaticResource TextBoxStyle}"/>
      <TextBox Text="TextBox" TextWrapping="Wrap"
         Style="{StaticResource TextBoxStyle}"/>
      <Button Content="Button" Style="{StaticResource ButtonStyle}"/>
      <Button Content="Button" Style="{StaticResource ButtonStyle}"/>
   </StackPanel>
</Grid>
```

In order to define the styles at the application level, instead of defining the styles in the `<UserControl.Resources>`, you move them to the `App.xaml` file within the element `<Application.Resources>`, as follows:

```
 <Application.Resources>
    <Style x:Key="TextBoxStyle" TargetType="TextBox">
        <Setter Property="FontSize" Value="22"/>
        <Setter Property="FontFamily" Value="Trebuchet MS"/>
        <Setter Property="Foreground" Value="#FFFF0000"/>
        <Setter Property="Margin" Value="5,5,5,5"/>
        <Setter Property="Width" Value="400"/>
    </Style>
```

```
    <Style x:Key="ButtonStyle" TargetType="Button">
        <Setter Property="FontSize" Value="20"/>
        <Setter Property="FontFamily" Value="Trebuchet MS"/>
        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="Width" Value="200"/>
        <Setter Property="Foreground" Value="#FF0000FF"/>
        <Setter Property="Margin" Value="5,5,5,5"/>
    </Style>
</Application.Resources>
```

That is all there is to it. Again, there are no changes at all to the controls themselves. For example, to use these styles on your `UserControl`, the XAML would still look like the following:

```
<Grid x:Name="LayoutRoot" Background="White" >
    <StackPanel HorizontalAlignment="Left" VerticalAlignment="Top">
        <TextBox Text="TextBox" TextWrapping="Wrap"
            Style="{StaticResource TextBoxStyle}"/>
        <TextBox Text="TextBox" TextWrapping="Wrap"
            Style="{StaticResource TextBoxStyle}"/>
        <Button Content="Button" Style="{StaticResource ButtonStyle}"/>
        <Button Content="Button" Style="{StaticResource ButtonStyle}"/>
    </StackPanel>
</Grid>
```

Silverlight Style Hierarchy

As I mentioned earlier in the chapter, Silverlight styles are hierarchical. When a control has a style set, Silverlight will first look for the style at the local level, within the document's `<UserControl.Resources>`. If the style is found, Silverlight will look no further. If the style is not found locally, it will look at the application level. If the style is not found there, an `XamlParseException` will be thrown.

In addition to locally defined styles overriding application-level styles, any properties that are defined inline in the control element itself will override properties within the style. For example, consider the following XAML:

```
<UserControl.Resources>
    <Style x:Key="TextBoxStyle" TargetType="TextBox">
        <Setter Property="FontSize" Value="22"/>
        <Setter Property="FontFamily" Value="Trebuchet MS"/>
        <Setter Property="Foreground" Value="#FFFF0000"/>
        <Setter Property="Margin" Value="5,5,5,5"/>
        <Setter Property="Width" Value="400"/>
```

```
    </Style>
    <Style x:Key="ButtonStyle" TargetType="Button">
        <Setter Property="FontSize" Value="20"/>
        <Setter Property="FontFamily" Value="Trebuchet MS"/>
        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="Width" Value="200"/>
        <Setter Property="Foreground" Value="#FF0000FF"/>
        <Setter Property="Margin" Value="5,5,5,5"/>
    </Style>
</UserControl.Resources>

<Grid x:Name="LayoutRoot" Background="White" >
    <StackPanel HorizontalAlignment="Left" VerticalAlignment="Top">
        <TextBox Text="TextBox" TextWrapping="Wrap"
            Style="{StaticResource TextBoxStyle}" FontSize="10"/>
        <TextBox Text="TextBox" TextWrapping="Wrap"
            Style="{StaticResource TextBoxStyle}"/>
        <Button Content="Button" Style="{StaticResource ButtonStyle}"/>
        <Button Content="Button" Style="{StaticResource ButtonStyle}"/>
    </StackPanel>
</Grid>
```

Both `TextBox` controls are set to the `TextBoxStyle` style; however, the first `TextBox` has an inline property defined for `FontSize`. Therefore, when you run the XAML, it will appear as shown in Figure 9-21.

Figure 9-21. *An example of inline properties overriding style properties*

Notice that even though `FontSize` was defined inline, the control still picked up the remaining properties from `TextBoxStyle`. However, a locally defined style will prevent any properties from being applied from an application-level style.

Summary

In this chapter, we looked at options for styling your Silverlight applications. You saw how to define style properties inline using both Visual Studio and Expression Blend. Then we explored defining styles with Silverlight styles, both at the document level and the application level. In the next chapter, we will look at using Expression Blend to define Silverlight transformations and animations.

CHAPTER 10

Transformations and Animation

Incorporating animation of objects in a web application can really enhance the UI. In the past, to implement this type of animation in a web site, you would most likely turn to Adobe Flash. The cool thing for Microsoft .NET developers is that now we can do it all within the technologies that we know, and better yet, we can code it using .NET. Personally, I consider this the most exciting aspect of Silverlight 2. For years, I have been struggling with the desire to put animations into my applications, but not doing so because I did not want to jump over to Flash. But that's no longer necessary. We can now do it all within .NET, my friends! This chapter will show you just how that's done.

Introduction to Silverlight 2 Animation

The term *animation* usually brings to mind cartoons or animated features like those that Disney has brought to life on the big screen. Artists create a number of images with slight variations that, when shown in rapid sequence, appear as fluid movement. Fundamental to any type of animation is the changing of some attribute of an object over time.

For Silverlight, the implementation of an animation is very straightforward. You change a property of an object gradually over time, such that you have the appearance of that object moving smoothly from one point to the next.

As an example, Figure 10-1 shows an icon bar that I created for one of my Silverlight 2 applications. As your mouse rolls over an icon in the bar, the icon grows; as the mouse leaves the icon, it shrinks back to its initial size. When you click one of the icons, the icon bounces, just as it does on the Mac OS X Dock.

Figure 10-1. *An animated application bar created with Silverlight*

In the example in Figure 10-1, for one of the icons, the animation that was created when the mouse was placed over the icon had two basic positions: at timestamp 0.00, the icon's `Width` and `Height` properties were set to 50 pixels; at timestamp 0.25, the `Width` and `Height` properties were set to 75 pixels. To make the transition smooth from timestamp 0.00 to 0.25, Silverlight creates a *spline*, which will generate all of the "frames" along the way to make the movement appear fluid to the human eye.

Silverlight Storyboards

In movies or cartoon animations, a *storyboard* is a sequence of sketches that depict changes of action over the duration of the film or cartoon. So, essentially, a storyboard is a timeline. In the same way, storyboards in Silverlight 2 are timelines. As an example, Figure 10-2 shows a storyboard for an application that animates the transformation of a circle and two rectangles.

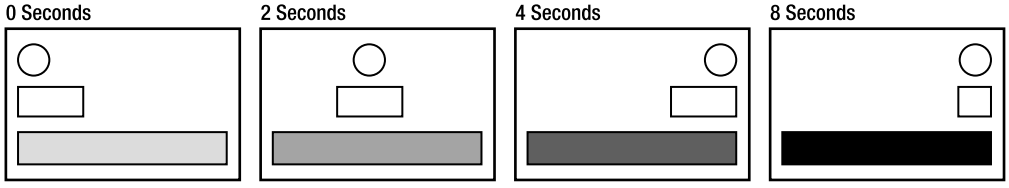

Figure 10-2. *Example of a storyboard*

In the storyboard in Figure 10-2, three objects are represented: a circle, a small rectangle, and a large rectangle. At the start of the storyboard's timeline, all three objects are on the left side of the document. After 2 seconds, the circle and smaller rectangle start to move toward the right side of the document. The larger rectangle starts to change its background

from white to black. At 4 seconds into the timeline, the circle and the smaller rectangle will have reached the right side of the document. At that time, the smaller rectangle will begin to turn into a square. At 8 seconds, the smaller rectangle will have turned into a square, and the larger rectangle will have turned fully black.

If you translate this storyboard into Silverlight animations, you will have four animations:

- Two animations that will cause the circle and the smaller square to move from the left to the right side of the document
- An animation that will change the background of the larger rectangle from white to black
- An animation to change the smaller rectangle into a square

Next, we will look at the different types of animations in Silverlight 2.

Types of Animation in Silverlight

There are two basic types of animations in Silverlight 2:

Linear interpolation animation: This type of animation smoothly and continuously varies property values over time.

Keyframe animation: With this type of animation, values change based on keyframes that have been added to a given point in the timeline.

Most commonly, keyframe animations are used in conjunction with a form of interpolation to smooth animations.

All types of animation in Silverlight 2 are derived from the `Timeline` class found in the `System.Windows.Media.Animation` namespace. The following types of animation are available:

- `ColorAnimation`
- `ColorAnimationUsingKeyFrames`
- `DoubleAnimation`
- `DoubleAnimationUsingKeyFrames`
- `ObjectAnimationUsingKeyFrames`
- `PointAnimation`
- `PointAnimationUsingKeyFrames`

Each of these animates a different type of object. For example, `ColorAnimation` animates the value of a `Color` property between two target values. Similarly, `DoubleAnimation` animates the value of a `Double` property, `PointAnimation` animates the value of a `Point` property, and `ObjectAnimation` animates the value of an `Object` property. Developers determine which animation type to use based on what they want to animate.

As an example, let's look at a very simple animation where we will increase the size of a rectangle over time, as shown in Figure 10-3. This example will allow us to dissect some of the properties involved with the animation.

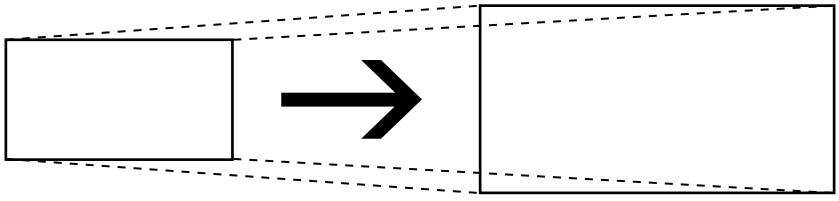

Figure 10-3. *Animation of growing a rectangle*

To perform this animation, we need to use a `DoubleAnimationUsingKeyFrames` animation, since we are modifying the `Width` and `Height` properties of the rectangle, both of which are properties of type `Double`. Let's look at the XAML used to perform this animation.

