PART V

Data

CHAPTER 19

Data Binding

Data binding is the time-honored tradition of pulling information out of an object and displaying it in your application's user interface, without writing the tedious code that does all the work. Often, rich clients use *two-way* data binding, which adds the ability to push information from the user interface back into some object—again, with little or no code. Because many Windows applications are all about data (and all of them need to deal with data some of the time), data binding is a top concern in a user interface technology like WPF.

Developers who are approaching WPF from a Windows Forms background will find that WPF data binding has many similarities. As in Windows Forms, WPF data binding allows you to create bindings that take information from just about any property of any object and stuff it into just about any property of any element. WPF also includes a set of list controls that can handle entire collections of information and allow you to navigate through them. However, there are significant changes in the way that data binding is implemented behind the scenes, some impressive new functionality, and a fair bit of tweaking and fine-tuning. Many of the same concepts apply, but the same code won't.

In this chapter, you'll learn how to use WPF data binding. You'll create declarative bindings that extract the information you need and display it in various types of elements. You'll also learn how to plug this system into a back-end database.

What's New Although the data binding basics remain the same, WPF 4.5 adds a number of small refinements. In this chapter, you'll learn about the improved VirtualizationPanel, which allows more fine-tuning (see the section "Improving Performance in Long Lists"). You'll also consider the INotifyDataErrorInfo interface for validation (see the section "Validation"), which was brought over from Silverlight and replaces the IDataErrorInfo interface.

Binding to a Database with Custom Objects

When developers hear the term *data binding*, they often think of one specific application—pulling information out of a database and showing it onscreen with little or no code.

As you saw in Chapter 8, data binding in WPF is a much more general tool. Even if your application never comes into contact with a database, it's still likely to use data binding to automate the way elements interact or translate an object model into a suitable display.

However, you can learn a lot about the details of object binding by considering a traditional example that queries and updates a table in a database. In this chapter, you'll use an example that retrieves a catalog of products. The first step in building this example is to create a custom data access component.

■ **Note** The downloadable code for this chapter includes the custom data access component and a database script that installs the sample data, so you can test all the examples. But if you don't have a test database server or you don't want to go to the trouble of creating a new database, you can use an alternate version of the data access component that's also included with the code. This version simply loads the data from a file, while still exposing the same set of classes and methods. It's perfect for testing but obviously impractical for a real application.

Building a Data Access Component

In professional applications, database code is not embedded in the code-behind class for a window but encapsulated in a dedicated class. For even better componentization, these data access classes can be pulled out of your application altogether and compiled in a separate DLL component. This is particularly true when writing code that accesses a database (because this code tends to be extremely performance-sensitive), but it's a good design no matter where your data lives.

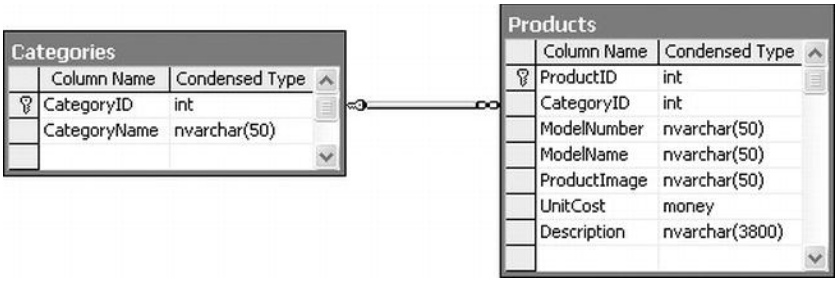
DESIGNING DATA ACCESS COMPONENTS

No matter how you plan to use data binding (or even if you don't), your data access code should always be coded in a separate class. This approach is the only way you have the slightest chance to make sure you can efficiently maintain, optimize, troubleshoot, and (optionally) reuse your data access code.

When creating a data class, you should follow a few basic guidelines in this section:

- *Open and close connections quickly:* Open the database connection in every method call, and close it before the method ends. This way, a connection can't be inadvertently left open. One way to ensure the connection is closed at the appropriate time is with a using block.
- *Implement error handling:* Use error handling to make sure that connections are closed even if an exception occurs.
- *Follow stateless design practices:* Accept all the information needed for a method in its parameters, and return all the retrieved data through the return value. This avoids complications in a number of scenarios (for example, if you need to create a multithreaded application or host your database component on a server).
- *Store the connection string in one place:* Ideally, this is the configuration file for your application.

The database component that's shown in the following example retrieves a table of product information from the Store database, which is a sample database for the fictional IBuySpy store included with some Microsoft case studies. Figure 19-1 shows two tables in the Store database and their schemas.



The data access class is exceedingly simple—it provides just a single method that allows the caller to retrieve one product record. Here's the basic outline:

```
public class StoreDB
{
    // Get the connection string from the current configuration file.
    private string connectionString = Properties.Settings.Default.StoreDatabase;

    public Product GetProduct(int ID)
    {
        ...
    }
}
```

The query is performed through a stored procedure in the database named GetProduct. The connection string isn't hard-coded—instead, it's retrieved through an application setting in the .config file for this application. (To view or set application settings, double-click the Properties node in the Solution Explorer, and then click the Settings tab.)

When other windows need data, they call the StoreDB.GetProduct() method to retrieve a Product object. The Product object is a custom object that has a sole purpose in life—to represent the information for a single row in the Products table. You'll consider it in the next section.

You have several options for making the StoreDB class available to the windows in your application:

- The window could create an instance of StoreDB whenever it needs to access the database.
- You could change the methods in the StoreDB class to be static.
- You could create a single instance of StoreDB and make it available through a static property in another class (following the "factory" pattern).

The first two options are reasonable, but both of them limit your flexibility. The first choice prevents you from caching data objects for use in multiple windows. Even if you don't want to use that caching right away, it's worth designing your application in such a way that it's easy to implement later. Similarly, the second approach assumes you won't have any instance-specific state that you need to retain in the StoreDB class. Although this is a good design principle, you might want to retain some details (such as the connection string) in memory. If you convert the StoreDB class to use static methods, it becomes much more difficult to access different instances of the Store database in different back-end data stores.

Ultimately, the third option is the most flexible. It preserves the switchboard design by forcing all the windows to work through a single property. Here's an example that makes an instance of StoreDB available through the Application class:

```
public partial class App : System.Windows.Application
{
    private static StoreDB storeDB = new StoreDB();
    public static StoreDB StoreDB
    {
        get { return storeDB; }
    }
}
```

In this book, we're primarily interested with how data objects can be bound to WPF elements. The actual process that deals with creating and filling these data objects (as well as other implementation details, such as whether StoreDB caches the data over several method calls, whether it uses stored procedures instead of inline queries, whether it fetches the data from a local XML file when offline, and so on) isn't our focus. However, just to get an understanding of what's taking place, here's the complete code:

```
public class StoreDB
{
    private string connectionString = Properties.Settings.Default.StoreDatabase;

    public Product GetProduct(int ID)
    {
        SqlConnection con = new SqlConnection(connectionString);
        SqlCommand cmd = new SqlCommand("GetProductByID", con);
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.AddWithValue("@ProductID", ID);

        try
        {
            con.Open();
            SqlDataReader reader = cmd.ExecuteReader(CommandBehavior.SingleRow);
            if (reader.Read())
            {
                // Create a Product object that wraps the
                // current record.
                Product product = new Product((string)reader["ModelNumber"],
                    (string)reader["ModelName"], (decimal)reader["UnitCost"],
                    (string)reader["Description"] ,
                    (string)reader["ProductImage"]);
                return(product);
            }
            else
            {
                return null;
            }
        }
        finally
        {
            con.Close();
        }
    }
}
```

■ **Note** Currently, the GetProduct() method doesn't include any exception handling code, so all exceptions will bubble up the calling code. This is a reasonable design choice, but you might want to catch the exception in GetProduct(), perform cleanup or logging as required, and then rethrow the exception to notify the calling code of the problem. This design pattern is called *caller inform*.

