Managing Data and Media in Microsoft Silverlight 4: A mashup of chapters from Packt's bestselling Silverlight books

Chapter No. 3
"An Introduction to Data Binding"
In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.3 "An Introduction to Data Binding"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Gastón C. Hillar has been working with computers since he was eight. He began programming with the legendary Texas TI-99/4A and Commodore 64 home computers in the early 80s.

He has a Bachelor's degree in Computer Science—graduated with honors—and an MBA (Masters in Business Administration)—graduated with an outstanding thesis. He worked as a developer, architect, and a project manager for many companies in Buenos Aires, Argentina. Now, he is an independent IT consultant and a freelance author always looking for new adventures around the world. He also works with electronics (he is an electronics technician). He is always researching about new technologies and writing about them. He owns an IT and electronics laboratory with many servers, monitors, and measuring instruments.


He is also the author of more than 40 books in Spanish about computer science, modern hardware, programming, systems development, software architecture, business applications, balanced scorecard applications, IT project management, the Internet, and electronics.

For More Information:
He contributes to Dr. Dobb's Go Parallel programming portal http://www.ddj.com/go-parallel/ and he is a guest blogger at Intel Software Network http://software.intel.com

He usually writes articles for Spanish magazines Mundo Linux, Solo Programadores, and Resistor.

Gill Cleeren is Microsoft Regional Director (http://www.theregion.com), Silverlight MVP (former ASP.NET MVP), and Telerik MVP. He lives in Belgium where he works as a .NET architect at Ordina (http://www.ordina.be/). He is passionate about .NET and always plays with the newest bits. In his role as Regional Director, Gill has given many sessions, webcasts, and training on new as well as existing technologies, such as Silverlight, ASP.NET, and WPF at conferences including TechEd Berlin 2010, TechDays Belgium – Switzerland – Sweden, DevDays NL, NDC Oslo Norway, DevReach Bulgaria, NRW Conference Germany, Spring Conference UK, Telerik Silverlight Roadshow in Sweden, and Telerik RoadShow UK. He is the author of Packt's Microsoft Silverlight 4 Data and Services Cookbook and is also the author of many articles in various developer magazines and for SilverlightShow.net. He organizes the yearly Community Day event in Belgium and also leads Visug (http://www.visug.be), the largest .NET user group in Belgium. You can find his blog at http://www.snowball.be and on Twitter by following @gillcleeren.

Kevin Dockx lives in Belgium and works at RealDolmen, one of Belgium's biggest ICT companies, where he is a 30-year old technical specialist/project leader on .NET web applications, mainly Silverlight, and a solution manager for Rich Applications (Silverlight, Windows Phone 7, WPF, Surface, and HTML5). His main focus is on all things related to Silverlight, but he still keeps an eye on the new developments concerning other products from the Microsoft .NET (Web) Stack. As a Silverlight enthusiast, he's a regular speaker on various national and international events, such as Microsoft DevDays in the Netherlands, Microsoft Techdays in Portugal, NDC Oslo Norway, and Community Day Belgium. He is the author of Packt's Microsoft Silverlight 4 Data and Services Cookbook and also writes articles for various Silverlight-related sites and magazines. His blog, which contains various tidbits on Silverlight, .NET, and the occasional rambling, can be found at http://blog.kevindockx.com/, and you can find him on Twitter as well: @KevinDockx.
Todd Snyder has been a software developer/architect for over 16 years. During that time, he has spent several years as a consultant providing technical guidance and leadership for the development of enterprise class systems on the Microsoft Platform. At Infragistics, he is a principal consultant that focuses on the design and construction of RIA and n-tier based applications. Todd is the co-leader for the New Jersey .Net user group (http://www.njdotnet.net/) and is a frequent speaker at trade shows, code camps, and Firestarters.

Joel Eden, PhD has been working in the area of user experience and design methods for over 10 years. Currently a Senior Interaction Designer working on UX Tools at Infragistics, he spent three years in the Infragistics Services group, consulting for external clients. Prior to Infragistics, he worked at multiple design agencies in the Philadelphia area, as well as working at Lockheed Martin's Advanced Technology Labs. Joel holds a B.S. in Computer Science, and a Ph.D in Information Science, both from Drexel University.

Jeffrey Smith has been a Visual Designer for six years. During that time he has been an Art Director at various agencies and studied special effects and animation at NYU. A convert from flash and flex, he has been working with .NET technologies for the past two years, specializing in WPF and Silverlight. At Infragistics, he is an UX Visual Designer that focuses on the design, implementation, and user experience. You can view some of his work at http://www.thinksinkstudio.com

Matthew Duffield is a .NET architect in designing and developing enterprise applications. He specializes in .NET with an emphasis on WPF, Silverlight, and WP7 development. He is a Microsoft MVP in Client Application Development and has an MSCD.NET certification. He also works in business intelligence, designing, and developing solutions for data warehouse and data mining initiatives. You can read his blog at mattduffield.wordpress.com and follow him on Twitter at @mattduffield.

Cameron Albert is an independent software development consultant with over 10 years of experience, specializing in Microsoft technologies such as Silverlight, WPF, WCF, SQL Server, and ASP.NET. Having worked in the medical, insurance, and media/entertainment industries, Cameron has been involved in a variety of development solutions featuring a broad range of technical issues. He also dabbles in game development utilizing Silverlight and maintains a blog detailing his exploits into the development world here: http://www.cameronAlbert.com.

For More Information:
Frank LaVigne has been hooked on with software development since he was 12, when he got his own Commodore 64 computer. Since then, he's worked as developer for financial firms on Wall Street and also in Europe. He has worked on various Tablet PC solutions and on building advanced user experiences in Silverlight and WPF. He lives in the suburbs of Washington, DC. He founded the CapArea.NET User Group Silverlight Special Interest Group and has been recognized by Microsoft as a Tablet PC MVP. He blogs regularly at www.FranksWorld.com.

Vibor Cipan is currently serving as the CEO and Partner of FatDUX Zagreb—a full service interactive UX and service design agency with offices around the world. Before joining FatDUX, Vibor, he worked at Microsoft Development Center in Copenhagen and before that, at Microsoft, Croatia. One thing, however, has stayed constant—his focus on user experience, service design, usability, and information architecture. He has been awarded the prestigious title of Microsoft Most Valuable Professional for three years in a row (and is still currently holding that title). He was the youngest awardee and the first one in the CEE, Europe to receive the award while being a full-time student.

For More Information:
Managing Data and Media in Microsoft Silverlight 4: A mashup of chapters from Packt's bestselling Silverlight books

A Packt Compendium is a book formed by drawing existing content from several related Packt titles. In other words, it is a mashup of published Packt content – Professional Expertise Distilled in the true sense. Such a compendium of Packt's content allows you to learn from each of the chapters' unique styles and Packt does its best to compile the chapters without breaking the narrative flow for the reader.

Please note that the chapters in this compendium were originally written and intended as a part of various separate Packt titles, so you might find that the information included in this instance is more akin to that of a standalone chapter, rather than creating step-by-step, continuous flowing prose. We are sure that you will find this medley a useful and valuable resource with which you can benefit from a range of Packt books—and their authors' expertise!

*Managing Data and Media in Microsoft Silverlight 4: A mashup of chapters from Packt's bestselling Silverlight* focuses on showing .NET developers how to interact with, and handle multiple sources of data in Silverlight business applications, and how to solve particular data problems following a practical hands-on approach, using real-world examples. This book is a collection of media- and data-based chapters from Packt's bestselling Silverlight books:

1. *Silverlight 4 User Interface Cookbook*
3. *Microsoft Silverlight 4 Data and Services Cookbook*
4. *Microsoft Silverlight 4 and SharePoint 2010 Integration*
5. *Microsoft Silverlight 4: Building Rich Enterprise Dashboards*
6. *3D Game Development with Microsoft Silverlight 3: Beginner’s Guide*

For More Information:  
Microsoft Silverlight is a programmable web browser plugin that enables features including animation, vector graphics, and audio-video playback features that characterize Rich Internet Applications. However, Silverlight is a great Line-Of-Business platform and is increasingly being used to build data-driven business applications. This book will enable .NET developers to get their finger on the pulse of data-driven business applications in Silverlight.

In this book, you will find content in various easy-to-follow styles such as a recipe-based cookbook format, tutorial-based beginner's guide, and a reference-styled handbook. The aim of this book is to provide a lot of valuable content to you from various other Packt Silverlight books. It is designed in such a way that you can refer to the topics chapter-by-chapter, and read them in no particular order. It offers clear examples to successfully perform the most important data-related tasks in Silverlight.

**What This Book Covers**

The book starts with discussion on layouts and content organization and covers all the options available to access data and communicate with services to get the most out of data in your Silverlight business applications, at the same time providing a rich user experience. Understand sophisticated data access techniques in your Silverlight business applications by binding data to Silverlight controls, validating data in Silverlight, getting data from services into Silverlight applications, and much more! Discover the tips, tricks, and hands on experience to create, customize, and design rich enterprise dashboards with Silverlight from a distinguished team of User Experience and Development authors.