```
<UserControl.Resources>
    <Storyboard x:Name="Storyboard1">
        <DoubleAnimationUsingKeyFrames
            BeginTime="00:00:00"
            Storyboard.TargetName="rectangle"
            Storyboard.TargetProperty="Width">
            <SplineDoubleKeyFrame KeyTime="00:00:02" Value="400"/>
        </DoubleAnimationUsingKeyFrames>
        <DoubleAnimationUsingKeyFrames
            BeginTime="00:00:00"
            Storyboard.TargetName="rectangle"
            Storyboard.TargetProperty="Height">
            <SplineDoubleKeyFrame KeyTime="00:00:02" Value="240"/>
        </DoubleAnimationUsingKeyFrames>
    </Storyboard>
</UserControl.Resources>

<Grid x:Name="LayoutRoot" Background="White" >
    <Rectangle
        Height="120"
        Width="200"
```

```
        HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Stroke="#FF000000"
        x:Name="rectangle"/>
</Grid>
```

A number of elements are required. First, the rectangle itself has a name defined. This is required, as the animation needs to be able to refer to the rectangle by its name.

Next, in the storyboard, we have two animations: one to animate the width and one to animate the height.

The `BeginTime` property tells Silverlight at what time during the storyboard the animation should begin. In both cases, we are starting the animations as soon as the storyboard is initiated (`BeginTime="00:00:00"`).

The `TargetName` property tells the animation which control is being animated. In this case, both animations are targeting the rectangle.

The final property set is `TargetProperty`. This is an attached property that refers to the property that is being animated. In the case of the first animation, `TargetProperty` is set to the rectangle's `Width` property. As the animation's value is changed, the value will be set to the `Width` property of the rectangle.

Finally, since this is a keyframe animation, keyframes are defined within the animation. In our case, only one keyframe is defined, 2 seconds (`KeyTime="00:00:02"`) into the storyboard. In the first animation, 2 seconds into the storyboard's timeline, the value of the `Width` property will be changed to `400`:

```
<SplineDoubleKeyFrame KeyTime="00:00:02" Value="400"/>
```

Programmatically Controlling Animations

Once your animations have been created, Silverlight needs to know when to trigger a given animation or storyboard. Silverlight 2 provides a number of functions that allow you to programmatically control your storyboard animations. Table 10-1 lists some common storyboard methods.

Table 10-1. *Common Storyboard Animation Methods*

Method	Description
`Begin()`	Initiates the storyboard
`Pause()`	Pauses the storyboard
`Resume()`	Resumes a paused storyboard
`Stop()`	Stops the storyboard
`Seek()`	Skips to a specific part of the storyboard animation

As an example, consider a simple animation where a rectangle grows and shrinks, repeating forever. We want to allow the user to control the animation through a simple UI. Clicking the Start button starts the animation, and clicking the Stop button stops it. In addition, if the user clicks the rectangle, it will pause and resume the animation. Here's the XAML to set up the application:

```
<UserControl.Resources>
  <Storyboard x:Name="MoveRect" RepeatBehavior="Forever">
    <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="rectangle" Storyboard.TargetProperty="Width">
      <SplineDoubleKeyFrame KeyTime="00:00:00" Value="200"/>
      <SplineDoubleKeyFrame KeyTime="00:00:03" Value="600"/>
      <SplineDoubleK
eyFrame KeyTime="00:00:06" Value="200"/>
    </DoubleAnimationUsingKeyFrames>
    <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="rectangle" Storyboard.TargetProperty="Height">
      <SplineDoubleKeyFrame KeyTime="00:00:00" Value="100"/>
      <SplineDoubleKeyFrame KeyTime="00:00:03" Value="300"/>
      <SplineDoubleKeyFrame KeyTime="00:00:06" Value="100"/>
    </DoubleAnimationUsingKeyFrames>
  </Storyboard>
</UserControl.Resources>

<Grid x:Name="LayoutRoot" Background="White" >
  <Rectangle Height="100" Width="200" Fill="#FF000AFF"
      Stroke="#FF000000" StrokeThickness="3" x:Name="rectangle" />
  <Button Height="24" Margin="200,416,340,40"
      Content="Start" Width="100" x:Name="btnStart" />
  <Button Height="24" Margin="340,416,200,40"
      Content="Stop" Width="100" x:Name="btnStop" />
</Grid>
```

The UI is shown in Figure 10-4.

To implement the desired behavior, we will wire up three event handlers in the `Page` constructor.

Figure 10-4. *The setup for the example of programmatically controlling animation*

To start the animation when the user clicks the Start button, we use the storyboard's `Begin()` method. To stop the animation, we use the storyboard's `Stop()` method. The pause/resume behavior is a bit trickier, but still not complicated. We include a private Boolean property called `Paused`, which we use to tell the code behind whether or not the animation is paused. To pause and resume the animation, we use the `Pause()` and `Resume()` methods. The code looks like this:

```
private bool Paused;
public Page()
{
    // Required to initialize variables
    InitializeComponent();
    this.btnStart.Click += new RoutedEventHandler(btnStart_Click);
    this.btnStop.Click += new RoutedEventHandler(btnStop_Click);
    this.rectangle.MouseLeftButtonUp +=
        new MouseButtonEventHandler(rectangle_MouseLeftButtonUp);
}

void rectangle_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    if (Paused)
    {
        this.MoveRect.Resume();
        Paused = false;
    }
```

```
    else
    {
        this.MoveRect.Pause();
        Paused = true;
    }
}

void btnStop_Click(object sender, RoutedEventArgs e)
{
    this.MoveRect.Stop();
}