Building a Data Object

The data object is the information package that you plan to display in your user interface. Any class works, provided it consists of public properties (fields and private properties aren't supported). In addition, if you want to use this object to make changes (via two-way binding), the properties cannot be read-only.

Here's the Product object that's used by StoreDB:

```
public class Product
{
    private string modelNumber;
    public string ModelNumber
    {
        get { return modelNumber; }
        set { modelNumber = value; }
    }

    private string modelName;
    public string ModelName
    {
        get { return modelName; }
        set { modelName = value; }
    }

    private decimal unitCost;
    public decimal UnitCost
    {
        get { return unitCost; }
        set { unitCost = value; }
    }

    private string description;
    public string Description
    {
        get { return description; }
        set { description = value; }
    }

    public Product(string modelNumber, string modelName,
      decimal unitCost, string description)
    {
        ModelNumber = modelNumber;
        ModelName = modelName;
```

```
        UnitCost = unitCost;
        Description = description;
    }
}
```

Displaying the Bound Object

The final step is to create an instance of the Product object and then bind it to your controls. Although you could create a Product object and store it as a resource or a static property, neither approach makes much sense. Instead, you need to use StoreDB to create the appropriate object at runtime and then bind that to your window.

■ **Note** Although the declarative no-code approach sounds more elegant, there are plenty of good reasons to mix a little code into your data-bound windows. For example, if you're querying a database, you probably want to handle the connection in your code so that you can decide how to handle exceptions and inform the user of problems.

Consider the simple window shown in Figure 19-2. It allows the user to supply a product code, and it then shows the corresponding product in the Grid in the lower portion of the window.

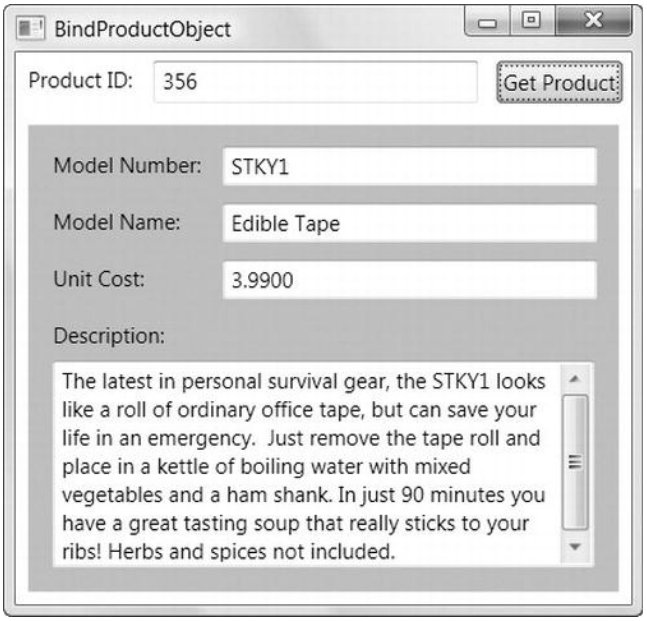


Figure 19-2. *Querying a product*

When you design this window, you don't have access to the Product object that will supply the data at runtime. However, you can still create your bindings without indicating the data source. You simply need to indicate the property that each element uses from the Product class.

Here's the full markup for displaying a Product object:

```
<Grid Name="gridProductDetails">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="*"></RowDefinition>
  </Grid.RowDefinitions>

  <TextBlock Margin="7">Model Number:</TextBlock>
  <TextBox Margin="5" Grid.Column="1"
   Text="{Binding Path=ModelNumber}"></TextBox>
  <TextBlock Margin="7" Grid.Row="1">Model Name:</TextBlock>
  <TextBox Margin="5" Grid.Row="1" Grid.Column="1"
   Text="{Binding Path=ModelName}"></TextBox>
  <TextBlock Margin="7" Grid.Row="2">Unit Cost:</TextBlock>
  <TextBox Margin="5" Grid.Row="2" Grid.Column="1"
   Text="{Binding Path=UnitCost}"></TextBox>
  <TextBlock Margin="7,7,7,0" Grid.Row="3">Description:</TextBlock>
  <TextBox Margin="7" Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2"
   TextWrapping="Wrap" Text="{Binding Path=Description}"></TextBox>
</Grid>
```

Notice that the Grid wrapping all these details is given a name so that you can manipulate it in code and complete your data bindings.

When you first run this application, no information will appear. Even though you've defined your bindings, no source object is available.

When the user clicks the button at runtime, you use the StoreDB class to get the appropriate product data. Although you could create each binding programmatically, this wouldn't make much sense (and it wouldn't save much code over just populating the controls by hand). However, the DataContext property provides a perfect shortcut. If you set it for the Grid that contains all your data binding expressions, all your binding expressions will use it to fill themselves with data.

Here's the event handling code that reacts when the user clicks the button:

```
private void cmdGetProduct_Click(object sender, RoutedEventArgs e)
{
    int ID;
    if (Int32.TryParse(txtID.Text, out ID))
    {
        try
        {
            gridProductDetails.DataContext = App.StoreDB.GetProduct(ID);
        }
        catch
        {
            MessageBox.Show("Error contacting database.");
        }
```

```
    }
    else
    {
        MessageBox.Show("Invalid ID.");
    }
}
```

BINDING WITH NULL VALUES

The current Product class assumes that it will get a full complement of product data. However, database tables frequently include nullable fields, where a null value represents missing or inapplicable information. You can reflect this reality in your data classes by using nullable data types for simple value types such as numbers and dates. For example, in the Product class, you can use `decimal?` instead of `decimal`. Of course, reference types, such as strings and full-fledged objects, always support null values.

The results of binding a null value are predictable: the target element shows nothing at all. For numeric fields, this behavior is useful because it distinguishes between a missing value (in which case the element shows nothing) and a zero value (in which case it shows the text "0"). However, it's worth noting that you can change how WPF handles null values by setting the TargetNullValue property in your binding expression. If you do, the value you supply will be displayed whenever the data source has a null value. Here's an example that shows the text "[No Description Provided]" when the Product.Description property is null:

```
Text="{Binding Path=Description, TargetNullValue=[No Description Provided]}"
```

The square brackets around the TargetNullValue text are optional. In this example, they're intended to help the user recognize that the displayed text isn't drawn from the database.