*Chapter 1, Layouts and General Content Organization,* covers important layout considerations to be made before we start building any application, regardless of being a web application built with Silverlight or a typical desktop application built with Windows Presentation Foundation.

*Chapter 2, Handling Data,* covers the process of collecting and handling data input from a customer and saving that input on the server. We also looked at how to bind data to control properties and how to provide simple data validation using the built-in visual states provided in the textbox control.

*Chapter 3, An Introduction to Data Binding,* explains how data binding allows us to build data-driven applications in Silverlight in a much easier and much faster way compared to old-school methods of displaying and editing data.

*Chapter 4, Advanced Data Binding,* explores the data binding engine that gives many points where we can extend or change the binding process.

*Chapter 5, The Data Grid,* shows how to work with the DataGrid. This is an essential control for applications that rely on (collections of) data.

For More Information:
Chapter 6, Talking to REST and WCF Data Services, here, we'll first look at talking with REST services from Silverlight. Secondly, we'll look at how to work with WCF Data Services (formerly known as ADO.NET Data Services), which are also pure REST services at their base.

Chapter 7, Interacting with Data on the SharePoint Server, will cover many topics that help us create simple and complex Line-Of-Business Silverlight RIAs that run as Silverlight Web Parts to interact with data in the SharePoint Server.

Chapter 8, Interacting with Rich Media and Animations, will cover many topics related to retrieving digital assets from SharePoint libraries through the SharePoint Silverlight Client Object Model and consuming them in a Silverlight RIA.

Chapter 9, Data Access Strategies, will introduce you to the features included in SharePoint 2010 for hosting Silverlight dashboard applications. We will explore how to set up a Silverlight web part, and use the SharePoint Silverlight Client Object Model to communicate with data hosted in SharePoint.

Chapter 10, Building Dashboards in SharePoint and Silverlight, will explore the different data access strategies you can use while building a Silverlight application. How to build your own custom data services using SOAP, REST, and OData, a walkthrough of how to consume externally-hosted services, and how the cross-domain security policy system works with Silverlight to call external services.

Chapter 11, Working with 3D Characters, will take 3D elements from popular and professional 3D DCC tools and we will show them rendered in real-time on the screen.
An Introduction to Data Binding

This chapter is taken from Silverlight 4 Data and Services Cookbook (Chapter 2) by Gill Cleeren, Kevin Dockx.

In this chapter, we will cover:

- Displaying data in Silverlight applications
- Creating dynamic bindings
- Binding data to another UI element
- Binding collections to UI elements
- Enabling a Silverlight application to automatically update its UI
- Obtaining data from any UI element it is bound to
- Using the different modes of data binding to allow persisting data
- Data binding from Expression Blend 4
- Using Expression Blend 4 for sample data generation

For More Information:
Introduction

Data binding allows us to build data-driven applications in Silverlight in a much easier and much faster way compared to old-school methods of displaying and editing data. This chapter takes a look at how data binding works. We’ll start by looking at the general concepts of data binding in Silverlight 4 in this chapter.

Analyzing the term data binding immediately reveals its intentions. It is a technique that allows us to bind properties of controls to objects or collections thereof.

The concept is, in fact, not new. Technologies such as ASP.NET, Windows Forms, and even older technologies such as MFC (Microsoft Foundation Classes) include data binding features. However, WPF’s data binding platform has changed the way we perform data binding; it allows loosely coupled bindings. The BindingsSource control in Windows Forms has to know of the type we are binding to, at design time. WPF’s built-in data binding mechanism does not. We simply define to which property of the source the target should bind. And at runtime, the actual data—the object to which we are binding—is linked. Luckily for us, Silverlight inherits almost all data binding features from WPF and thus has a rich way of displaying data.

A binding is defined by four items:

- **The source or source object**: This is the data we are binding to. The data that is used in data binding scenarios is in-memory data, that is, objects. Data binding itself has nothing to do with the actual data access. It works with the objects that are a result of reading from a database or communicating with a service. A typical example is a Customer object.

- **A property on the source object**: This can, for example, be the Name property of the Customer object.

- **The target control**: This is normally a visual control such as a TextBox or a ListBox control. In general, the target can be a DependencyObject. In Silverlight 2 and Silverlight 3, the target had to derive from FrameworkElement; this left out some important types such as transformations.

- **A property on the target control**: This will, in some way—directly or after a conversion—display the data from the property on the source.

The data binding process can be summarized in the following image:

![Data Binding Process Diagram](image)

For More Information:
In the previous image, we can see that the data binding engine is also capable of synchronization. This means that data binding is capable of updating the display of data automatically. If the value of the source changes, Silverlight will change the value of the target as well without us having to write a single line of code. Data binding isn't a complete black box either. There are hooks in the process, so we can perform custom actions on the data flowing from source to target, and vice versa. These hooks are the converters.

Our applications can still be created without data binding. However, the manual process—that is getting data and setting all values manually on controls from code-behind—is error prone and tedious to write. Using the data-binding features in Silverlight, we will be able to write more maintainable code faster.

In this chapter, we'll explore how data binding works. We'll start by building a small data-driven application, which contains the most important data binding features, to get a grasp of the general concepts. We'll also see that data binding isn't tied to just binding single objects to an interface; binding an entire collection of objects is supported as well. We'll also be looking at the binding modes. They allow us to specify how the data will flow (from source to target, target to source, or both). We'll finish this chapter by looking at the support that Blend 4 provides to build applications that use data binding features.

In the recipes of this chapter, we'll assume that we are building a simple banking application using Silverlight. Each of the recipes in this chapter will highlight a part of this application where the specific feature comes into play. The following screenshot shows the resulting Silverlight banking application:

For More Information:
Displaying data in Silverlight applications

When building Silverlight applications, we often need to display data to the end user. Applications such as an online store with a catalogue and a shopping cart, an online banking application and so on, need to display data of some sort.

Silverlight contains a rich data binding platform that will help us to write data-driven applications faster and using less code. In this recipe, we’ll build a form that displays the data of the owner of a bank account using data binding.

Getting ready

To follow along with this recipe, you can use the starter solution located in the Chapter03/SilverlightBanking_Displaying_Data_Starter folder in the code bundle available on the Packt website. The finished application for this recipe can be found in the Chapter03/SilverlightBanking_Displaying_Data_Completed folder.

How to do it...

Let’s assume that we are building a form, part of an online banking application, in which we can view the details of the owner of the account. Instead of wiring up the fields of the owner manually, we’ll use data binding. To get data binding up and running, carry out the following steps:

1. Open the starter solution, as outlined in the Getting Ready section.
2. The form we are building will bind to data. Data in data binding is in-memory data, not the data that lives in a database (it can originate from a database though). The data to which we are binding is an instance of the Owner class. The following is the code for the class. Add this code in a new class file called Owner in the Silverlight project.

   ```csharp
   public class Owner
   {
       public int OwnerId { get; set; }
       public string FirstName { get; set; }
       public string LastName { get; set; }
       public string Address { get; set; }
       public string ZipCode { get; set; }
       public string City { get; set; }
   }
   ```

For More Information:  
public string State { get; set; }
public string Country { get; set; }
public DateTime BirthDate { get; set; }
public DateTime CustomerSince { get; set; }
public string ImageName { get; set; }
public DateTime LastActivityDate { get; set; }
public double CurrentBalance { get; set; }
public double LastActivityAmount { get; set; }

3. Now that we've created the class, we are able to create an instance of it in the MainPage.xaml.cs file, the code-behind class of MainPage.xaml. In the constructor, we call the InitializeOwner method, which creates an instance of the Owner class and populates its properties.

private Owner owner;
public MainPage()
{
    InitializeComponent();
    //initialize owner data
    InitializeOwner();
}
private void InitializeOwner()
{
    owner = new Owner();
    owner.OwnerId = 1234567;
    owner.FirstName = "John";
    owner.LastName = "Smith";
    owner.Address = "Oxford Street 24";
    owner.ZipCode = "W1A";
    owner.City = "London";
    owner.Country = "United Kingdom";
    owner.State = "NA";
    owner.ImageName = "man.jpg";
    owner.LastActivityAmount = 100;
    owner.LastActivityDate = DateTime.Today;
    owner.CurrentBalance = 1234.56;
    owner.BirthDate = new DateTime(1953, 6, 9);
    owner.CustomerSince = new DateTime(1999, 12, 20);
}
4. Let's now focus on the form itself and build its UI. For this sample, we're not making the data editable. So for every field of the Owner class, we'll use a TextBlock. To arrange the controls on the screen, we'll use a Grid called OwnerDetailsGrid. This Grid can be placed inside the LayoutRoot Grid.