void btnStart_Click(object sender, RoutedEventArgs e)
{
    this.MoveRect.Begin();
}
```

That's all there is to it!

So far in this chapter, we have looked at some very simple animations. Of course, in reality, animations can get much more complex. One of the key advantages you have as a developer is that there are tools to assist you with these animations. Expression Blend is the tool to use when designing your Silverlight 2 animations.

Using Expression Blend to Create Animations

Although you can use Visual Studio 2008 to create your animations in Silverlight, Visual Studio does not include designer tools to assist you. If you are going to build animations programmatically, Visual Studio is the way to go. But if you are creating your animations in design mode, Expression Blend 2 has the tools that allow you to do this easily.

Viewing a Storyboard in the Expression Blend Timeline

The primary asset within Expression Blend for animations is the Objects and Timeline panel. Up to this point, we have focused on the object side of the Objects and Timeline panel. With animations, it is all about the timeline. With a storyboard selected, the timeline appears as shown in Figure 10-5.

Figure 10-5. *Expression Blend's timeline for a storyboard*

The timeline in Figure 10-5 is actually the implemented timeline for the storyboard shown earlier in Figure 10-2. The three objects in the storyboard are listed in the Objects and Timeline panel. To the right of each of these objects, you see the timeline with just over 10 seconds showing horizontally. At time 0, there are three keyframes added, indicating that some animation action is taking place at that time. Then, at 4 seconds into the timeline, you see two keyframes providing the end point of the circle and smaller rectangle's movement from left to right. At 8 seconds through the timeline, there are two final keyframes: one providing an end point for the smaller rectangle turning into a square and one changing the larger rectangle to black.

To better understand how Expression Blend can help you build your animations, let's run through an exercise.

Try It Out: Creating an Animation with Expression Blend

In this exercise, you'll create the classic bouncing ball animation using Expression Blend. You'll create an animation that will make a red ball drop and bounce on a black rectangle until it comes to rest. You'll start off with a very simple animation, and then add to it to make it progressively more realistic.

1. Create a new Silverlight application in Expression Blend 2 named `Ch10_BlendAnimations`.

2. Add an `Ellipse` control with red fill and a black border near the top center of the grid. Next, add a `Rectangle` control to the very bottom of the grid, and have it stretch all the way from left to right. Set the fill color and border color to black. Your application should appear similar to Figure 10-6.

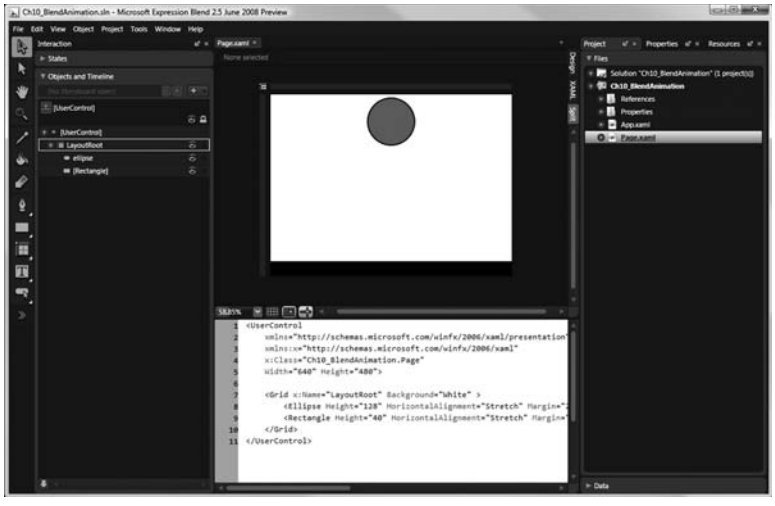

Figure 10-6. *Initial application layout*

3. The first step in creating an animation is to create a new storyboard. On the Objects and Timeline panel, click the button with the plus sign, to the right of the text "(No Storyboard open)," as shown in Figure 10-7. This opens the Create Storyboard Resource dialog box.

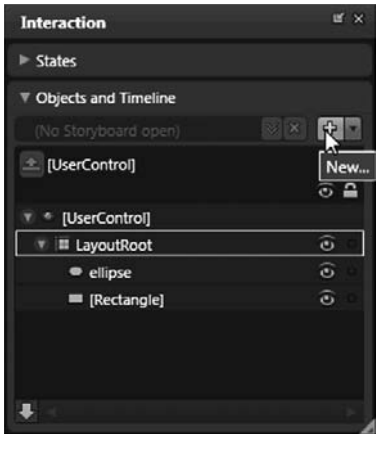

Figure 10-7. *Click the plus button to create a new storyboard.*

4. In the Create Storyboard Resource dialog box, enter `BounceBall` in the Name (Key) text box, as shown in Figure 10-8. This will be the name of your storyboard.

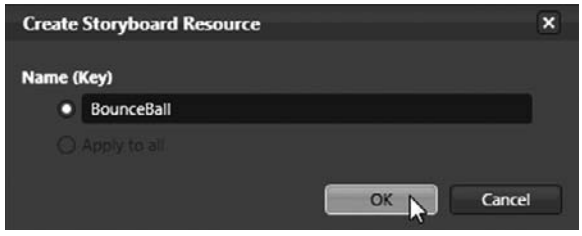

Figure 10-8. *Name your storyboard in the Create Storyboard Resource dialog box.*

5. When the storyboard is created, the timeline will be visible on the right side of the Objects and Timeline panel. To better see this, switch to the Animation workspace in Expression Blend by selecting Window ➤ Active Workspace ➤ Animation Workspace. Your workspace should now look similar to Figure 10-9.

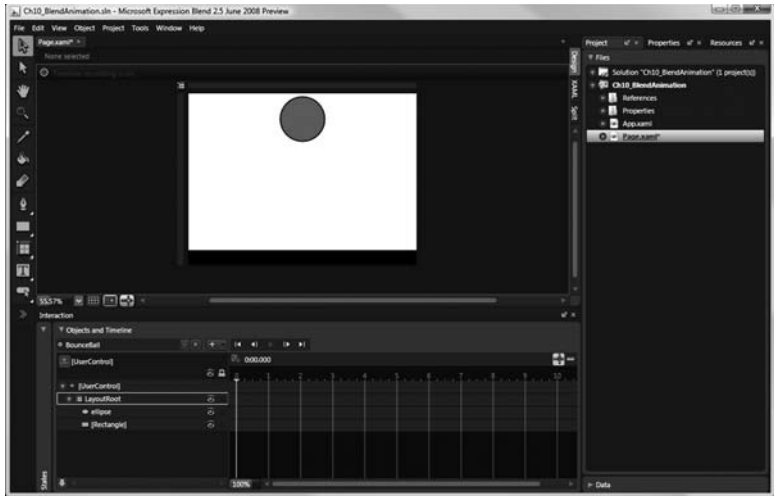

Figure 10-9. *The Animation workspace in Expression Blend*

Your animation will have many keyframes, as the ball will be moving up and down as it "bounces" on the rectangle. To simplify things, every change of direction will cause the need for a new keyframe. For your first keyframe, you will simply take the ball and drop it onto the top of the rectangle. To do this, you need to add a new keyframe and move the ball to its new position on the grid.

6. Make sure the artboard is surrounded in a red border with "Timeline recording is on" in the upper-right corner. If this is not the case, make certain that `BounceBall` is selected for the storyboard in the Object and Timeline panel, and you can click the red circle in the top-left corner to toggle between recording and not recording.

7. Move the playhead (the yellow vertical line on the timeline with the down arrow at the top), to position 3 (3 seconds), as shown in Figure 10-10.

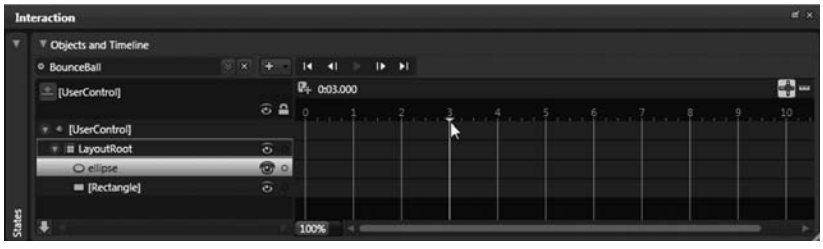

Figure 10-10. *Moving the playhead on the timeline*

8. With the playhead at 3 seconds, select the ellipse and move it down so that it is positioned directly below its starting point, but touching the black rectangle, as shown in Figure 10-11.

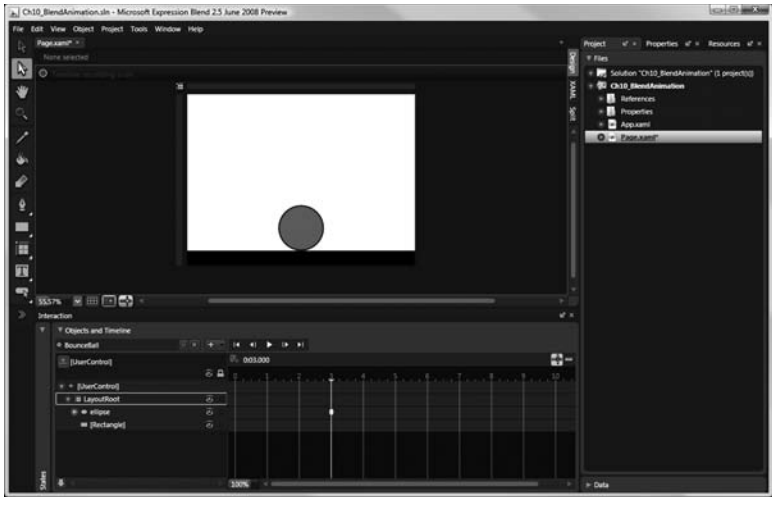

Figure 10-11. *Repositioned ball on our grid*

If you look carefully at the timeline, you'll notice that a red circle has shown up to the left of the `Ellipse` control in the Objects and Timeline panel, with a white arrow indicating that the object contains an animation. In addition, in the timeline, at position 3 seconds, a white ellipse has appeared to the right of the `Ellipse` control. This is how Expression Blend visually represents a keyframe.

At the top of the timeline, you will see buttons for navigating forward and backward between the frames in the animation. In addition, there is a play button that lets you view the animation.

9. Click the play button to view the animation. If you followed the steps properly, you will see the ball start at the top of the grid and slowly move to the top of the rectangle.

   You just created your first animation! However, it isn't very realistic. In a real environment, the ball would accelerate as it fell toward the rectangle. So its movement would start out slow and speed up. You can mimic this behavior by modifying your keyframe and adding a spline.

10. Select the newly added keyframe in the timeline. (When the keyframe is selected, it will turn gray instead of white).

    Once the keyframe is selected, in the Properties panel, you will see a section titled Easing. This section allows you to adjust the `KeySpline` property. By default, the interpolation between the two keyframes is linear. However, for this example, you want to speed up the ball as it gets closer to the second keyframe.

11. Click and drag the dot in the upper-right corner of the `KeySpline` grid (the end point of the right side of the line), and drag it down so it appears as shown in Figure 10-12.

12. Click the play button at the top of the timeline. This time, you will see that the circle starts to drop slowly and then speeds up the closer it gets to the rectangle. This makes for a much more realistic animation.

13. Next, the circle is going to bounce back up after impacting the rectangle. With recording still on, move the playhead to 6 seconds on the timeline, and then move the circle directly up from its current position to about three-fourths its initial starting point.

14. Select the new keyframe that is created, and navigate to the Easing section of the Properties panel. This time, you want the movement to start out fast and slow down as the circle reaches its apex. To get this effect, move the bottom-left dot up so the `KeySpline` curve appears as shown in Figure 10-13.

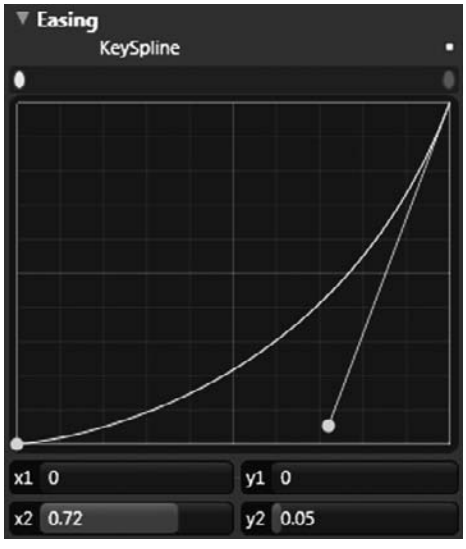

Figure 10-12. *Adjusting the KeySpline property for the ball dropping*

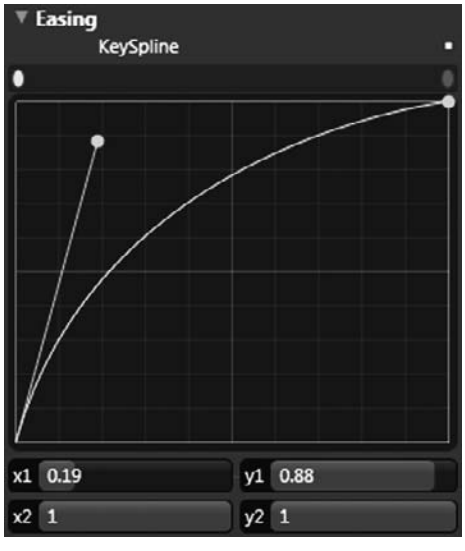

Figure 10-13. *Adjusting the KeySpline property for the ball rising*

15. Click the play button above the timeline to see the animation you have so far. The circle will fall with increasing speed, and then bounce back up with decreasing speed. So far so good, but what goes up, must come down.

16. Move the playhead to 8 seconds, and move the circle up about one-fourth its initial position and adjust the `KeySpline` property to match Figure 10-12. Sticking with the pattern, move the playhead to 10 seconds, and move the circle down to the top of the rectangle. The `KeySpline` curve should match Figure 10-13. Repeat this pattern at 11 seconds, and then 11.5 seconds.

17. Click the play button. You should see the circle bounce on the rectangle as you would expect. The final timeline will appear as shown in Figure 10-14.

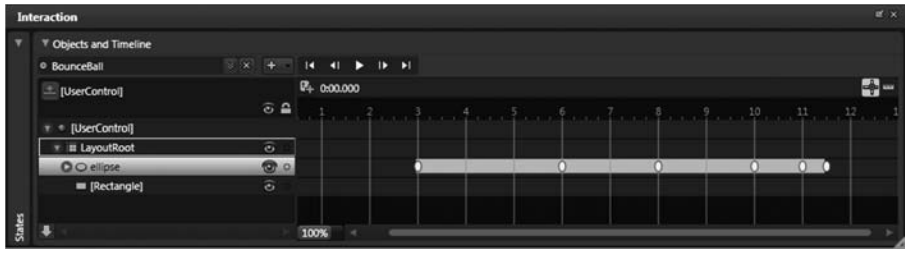

Figure 10-14. *Final timeline for bouncing ball*

Next, you need to tell Silverlight when the animation should take place. We will keep it simple and have the animation start when the page is loaded.

18. Navigate to the code behind for the `Page.xaml` file. In the `Page()` constructor, add the event handler for the `Loaded` event, as follows:

```
public Page()
{
    // Required to initialize variables
    InitializeComponent();
    this.Loaded += new RoutedEventHandler(Page_Loaded);
}

void Page_Loaded(object sender, RoutedEventArgs e)
{
    this.BounceBall.Begin();
}
```

19. Run the application. At this point, you should see the ball bounce on the rectangle. You might see something like what is shown in Figure 10-15.

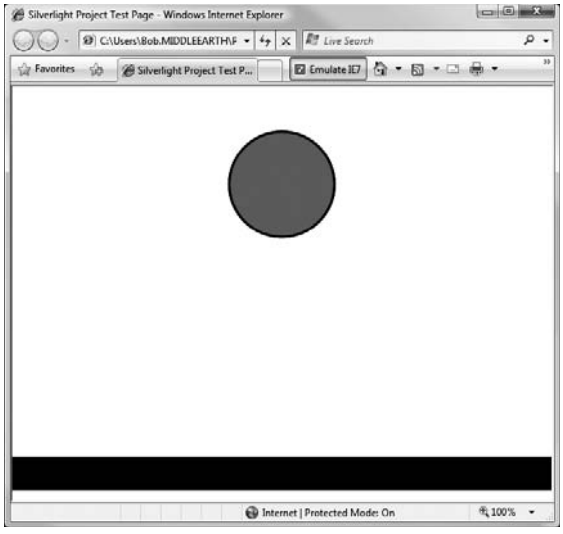

Figure 10-15. *Finished bouncing ball animation application*

In this section, we discussed animations in Silverlight 2. You should be comfortable creating new animations for your application in Expression Blend, and modifying and programming against those animations in Visual Studio 2008. The next section addresses transformations in Silverlight 2.

Creating Transformations in Silverlight

Silverlight 2 includes a number of 2D *transforms*, which are used to change the appearance of objects. Transforms in Silverlight are defined using a transformation matrix, which is a mathematical construct for mapping points from one coordinate space to another. If this sounds a bit confusing, do not fear, Silverlight 2 abstracts this matrix.

Silverlight 2 supports four transformation types: rotation, scaling, skewing, and translation.

■ **Note** You can also define your own transformation matrix, if you need to modify or combine the four transformation types. See *Pro Silverlight 2* by Matthew MacDonald (Apress, 2008) for details on how to do this.

Transformation Types

Figure 10-16 shows a Silverlight 2 application that has been divided into four grid cells. Each cell contains two rectangles that have their width and height set to 100 pixels. One

of the rectangles in each cell has a border with its width set to 1 pixel, and the other has a border with its width set to 5 pixels. The rectangle with the thicker border was then transformed, so you can see the result of the transformation.

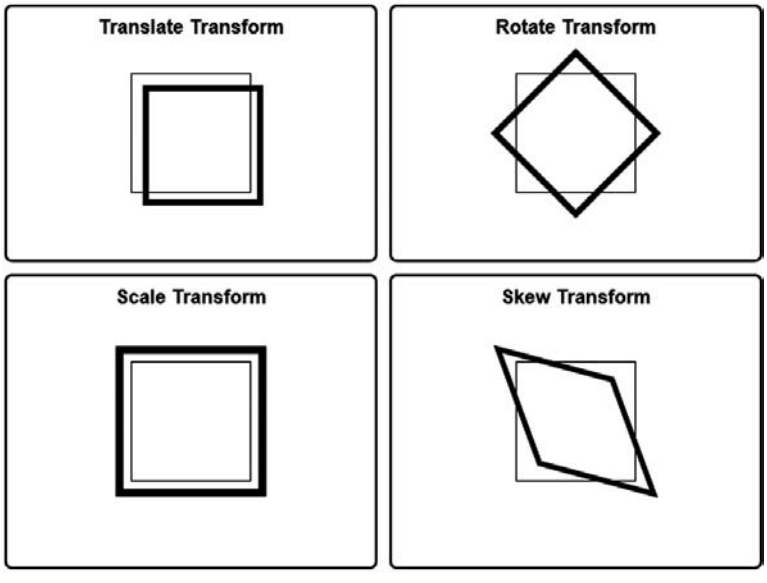

Figure 10-16. *Examples of the four transformation types*

ScaleTransform

The `ScaleTransform` type allows you to transform the size of a Silverlight object. The `ScaleX` property is used to scale the object on the horizontal axis, and the `ScaleY` property is used to scale the object on the vertical axis. The values of these properties are multiples of the object's original size. For example, setting the `ScaleX` property to `2` will double the size of the object on the horizontal axis. The following XAML was used to create the `ScaleTransform` in Figure 10-16.

```
<Rectangle Height="100" Width="100" Stroke="#FF000000" Grid.Row="1" Grid.Column="0"
    StrokeThickness="5" RenderTransformOrigin="0.5,0.5">
  <Rectangle.RenderTransform>
    <TransformGroup>
      <ScaleTransform ScaleX="1.25" ScaleY="1.25"/>
    </TransformGroup>
  </Rectangle.RenderTransform>
</Rectangle>
```

SkewTransform

The SkewTransform type allows you to skew a Silverlight object horizontally and vertically. The SkewTransform is used most commonly to create a 3D effect for an object. The AngleX property is used to skew the object horizontally, and AngleY is used to skew the object vertically. The following XAML was used to create the SkewTransform in Figure 10-16:

```
<Rectangle Height="100" Width="100" Stroke="#FF000000" Grid.Row="1" Grid.Column="1"
    StrokeThickness="5" RenderTransformOrigin="0.5,0.5">
  <Rectangle.RenderTransform>
    <TransformGroup>
      <SkewTransform AngleX="20" AngleY="15"/>
    </TransformGroup>
  </Rectangle.RenderTransform>
</Rectangle>
```

RotateTransform

The RotateTransform type allows you to rotate a Silverlight object by a specified angle around a specified center point. The angle is specified by the Angle property, and the center point is specified by the RenderTransformOrigin property. When you create a RotateTransform for a rectangle in Expression Blend, by default, it will set RenderTransformOrigin to 0.5, 0.5, which is the center of the object. You can also specify the center point using the CenterX and CenterY properties on the RotateTransform element. The following is the XAML to produce the RotateTransform in Figure 10-16:

```
<Rectangle Height="100" Width="100" Stroke="#FF000000" Grid.Row="0" Grid.Column="1"
    StrokeThickness="5" RenderTransformOrigin="0.5,0.5">
  <Rectangle.RenderTransform>
    <TransformGroup>
      <RotateTransform Angle="45"/>
    </TransformGroup>
  </Rectangle.RenderTransform>
</Rectangle>
```

TranslateTransform

The TranslateTransform type allows you to change the position of a Silverlight object, both horizontally and vertically. The X property controls the position change on the horizontal axis, and the Y property controls the change to the vertical axis. The following XAML was used to create the TranslateTransform in Figure 10-16:

```
<Rectangle Height="100" Width="100" Stroke="#FF000000" Grid.Row="0" Grid.Column="0"
    StrokeThickness="5" RenderTransformOrigin="0.5,0.5">
  <Rectangle.RenderTransform>
    <TransformGroup>
```

```
      <TranslateTransform X="10" Y="10"/>
    </TransformGroup>
  </Rectangle.RenderTransform>
</Rectangle>
```

Now that we have covered the basics of transforms in Silverlight 2, let's run through a quick exercise that will give you a chance to try them out for yourself.

Try It Out: Using Expression Blend to Transform Silverlight Objects

In this exercise, you'll use Expression Blend to add and animate transformations.

1. Create a new Silverlight application in Expression Blend called `Ch10_BlendTransforms`. Add two `ColumnDefinition` elements and two `RowDefinition` elements so the root `Grid` is equally divided into four cells, as follows:

```
<Grid x:Name="LayoutRoot" Background="White" >
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
</Grid>
```

2. Next, add two rectangles to each of the cells that you just created. Create two sets of rectangles: one set with `StrokeThickness="1"` and another with `StrokeThickness="5"`. Also, name the second set of rectangles `recTrans`. Add the following code:

```
<Grid x:Name="LayoutRoot" Background="White" >

  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
```

```
<Rectangle Grid.Row="0" Grid.Column="0" Height="100"
    Width="100" Stroke="#FF000000" StrokeThickness="1" />
<Rectangle Grid.Row="0" Grid.Column="1" Height="100"
    Width="100" Stroke="#FF000000" StrokeThickness="1"  />
<Rectangle Grid.Row="1" Grid.Column="0" Height="100"
    Width="100" Stroke="#FF000000" StrokeThickness="1"  />
<Rectangle Grid.Row="1" Grid.Column="1" Height="100"
    Width="100" Stroke="#FF000000" StrokeThickness="1"  />

<Rectangle Grid.Row="0" Grid.Column="0" Height="100"
    Width="100" Stroke="#FF000000" StrokeThickness="5" x:Name="recTrans" />
<Rectangle Grid.Row="0" Grid.Column="1" Height="100"
    Width="100" Stroke="#FF000000" StrokeThickness="5" x:Name="recRotate" />
<Rectangle Grid.Row="1" Grid.Column="0" Height="100"
    Width="100" Stroke="#FF000000" StrokeThickness="5" x:Name="rectScale" />
<Rectangle Grid.Row="1" Grid.Column="1" Height="100"
    Width="100" Stroke="#FF000000" StrokeThickness="5" x:Name="rectSkew" />