Updating the Database

You don't need to do anything extra to enable data object updates with this example. The TextBox.Text property uses two-way binding by default, which means that the bound Product object is modified as you edit the text in the text boxes. (Technically, each property is updated when you tab to a new field, because the default source update mode for the TextBox.Text property is LostFocus. To review the different update modes that binding expressions support, refer to Chapter 8.)

You can commit changes to the database at any time. All you need is to add an UpdateProduct() method to the StoreDB class and an Update button to the window. When clicked, your code can grab the current Product object from the data context and use it to commit the update:

```
private void cmdUpdateProduct_Click(object sender, RoutedEventArgs e)
{
    Product product = (Product)gridProductDetails.DataContext;
    try
    {
        App.StoreDB.UpdateProduct(product);
    }
    catch
    {
        MessageBox.Show("Error contacting database.");
    }
}
```

This example has one potential stumbling block. When you click the Update button, the focus changes to that button, and any uncommitted edit is applied to the Product object. However, if you set the Update button to be a default button (by setting IsDefault to true), there's another possibility. A user could make a change in one of the fields and hit Enter to trigger the update process without committing the last change. To avoid this possibility, you can explicitly force the focus to change before you execute any database code, like this:

```
FocusManager.SetFocusedElement(this, (Button)sender);
```

Change Notification

The Product binding example works so well because each Product object is essentially fixed—it never changes (except if the user edits the text in one of the linked text boxes).

For simple scenarios, where you're primarily interested in displaying content and letting the user edit it, this behavior is perfectly acceptable. However, it's not difficult to imagine a different situation, where the bound Product object might be modified elsewhere in your code. For example, imagine an Increase Price button that executes this line of code:

```
product.UnitCost *= 1.1M;
```

■ **Note** Although you could retrieve the Product object from the data context, this example assumes you're also storing it as a member variable in your window class, which simplifies your code and requires less type casting.

When you run this code, you'll find that even though the Product object has been changed, the old value remains in the text box. That's because the text box has no way of knowing that you've changed a value.

You can use three approaches to solve this problem:

- You can make each property in the Product class a dependency property using the syntax you learned about in Chapter 4. (In this case, your class must derive from DependencyObject.) Although this approach gets WPF to do the work for you (which is nice), it makes the most sense in elements—classes that have a visual appearance in a window. It's not the most natural approach for data classes like Product.
- You can raise an event for each property. In this case, the event must have the name *PropertyName*Changed (for example, UnitCostChanged). It's up to you to fire the event when the property is changed.
- You can implement the System.ComponentModel.INotifyPropertyChanged interface, which requires a single event named PropertyChanged. You must then raise the PropertyChanged event whenever a property changes and indicate which property has changed by supplying the property name as a string. It's still up to you to raise this event when a property changes, but you don't need to define a separate event for each property.

The first approach relies on the WPF dependency property infrastructure, while both the second and the third rely on events. Usually, when creating a data object, you'll use the third approach. It's the simplest choice for non-element classes.

■ **Note** You can actually use one other approach. If you suspect a change has been made to a bound object and that bound object doesn't support change notifications in any of the proper ways, you can retrieve the BindingExpression object (using the FrameworkElement.GetBindingExpression() method) and call BindingExpression.UpdateTarget() to trigger a refresh. Obviously, this is the most awkward solution—you can almost see the duct tape that's holding it together.

Here's the definition for a revamped Product class that uses the INotifyPropertyChanged interface, with the code for the implementation of the PropertyChanged event:

```
public class Product : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
          PropertyChanged(this, e);
    }
}
```

Now you simply need to fire the PropertyChanged event in all your property setters:

```
private decimal unitCost;
public decimal UnitCost
{
    get { return unitCost; }
    set {
        unitCost = value;
        OnPropertyChanged(new PropertyChangedEventArgs("UnitCost"));
    }
}
```

If you use this version of the Product class in the previous example, you'll get the behavior you expect. When you change the current Product object, the new information will appear in the text box immediately.

■ **Tip** If several values have changed, you can call OnPropertyChanged() and pass in an empty string. This tells WPF to reevaluate the binding expressions that are bound to any property in your class.

Binding to a Collection of Objects

Binding to a single object is quite straightforward. But life gets more interesting when you need to bind to some collection of objects—for example, all the products in a table.

Although every dependency property supports the single-value binding you've seen so far, collection binding requires an element with a bit more intelligence. In WPF, all the classes that derive from ItemsControl have the ability to show an entire list of items. Data binding possibilities include the ListBox, ComboBox, ListView, and DataGrid (and the Menu and TreeView for hierarchical data).

■ **Tip** Although it seems like WPF offers a relatively small set of list controls, these controls allow you to show your data in a virtually unlimited number of ways. That's because the list controls support data templates, which allow you to control exactly how items are displayed. You'll learn more about data templates in Chapter 20.

To support collection binding, the ItemsControl class defines the three key properties listed in Table 19-1.

Table 19-1. *Properties in the ItemsControl Class for Data Binding*

Name	Description
`ItemsSource`	Points to the collection that has all the objects that will be shown in the list.
`DisplayMemberPath`	Identifies the property that will be used to create the display text for each item.
`ItemTemplate`	Accepts a data template that will be used to create the visual appearance of each item. This property is far more powerful than `DisplayMemberPath`, and you'll learn how to use it in Chapter 20.

At this point, you're probably wondering exactly what type of collections you can stuff in the ItemSource property. Happily, you can use just about anything. All you need is support for the IEnumerable interface, which is provided by arrays, all types of collections, and many more specialized objects that wrap groups of items. However, the support you get from a basic IEnumerable interface is limited to read-only binding. If you want to edit the collection (for example, you want to allow inserts and deletions), you need a bit more infrastructure, as you'll see shortly.

Displaying and Editing Collection Items

Consider the window shown in Figure 19-3, which shows a list of products. When you choose a product, the information for that product appears in the bottom section of the window, where you can edit it. (In this example, a GridSplitter lets you adjust the space given to the top and bottom portions of the window.)

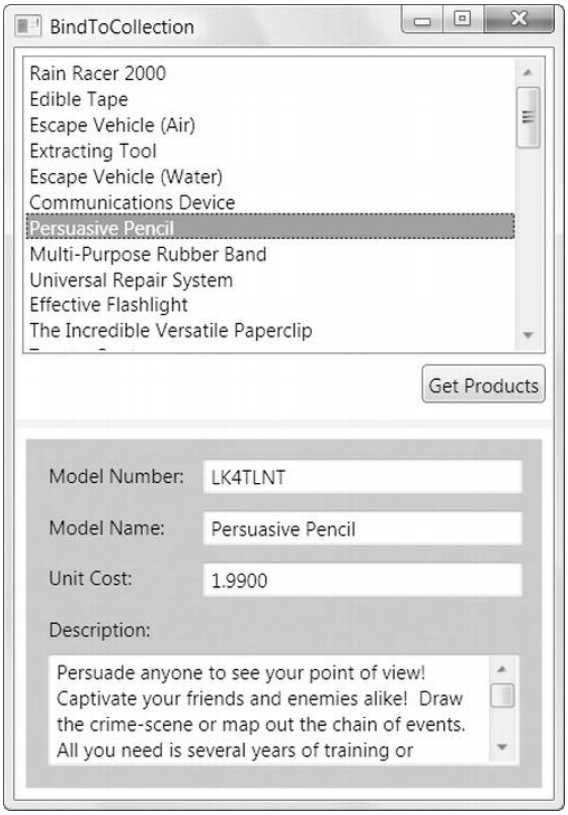


Figure 19-3. *A list of products*

To create this example, you need to begin by building your data access logic. In this case, the StoreDB.GetProducts() method retrieves the list of all the products in the database using the GetProducts stored procedure. A Product object is created for each record and added to a generic List collection. (You could use any collection here—for example, an array or a weakly typed ArrayList would work equivalently.)