We will want the Text property of each TextBlock to be bound to a specific property of the Owner instance. This can be done by specifying this binding using the Binding "markup extension" on this property.

```xml
<Grid x:Name="OwnerDetailsGrid"
     VerticalAlignment="Stretch"
     HorizontalAlignment="Left"
     Background="LightGray"
     Margin="3 5 0 0"
     Width="300">
    <Grid.RowDefinitions>
        <RowDefinition Height="100"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="30"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>

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

    <Image x:Name="OwnerImage"
           Grid.Row="0"
           Width="100"
           Height="100"
           Stretch="Uniform"
           HorizontalAlignment="Left"
           Margin="3"
           Source="/CustomerImages/man.jpg"
           Grid.ColumnSpan="2">
    </Image>

    <TextBlock x:Name="OwnerIdTextBlock"
Grid.Row="1"
FontWeight="Bold"
Margin="2"
Text="Owner ID:">
</TextBlock>
<TextBlock x:Name="FirstNameTextBlock"
Grid.Row="2"
FontWeight="Bold"
Margin="2"
Text="First name:">
</TextBlock>
<TextBlock x:Name="LastNameTextBlock"
Grid.Row="3"
FontWeight="Bold"
Margin="2"
Text="Last name:">
</TextBlock>
<TextBlock x:Name="AddressTextBlock"
Grid.Row="4"
FontWeight="Bold"
Margin="2"
Text="Address:">
</TextBlock>
<TextBlock x:Name="ZipCodeTextBlock"
Grid.Row="5"
FontWeight="Bold"
Margin="2"
Text="Zip code:">
</TextBlock>
<TextBlock x:Name="CityTextBlock"
Grid.Row="6"
FontWeight="Bold"
Margin="2"
Text="City:">
</TextBlock>
<TextBlock x:Name="StateTextBlock"
Grid.Row="7"
FontWeight="Bold"
Margin="2"
Text="State:">
</TextBlock>
<TextBlock x:Name="CountryTextBlock"
Grid.Row="8"
FontWeight="Bold"
Margin="2"
Text="Country:">
</TextBlock>
An Introduction to Data Binding

<TextBlock x:Name="BirthDateTextBlock"
    Grid.Row="9"
    FontWeight="Bold"
    Margin="2"
    Text="Birthdate:">
</TextBlock>

<TextBlock x:Name="CustomerSinceTextBlock"
    Grid.Row="10"
    FontWeight="Bold"
    Margin="2"
    Text="Customer since:">
</TextBlock>

<TextBlock x:Name="OwnerIdValueTextBlock"
    Grid.Row="1"
    Grid.Column="1"
    Margin="2"
    Text="{Binding OwnerId}">
</TextBlock>

<TextBlock x:Name="FirstNameValueTextBlock"
    Grid.Row="2"
    Grid.Column="1"
    Margin="2"
    Text="{Binding FirstName}">
</TextBlock>

<TextBlock x:Name="LastNameValueTextBlock"
    Grid.Row="3"
    Grid.Column="1"
    Margin="2"
    Text="{Binding LastName}">
</TextBlock>

<TextBlock x:Name="AddressValueTextBlock"
    Grid.Row="4"
    Grid.Column="1"
    Margin="2"
    Text="{Binding Address}">
</TextBlock>

<TextBlock x:Name="ZipCodeValueTextBlock"
    Grid.Row="5"
    Grid.Column="1"
    Margin="2"
    Text="{Binding ZipCode}">
</TextBlock>

<TextBlock x:Name="CityValueTextBlock"
    Grid.Row="6"
    Grid.Column="1"
5. At this point, all the controls know what property they need to bind to. However, we haven't specified the actual link. The controls don't know about the Owner instance we want them to bind to. Therefore, we can use DataContext. We specify the DataContext of the OwnerDetailsGrid to be the Owner instance. Each control within that container can then access the object and bind to its properties. Setting the DataContext in done using the following code:

```csharp
public MainPage()
{
    InitializeComponent();
    //initialize owner data
    InitializeOwner();
    OwnerDetailsGrid.DataContext = owner;
}
```

The result can be seen in the following screenshot:

![Owner Details Grid](image)
How it works...

Before we take a look at the specifics of data binding, let's see what code we would need to write if Silverlight did not support data binding. The following is the ManualOwner class and we will be binding an instance of this class manually:

```csharp
public class ManualOwner
{
    public int OwnerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public string ZipCode { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }
    public DateTime BirthDate { get; set; }
    public DateTime CustomerSince { get; set; }
    public string ImageName { get; set; }
    public DateTime LastActivityDate { get; set; }
    public double CurrentBalance { get; set; }
    public double LastActivityAmount { get; set; }
}
```

The XAML code would look the same, apart from the binding markup extensions that are absent as we aren't using the data binding functionality. The following is a part of the code that has no data binding markup extensions:

```xml
<TextBlock x:Name="OwnerIdValueTextBlock"
    Grid.Row="1"
    Grid.Column="1"
    Margin="2" >
</TextBlock>
<TextBlock x:Name="FirstNameValueTextBlock"
    Grid.Row="2"
    Grid.Column="1"
    Margin="2" >
</TextBlock>
<TextBlock x:Name="LastNameValueTextBlock"
    Grid.Row="3"
    Grid.Column="1"
    Margin="2" >
</TextBlock>
<TextBlock x:Name="AddressValueTextBlock"
    Grid.Row="4"
    Grid.Column="1"
    Margin="2" >
</TextBlock>
```

For More Information:
An Introduction to Data Binding

Of course, the DataContext would also not be needed. Instead, we would manually have to link all the TextBlock controls with a property of the ManualOwner from code-behind as shown in the following code. As can be seen, this is not the most exciting code one can write!

```csharp
public MainPage()
{
    InitializeComponent();
    //initialize owner data
    InitializeOwner();
    SetOwnerValues();
}
private void SetOwnerValues()
{
    OwnerIdValueTextBlock.Text = owner.OwnerId.ToString();
    FirstNameValueTextBlock.Text = owner.FirstName;
    LastNameValueTextBlock.Text = owner.LastName;
    AddressValueTextBlock.Text = owner.Address;
    //other values go here
}
```

It's also easy to make errors this way. When a field gets added to the ManualOwner, we need to remember the places in which we have to update our code manually.

However, we can do better using data binding. Data binding enables us to write less code and have fewer opportunities to make errors.

Silverlight's data binding features allow us to bind the properties of the Owner instance to the Text property of the TextBlock controls using the Binding "markup extension". A markup extension can be recognized by a pair of curly braces ({}). It's basically a signal for the XAML parser that more needs to be done than simple attribute parsing. In this case, an instance of the System.Windows.Data.Binding is to be created for data binding to happen. The created Binding instance will bind the source object with the target control.

Looking back at the XAML code, we find that this binding is achieved for each TextBlock using the following code:

```xml
<TextBlock Text="{Binding CustomerSince}" />
```

This is, in fact, the shortened format. We could have written it as the following code:

```xml
<TextBlock Text="{Binding Path=CustomerSince}" />
```

The format for the binding is generally the following:

```xml
<TargetControl TargetProperty="{Binding SourceProperty, SomeBindingProperties}" />
```

For More Information:
Note that using SomeBindingProperties, more options can be specified when creating the binding. For example, we can specify that data should not only flow from source object to target control, but also vice versa. We'll explore a whole list of extra binding properties in the next recipes.

Are we missing something? Each control knows what it should bind to, but we haven't specified the actual source of the data. This is done using the DataContext. We set the Owner instance to be the DataContext of the Grid containing the controls. All controls within the Grid can access the data. We'll look at the DataContext in a later recipe.

Finally, there is one important point to note; we can't just bind everything. Basically, there are two rules we must follow:

1. The target object must be a DependencyObject (System.Windows.DependencyObject). In Silverlight 2 and Silverlight 3, the target could be a FrameworkElement instance only. FrameworkElement is lower in the class hierarchy than DependencyObject. Because of this, some important objects could not be used in data binding scenarios such as Transformations. Silverlight 4 has solved this problem.

2. The target property must be a dependency property. Again, don't panic, as almost all properties on UI controls (such as text, foreground and so on) are dependency properties.

Dependency properties were introduced with WPF and can be considered as properties on steroids. They include a mechanism that at any point in time determines what the value of the property should be, based on several influences working on the property such as data binding, styling, and so on. They can be considered as the enabler for animations, data binding, styling, and so on.


There's more...

Instead of creating the Owner instance in code, we can create it from XAML as well. First, we need to map the CLR namespace to an XML namespace as follows:

```xml
xmlns:local="clr-namespace:SilverlightBanking"
```

For More Information:

In the Resources collection of the container (the UserControl), we instantiate the type like this:

```xml
<UserControl.Resources>
  <local:Owner x:Key="localOwner"
    City="London"
    Country="United Kingdom"
    FirstName="John"
    LastName="Smith"
    OwnerId="1234567 ...">
  </local:Owner>
</UserControl.Resources>
```

The actual binding is almost the same, apart from specifying the source. We are not using the DataContext now, but we need to use the Source in each binding, referring to the item in the Resources:

```xml
<TextBlock x:Name="OwnerIdValueTextBlock"
  Grid.Row="1"
  Grid.Column="1"
  Margin="2"
  Text="{Binding OwnerId,
    Source={StaticResource localOwner}}" />
</TextBlock>
<TextBlock x:Name="FirstNameValueTextBlock"
  Grid.Row="2"
  Grid.Column="1"
  Margin="2"
  Text="{Binding FirstName,
    Source={StaticResource localOwner}}" />
</TextBlock>
<TextBlock x:Name="LastNameValueTextBlock"
  Grid.Row="3"
  Grid.Column="1"
  Margin="2"
  Text="{Binding LastName,
    Source={StaticResource localOwner}}" />
</TextBlock>
```

Whether binding from XAML is useful or not depends on the scenario. In most scenarios, we bind to objects that are created at runtime from code-behind. In this case, binding from XAML isn't possible.

**See also**

The DataContext makes its first appearance in this recipe, but we'll look at it in more detail in the *Obtaining data from any UI element it is bound to* recipe in this chapter.

For More Information:  
Creating dynamic bindings

In the previous recipe, you've learned how to use data binding in XAML. This is often useful because it allows you to show data easily to your user, for example, showing user information or a list of products. In this recipe, you'll learn how to do exactly the same in C# code, instead of XAML. This can be useful in situations where you want to bind a dependency property to the property of an object that you'll know only at runtime.

Getting ready

For this recipe, we can continue from the solution that was completed in the previous recipe. Alternatively, you can find the starter solution in the Chapter03/SilverlightBanking_Dynamic_Bindings_Starter folder in the code bundle that is available on the Packt website. Also, the completed solution can be found in the Chapter03/SilverlightBanking_Dynamic_Bindings_Completed folder.

How to do it...

We're going to change the code from the previous recipe, so we can create the bindings in C#, instead of XAML. To do this, we'll carry out the following steps:

1. Open the solution created in the previous recipe, Displaying data in Silverlight applications, locate the grid named OwnersDetailsGrid in MainPage.xaml, and remove the Binding syntax from the XAML code for each TextBlock as shown in the following code:

```xml
<TextBlock x:Name="OwnerIdValueTextBlock"
    Grid.Row="1"
    Grid.Column="1"
    Margin="2">
</TextBlock>
<TextBlock x:Name="FirstNameValueTextBlock"
    Grid.Row="2"
    Grid.Column="1"
    Margin="2">
</TextBlock>
<TextBlock x:Name="LastNameValueTextBlock"
    Grid.Row="3"
    Grid.Column="1"
    Margin="2">
</TextBlock>
<TextBlock x:Name="AddressValueTextBlock"
    Grid.Row="4"
    Grid.Column="1"
    Margin="2">
</TextBlock>
```

For More Information:

An Introduction to Data Binding

</TextBlock>
<TextBlock x:Name="ZipCodeValueTextBlock"
    Grid.Row="5"
    Grid.Column="1"
    Margin="2">
</TextBlock>
<TextBlock x:Name="CityValueTextBlock"
    Grid.Row="6"
    Grid.Column="1"
    Margin="2">
</TextBlock>
<TextBlock x:Name="StateValueTextBlock"
    Grid.Row="7"
    Grid.Column="1"
    Margin="2">
</TextBlock>
<TextBlock x:Name="CountryValueTextBlock"
    Grid.Row="8"
    Grid.Column="1"
    Margin="2">
</TextBlock>
<TextBlock x:Name="BirthDateValueTextBlock"
    Grid.Row="9"
    Grid.Column="1"
    Margin="2">
</TextBlock>
<TextBlock x:Name="CustomerSinceValueTextBlock"
    Grid.Row="10"
    Grid.Column="1"
    Margin="2">
</TextBlock>

2. Open the code-behind MainPage.xaml.cs file. Here, we're going to create the same bindings in the C# code. In the constructor, after the call to InitializeComponent(), add the following code:

OwnerIdValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("OwnerId"));
FirstNameValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("FirstName"));
LastNameValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("LastName"));
AddressValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("Address"));
ZipCodeValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("ZipCode"));
CityValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("City"));
StateValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("State"));
CountryValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("Country"));
BirthDateValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("BirthDate"));
CustomerSinceValueTextBlock.SetBinding(TextBlock.TextProperty,
    new Binding("CustomerSince"));

For More Information:
new Binding("City");
StateValueTextBlock.SetBinding(TextBlock.TextProperty,
   new Binding("State");
CountryValueTextBlock.SetBinding(TextBlock.TextProperty,
   new Binding("Country");
BirthDateValueTextBlock.SetBinding(TextBlock.TextProperty,
   new Binding("BirthDate");
CustomerSinceValueTextBlock.SetBinding(TextBlock.TextProperty,
   new Binding("CustomerSince");

3. We can now build and run the application, and you'll notice that the correct data is
   still displayed in the details form. The result can be seen in the following screenshot:

   ![User details form]

Owner ID: 1234567
First name: John
Last name: Smith
Address: Oxford Street 24
Zip code: W1A
City: London
State: NA
Country: United Kingdom
Birthdate: 6/6/1953 12:00:00 AM
Customer since: 12/20/1999 12:00:00 AM
Current balance: 1234.56
Last activity on: 1/3/2010 12:00:00 AM
Amount: 100

How it works...

This recipe shows you how to set the binding using C# syntax. Element.SetBinding
expects two parameters, a dependency property and a binding object. The first parameter
defines the DependencyProperty of the element you want to bind. The second parameter
defines the binding by passing a string that refers to the property path of the object to which
you are binding.

For More Information:
www.packtpub.com/managing-data-and-media-in-silverlight-4-packts-
bestselling-books/book
An Introduction to Data Binding

There's more...

In our example, we've used `new Binding("path")` as the syntax. The binding object, however, has different properties that you can set and which can be of interest. A few of these properties are `Converter`, `ConverterParameter`, `ElementName`, `Path`, `Mode`, and `ValidatesOnExceptions`.

To know when and how to use these properties, have a look at the other recipes in this chapter and the next which explain all the possibilities in detail. They are, however, already mentioned in this recipe to make it clear you can do everything that is required as far as bindings are concerned in both C# and XAML.

Binding data to another UI element

Sometimes, the value of the property of an element is directly dependent on the value of the property of another element. In this case, you can create a binding in XAML called an element binding or element-to-element binding. This binding links both values. If needed, the data can flow bidirectionally.

In the banking application, we can add a loan calculator that allows the user to select an amount and the number of years in which they intend to pay the loan back to the bank, including (of course) a lot of interest.

Getting ready

To follow this recipe, you can either continue with your solution from the previous recipe or use the provided solution that can be found in the Chapter03/SilverlightBanking_Element_Binding_Starter folder in the code bundle that is available on the Packt website. The finished application for this recipe can be found in the Chapter03/SilverlightBanking_Element_Binding_Completed folder.

How to do it...

To build the loan calculator, we'll use `Slider` controls. Each `Slider` is bound to a `TextBlock` using an element-to-element binding to display the actual value. Let's take a look at the steps we need to follow to create this binding:

1. We will build the loan calculator as a separate screen in the application. Add a new child window called `LoanCalculation.xaml`. To do so, right-click on the Silverlight project in the Solution Explorer, select Add | New Item..., and choose Silverlight Child Window under Visual C#.
2. Within MainPage.xaml, add a **Click** event on the LoanCalculationButton as shown in the following code:

```
<Button x:Name="LoanCalculationButton"
       Click="LoanCalculationButton_Click" />
```

3. In the code-behind's event handler for this **Click** event, we can trigger the display of this new screen with the following code:

```csharp
private void LoanCalculationButton_Click(object sender, RoutedEventArgs e)
{
    LoanCalculation loanCalculation = new LoanCalculation();
    loanCalculation.Show();
}
```

4. The UI of the LoanCalculation.xaml is quite simple—it contains two **Slider** controls. Each **Slider** control has set values for its Minimum and Maximum values (not all UI code is included here; the complete listing can be found in the finished sample code) as shown in the following code:

```
<Slider x:Name="AmountSlider"
       Minimum="10000"
       Maximum="1000000"
       SmallChange="10000"
       LargeChange="10000"
       Width="300" />

<Slider x:Name="YearSlider"
       Minimum="5"
       Maximum="30"
       SmallChange="1"
       LargeChange="1"
       Width="300"
       UseLayoutRounding="True">
</Slider>
```

5. As dragging a **Slider** does not give us proper knowledge of where we are exactly between the two values, we add two **TextBlock** controls. We want the **TextBlock** controls to show the current value of the **Slider** control, even while dragging. This can be done by specifying an element-to-element binding as shown in the following code:

```
<TextBlock x:Name="AmountTextBlock"
           Text="{Binding ElementName=AmountSlider, Path=Value}"
 />