</Grid>
```

At this point, your application should have four squares equally spaced in the four cells of your application. The next step will be to introduce your transforms, but instead of just adding the transforms, you are going to animate the transformation taking place.

3. Using the techniques discussed earlier in this chapter, create a new storyboard called `TransformElements`.

4. You will perform the transformations over 2 seconds, so move the playhead on the timeline to 2 seconds. Select the rectangle named `recTrans`. In the Properties panel, find the Transform section. Select the Translate tab. Set `X` and `Y` to `25`. This will cause the top-left square to move down and to the right, as shown in Figure 10-17.

5. Highlight the rectangle named `recRotate`. In the Transform section of the Properties panel, select the Rotate tab. Set the `Angle` property to `45`. The top-right square will rotate 45 degrees, as shown in Figure 10-18.

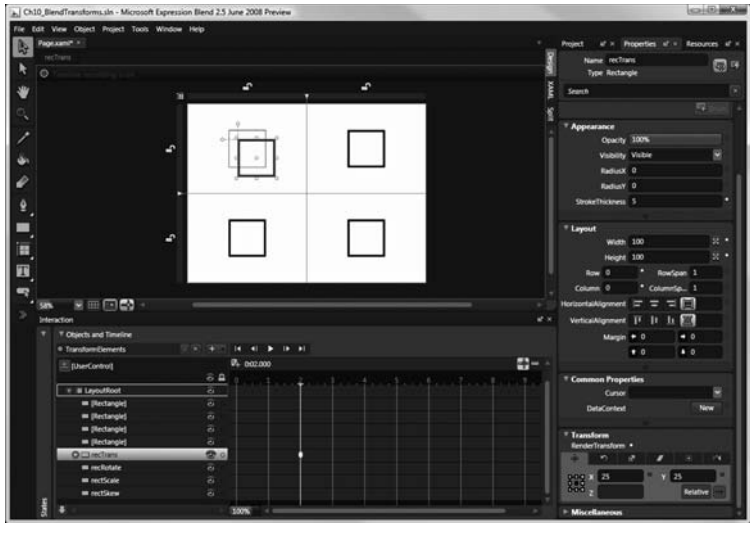

Figure 10-17. *Adding the TranslateTransform*

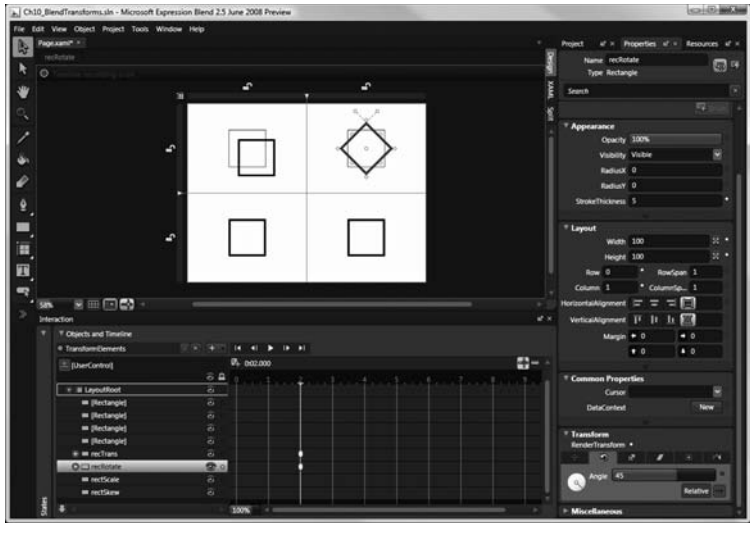

Figure 10-18. *Adding the RotateTransform*

6. Select the rectangle named `rectScale`. In the Transform section of the Properties panel, select the Scale tab. Set the values of the `X` and `Y` properties to `1.5`, which will scale the bottom-left square 1.5x, or 150%, as shown in Figure 10-19.

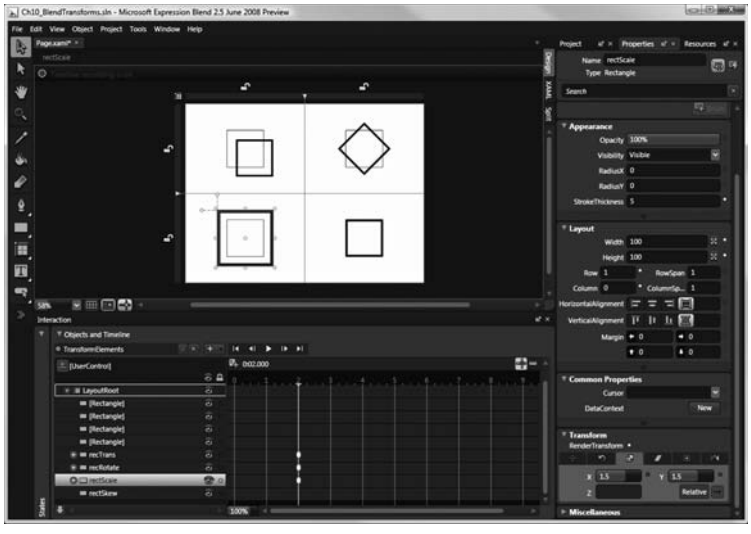

Figure 10-19. *Adding the ScaleTransform*

7. Select the rectangle named `rectSkew`. In the Transform section of the Properties panel, select the Skew tab. Set the values of the `X` and `Y` properties to `20`. This will cause the square to skew into a diamond shape, as shown in Figure 10-20.

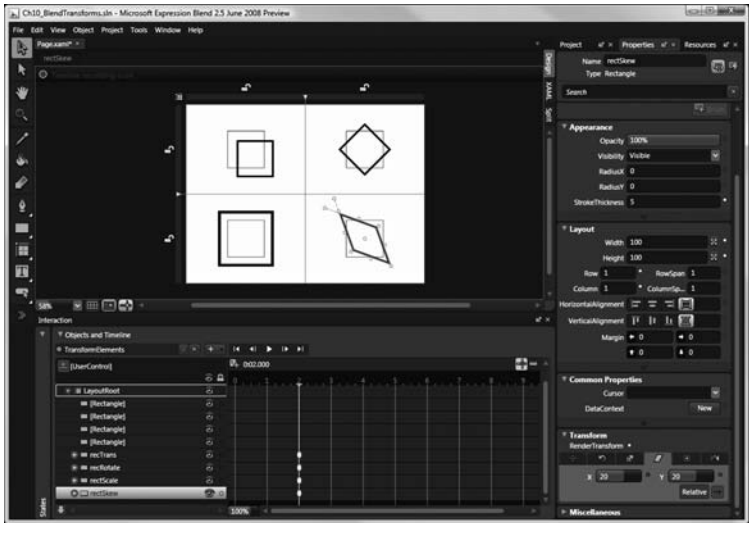

Figure 10-20. *Adding the SkewTransform*

8. Click the play button at the top of the timeline, and watch the objects transform from their original shapes and locations.

As you've seen in this exercise, applying transformations is pretty straightforward.

Summary

This chapter covered creating animations in Silverlight 2. We looked at animations from a high level, discussed the different elements that make up an animation in Silverlight 2, and explored how to programmatically control animations in the code behind. We also looked at how Expression Blend 2 helps you create complex animations. Then we shifted our focus to transformations in Silverlight 2. We looked at each of the four transform types, and then created a simple Silverlight application utilizing transforms.

In the following chapter, we will look at the more advanced topic of creating your own Silverlight 2 custom controls. Custom controls allow you to create Silverlight functionality that can be easily reused in different Silverlight applications.

CHAPTER 11

Custom Controls

So far in this book, you have learned about the many elements of Silverlight 2 and how they can be used to build RIAs. But what if Silverlight doesn't offer the specific functionality you need for an application? In that case, you may want to create a custom control to provide that additional functionality.

The actual procedure for creating custom controls is not that terribly difficult, but understanding the process can be. Under the hood, Silverlight performs some complex work, but most Silverlight developers do not need to know these details. However, in order to understand custom controls and the process used to build them, we must dive in and see how Silverlight ticks.

In this chapter, we will examine when it is appropriate to write custom controls in Silverlight 2. Then we will look at the Silverlight Control Toolkit and the controls it offers for developers to use in their applications. Next, we will explore the different aspects of the Silverlight 2 control model. Finally, we will build a custom control for Silverlight 2.

When to Write Custom Controls

When you find that none of the existing Silverlight controls do exactly what you want, creating a custom control is not always the solution. In fact, in most cases, you should be able to get by without writing custom controls. Due to the flexibility built into the Silverlight 2 controls, you can usually modify an existing one to suit your needs.

As a general rule, if your goal is to modify the appearance of a control, there is no need to write a custom control. Silverlight controls that are built properly, following Microsoft's best practices, will adopt the Parts and States model, which calls for complete separation of the logical and visual aspects of your control. Due to this separation, developers can change the appearance of controls, and even change transitions of the controls between different states, without needing to write custom controls.

So, just when is creating a custom control the right way to go? Here are the primary reasons for writing custom controls:

Abstraction of functionality: When developing your applications, you may need to implement some functionality that can be achieved using Silverlight 2's out-of-the-box support. However, if this functionality needs to be reused often in your application, you may choose to create a custom control that abstracts the functionality, in order to simplify the application. An example of this would be if you wanted to have two text boxes next to each other for first and last names. Instead of always including two `TextBox` controls in your XAML, you could write a custom control that would automatically include both text boxes and would abstract the behavior surrounding the text boxes.

Modification of functionality: If you would like to change the way a Silverlight 2 control behaves, you can write a custom control that implements that behavior, perhaps inheriting from an existing control. An example of this would be if you wanted to create a button that pops up a menu instead of simply triggering a click method.

Creation of new functionality: The most obvious reason for writing a custom control in Silverlight 2 is to add functionality that does not currently exist in Silverlight. As an example, you could write a control that acts as a floating window that can be dragged and resized.

Although these are valid reasons for creating custom controls, there is one more resource you should check before you do so: the Silverlight Control Toolkit.

Silverlight Control Toolkit

Upon the release of Silverlight 2, Microsoft announced the Silverlight Control Toolkit, an open source project located on CodePlex at `http://www.codeplex.com/SilverlightToolkit`. This toolkit provides additional components and controls that you can download for use in your Silverlight applications. For example, it includes the fully functional charting controls shown in Figure 11-1.

Microsoft's target is to eventually have more than 100 controls available through this open source toolkit. For developers, this means that as Silverlight 2 matures, more and more controls will be available for use in your applications.

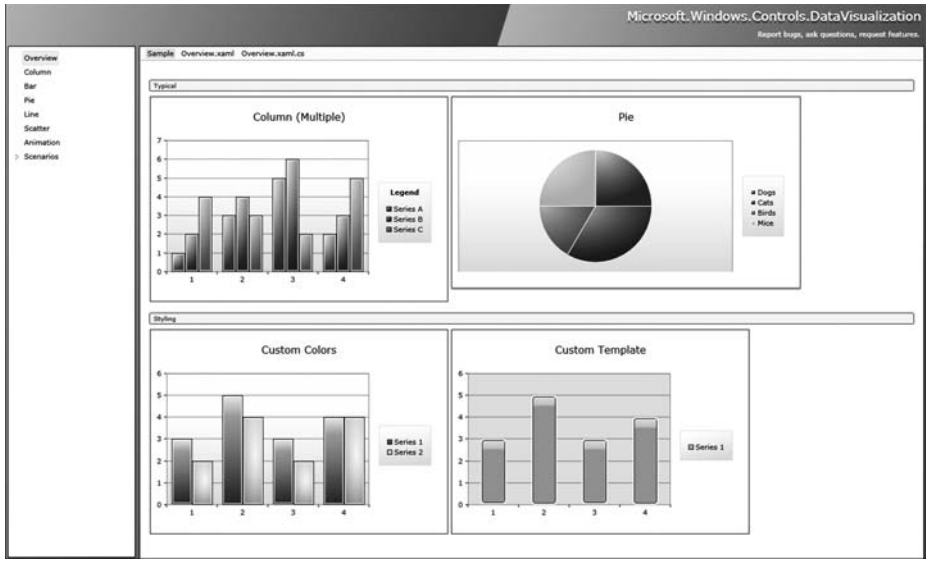

Figure 11-1. *Charting control in the Silverlight Control Toolkit*

The Silverlight Control Toolkit contains four "quality bands" that describe the specific control's maturity level: experimental, preview, stable, and mature. With the initial announcement of the Silverlight Control Toolkit, the following twelve controls (six within the preview band and six in the stable band) are available for download (including the full source code):

- `AutoCompleteBox`
- `NumericUpDown`
- `Viewbox`
- `Expander`
- `ImplicitStyleManager`
- `Charting`
- `TreeView`
- `DockPanel`
- `WrapPanel`

- `Label`
- `HeaderedContentControl`
- `HeaderedItemsControl`

This toolkit is an excellent resource for Silverlight 2 developers. You can use these controls as is in your applications, or you can use the source code to modify your own controls. They are also a great way to learn how to build custom controls, because you can examine their source code. In order to understand that source code, you will need to know about the Silverlight control model.

Silverlight Control Model

Before you start to build custom controls for Silverlight 2, you should understand the key concepts of the Silverlight 2 control model. In this section, we will look at two of these concepts:

- The Parts and States model
- Dependency properties

Parts and States Model

Following Microsoft's best practices, Silverlight 2 controls are built with a strict separation between the visual aspects of the control and the logic behind the control. This allows developers to create templates for existing controls that will dramatically change the visual appearance and the visual behaviors of a control, without needing to write any code. This separation is called for by the Parts and States model. The visual aspects of controls are managed by Silverlight's Visual State Manager (VSM).

■ **Note** You are not required to adhere to the Parts and State model when developing custom controls. However, developers are urged to do so in order to follow the best practices outlined by Microsoft.

The Parts and States model uses the following terminology:

Parts: Named elements contained in a control template that are manipulated by code in some way are called *parts*. For example, a simple `Button` control could consist of a rectangle that is the body of the button and a text block that represents the text on the control.

States: A control will always be in a *state*. For a `Button` control, different states include when the mouse is hovered over the button, when the mouse is pressed down on the button, and when neither is the case (its default or normal state). The visual look of control is defined by its particular state.

Transitions: When a control changes from one state to another—for example, when a `Button` control goes from its normal state to having the mouse hovered over it—its visual appearance may change. In some cases, this change may be animated to provide a smooth visual transition from the states. These animations are defined in the Parts and States model by *transitions*.

State group: According to the Parts and States model, control states can be grouped into mutually exclusive groups. A control cannot be in more than one state within the same state group at the same time.

Dependency Properties

Properties are a common part of object-oriented programming and familiar to .NET developers. Here is a typical property definition:

```
private string _name;
public string Name
{
    get { return _name; }
    set { _name = value; }
}
```

In Silverlight 2 and WPF, Microsoft has added some functionality to the property system. This new system is referred to as the *Silverlight 2 property system*. Properties created based on this new property system are called *dependency properties*.

In a nutshell, dependency properties allow Silverlight 2 to determine the value of a property dynamically from a number of different inputs, such as data binding or template binding. As a general rule, if you want to be able to style a property or to have it participate in data binding or template binding, it must be defined as a dependency property.

You define a property as a dependency property using the `DependencyProperty` object, as shown in the following code snippet:

```
public static readonly DependencyProperty NameProperty =
    DependencyProperty.Register(
        "Name",
        typeof(string),
        typeof(MyControl),
        null
        );
```

```
public int Name
{
    get
    {
        return (string)GetValue(NameProperty);
    }
    set
    {
        SetValue(NameProperty, value);
    }
}
```

This example defines the `Name` property as a dependency property. It declares a new object of type `DependencyProperty` called `NameProperty`, following the naming convention detailed by Microsoft. `NameProperty` is set equal to the return value of the `DependencyProperty.Register()` method, which registers a dependency property within the Silverlight 2 property system.

The `DependencyProperty.Register()` method is passed a number of arguments:

- The name of the property that you are registering as a dependency property—`Name` in this example.
- The data type of the property you are registering—`string` in this example.
- The data type of the object that is registering the property—`MyControl` in this example.
- Metadata that should be registered with the dependency property. Most of the time, this will be used to hook up a callback method that will be called whenever the property's value is changed. This example simply passes `null`. In the next section, you will see how this last argument is used.

Now that we have discussed custom controls in Silverlight 2 from a high level, it's time to see how to build your own.

Creating Custom Controls in Silverlight 2

As I mentioned at the beginning of the chapter, creating a custom control does not need to be difficult. Of course, the work involved depends on how complex your control needs to be. As you'll see, the custom control you'll create in this chapter is relatively simple. Before we get to that exercise, let's take a quick look at the two options for creating custom controls.

Implementing Custom Functionality

You have two main options for creating custom functionality in Silverlight 2:

With a `UserControl`: The simplest way to create a piece of custom functionality is to implement it with a `UserControl`. Once the `UserControl` is created, you can then reuse it across your application.

As a custom control: The content that is rendered is built from scratch by the developer. This is by far the most complex option for creating a custom control. You would need to do this when you want to implement functionality that is unavailable with the existing controls in Silverlight 2.

In this chapter's exercise, we will take the custom control approach.

Try It Out: Building a Custom Control

In this exercise, you will build your own "cooldown" button. This button will be disabled for a set number of seconds—its cooldown duration—after it is clicked. If you set the cooldown to be 3 seconds, then after you click the button, you will not be able to click it again for 3 seconds.

For demonstration purposes, you will not use the standard Silverlight 2 `Button` control as the base control. Instead, you will create a custom control that implements `Control`. This way, I can show you how to create a control with a number of states.

The cooldown button will have five states, implemented in two state groups. The `NormalStates` state group will have these states:

- `Pressed`: The button is being pressed. When it is in this state, the thickness of the button's border will be reduced.
- `MouseOver`: The mouse is hovering over the button. When it is in this state, the thickness of the button's border will be increased.
- `Normal`: The button is in its normal state.

It will also have a state group named `CoolDownStates`, which will contain two states:

- `Available`: The button is active and available to be clicked.
- `CoolDown`: The button is in its cooldown state, and therefore is not active. You will place a rectangle over top of the button that is of 75% opacity. In addition, you will disable all other events while the button is in this state.

Keep in mind that this is only an example, and it has many areas that could use improvement. The goal of the exercise is not to produce a control that you will use in your applications, but rather to demonstrate the basic steps for creating a custom control in Silverlight 2.

Setting Up the Control Project

Let's get started by creating a new project for the custom control.

1. In Visual Studio 2008, create a new Silverlight application named `Ch11_CoolDownButton` and allow Visual Studio to create a Web Site project to host your application.

2. From Solution Explorer, right-click the solution and select Add ➤ New Project.

3. In the Add New Project dialog box, select the Silverlight Class Library template and name the library `CoolDownButton`, as shown in Figure 11-2.

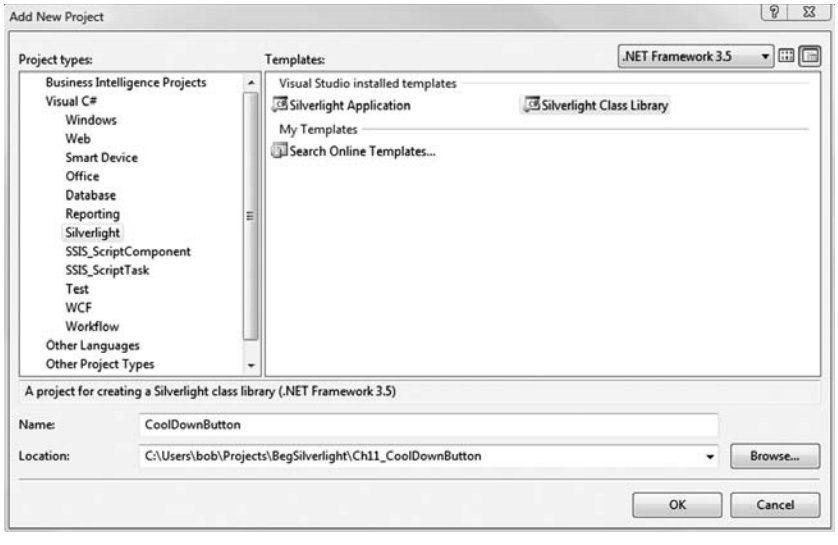

Figure 11-2. *Adding the Silverlight Class Library to the project*

4. By default, Visual Studio will create a class named `Class1.cs`. Delete this file from the project.

5. Right-click the `CoolDownButton` project and select Add ➤ New Item.

6. In the Add New Item dialog box, select the Class template and name the class `CoolDownButtonControl`, as shown in Figure 11-3.

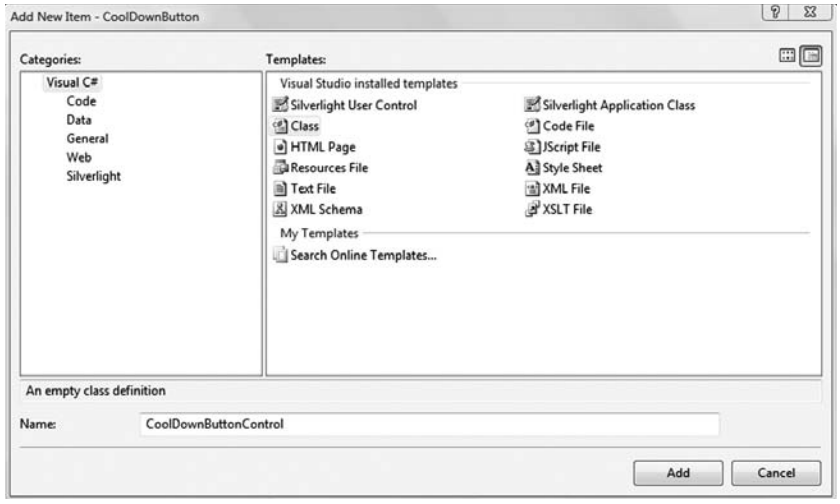

Figure 11-3. *Adding the new class to the project*

Defining Properties and States

Now you're ready to create the control. Let's begin by coding the properties and states.

1. Set the control class to inherit from `Control`, in order to gain the base Silverlight 2 control functionality, as follows:

```
namespace CoolDownButton
{
    public class CoolDownButtonControl : Control
    {

    }
}
```

2. Now add the control's public properties, as follows:

```
public static readonly DependencyProperty CoolDownSecondsProperty =
    DependencyProperty.Register(
        "CoolDownSeconds",
        typeof(int),
        typeof(CoolDownButtonControl),
        new PropertyMetadata(
            new PropertyChangedCallback(
                CoolDownButtonControl.OnCoolDownSecondsPropertyChanged
                )
```

```
            )
        );

    public int CoolDownSeconds
    {
        get
        {
            return (int)GetValue(CoolDownSecondsProperty);
        }
        set
        {
            SetValue(CoolDownSecondsProperty, value);
        }
    }

    private static void OnCoolDownSecondsPropertyChanged(
        DependencyObject d, DependencyPropertyChangedEventArgs e)
    {
        CoolDownButtonControl cdButton = d as CoolDownButtonControl;

        cdButton.OnCoolDownButtonChange(null);
    }

    public static readonly DependencyProperty ButtonTextProperty =
        DependencyProperty.Register(
            "ButtonText",
            typeof(string),
            typeof(CoolDownButtonControl),
            new PropertyMetadata(
                new PropertyChangedCallback(
                    CoolDownButtonControl.OnButtonTextPropertyChanged
                    )
                )
            );

    public string ButtonText
    {
        get
        {
            return (string)GetValue(ButtonTextProperty);
        }
        set
```

```
    {
        SetValue(ButtonTextProperty, value);
    }
}

private static void OnButtonTextPropertyChanged(
    DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    CoolDownButtonControl cdButton = d as CoolDownButtonControl;
    cdButton.OnCoolDownButtonChange(null);
}

protected virtual void OnCoolDownButtonChange(RoutedEventArgs e)
{

}
```

As explained earlier in the chapter, in order for your properties to allow data binding, template binding, styling, and so on, they must be dependency properties. In addition to the dependency properties, you added two callback methods that will be called when the properties are updated. By naming convention, the `CoolDownSeconds` property has a `DependencyProperty` object named `CoolDownSecondsProperty` and a callback method of `onCoolDownSecondsPropertyChanged()`. So you need to watch out, or your names will end up very long, as they have here.

3. Add some private members to contain state information, as follows:

```
namespace CoolDownButton
{
    public class CoolDownButtonControl : Control
    {
        ...

        private FrameworkElement corePart;
        private bool isPressed, isMouseOver, isCoolDown;
        private DateTime pressedTime;

    }
}
```

The `corePart` members are of type `FrameworkElement` and will hold the instance of the main part, which will respond to mouse events. The `isPressed`, `isMouseOver`, and `isCoolDown` Boolean members will be used to help keep track of the current button state. And the `pressedTime` member will record the time that the button was clicked in order to determine when the cooldown should be removed.

4. Add a helper method called `GoToState()`, which will assist in switching between the states of the control.

```
private void GoToState(bool useTransitions)
{
    //  Go to states in NormalStates state group
    if (isPressed)
    {
        VisualStateManager.GoToState(this, "Pressed", useTransitions);
    }
    else if (isMouseOver)
    {
        VisualStateManager.GoToState(this, "MouseOver", useTransitions);
    }
    else
    {
        VisualStateManager.GoToState(this, "Normal", useTransitions);
    }

    //  Go to states in CoolDownStates state group
    if (isCoolDown)
    {
        VisualStateManager.GoToState(this, "CoolDown", useTransitions);
    }
    else
    {
        VisualStateManager.GoToState(this, "Available", useTransitions);
    }
}
```

This method will check the private members you added in the previous step to determine in which state the control should be. When the proper state is determined, the `VisualStateManager.GoToState()` method is called, passing it the control, the name of the state, and whether or not the control should use transitions when switching from the current state to this new state (whether or not an animation should be shown).

Now let's turn our attention to the visual aspect of the control.

Defining the Control's Appearance

The default control template is placed in a file named `generic.xaml`, which is located in a folder named `themes`. These names are required. The `generic.xaml` is a resource dictionary that defines the built-in style for the control. You need to add the folder and file, make some adjustments to the file, and then add the XAML to set the control's appearance.

1. To add the required folder, right-click the `CoolDownButton` project and select Add ➤ New Folder. Name the folder `themes`.

2. Right-click the newly added `themes` folder and select Add ➤ New Item.

3. In the Add New Item dialog box, select the Silverlight User Control template and name the file `generic.xaml`, as shown in Figure 11-4. Click Add and confirm that the `generic.xaml` file was added within the `themes` folder.

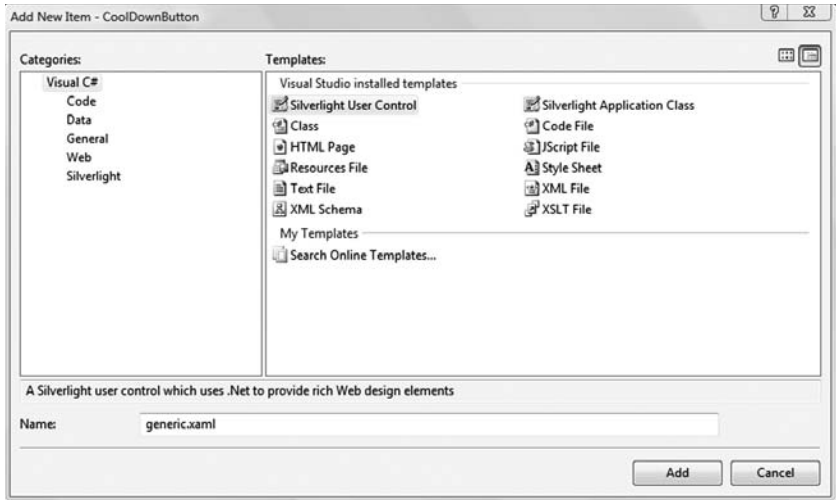

Figure 11-4. *Adding the generic.xaml resource dictionary*

4. In Solution Explorer, expand the `generic.xaml` file to see the `generic.xaml.cs` file. Right-click it and delete this code-behind file.

5. Right-click the `generic.xaml` file and select Properties. Change the Build Action to Resource and remove the resource for the Custom Tool property, as shown in Figure 11-5.

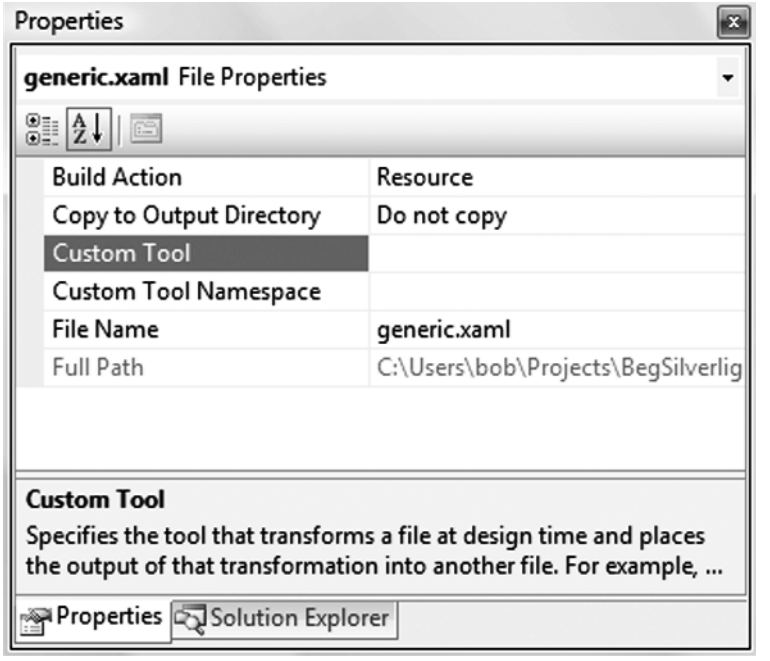

Figure 11-5. *The Properties panel for generic.xaml*

6. Open the `generic.xaml` file. You will see that, by default, the file has the following contents:

```
<UserControl x:Class="CoolDownButton.themes.generic"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">

    </Grid>
</UserControl>
```

7. You need to change the `generic.xaml` file to be a resource dictionary. To do this, replace the `UserControl` tag with a `ResourceDictionary` tag. Then remove the `Width` and `Height` definitions and add a new `xmlns` for the `CoolDownButton`. Finally, remove the `Grid` definition. Your code should look like this:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:begSL2="clr-namespace:CoolDownButton">
</ResourceDictionary>
```

8. Now you can add the actual XAML that will make up the control. First, add a `Style` tag, with the `TargetType` set to `CoolDownButtonControl`. Then add a `Setter` for the control template, and within that, add the `ControlTemplate` definition, again with `TargetType` set to `CoolDownButtonControl`. The control will consist of two `Rectangle` components: one for the button itself, named `coreButton`, one for the 75% opacity overlay that will displayed when the button is in its `CoolDown` state. It will also have a `TextBlock` component to contain the text of the button. This defines the control in the default state. Therefore, the opacity of the overlay rectangle is set to 0% to start, because the overlay should not be visible by default. The additions are as follows:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:begSL2="clr-namespace:CoolDownButton">
<Style TargetType="begSL2:CoolDownButtonControl">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="begSL2:CoolDownButtonControl">
                <Grid x:Name="LayoutRoot">
                    <Rectangle
                        StrokeThickness="4"
                        Stroke="Navy"
                        Fill="AliceBlue"
                        RadiusX="4"
                        RadiusY="4"
                        x:Name="innerButton" />
                    <TextBlock
                        HorizontalAlignment="Center"
                        VerticalAlignment="Center"
                        Text="Test"
                        TextWrapping="Wrap"/>
                    <Rectangle
                        Opacity="0"
                        Fill="#FF000000"
                        Stroke="#FF000000"
                        RenderTransformOrigin="0.5,0.5"
                        RadiusY="4" RadiusX="4"
                        x:Name="corePart">
                        <Rectangle.RenderTransform>
                            <TransformGroup>
                                <ScaleTransform
```

```
                                    ScaleX="1"
                                    ScaleY="1"/>
                            </TransformGroup>
                        </Rectangle.RenderTransform>
                    </Rectangle>
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
</ResourceDictionary>
```

9. Now that you have defined the default appearance of the control, you need to add the `VisualStateGroups`, along with the different states for the control. To do this, add the following code directly below the `Grid` definition and above the first `Rectangle`. Notice that for each state, a `Storyboard` is used to define the state's visual appearance.

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="NormalStates">

        <VisualState x:Name="Normal"/>

        <VisualState x:Name="MouseOver" >
            <Storyboard >
                <DoubleAnimation
                    Storyboard.TargetName="innerButton"
                    Storyboard.TargetProperty="(UIElement.StrokeThickness)"
                    Duration="0" To="6"/>
            </Storyboard>

        </VisualState>
        <VisualState x:Name="Pressed">

            <Storyboard>
                <DoubleAnimation
                    Storyboard.TargetName="innerButton"
                    Storyboard.TargetProperty="(UIElement.StrokeThickness)"
                    Duration="0" To="2"/>
            </Storyboard>

        </VisualState>
```

```
        </VisualStateGroup>

        <VisualStateGroup x:Name="CoolDownStates">

            <VisualState x:Name="Available"/>
            <VisualState x:Name="CoolDown">
                <Storyboard>
                    <DoubleAnimation
                        Storyboard.TargetName="corePart"
                        Storyboard.TargetProperty="(UIElement.Opacity)"
                        Duration="0" To=".75"/>
                </Storyboard>
            </VisualState>

        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
```

Now we need to turn our attention back to the `CoolDownButtonControl.cs` file to finish up the logic behind the control.