Here's the GetProducts() code:

```
public List<Product> GetProducts()
{
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand("GetProducts", con);
    cmd.CommandType = CommandType.StoredProcedure;

    List<Product> products = new List<Product>();
    try
    {
        con.Open();
        SqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            // Create a Product object that wraps the
```

```
            // current record.
            Product product = new Product((string)reader["ModelNumber"],
              (string)reader["ModelName"], (decimal)reader["UnitCost"],
              (string)reader["Description"], (string)reader["CategoryName"],
              (string)reader["ProductImage"]);

            // Add to collection
            products.Add(product);
        }
    }
    finally
    {
        con.Close();
    }
    return products;
}
```

When the Get Products button is clicked, the event handling code calls the GetProducts() method and supplies it as the ItemsSource for list. The collection is also stored as a member variable in the window class for easier access elsewhere in your code.

```
private List<Product> products;

private void cmdGetProducts_Click(object sender, RoutedEventArgs e)
{
    products = App.StoreDB.GetProducts();
    lstProducts.ItemsSource = products;
}
```

This successfully fills the list with Product objects. However, the list doesn't know how to display a product object, so it will simply call the ToString() method. Because this method hasn't been overridden in the Product class, this has the unimpressive result of showing the fully qualified class name for every item (see Figure 19-4).

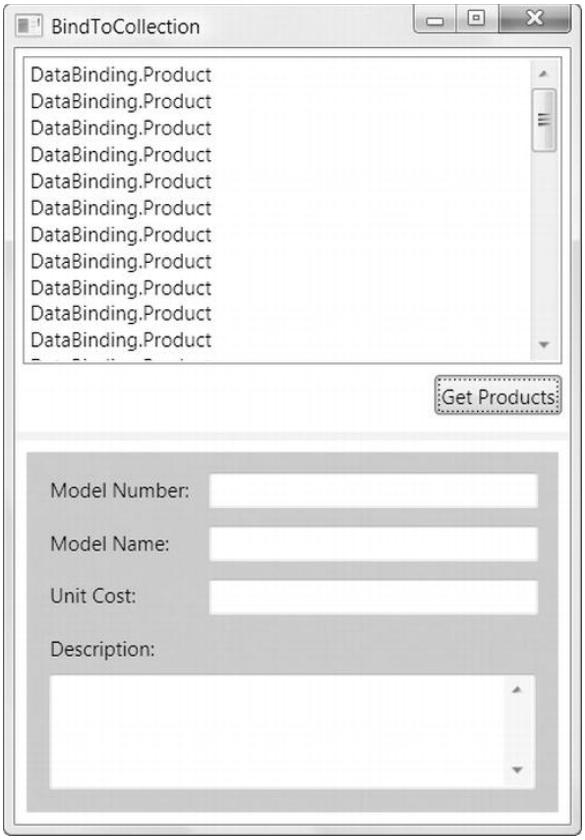


Figure 19-4. *An unhelpful bound list*

You have three options to solve this problem:

- *Set the DisplayMemberPath property of the list*: For example, set this to ModelName to get the result shown in Figure 19-4.
- *Override the ToString() method to return more useful information*: For example, you could return a string with the model number and model name of each item. This approach gives you a way to show more than one property in the list (for example, it's great for combining the FirstName and LastName properties in a Customer class). However, you still don't have much control over how the data is presented.
- *Supply a data template:* This way, you can show any arrangement of property values (along with fixed text). You'll learn how to use this trick in Chapter 20.

After you've decided how to display information in the list, you're ready to move on to the second challenge: displaying the details for the currently selected item in the grid that appears below the list. You could handle this challenge by responding to the SelectionChanged event and manually changing the data context of the grid, but there's a quicker approach that doesn't require any code. You simply need to set a binding expression for the Grid.DataContent property that pulls the selected Product object out of the list, as shown here:

```
<Grid DataContext="{Binding ElementName=lstProducts, Path=SelectedItem}">
  ...
</Grid>
```

When the window first appears, nothing is selected in the list. The ListBox.SelectedItem property is null, and therefore the Grid.DataContext is too, and no information appears. As soon as you select an item, the data context is set to the corresponding object, and all the information appears.

If you try this example, you'll be surprised to see that it's already fully functional. You can edit product items, navigate away (using the list), and then return to see that your edits were successfully committed. In fact, you can even change a value that affects the display text in the list. If you modify the model name and tab to another control, the corresponding entry in the list is refreshed automatically. (Experienced developers will recognize this as a frill that Windows Forms applications lacked.)

■ **Tip** To prevent a field from being edited, set the IsLocked property of the text box to true, or, better yet, use a read-only control like a TextBlock.

MASTER-DETAILS DISPLAY

As you've seen, you can bind other elements to the SelectedItem property of your list to show more details about the currently selected item. Interestingly, you can use a similar technique to build a master-details display of your data. For example, you can create a window that shows a list of categories and a list of products. When the user chooses a category in the first list, you can show just the products that belong to that category in the second list.

To pull this off, you need to have a *parent* data object that provides a collection of related *child* data objects through a property. For example, you could build a Category product that provides a property named Category.Products with the products that belong to that category. (In fact, you can find an example of a Category class that's designed like this in Chapter 21.) You can then build a master-details display with two lists. Fill your first list with Category objects. To show the related products, bind your second list—the list that displays products—to the SelectedItem.Products property of the first list. This tells the second list to grab the current Category object, extract its collection of linked Product objects, and display them.

You can find an example that uses related data in Chapter 21, with a TreeView that shows a categorized list of products.

Of course, to complete this example, from an application perspective you'll need to supply some code. For example, you might need an UpdateProducts() method that accepts your collection or products and executes the appropriate statements. Because an ordinary .NET object doesn't provide any change tracking, this is a situation where you might want to consider using the ADO.NET DataSet (as described a little later in this chapter). Alternatively, you might want to force users to update records one at a time. (One option is to disable the list when text is modified in a text box and force the user to then cancel the change by clicking Cancel or apply it immediately by clicking Update.)

Inserting and Removing Collection Items

One limitation of the previous example is that it won't pick up changes you make to the collection. It notices changed Product objects, but it won't update the list if you add a new item or remove one through code.

For example, imagine you add a Delete button that executes this code:

```
private void cmdDeleteProduct_Click(object sender, RoutedEventArgs e)
{
    products.Remove((Product)lstProducts.SelectedItem);
}
```

The deleted item is removed from the collection, but it remains stubbornly visible in the bound list.