<TextBlock x:Name="MonthTextBlock"
           Text="{Binding ElementName=YearSlider, Path=Value}"
 />
```

---

For More Information:

6. Add a Button that will perform the actual calculation called CalculateButton and a TextBlock called PaybackTextBlock to show the results. This can be done using the following code:

```xml
<Button x:Name="CalculateButton"
    Content="Calculate"
    Click="CalculateButton_Click">
</Button>
<TextBlock x:Name="PaybackTextBlock"></TextBlock>
```

7. The code for the actual calculation that is executed when the Calculate button is clicked uses the actual value for either the Slider or the TextBlock. This is shown in the following code:

```csharp
private double percentage = 0.0345;
private void CalculateButton_Click(object sender, RoutedEventArgs e)
{
    double requestedAmount = AmountSlider.Value;
    int requestedYears = (int)YearSlider.Value;
    for (int i = 0; i < requestedYears; i++)
    {
        requestedAmount += requestedAmount * percentage;
    }
    double monthlyPayback =
        requestedAmount / (requestedYears * 12);
    PaybackTextBlock.Text =
        "€" + Math.Round(monthlyPayback, 2);
}
```

Having carried out the previous steps, we now have successfully linked the value of the Slider controls and the text of the TextBlock controls. The following screenshot shows the LoanCalculation.xaml screen as it is included in the finished sample code containing some extra markup:

![LoanCalculation Screen](image)

For More Information:
How it works...

An element binding links two properties of two controls directly from XAML. It allows creating a Binding where the source object is another control. For this to work, we need to create a Binding and specify the source control using the ElementName property. This is shown in the following code:

```xml
<TextBlock Text="{Binding ElementName=YearSlider, Path=Value}"/>
</TextBlock>
```

Element bindings were added in Silverlight 3. Silverlight 2 did not support this type of binding.

There's more...

An element binding can also work in both directions, that is, from source to target and vice versa. This can be achieved by specifying the Mode property on the Binding and setting it to TwoWay.

The following is the code for this. In this code, we replaced the TextBlock by a TextBox. When entering a value in the latter, the Slider will adjust its position:

```xml
<TextBox x:Name="AmountTextBlock"
        Text="{Binding ElementName=AmountSlider, Path=Value, Mode=TwoWay}"/>
</TextBox>
```

Element bindings without bindings

Achieving the same effect in Silverlight 2—which does not support this feature—is also possible, but only through the use of an event handler as shown in the following code. Element bindings eliminate this need:

```csharp
private void AmountSlider_ValueChanged(object sender, RoutedPropertyChangedEventArgs<double> e)
{
    AmountSlider.Value = Math.Round(e.NewValue);
    AmountTextBlock.Text = AmountSlider.Value.ToString();
}
```

See also

Element-to-element bindings can be easily extended to use converters. For more information on TwoWay bindings, take a look at the Using the different modes of data binding to allow persisting data recipe in this chapter.

For More Information:
Binding collections to UI elements

Often, you'll want to display lists of data in your application such as a list of shopping items, a list of users, a list of bank accounts, and so on. Such a list typically contains a bunch of items of a certain type that have the same properties and need to be displayed in the same fashion.

We can use data binding to easily bind a collection to a Silverlight control (such as a ListBox or DataGrid) and use the same data binding possibilities to define how every item in the collection should be bound. This recipe will show you how to achieve this.

Getting ready

For this recipe, you can find the starter solution in the Chapter03/SilverlightBanking_Binding_Collections_Starter folder and the completed solution in the Chapter03/SilverlightBanking_Binding_Collections_Completed folder in the code bundle that is available on the Packt website.

How to do it...

In this recipe, we'll create a ListBox bound to a collection of activities. To complete this task, carry out the following steps:

1. We'll need a collection of some kind. We'll create a new type, that is, AccountActivity. Add the AccountActivity class to your Silverlight project as shown in the following code:

   ```csharp
   public class AccountActivity
   {
       public int ActivityId {get; set;}
       public double Amount { get; set; }
       public string Beneficiary { get; set; }
       public DateTime ActivityDate { get; set; }
       public string ActivityDescription { get; set; }
   }
   
   Add an ObservableCollection of AccountActivity to MainPage.xaml.cs using the following code:
   
   private ObservableCollection<AccountActivity> accountActivitiesCollection;
   ```

For More Information:

2. Now, we'll instantiate `accountActivitiesCollection` and fill it with data.
   To do this, add the following code to `MainPage.xaml.cs`:

   ```csharp
   private void InitializeActivitiesCollection()
   {
       accountActivitiesCollection = new ObservableCollection<AccountActivity>();
       AccountActivity accountActivity1 = new AccountActivity();
       accountActivity1.ActivityId = 1;
       accountActivity1.Amount = -33;
       accountActivity1.Beneficiary = "Smith Woodworking Shop London";
       accountActivity1.ActivityDescription = "Paid by credit card";
       accountActivity1.ActivityDate = new DateTime(2009, 9, 1);
       accountActivitiesCollection.Add(accountActivity1);
       AccountActivity accountActivity2 = new AccountActivity();
       accountActivity2.ActivityId = 2;
       accountActivity2.Amount = 1000;
       accountActivity2.Beneficiary = "ABC Infrastructure";
       accountActivity2.ActivityDescription = "Paycheck September 2009";
       accountActivity2.ActivityDate = new DateTime(2009, 9, 1);
       accountActivitiesCollection.Add(accountActivity2);
   }
   
   This creates a collection with two items. You can add more if you want to.

3. Add the following code to the `MainPage` constructor to call the method you created in the previous step:
   ```csharp
   InitializeActivitiesCollection();
   ```

4. We're going to need a control to display these `AccountActivity` items. To do this, add a `ListBox` called `AccountActivityListBox`. This `ListBox` defines a `DataTemplate` that defines how each `AccountActivity` is displayed.

   ```xml
   <ListBox x:Name="AccountActivityListBox"
           Width="600"
           Grid.Row="1">
       <ListBox.ItemTemplate>
           <DataTemplate>
               <Grid>
                   <Grid.RowDefinitions>
                       <RowDefinition></RowDefinition>
                       <RowDefinition></RowDefinition>
                   </Grid.RowDefinitions>
                   <Grid.ColumnDefinitions>
                       <ColumnDefinition Width="150">
                   </ColumnDefinition>
               </Grid>
           </DataTemplate>
       </ListBox.ItemTemplate>
   </ListBox>
   ```

For More Information:
An Introduction to Data Binding

5. In the MainPage constructor, set the ObservableCollection of AccountActivity you created in step 2 as the ItemsSource of the ListBox as shown in the following code:

```csharp
AccountActivityListBox.ItemsSource = accountActivitiesCollection;
```

6. If we build and run the application now, we'll see that a list of AccountActivity items is displayed as shown in the following screenshot:

For More Information:
How it works...

The first three steps aren’t important for people who have worked with collections before. A class is created to define the type of items that are held by the collection, which is initialized and then items are added to it. The default collection type to use in Silverlight is **ObservableCollection**. We’re using this collection type here. (For more information about this, have a look at the There’s more... section in this recipe.)

The real magic happens in steps 4 and 5. In step 4, we are creating a **ListBox**, which has an **ItemTemplate** property. This **ItemTemplate** property should contain a **DataTemplate**, and it’s this **DataTemplate** that defines how each item of the collection should be visualized. So, the **DataTemplate** corresponds to one item of your collection: one **AccountActivity**. This means we can use the data binding syntax that binds to properties of an **AccountActivity** in this **DataTemplate**.

When the **ItemsSource** property of the **ListBox** gets set to the **ObservableCollection** of **AccountActivity**, each **AccountActivity** in the collection is evaluated and visualized as defined in the **DataTemplate**.

There’s more...

An **ObservableCollection** is the default collection type you’ll want to use in a Silverlight application because it’s a collection type that implements the **INotifyCollectionChanged** interface. This makes sure that the UI can automatically be updated when the collection is changed (by adding or deleting an item). More on this can be found in the *Enabling a Silverlight application to automatically update its UI* recipe.

The same principle applies for the properties of classes that implement the **INotifyPropertyChanged** interface. More on this can be found in the same recipe, that is, *Enabling a Silverlight application to automatically update its UI*.

---

For More Information:

In this recipe, we're using a ListBox to visualize our ObservableCollection. However, every control that inherits the ItemsControl class (directly or indirectly) can be used in this way, such as a ComboBox, TreeView, DataGrid, WrapPanel, and so on. For more information on what operations can be performed using DataGrid, have a look at Chapter 5, The Data Grid.

See also

To learn how an ObservableCollection enables a UI to be automatically updated, have a look at the Enabling a Silverlight application to automatically update its UI recipe.

Enabling a Silverlight application to automatically update its UI

In the previous recipes, we looked at how we can display data more easily using data binding for both single objects as well as collections. However, there is another feature that data binding offers us for free, that is, automatic synchronization between the target and the source. This synchronization will make sure that when the value of the source property changes, this change will be reflected in the target object as well (being a control on the user interface). This also works in the opposite direction—when we change the value of a bound control, this change will be pushed to the data object as well. Silverlight's data binding engine allows us to opt-in to this synchronization process. We can specify if we want it to work—and if so, in which direction(s)—using the mode of data binding.