Handling Control Events

To complete the control, you need to handle its events and define its control contract.

1. First, you must get an instance of the core part. Referring back to step 8 in the "Defining the Control's Appearance" section, you'll see that this is the overlay rectangle named `corePart`. This is the control on top of the other controls, so it is the one that will accept the mouse events. To get the instance of `corePart`, use the `GetChildElement()` method. Call this method in the `OnApplyTemplate()` method that is called whenever a template is applied to the control, as follows:

```
public override void OnApplyTemplate()
{
    base.OnApplyTemplate();

    CorePart = (FrameworkElement)GetTemplateChild("corePart");

    GoToState(false);
}
```

```
private FrameworkElement CorePart
{
    get
    {
        return corePart;
    }

    set
    {
        corePart = value;
    }
}
```

Notice that this method calls the base `OnApplyTemplate()` method, and then calls the `GoToState()` method, passing it `false`. This is the first time that the `GoToState()` method will be called, and you are passing it `false` so that it does not use any transitions while changing the state. The initial view of the control should not have any animations to get it to the initial state.

2. At this point, you need to wire up event handlers to handle the mouse events. First, create the event handlers themselves, as follows:

```
void corePart_MouseEnter(object sender, MouseEventArgs e)
{
    isMouseOver = true;
    GoToState(true);
}

void corePart_MouseLeave(object sender, MouseEventArgs e)
{
    isMouseOver = false;
    GoToState(true);
}

void corePart_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    isPressed = true;
    GoToState(true);
}
```

```
void corePart_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    isPressed = false;
    isCoolDown = true;
    pressedTime = DateTime.Now;
    GoToState(true);
}
```

3. Next, wire up the handlers to the events. You can do this in the `CorePart` property's setter, as follows. Note that in the case where more than one template is applied, before wiring up the event handlers, you need to make sure to remove any existing event handlers.

```
private FrameworkElement CorePart
{
    get
    {
        return corePart;
    }

    set
    {
        FrameworkElement oldCorePart = corePart;

        if (oldCorePart != null)
        {
            oldCorePart.MouseEnter -=
                new MouseEventHandler(corePart_MouseEnter);
            oldCorePart.MouseLeave -=
                new MouseEventHandler(corePart_MouseLeave);
            oldCorePart.MouseLeftButtonDown -=
                new MouseButtonEventHandler(
                    corePart_MouseLeftButtonDown);
            oldCorePart.MouseLeftButtonUp -=
                new MouseButtonEventHandler(
                    corePart_MouseLeftButtonUp);
        }

        corePart = value;
```

```
            if (corePart != null)
            {
                corePart.MouseEnter +=
                    new MouseEventHandler(corePart_MouseEnter);
                corePart.MouseLeave +=
                    new MouseEventHandler(corePart_MouseLeave);
                corePart.MouseLeftButtonDown +=
                    new MouseButtonEventHandler(
                        corePart_MouseLeftButtonDown);
                corePart.MouseLeftButtonUp +=
                    new MouseButtonEventHandler(
                        corePart_MouseLeftButtonUp);
            }
        }
    }
```

4. Recall that when the button is clicked, you need to make sure the button is disabled for however many seconds are set as the cooldown period. To do this, first create a method that checks to see if the cooldown time has expired, as follows:

```
private bool CheckCoolDown()
{
    if (!isCoolDown)
    {
        return false;
    }
    else
    {
        if (DateTime.Now > pressedTime.AddSeconds(CoolDownSeconds))
        {
            isCoolDown = false;
            return false;
        }
        else
        {
            return true;
        }
    }
}
```

The logic behind this method is pretty simple. If the `isCoolDown` flag is `true`, then you are simply checking to see if the current time is greater than the `pressedTime` added to the cooldown. If so, you reset the `isCoolDown` flag and return `false`; otherwise, you return `true`.

5. Now you need to surround the code in each of the event handlers with a call to the `CheckCoolDown()` method, as follows. If the cooldown has not yet expired, none of the event handlers should perform any action.

```
void corePart_MouseEnter(object sender, MouseEventArgs e)
{
    if (!CheckCoolDown())
    {
        isMouseOver = true;
        GoToState(true);
    }
}

void corePart_MouseLeave(object sender, MouseEventArgs e)
{
    if (!CheckCoolDown())
    {
        isMouseOver = false;
        GoToState(true);
    }
}

void corePart_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    if (!CheckCoolDown())
    {
        isPressed = true;
        GoToState(true);
    }
}

void corePart_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    if (!CheckCoolDown())
    {
        isPressed = false;
```

```
            isCoolDown = true;
            pressedTime = DateTime.Now;
            GoToState(true);
        }
    }
```

6. Recall that in step 2 of the "Defining Properties and States" section, you created a method called `OnCoolDownButtonChange()`. At that time, you did not place anything in this method. This is the method that is called whenever there is a notification change to a dependency property. When a change occurs, you need to call `GoToState()` so the control can reflect the changes, as follows:

```
protected virtual void OnCoolDownButtonChange(RoutedEventArgs e)
{
    GoToState(true);
}
```

7. Next, create a constructor for your control and apply the default style key. In many cases, this will simply be the type of your control itself.

```
public CoolDownButtonControl()
{
    DefaultStyleKey = typeof(CoolDownButtonControl);
}
```

8. The final step in creating the control is to define a control contract that describes your control. This is required in order for your control to be modified by tools such as Expression Blend 2. This contract consists of a number of attributes that are placed directly in the control class, as follows. These attributes are used only by tools; they are not used by the runtime.

```
namespace CoolDownButton
{
    [TemplatePart(Name = "Core", Type = typeof(FrameworkElement))]
    [TemplateVisualState(Name = "Normal", GroupName = "NormalStates")]
    [TemplateVisualState(Name = "MouseOver", GroupName = " NormalStates")]
    [TemplateVisualState(Name = "Pressed", GroupName = " NormalStates")]
    [TemplateVisualState(Name = "CoolDown", GroupName="CoolDownStates")]
    [TemplateVisualState(Name = "Available", GroupName="CoolDownStates")]
    public class CoolDownButtonControl : Control
    {
    }
}
```

This completes the creation of the custom control.

Compiling and Testing the Control

Now you're ready to try out your new control.

1. Compile your control.

2. If everything compiles correctly, you need create an instance of your control in your `Ch11_CoolDownButton` project. To do this, right-click the `Ch11_CoolDownButton` project in Solution Explorer and select Add Reference. In the Add Reference dialog box, select the Projects tab and choose `CoolDownButton`, as shown in Figure 11-6. Then click OK.

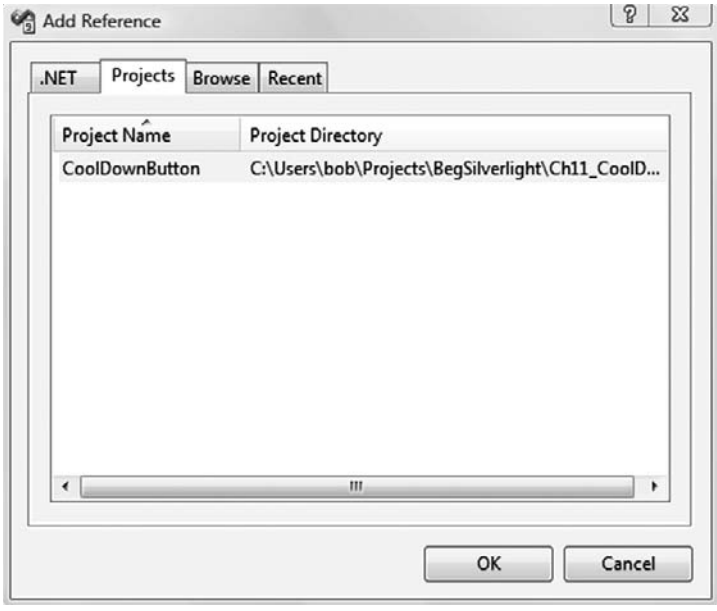

Figure 11-6. *Adding a reference to your control*

3. Navigate to your `Page.xaml` file within the `Ch11_CoolDownButton project`. First add a new `xmlns` to the `UserControl` definition, and then add an instance of your control, as follows:

```
<UserControl x:Class="Ch11_CoolDownButton.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:begSL2="clr-namespace:CoolDownButton;assembly=CoolDownButton"
    Width="400" Height="300">
```

```
    <Grid x:Name="LayoutRoot" Background="White">
        <begSL2:CoolDownButtonControl
            CoolDownSeconds="3"
            Width="150" Height="60" />
    </Grid>
</UserControl>
```

4. Run the project. You should see your button.

5. Test the states of your button. When you move the mouse over the button, the border thickness will increase. Click the mouse on the button, and the border will decrease. When you release the mouse button on the button, the border will go back to normal, and the overlay will appear. You can continue to move the mouse over the button, and you will notice that it will not respond to your events until 3 seconds have passed. Figure 11-7 shows the various control states.

Figure 11-7. *Button states*

Clearly, this cooldown button has a lot of room for improvement. However, the goal was to show you the basic steps involved in creating a custom control. As you most certainly could tell, the process is pretty involved, but the rewards of following the best practices are worth it. When the control is built properly like this, you can apply custom templates to it to dramatically change its appearance, without needing to rewrite any of the code logic.

Summary

Without a doubt, this was the most complex content that we have covered in this book. The goal was to give you a basic understanding of what is involved in creating custom controls the right way in Silverlight 2.

In this chapter, we looked at when you might want to create a custom control. Then you learned about some of the key concepts within the Silverlight 2 control model, including the Parts and States model and dependency properties. Finally, you built your own custom control.

Index

D

S

T

U

V

W

X

Y