To enable collection change tracking, you need to use a collection that implements the INotifyCollectionChanged interface. Most generic collections don't, including the List collection used in the current example. In fact, WPF includes a single collection that uses INotifyCollectionChanged: the ObservableCollection class.

■ **Note** If you have an object model that you're porting over from the Windows Forms world, you can use the Windows Forms equivalent of ObservableCollection, which is BindingList. The BindingList collection implements IBindingList instead of INotifyCollectionChanged, which includes a ListChanged event that plays the same role as the INotifyCollectionChanged.CollectionChanged event.

You can derive a custom collection from ObservableCollection to customize the way it works, but that's not necessary. In the current example, it's enough to replace the List<Product> object with an ObservableCollection<Product>, as shown here:

```
public List<Product> GetProducts()
{
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand("GetProducts", con);
    cmd.CommandType = CommandType.StoredProcedure;

    ObservableCollection<Product> products = new ObservableCollection<Product>();
    ...
```

The return type can be left as List<Product>, because the ObservableCollection class derives from the List class. To make this example just a bit more generic, you could use ICollection<Product> for the return type, because the ICollection interface has all the members you need to use.

Now, if you remove or add an item programmatically, the list is refreshed accordingly. Of course, it's still up to you to create the data access code that takes place before the collection is modified—for example, the code that removes the product record from the back-end database.

Binding to the ADO.NET Objects

All the features you've learned about with custom objects also work with the ADO.NET disconnected data objects.

For example, you could create the same user interface you see in Figure 19-4 but use the DataSet, DataTable, and DataRow on the back end, rather than the custom Product class and the ObservableCollection.

To try it, start by considering a version of the GetProducts() method that extracts the same data but packages it into a DataTable:

```
public DataTable GetProducts()
{
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand("GetProducts", con);
    cmd.CommandType = CommandType.StoredProcedure;
    SqlDataAdapter adapter = new SqlDataAdapter(cmd);

    DataSet ds = new DataSet();
    adapter.Fill(ds, "Products");
    return ds.Tables[0];
}
```

You can retrieve this DataTable and bind it to the list in almost the same way you did with the ObservableCollection. The only difference is that you can't bind directly to the DataTable itself. Instead, you need to go through an intermediary known as the DataView. Although you can create a DataView by hand, every DataTable has a ready-made DataView object available through the DataTable.DefaultView property.

■ **Note** This limitation is nothing new. Even in a Windows Forms application, all DataTable data binding goes through a DataView. The difference is that the Windows Forms universe can conceal this fact. It allows you to write code that appears to bind directly to a DataTable, when in reality it uses the DataView that's provided by the DataTable.DefaultView property.

Here's the code you need:

```
private DataTable products;

private void cmdGetProducts_Click(object sender, RoutedEventArgs e)
{
    products = App.StoreDB.GetProducts();
    lstProducts.ItemsSource = products.DefaultView;
}
```

Now the list will create a separate entry for each DataRow object in the DataTable.Rows collection. To determine what content is shown in the list, you need to set DisplayMemberPath property with the name of the field you want to show or use a data template (as described in Chapter 20).

The nice aspect of this example is that once you've changed the code that fetches your data, you don't need to make any more modifications. When an item is selected in the list, the Grid underneath grabs the selected item for its data context. The markup you used with the ProductList collection still works, because the property names of the Product class match the field names of the DataRow.

Another nice feature in this example is that you don't need to take any extra steps to implement change notifications. That's because the DataView class implements the IBindingList interface, which allows it to notify the WPF infrastructure if a new DataRow is added or an existing one is removed.

However, you do need to be a little careful when removing a DataRow object. It might occur to you to use code like this to delete the currently selected record:

```
products.Rows.Remove((DataRow)lstProducts.SelectedItem);
```

This code is wrong on two counts. First, the selected item in the list isn't a DataRow object—it's a thin DataRowView wrapper that's provided by the DataView. Second, you probably don't want to remove your DataRow from the collection of rows in the table. Instead, you probably want to mark it as deleted so that when you commit the changes to the database, the corresponding record is removed.

Here's the correct code, which gets the selected DataRowView, uses its Row property to find the corresponding DataRow object, and calls its Delete() method to mark the row for upcoming deletion:

```
((DataRowView)lstProducts.SelectedItem).Row.Delete();
```

At this point, the scheduled-to-be-deleted DataRow disappears from the list, even though it's technically still in the DataTable.Rows collection. That's because the default filtering settings in the DataView hide all deleted records. You'll learn more about filtering in Chapter 21.

Binding to a LINQ Expression

WPF supports Language Integrated Query (LINQ), which is an all-purpose query syntax that works across a variety of data sources and is closely integrated with the C# language. LINQ works with any data source that has a LINQ provider. Using the support that's included with .NET, you can use similarly structured LINQ queries to retrieve data from an in-memory collection, an XML file, or a SQL Server database. And as with other query languages, LINQ allows you to apply filtering, sorting, grouping, and transformations to the data you retrieve.

Although LINQ is somewhat outside the scope of this chapter, you can learn a lot from a simple example. For example, imagine you have a collection of Product objects, named *products*, and you want to create a second collection that contains only those products that exceed $100 in cost. Using procedural code, you can write something like this:

```
// Get the full list of products.
List<Product> products = App.StoreDB.GetProducts();

// Create a second collection with matching products.
List<Product> matches = new List<Product>();
foreach (Product product in products)
{
    if (product.UnitCost >= 100)
    {
        matches.Add(product);
    }
}
```

Using LINQ, you can use the following *expression*, which is far more concise:

```
// Get the full list of products.
List<Product> products = App.StoreDB.GetProducts();

// Create a second collection with matching products.
IEnumerable<Product> matches = from product in products
          where product.UnitCost >= 100
          select product;
```

This example uses LINQ to Collections, which means it uses a LINQ expression to query the data in an in-memory collection. LINQ expressions use a set of new language keywords, including from, in, where, and select. These LINQ keywords are a genuine part of the C# language.

■ **Note** A full discussion of LINQ is beyond the scope of this book. For a detailed treatment, refer to the huge catalog of LINQ examples at `http://tinyurl.com/y9vp4vu`.

LINQ revolves around the IEnumerable<T> interface. No matter what data source you use, every LINQ expression returns some object that implements IEnumerable<T>. Because IEnumerable<T> extends IEnumerable, you can bind it in a WPF window just as you bind an ordinary collection:

```
lstProducts.ItemsSource = matches;
```

Unlike ObservableCollection and the DataTable classes, the IEnumerable<T> interface does not provide a way to add or remove items. If you need this capability, you need to first convert your IEnumerable<T> object into an array or List collection using the ToArray() or ToList() method.

Here's an example that uses ToList() to convert the result of a LINQ query (shown previously) into a strongly typed List collection of Product objects:

```
List<Product> productMatches = matches.ToList();
```

■ **Note** ToList() is an extension method, which means it's defined in a different class from the one in which is used. Technically, ToList() is defined in the System.Linq.Enumerable helper class, and it's available to all IEnumerable<T> objects. However, it won't be available if the Enumerable class isn't in scope, which means the code shown here will not work if you haven't imported the System.Linq namespace.