The synchronization works for both single objects bound to the UI as well as entire collections. But for it to work, an interface needs to be implemented in either case.

This synchronization process is what we'll be looking at in this recipe.

Getting ready

If you want to follow along with this recipe, you can either use the code from the previous recipes or use the provided solution in the Chapter03/SilverlightBanking_Update_UI_Starter folder in the code bundle that is available on the Packt website. The finished solution for this recipe can be found in the Chapter03/SilverlightBanking_Update_UI_Completed folder.
How to do it...

In this recipe, we'll look at how Silverlight does automatic synchronization, both for a single object and for a collection of objects. To demonstrate both types of synchronization, we'll use a timer that adds another activity on the account every 10 seconds. A single instance of the Owner class is bound to the UI. However, the newly added activities will cause the CurrentBalance, LastActivity, and LastActivityAmount properties of the Owner class to get updated. Also, these activities on the account will be reflected in the list of activities. The following are the steps to achieve automatic synchronization:

1. For the data binding engine to notice changes on the source object, the source needs to send a notification that the value of one of its properties has changed. By default, the Owner class does not do so. The original Owner class is shown by the following code:

   ```csharp
   public class Owner
   {
       public int OwnerId { get; set; }
       public string FirstName { get; set; }
       public string LastName { get; set; }
       public string Address { get; set; }
       public string ZipCode { get; set; }
       public string City { get; set; }
       public string State { get; set; }
       public string Country { get; set; }
       public DateTime BirthDate { get; set; }
       public DateTime CustomerSince { get; set; }
       public string ImageName { get; set; }
       public DateTime LastActivityDate { get; set; }
       public double CurrentBalance { get; set; }
       public double LastActivityAmount { get; set; }
   }
   ```

2. To make this class support notifications, an interface has to be implemented, namely the INotifyPropertyChanged interface. This interface defines one event, that is, the PropertyChanged event. Whenever one of the properties changes, this event should be raised. The changed Owner class is shown in the following code. (Only two properties are shown as they are all similar; the rest can be found in the finished solution in the book sample code.)

   ```csharp
   public class Owner : INotifyPropertyChanged
   {
       private double currentBalance;
       private string firstName;
       public event PropertyChangedEventHandler PropertyChanged;
       public string FirstName
   ```

For More Information:
An Introduction to Data Binding

```csharp
{
    get
    {
        return firstName;
    }
    set
    {
        firstName = value;
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs("FirstName"));
    }
}
public double CurrentBalance
{
    get
    {
        return currentBalance;
    }
    set
    {
        currentBalance = value;
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs("CurrentBalance"));
    }
}
}
```

3. To simulate updates, we'll use a DispatcherTimer in the MainPage. With every tick of this timer, a new activity on the account is created. We'll count the new value of the CurrentBalance with every tick and update the value of the LastActivityDate and LastActivityAmount as shown in the following code:

```csharp
private DispatcherTimer timer;
private int currentActivityId = 11;
public MainPage()
{
    InitializeComponent();
    //initialize owner data
    InitializeOwner();
    OwnerDetailsGrid.DataContext = owner;
    timer = new DispatcherTimer();
    timer.Interval = new TimeSpan(0, 0, 10);
    timer.Tick += new EventHandler(timer_Tick);
}
```
timer.Start();
}

void timer_Tick(object sender, EventArgs e)
{
    currentActivityId++;
    double amount = 0 - new Random().Next(100);
    AccountActivity newActivity = new AccountActivity();
    newActivity.ActivityId = currentActivityId;
    newActivity.Amount = amount;
    newActivity.Beneficiary = "Money withdrawal";
    newActivity.ActivityDescription = "ATM In Some Dark Alley";
    newActivity.ActivityDate = new DateTime(2009, 9, 18);
    owner.CurrentBalance += amount;
    owner.LastActivityDate = DateTime.Now;
    owner.LastActivityAmount = amount;
}

4. In XAML, the TextBlock controls are bound as mentioned before. If no Mode is specified, OneWay is assumed. This causes updates of the source to be reflected in the target as shown in the following code:

   <TextBlock x:Name="CountryValueTextBlock"
            Grid.Row="8"
            Grid.Column="1"
            Margin="2"
            Text="{Binding Country}" />
   <TextBlock x:Name="BirthDateValueTextBlock"
            Grid.Row="9"
            Grid.Column="1"
            Margin="2"
            Text="{Binding BirthDate}" />
   <TextBlock x:Name="CustomerSinceValueTextBlock"
            Grid.Row="10"
            Grid.Column="1"
            Margin="2"
            Text="{Binding CustomerSince}" />

5. If we run the application now, after 10 seconds, we'll see the values changing. The values can be seen in the following screenshot:
6. In the Binding collections to UI elements recipe, we saw how to bind a list of AccountActivity items to a ListBox. If we want the UI to update automatically when changes occur in the list (when a new item is added or an existing item is removed), then the list to which we bind should implement the INotifyCollectionChanged interface. Silverlight has a built-in list that implements this interface, namely the ObservableCollection<T>. If we were binding to a List<T>, then these automatic updates wouldn't work. Working with an ObservableCollection<T> is no different than working with a List<T>. In the following code, we're creating the ObservableCollection<AccountActivity> and adding items to it:

```csharp
private ObservableCollection<AccountActivity> accountActivitiesCollection;
private void InitializeActivitiesCollection()
{
    accountActivitiesCollection = new ObservableCollection<AccountActivity>();
    AccountActivity accountActivity1 = new AccountActivity();
    accountActivity1.ActivityId = 1;
    accountActivity1.Amount = -33;
    accountActivity1.Beneficiary = "Smith Woodworking Shop London";
    accountActivity1.ActivityDescription = "Paid by credit card";
    accountActivity1.ActivityDate = new DateTime(2009, 9, 1);
    accountActivitiesCollection.Add(accountActivity1);
}
```

7. Update the Tick event, so that each new Activity is added to the collection:

```csharp
void timer_Tick(object sender, EventArgs e)
{
    ...
    AccountActivity newActivity = new AccountActivity();
    ...
    accountActivitiesCollection.Add(newActivity);
    ...
}
```

8. To bind this collection to the ListBox, we use the ItemsSource property. The following code can be added to the constructor to create the collection and perform the binding:

```csharp
InitializeActivitiesCollection();
AccountActivityListBox.ItemsSource = accountActivitiesCollection;
```

When we run the application now, we see that all added activities appear in the ListBox control. With every tick of the Timer, a new activity is added and the UI refreshes automatically.
How it works...

In some scenarios, we might want to view changes to the source object in the user interface immediately. Silverlight's data binding engine can automatically synchronize the source and target for us, both for single objects and for collections.

**Single objects**

If we want the target controls on the UI to update automatically if a property value of an instance changes, then the class to which we are binding should implement the INotifyPropertyChanged interface. This interface defines just one event—PropertyChanged. It is defined in the System.ComponentModel namespace using the following code:

```csharp
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

This event should be raised whenever the value of a property changes. The name of the property that has changed is passed as the parameter for the instance of PropertyChangedEventArgs.

A binding in XAML is set to OneWay by default. OneWay allows updates to be passed on to the target. (For more information on binding modes, refer to the Using the different modes of data binding to allow persisting data recipe.) If we had set the binding to Mode=OneTime, then only the initial values would have been loaded.

Now, what exactly happens when we bind to a class that implements this interface? Whenever we do so, Silverlight's data binding engine will notice this and will automatically start to check if the PropertyChanged event is raised by an instance of the class. It will react to this event, thereby resulting in an update of the target.

**Collections**

Whenever a collection changes, we might want to get updates of this collection as well. In this example, we want to view the direct information of all the activities on the account. Normally, we would have placed these in a List<T>. However, List<T> does not raise an event when items are being added or deleted. Similar to INotifyPropertyChanged, an interface exists so that a list/collection should implement for data binding to pick up those changes. This interface is known as INotifyCollectionChanged.

We didn't directly create a class that implements this interface. However, we used an ObservableCollection<T>. This collection already implemented this interface for us.

Whenever items are being added, deleted, or the collection gets refreshed, an event will be raised on which the data binding engine will bind itself. As for single objects, changes will be reflected in the UI immediately.

For More Information:
Cleaning up the code

In the code for the Owner class, we have inputted all the properties as shown in the following code:

```csharp
public double CurrentBalance
{
    get
    {
        return currentBalance;
    }
    set
    {
        currentBalance = value;
        if (currentBalance != null)
            PropertyChanged(this, new PropertyChangedEventArgs("CurrentBalance"));
    }
}
```

It's a good idea to move the check whether the event is null (which means that there is no one actually subscribed to the event) and the raising of the event to a separate method as shown in the following code:

```csharp
public void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

public double CurrentBalance
{
    get
    {
        return currentBalance;
    }
    set
    {
        if (currentBalance != value)
        {
            currentBalance = value;
            OnPropertyChanged("CurrentBalance");
        }
    }
}
```
It may also be a good idea to move this method to a base class and have the entities inherit from this class as shown in the following code:

```csharp
public class BaseEntity : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
    public class Owner : BaseEntity
    {
    }
}
```

While automatic synchronization is a nice feature that comes along with data binding for free, it's not always needed. Sometimes it's not even wanted. Therefore, implement the interfaces that are described here only when the application needs them. It's an opt-in model.

## Obtaining data from any UI element it is bound to

When a user who is working with your application performs a certain action, it's often essential to know on what object this action will be executed. For example, if a user clicks on a **Delete** button on an item, it's essential that you know which item is clicked so that you can write the correct code to delete that item. Also, when a user wants to edit an item in a list, it's necessary that you—the programmer—know which item in the list the user wants to edit.

In Silverlight, there is a very easy mechanism called **DataContext** that helps us in this task. In this recipe, we're going to use the **DataContext** to get the data when we need it.

### Getting ready

If you want to follow along with this recipe, you can either use the code from the previous recipes or use the provided solution in the `Chapter03/SilverlightBanking_Obtaining_Data_Starter` folder in the code bundle that is available on the Packt website. The completed solution for this recipe can be found in the `Chapter03/SilverlightBanking_Obtaining_Data_Completed` folder.
How to do it...

We're going to create a Details... button for each item in the ListBox containing AccountActivities. This Details... button will open a new ChildWindow that will display details about the selected AccountActivity. To achieve this, carry out the following steps:

1. We'll start by opening the solution we've created by following all the steps of the Binding data to collections recipe. We add a new item to the Silverlight project—a ChildWindow named ActivityDetailView—and add the following code to the XAML defining this new control:

```xml
<Grid x:Name="LayoutRoot" Margin="2">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid x:Name="OwnerDetailsGrid">
        <Grid.RowDefinitions>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <TextBlock x:Name="ActivityIdTextBlock" Grid.Row="0" FontWeight="Bold" Margin="2" Text="Activity ID:">
        </TextBlock>
        <TextBlock x:Name="BeneficiaryTextBlock" Grid.Row="1" FontWeight="Bold" Margin="2" Text="Beneficiary:">
        </TextBlock>
    </Grid>
</Grid>
```

For More Information:
For More Information:
An Introduction to Data Binding

2. Next, we open ActivityDetailView.xaml.cs and add the following code:

```csharp
public ActivityDetailView(AccountActivity activity)
{
    InitializeComponent();
    this.DataContext = activity;
}

private void btnOK_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = true;
}
```

3. Now, we open MainPage.xaml, locate the ListBox named AccountActivityListBox, and add a button named btnDetails to the DataTemplate of that ListBox. This is shown in the following code:

```xml
<Button x:Name="btnDetails"
    Grid.Row="1"
    Grid.Column="2"
    HorizontalAlignment="Right"
    Content="Details..." 
    Click="btnDetails_Click">
</Button>
```

4. Add the following C# code to MainPage.xaml.cs to handle the Click event of the button we've added in the previous step:

```csharp
private void btnDetails_Click(object sender, RoutedEventArgs e)
{
    ActivityDetailView activityDetailView = new ActivityDetailView((AccountActivity)((Button)sender).DataContext);
    activityDetailView.Show();
}
```

For More Information:
5. We can now build and run the solution. When you click on the Details... button, you'll see the details of the selected AccountActivity in a ChildWindow. You can see the result in the following screenshot:

![Screenshot of AccountActivity details](image)

**How it works...**

Once the DataContext of a general control has been set (any CLR object can be used as DataContext), each child item of that control refers to the same DataContext.

For example, if we have a UserControl containing a Grid that has three columns, with aTextBox in the first two and a Button in the last column, and if the DataContext of the UserControl gets set to an object of the Person type, then the Grid, TextBox, and Button would have that same Person object as their DataContext. To be more precise, if the DataContext of an item hasn't been set, then Silverlight will find out if the parent of that item in the visual tree has its DataContext set to an object and use that DataContext as the DataContext of the child item. Silverlight keeps on trickling right up to the uppermost level of the application.

If you use an ItemsControl such as a ListBox and give it a collection as an ItemsSource, then the DataContext of that ListBox is the collection you bound it to.

---

For More Information:

An Introduction to Data Binding

Following the same logic, the DataContext of one ListBoxItem is one item from the collection. In our example, one item is defined by a DataTemplate containing a Grid, various TextBlocks, and a Button. Due to the fact that Silverlight keeps on trickling up to look for a valid DataContext, the DataContext of the Grid, all the TextBlocks, and the Button are the same; they’re one item from the ItemsSource collection of the ListBox.

With this in mind, we can now access the data that is bound to any UI element of our ListBoxItem. The data we need is the DataContext of the button we’re clicking.

The click event of this button has a sender parameter—the Button itself. To access the DataContext, we cast the sender parameter to a Button object. As we know that the ListBox is bound to an ObservableCollection of AccountActivity, we can cast the DataContext to type AccountActivity. To show the details window, all we need to do now is pass this object to the constructor of the details ChildWindow.

See also

The DataContext is important when you’re working with data binding as it’s the DataContext of an element that’s looked at as the source of the binding properties. You can learn more about data binding and the various possibilities it offers by looking at almost any recipe in this chapter.

Using the different modes of data binding to allow persisting data

Until now, the data has flowed from the source to the target (the UI controls). However, it can also flow in the opposite direction, that is, from the target towards the source. This way, not only can data binding help us in displaying data, but also in persisting data.

The direction of the flow of data in a data binding scenario is controlled by the Mode property of the Binding. In this recipe, we’ll look at an example that uses all the Mode options and in one go, we’ll push the data that we enter ourselves to the source.

Getting ready

This recipe builds on the code that was created in the previous recipes, so if you’re following along, you can keep using that codebase. You can also follow this recipe from the provided start solution. It can be found in the Chapter03/SilverlightBanking_Binding_Modes_Starter folder in the code bundle that is available on the Packt website. The Chapter03/SilverlightBanking_Binding_Modes_Completed folder contains the finished application of this recipe.

For More Information:

How to do it...

In this recipe, we'll build the "edit details" window of the Owner class. On this window, part of the data is editable, while some isn't. The editable data will be bound using a TwoWay binding, whereas the non-editable data is bound using a OneTime binding. The Current balance of the account is also shown—which uses the automatic synchronization—based on the INotifyPropertyChanged interface implementation. This is achieved using OneWay binding. The following is a screenshot of the details screen:

Let's go through the required steps to work with the different binding modes:

1. Add a new Silverlight child window called OwnerDetailsEdit.xaml to the Silverlight project.
2. In the code-behind of this window, change the default constructor—so that it accepts an instance of the Owner class—as shown in the following code:

```csharp
private Owner owner;
public OwnerDetailsEdit(Owner owner)
{
    InitializeComponent();
    this.owner = owner;
}
```

For More Information:
An Introduction to Data Binding

3. In MainPage.xaml, add a Click event on the OwnerDetailsEditButton:
   <Button x:Name="OwnerDetailsEditButton"
           Click="OwnerDetailsEditButton_Click" />

4. In the event handler, add the following code, which will create a new instance of the OwnerDetailsEdit window, passing in the created Owner instance:
   private void OwnerDetailsEditButton_Click(object sender, RoutedEventArgs e)
   {
       OwnerDetailsEdit ownerDetailsEdit = new OwnerDetailsEdit(owner);
       ownerDetailsEdit.Show();
   }

5. The XAML of the OwnerDetailsEdit is pretty simple. Take a look at the completed solution (Chapter03/SilverlightBanking_Binding_Modes_Completed) for a complete listing. Don't forget to set the passed Owner instance as the DataContext for the OwnerDetailsGrid. This is shown in the following code:
   OwnerDetailsGrid.DataContext = owner;

6. For the OneWay and TwoWay bindings to work, the object to which we are binding should be an instance of a class that implements the INotifyPropertyChanged interface. In our case, we are binding an Owner instance. This instance implements the interface correctly. The following code illustrates this:
   public class Owner : INotifyPropertyChanged
   {
       public event PropertyChangedEventHandler PropertyChanged;
       ...
   }

7. Some of the data may not be updated on this screen and it will never change. For this type of binding, the Mode can be set to OneTime. This is the case for the OwnerId field. The users should neither be able to change their ID nor should the value of this field change in the background, thereby requiring an update in the UI. The following is the XAML code for this binding:
   <TextBlock x:Name="OwnerIdValueTextBlock"
             Text="{Binding OwnerId, Mode=OneTime}" />

8. The CurrentBalance TextBlock at the bottom does not need to be editable by the user (allowing a user to change his or her account balance might not be beneficial for the bank), but it does need to change when the source changes. This is the automatic synchronization working for us and it is achieved by setting the Binding to Mode=OneWay. This is shown in the following code:
   <TextBlock x:Name="CurrentBalanceValueTextBlock"
             Text="{Binding CurrentBalance, Mode=OneWay}" />

For More Information:
9. The final option for the Mode property is TwoWay. TwoWay bindings allow us to persist data by pushing data from the UI control to the source object. In this case, all other fields can be updated by the user. When we enter a new value, the bound Owner instance is changed. TwoWay bindings are illustrated using the following code:

```xml
<TextBox x:Name="FirstNameValueTextBlock"
    Text="{Binding FirstName, Mode=TwoWay}"/>
</TextBox>
```

We've applied all the different binding modes at this point. Notice that when you change the values in the pop-up window, the details on the left of the screen are also updated. This is because all controls are in the background bound to the same source object as shown in the following screenshot:

![Screen shot showing binding modes](image)

**How it works...**

When we looked at the basics of data binding, we saw that a binding always occurs between a source and a target. The first one is normally an in-memory object, but it can also be a UI control. The second one will always be a UI control.

Normally, data flows from source to target. However, using the Mode property, we have the option to control this.

A OneTime binding should be the default for data that does not change when displayed to the user. When using this mode, the data flows from source to target. The target receives the value initially during loading and the data displayed in the target will never change. Quite logically, even if a OneTime binding is used for a TextBox, changes done to the data by the user will not flow back to the source. IDs are a good example of using OneTime bindings. Also, when building a catalogue application, OneTime bindings can be used, as we won't change the price of the items that are displayed to the user (or should we...?).

For More Information:

An Introduction to Data Binding

We should use a OneWay binding for binding scenarios in which we want an up-to-date display of data. Data will flow from source to target here also, but every change in the values of the source properties will propagate to a change of the displayed values. Think of a stock market application where updates are happening every second. We need to push the updates to the UI of the application.

The TwoWay bindings can help in persisting data. The data can now flow from source to target, and vice versa. Initially, the values of the source properties will be loaded in the properties of the controls. When we interact with these values (type in a textbox, drag a slider, and so on), these updates are pushed back to the source object. If needed, conversions can be done in both directions.

There is one important requirement for the OneWay and TwoWay bindings. If we want to display up-to-date values, then the INotifyPropertyChanged interface should be implemented. The OneTime and OneWay bindings would have the same effect, even if this interface is not implemented on the source. The TwoWay bindings would still send the updated values if the interface was not implemented; however, they wouldn't notify about the changed values. It can be considered as a good practice to implement the interface, unless there is no chance that the updates of the data would be displayed somewhere in the application. The overhead created by the implementation is minimal.

There's more...

Another option in the binding is the UpdateSourceTrigger. It allows us to specify when a TwoWay binding will push the data to the source. By default, this is determined by the control. For a TextBox, this is done on the LostFocus event; and for most other controls, it's done on the PropertyChanged event.

The value can also be set to Explicit. This means that we can manually trigger the update of the source.

```csharp
BindingExpression expression = this.FirstNameValueTextBlock.GetBindingExpression(TextBox.TextProperty);
expression.UpdateSource();
```

See also

Changing the values that flow between source and target can be done using converters.

For More Information:

**Data binding from Expression Blend 4**

While creating data bindings is probably a task mainly reserved for the developer(s) in the team, Blend 4—the design tool for Silverlight applications—also has strong support for creating and using bindings.

In this recipe, we'll build a small data-driven application that uses data binding. We won't manually create the data binding expressions; we'll use Blend 4 for this task.

**How to do it...**

For this recipe, we'll create a small application from scratch that allows us to edit the details of a bank account owner. In order to achieve this, carry out the following steps:

1. We'll need to open Blend 4 and go to **File | New Project...** In the **New Project** dialog box, select **Silverlight 4 Application + Website**. Name the project **SilverlightOwnerEdit** and click on the **OK** button. Blend will now create a Silverlight application and a hosting website.

2. We'll start by adding a new class called **Owner**. Right-click on the Silverlight project and select **Add New Item...**. In the dialog box that appears, select the **Class** template and click on the **OK** button. The following is the code for the **Owner** class and it can be edited inside Blend 4:

   ```csharp
   public class Owner
   {
       public string Name {get; set;}
       public int CurrentBalance {get;set;}
       public DateTime LastActivityDate {get;set;}
   }
   ```

3. In the code-behind of **MainPage.xaml**, create an instance of the **Owner** class and set it as the **DataContext** for the **LayoutRoot** of the page.

   ```csharp
   public partial class MainPage : UserControl
   {
       public Owner owner;
       public MainPage()
       {
           // Required to initialize variables
           InitializeComponent();
           owner = new Owner()
           {
               Name="Gill Cleeren",
               CurrentBalance=300,
           }
   }
   ```

For More Information:

An Introduction to Data Binding

```csharp
    LastActivityDate = DateTime.Now.Date

    LayoutRoot.DataContext = owner;
```
7. In the dialog box that appears, we can now couple the Name property of the Owner type to the Text property of the TextBox. Under the Explicit Data Context tab, mark the Use a custom path expression checkbox and enter Name as the value. Click on the down arrow so that the advanced properties are expanded and mark TwoWay as the Binding direction. The other properties are similar as shown in the following screenshot:

![Create Data Binding dialog box](image)

**How it works...**

Let’s look at the resulting XAML code for a moment. Blend created the bindings for us automatically taking into account the required options such as Mode=TwoWay. This is shown in the following code:

```xml
<TextBox Grid.Column="1"
     Text="{Binding Name, Mode=TwoWay,
     UpdateSourceTrigger=Default}"
     TextWrapping="Wrap"/>

<TextBox Grid.Column="1"
     Grid.Row="2"
     Text="{Binding LastActivityDate, Mode=TwoWay,
     UpdateSourceTrigger=Default}"
     TextWrapping="Wrap"/>

<TextBox Grid.Column="1"
```

For More Information:  
An Introduction to Data Binding

Grid.Row="1"
Text="{Binding CurrentBalance, Mode=TwoWay,
       UpdateSourceTrigger=Default}"
TextWrapping="Wrap"/>

When we have to create many bindings, it's often easier to do so through these dialog boxes than typing them manually in Visual Studio.

**Using Expression Blend 4 for sample data generation**

Expression Blend 4 contains a feature that is capable of generating the sample data while developing an application. It visualizes the data on which we are working and provides us with an easier way to create an interface for a data-driven application. This feature was added to Blend in version 3.

**How to do it...**

In this recipe, we'll build a small management screen for the usage of the bank employees. It will show an overview of the bank account owners. We wouldn't want to waste time with the creation of (sample) data, so we'll hand over this task to Blend. The following are the steps we need to follow for the creation of this data:

1. Open Blend 4 and go to **File | New Project...** In the dialog box that appears, select **Silverlight 4 Application + Website**. Name the project as **SilverlightBankingManagement** and click on the **OK** button. Blend will now create a Silverlight application and a hosting website.

2. With **MainPage.xaml** open in either the **Design View** or the **Split View**, go to the **Data** window. In this window, click on the **Add sample data source** icon and select **Define New Sample Data...** as shown in the following screenshot:

![Define New Sample Data](image)

3. In the **Define New Sample Data** dialog box that appears, specify the **Data source name** as **OwnerDataSource**. We have the option to either embed this data source in the usercontrol (**This document**) or make it available for the entire project (**Project**). Select the latter option by selecting the **Project** radio button and clicking on the **OK** button.

For More Information:

The last option in this window—**Enable sample data when application is running**—allows us to switch off the sample data while running the compiled application. If we leave the checkbox checked, then the sample data will be used for the design time as well as the runtime. We'll keep this option enabled.

Blend will now generate the data source for us. The result is shown in the following screenshot:

4. By default, a **Collection** is created and it contains items with two properties. Each property has a type. Start by adding two more properties by clicking on the + sign next to the **Collection** and select the **Add simple property** option.

Rename **Property1** to **Name**. Now, change the type options by clicking on the **Change property type** icon and selecting **Name** as the format. The other properties are similar and are shown in the following screenshot:

5. For the **Image** type, we can select a folder that contains images. Blend will then copy these images to the **SampleData** subfolder inside the project.

6. We're now ready to use the sample data—for example—in a master-detail scenario. A **ListBox** will contain all the **Owner** data from which we can select an instance. The details are shown in a **Grid** using some **TextBlock** controls. Make sure that the **Data** window is set to **List Mode** and drag the collection on to the design surface. This will trigger the creation of a listbox in which the items are formatted, so we can see the details.

---

**For More Information:**

7. Now, to view the details, we have to set the **Data** window to the **Details Mode**. Then, instead of dragging the collection, we select the properties that we want to see in the detail view and drag those onto the design surface. The result should be similar to the following screenshot:

Thus, Blend created all the data binding code in XAML as well as the sample data. For each different type, it generated different values.

**For More Information:**

Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.