The ToList() method causes the LINQ expression to be evaluated immediately. The end result is an ordinary collection, which you can deal with in all the usual ways. For example, you can wrap it in an ObservableCollection to get notification events, so any changes you make are reflected in bound controls immediately:

```
ObservableCollection<Product> productMatchesTracked =
  new ObservableCollection<Product>(productMatches);
```

You can then bind the productMatchesTracked collection to a control in your window.

DESIGNING DATA FORMS IN VISUAL STUDIO

Writing data access code and filling in dozens of binding expressions can take a bit of time. And if you create several WPF applications that work with databases, you're likely to find that you're writing similar code and markup in all of them. That's why Visual Studio includes the ability to generate data access code and insert data-bound controls *automatically*.

To use these features, you need to first create a Visual Studio *data source*. (A data source is a definition that allows Visual Studio to recognize your back-end data provider and provide code generation services that use it.) You can create a data source that wraps a database, a web service, an existing data access class, or a SharePoint server. The most common choice is to create an *entity data model*, which is a set of

generated classes that models the tables in a database and allows you to query them, somewhat like the data access component used in this chapter. The benefit is obvious—the entity data model allows you to avoid writing the often tedious data code. The disadvantages are just as clear—if you want the data logic to work exactly the way you want, you'll need to spend some time tweaking options, finding the appropriate extensibility points, and wading through the lengthy code. Examples where you might want fine-grained control over data access logic include if you need to call specific stored procedures, cache the results of a query, use a specific concurrency strategy, or log your data access operations. These feats are usually possible with an entity data model, but they take more work and may mitigate the benefit of automatically generated code.

To create a data source, choose Data ~TRA Add New Data Source to start the Data Source Configuration Wizard, which will ask you to choose your data source (in this case, a database) and then prompt you for additional information (such as the tables and fields you want to query). Once you've added your data source, you can use the Data Sources window to create bound controls. The basic approach is pure simplicity. First choose Data ~TRA Show Data Sources to see the Data Source window, which outlines the tables and fields you've chosen to work with. Then you can drag an individual field from the Data Sources window onto the design surface of a window (to create a bound TextBlock, TextBox, ListBox, or other control) or drag entire tables (to create a bound DataGrid or ListView).

WPF's data form features give you a quick and nifty way to build data-driven applications, but they don't beat knowing what's actually going on. They may be a good choice if you need straightforward data viewing or data editing and you don't want to spend a lot of time fiddling with features and fine-tuning your user interface. They're often a good fit for conventional line-of-business applications. If you'd like to learn more, you can find the official documentation at `http://tinyurl.com/d2taskv`.

Improving Performance in Long Lists

If you deal with huge amounts of data—for example, tens of thousands of records rather than just a few hundred—you know that a good data binding system needs more than sheer features. It also needs to be able to handle a huge amount of data without slowing to a crawl or swallowing an inordinate amount of memory. Fortunately, WPF's list controls are optimized to help you.

In the following sections, you'll learn about several performance enhancements for large lists that are supported by all WPF's list controls (that is, all controls that derive from ItemsControl), including the lowly ListBox and ComboBox and the more specialized ListView, TreeView, and DataGrid that you'll meet in Chapter 22.

Virtualization

The most important feature that WPF's list controls provide is *UI virtualization*, a technique where the list creates container objects for the currently displayed items only. For example, if you have a ListBox control with 50,000 records but the visible area holds only 30 records, the ListBox will create just 30 ListBoxItem objects (plus a few more to ensure good scrolling performance). If the ListBox didn't support UI virtualization, it would need to generate a full set of 50,000 ListBoxItem objects, which would clearly take more memory. More significantly, allocating these objects would take a noticeable amount of time, briefly locking up the application when your code sets the ListBox.ItemsSource property.

The support for UI virtualization isn't actually built into the ListBox or the ItemsControl class. Instead, it's hardwired into the VirtualizingStackPanel container, which functions like a StackPanel except for the added benefit of virtualization support. The ListBox, ListView, and DataGrid automatically use a VirtualizingStackPanel to lay out their children. As a result, you don't need to take any additional steps to get virtualization support. However, the ComboBox class uses the standard nonvirtualized StackPanel. If

you need virtualization support, you must explicitly add it by supplying a new ItemsPanelTemplate, as shown here:

```
<ComboBox>
  <ComboBox.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel></VirtualizingStackPanel>
    </ItemsPanelTemplate>
  </ComboBox.ItemsPanel>
</ComboBox>
```

The TreeView (see Chapter 22) is another control that supports virtualization but, by default, has it switched off. The issue is that the VirtualizingStackPanel didn't support hierarchical data in early releases of WPF. Now it does, but the TreeView disables the feature to guarantee ironclad backward compatibility. Fortunately, you can turn it on with just a single property setting, which is always recommended in trees that contain large amounts of data:

```
<TreeView VirtualizingStackPanel.IsVirtualizing="True" ... >
```

■ **Note** Technically VirtualizingStackPanel inherits from the abstract class VirtualizingPanel. If you want to use a different type of virtualization panel (for example, a Grid that supports virtualization), you'll need to purchase one from a third-party component vendor.

A number of factors can break UI virtualization, sometimes when you don't expect it:

- *Putting your list control in a ScrollViewer:* The ScrollViewer provides a window onto its child content. The problem is that the child content is given unlimited "virtual" space. In this virtual space, the ListBox renders itself at full size, with all of its child items on display. As a side effect, each item gets its own memory-hogging ListBoxItem object. This problem occurs any time you place a ListBox in a container that doesn't attempt to constrain its size; for example, the same problem crops up if you pop it into a StackPanel instead of a Grid.
- *Changing the list's control template and failing to use the ItemsPresenter:* The ItemsPresenter uses the ItemsPanelTemplate, which specifies the VirtualizingStackPanel. If you break this relationship or if you change the ItemsPanelTemplate yourself so it doesn't use a VirtualizingStackPanel, you'll lose the virtualization feature.
- *Not using data binding:* It should be obvious, but if you fill a list programmatically—for example, by dynamically creating the ListBoxItem objects you need—no virtualization will occur. Of course, you can consider using your own optimization strategy, such as creating just those objects that you need and only creating them at the time they're needed. You'll see this technique in action with a TreeView that uses just-in-time node creation to fill a directory tree in Chapter 22.

If you have a large list, you need to avoid these practices to ensure good performance.

Even when you're using UI virtualization, you still have to pay the price of instantiating your in-memory data objects. For example, in the 50,000-item ListBox example, you'll have 50,000 data objects floating around, each with distinct data about one product, customer, order record, or something else. If you want to optimize this portion of your application, you can consider using *data virtualization*—a

technique where you fetch only a batch of records at a time. Data virtualization is a more complex technique, because it assumes the cost of retrieving the data is lower than the cost of maintaining it. This might not be true, depending on the sheer size of the data and the time required to retrieve it. For example, if your application is continually connecting to a network database to fetch more product information as the user scrolls through a list, the end result may be slow scrolling performance and an increased load on the database server.

WPF does not have any controls or classes that support data virtualization. However, that hasn't stopped enterprising developers from creating the missing piece: a "fake" collection that pretends to have all the items but doesn't query the back-end data source until the control requires that data. You can find solid examples of this work at `http://bea.stollnitz.com/blog/?p=344` and `http://bea.stollnitz.com/blog/?p=378`.

Item Container Recycling

Ordinarily, as you scroll through a virtualized list, the control continually creates new item container objects to hold the newly visible items. For example, as you scroll through the 50,000-item ListBox, the ListBox will generate new ListBoxItem objects. But if you enable item container recycling, the ListBox will keep a small set of ListBoxItem objects alive and simply reuse them for different rows by loading them with new data as you scroll.

```
<ListBox VirtualizingStackPanel.VirtualizationMode="Recycling" ... >
```

Item container recycling improves scrolling performance and reduces memory consumption, because there's no need for the garbage collector to find old item objects and release them. Once again, this feature is disabled by default for all controls except the DataGrid to ensure backward compatibility. If you have a large list, you should always turn it on.

Cache Length

As you've already learned, the VirtualizingStackPanel creates a few extra items beyond the ones it shows. That way, it's ready to show these items immediately, when you start scrolling.

In previous versions of WPF, the number of extra items was hard-coded into the VirtualizingStackPanel. But in WPF 4.5, you can fine tune the exact number using two VirtualizingStackPanel properties: CacheLength and CacheLengthUnit. CacheLengthUnit lets you choose how you want to specify the number of extra items—as a number of items, as a number of pages (where a single page includes all the items that fit into the visible "window" of the control), or as a number of pixels (which might make sense if your items show pictures and are different sizes).

The default CacheLength and CacheLengthUnit properties store an extra page of items before and after the currently visible items, like this:

```
<ListBox VirtualizingStackPanel.CacheLength="1"
 VirtualizingStackPanel.CacheLengthUnit="Page" ... />
```

Here's how you might store exactly 100 items before and 100 items after:

```
<ListBox VirtualizingStackPanel.CacheLength="100"
 VirtualizingStackPanel.CacheLengthUnit="Item" ... />
```

And here's how you might store 100 items before and 500 items after (perhaps because you expect users to spend most of their time scrolling down the list rather than up it):

```
<ListBox VirtualizingStackPanel.CacheLength="100,500"
 VirtualizingStackPanel.CacheLengthUnit="Item" ... />
```

It's worth noting that the cache of extra items is filled in the background. That means that the VirtualizingStackPanel will show the visible set of items immediately, as soon as it's created. After that, the VirtualizingStackPanel will begin filling the cache on a lower priority background thread, so it can't lock up your application.

Deferred Scrolling

To further improve scrolling performance, you can switch on *deferred scrolling*. With deferred scrolling, the list display isn't updated when the user drags the thumb along the scroll bar. It's refreshed only once the user releases the thumb. By comparison, when you use ordinary scrolling, the list is refreshed as you drag so that it shows your changing position.

As with item container recycling, you need to explicitly enable deferred scrolling:

```
<ListBox ScrollViewer.IsDeferredScrollingEnabled="True" ... />
```

Clearly, there's a trade-off here between responsiveness and ease of use. If you have complex templates and lots of data, deferred scrolling may be preferred for its blistering speed. But otherwise, your users may prefer the ability to see where they're going as they scroll.

Ordinarily, the VirtualizingStackPanel uses *item-based* scrolling. That means that when you scroll down a small amount, the next item pops into view. You can't scroll down to see just *part* of an item. The smallest amount the panel scrolls is one complete item, whether you click the scroll bar, click a scroll arrow, or call a method like ListBox.ScrollIntoView().

However, you can override this behavior and use *pixel-based* scrolling by setting the VirtualizingStackPanel.ScrollUnit property to Pixel:

```
<ListBox VirtualizingStackPanel.ScrollUnit="Pixel" ... />
```

Now clipping is allowed, which means that as you scroll down you may see just part of an item.

The choice between item-based and pixel-based scrolling depends on the type of content you're showing in the list and your personal preference. In general, pixel-based scrolling is smoother because it allows smaller scrolling intervals, while item-based scrolling is cleaner because the full content of an item is always visible.

Validation

Another key ingredient in any data binding scenario is *validation*—in other words, logic that catches incorrect values and refuses them. You can build validation directly into your controls (for example, by responding to input in the text box and refusing invalid characters), but this low-level approach limits your flexibility.

Fortunately, WPF provides a validation feature that works closely with the data binding system you've explored. Validation gives you two more options to catch invalid values:

- *You can raise errors in your data object:* To notify WPF of an error, simply throw an exception from a property set procedure. Ordinarily, WPF ignores any exceptions that are thrown when setting a property, but you can configure it to show a more helpful visual indication. Other options to implement include the INotifyDataErrorInfo or IDataErrorInfo interface in your data class, which gives you the ability to indicate errors without throwing exceptions.

- *You can define validation at the binding level:* This gives you the flexibility to use the same validation regardless of the input control. Even better, because you define your validation in a distinct class, you can easily reuse it with multiple bindings that store similar types of data.

In general, you'll use the first approach if your data objects already have hardwired validation logic in their property set procedures and you want to take advantage of that logic. You'll use the second approach when you're defining validation logic for the first time and you want to reuse it in different contexts and with different controls. However, some developers choose to use both techniques. They use validation in the data object to defend against a small set of fundamental errors and use validation in the binding to catch a wider set of user-entry errors.

■ **Note** Validation applies only when a value from the target is being used to update the source—in other words, when you're using a TwoWay or OneWayToSource binding.

Validation in the Data Object

Some developers build error checking directly into their data objects. For example, here's a modified version of the Product.UnitPrice property that disallows negative numbers:

```
public decimal UnitCost
{
    get { return unitCost; }
    set
    {
        if (value < 0)
            throw new ArgumentException("UnitCost cannot be negative.");
        else
        {
            unitCost = value;
            OnPropertyChanged(new PropertyChangedEventArgs("UnitCost"));
        }
    }
}
```

The validation logic shown in this example prevents negative price values, but it doesn't give the user any feedback about the problem. As you learned earlier, WPF quietly ignores data binding errors that occur when setting or getting properties. In this case, the user won't have any way of knowing that the update has been rejected. In fact, the incorrect value will remain in the text box—it just won't be applied to the bound data object. To improve this situation, you need the help of the ExceptionValidationRule, which is described next.

DATA OBJECTS AND VALIDATION

Whether it's a good approach to place validation logic in a data object is a matter of never-ending debate.

This approach has some advantages; for example, it catches all errors all the time, whether they occur because of an invalid user edit, a programming mistake, or a calculation that's based on other invalid data.

However, this has the disadvantage of making the data objects more complex and moving validation code that's intended for an application's front end deeper into the back-end data model.

If applied carelessly, property validation can inadvertently rule out perfectly reasonable uses of the data object. They can also lead to inconsistencies and actually *compound* data errors. (For example, it might not make sense for the UnitsInStock to hold a value of –10, but if the underlying database stores this value, you might still want to create the corresponding Product object so you can edit it in your user interface.) Sometimes, problems like these are solved by creating yet another layer of objects—for example, in a complex system, developers might build a rich business object model overtop the bare-bones data object layer.

In the current example, the StoreDB and Product classes are designed to be part of a back-end data access component. In this context, the Product class is simply a glorified package that lets you pass information from one layer of code to another. For that reason, validation code really doesn't belong in the Product class.

The ExceptionValidationRule

The ExceptionValidationRule is a prebuilt validation rule that tells WPF to report all exceptions. To use the ExceptionValidationRule, you must add it to the Binding.ValidationRules collection, as shown here:

```
<TextBox Margin="5" Grid.Row="2" Grid.Column="1">
  <TextBox.Text>
    <Binding Path="UnitCost">
      <Binding.ValidationRules>
        <ExceptionValidationRule></ExceptionValidationRule>
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

This example uses both a value converter and a validation rule. Usually, validation is performed before the value is converted, but the ExceptionValidationRule is a special case. It catches exceptions that occur at any point, including exceptions that occur if the edited value can't be cast to the correct data type, exceptions that are thrown by the property setter, and exceptions that are thrown by the value converter.

So, what happens when validation fails? Validation errors are recorded using the attached properties of the System.Windows.Controls.Validation class. For each failed validation rule, WPF takes three steps:

- It sets the attached Validation.HasError property to true on the bound element (in this case, the TextBox control).
- It creates a ValidationError object with the error details (as returned from the ValidationRule.Validate() method) and adds that to the attached Validation.Errors collection.
- If the Binding.NotifyOnValidationError property is set to true, WPF raises the Validation.Error attached event on the element.

The visual appearance of your bound control also changes when an error occurs. WPF automatically switches the template that a control uses when its Validation.HasError property is true to the template that's defined by the attached Validation.ErrorTemplate property. In a text box, the new template changes the outline of the box to a thin red border.

In most cases, you'll want to augment the error indication in some way and give specific information about the error that caused the problem. You can use code that handles the Error event, or you can supply

a custom control template that provides a different visual indication. But before you tackle either of these tasks, it's worth considering two other ways WPF allows you to catch errors—by using the INotifyDataErrorInfo or IDataErrorInfo in your data objects and by writing custom validation rules.

The INotifyDataErrorInfo Interface

Many object-orientation purists prefer not to raise exceptions to indicate user input errors. There are several possible reasons, including the following: a user input error isn't an exceptional condition, error conditions may depend on the interaction between multiple property values, and it's sometimes worthwhile to hold on to incorrect values for further processing rather than reject them outright. WPF provides two interfaces that allow you to build objects that report errors without throwing exceptions. These interfaces are IDataErrorInfo and INotifyDataErrorInfo.

■ **Note** The IDataErrorInfo and INotifyDataErrorInfo interfaces have the same goal—they replace aggressive unhandled exceptions with a more polite system of error notification. The IDataErrorInfo interface is the original error-tracking interface, which dates back to the first version of .NET. WPF includes it for backward compatibility. The INotifyDataErrorInfo interface is a similar but richer interface that was created for Silverlight and ported to WPF for version 4.5. It has support for additional features, such as multiple errors per property and asynchronous validation. It's the one you'll use in this section.

The following example shows how to use the INotifyDataErrorInfo interface to detect problems with the Product object. The first step is to implement the interface:

```
public class Product : INotifyPropertyChanged, INotifyDataErrorInfo
{ ... }
```

The INotifyDataErrorInfo interface requires just three members. The ErrorsChanged event fires when an error is added or removed. The HasErrors property returns true or false to indicate whether the data object has errors. Finally, the GetErrors() method provides the full error information.

Before you can implement these methods, you need a way to track the errors in your code. The best bet is a private collection, like this:

```
private Dictionary<string, List<string>> errors =
  new Dictionary<string, List<string>>();
```

At first glance, this collection looks a little odd. To understand why, you need to know two facts. First, the INotifyDataErrorInfo interface expects you to link your errors to a specific property. Second, each property can have one or more errors. The easiest way to track this error information is with a Dictionary<T, K> collection that's indexed by property name:

```
private Dictionary<string, List<string>> errors =
  new Dictionary<string, List<string>>();
```

Each entry in the dictionary is itself a collection of errors. This example uses a simple List<Of T> List<T> of strings:

```
private Dictionary<string, List<string>> errors =
  new Dictionary<string, List<string>>();
```

However, you could use a full-fledged error object to bundle together multiple pieces of information about the error, including details such as a text message, error code, severity level, and so on.

Once you have this collection in place, you simply need to add to it when an error occurs (and remove the error information if the error is corrected). To make this process easier, the Product class in this example adds a pair of private methods named SetErrors() and ClearErrors():

```
public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

private void SetErrors(string propertyName, List<string> propertyErrors)
{
    // Clear any errors that already exist for this property.
    errors.Remove(propertyName);

    // Add the list collection for the specified property.
    errors.Add(propertyName, propertyErrors);

    // Raise the error-notification event.
    if (ErrorsChanged != null)
        ErrorsChanged(this, new DataErrorsChangedEventArgs(propertyName));
}

private void ClearErrors(string propertyName)
{
    // Remove the error list for this property.
    errors.Remove(propertyName);

    // Raise the error-notification event.
    if (ErrorsChanged != null)
        ErrorsChanged(this, new DataErrorsChangedEventArgs(propertyName));
}
```

And here's the error-handling logic that ensures that the Product.ModelNumber property is restricted to a string of alphanumeric characters. (Punctuation, spaces, and other special characters are not allowed.)

```
private string modelNumber;
public string ModelNumber
{
    get { return modelNumber; }
    set
    {
        modelNumber = value;

        bool valid = true;
        foreach (char c in modelNumber)
        {
            if (!Char.IsLetterOrDigit(c))
            {
                valid = false;
                break;
            }
        }
        if (!valid)
        {
            List<string> errors = new List<string>();
```

```
            errors.Add("The ModelNumber can only contain letters and numbers.");
            SetErrors("ModelNumber", errors);
        }
        else
        {
            ClearErrors("ModelNumber");
        }

        OnPropertyChanged(new PropertyChangedEventArgs("ModelNumber"));
    }
}
```

The final step is to implement the GetErrors() and HasErrors() methods. The GetErrors() method returns the list of errors for a specific property (or all the errors for all the properties). The HasErrors() property returns true if the Product class has one or more errors.

```
public IEnumerable GetErrors(string propertyName)
{
    if (string.IsNullOrEmpty(propertyName))
    {
        // Provide all the error collections.
        return (errors.Values);
    }
    else
    {
        // Provice the error collection for the requested property
        // (if it has errors).
        if (errors.ContainsKey(propertyName))
        {
            return (errors[propertyName]);
        }
        else
        {
            return null;
        }
    }
}

public bool HasErrors
{
    get
    {
        // Indicate whether the entire Product object is error-free.
        return (errors.Count > 0);
    }
}
```

To tell WPF to use the INotifyDataErrorInfo interface and use it to check for errors when a property is modified, the ValidatesOnNotifyDataErrors property of the binding must be true:

```
<TextBox Margin="5" Grid.Row="2" Grid.Column="1" x:Name="txtModelNumber"
  Text="{Binding Path=ModelNumber, Mode=TwoWay, ValidatesOnNotifyDataErrors=True,
```

```
NotifyOnValidationError=True}"></TextBox>
```

Technically, you don't need to explicitly set ValidatesOnNotifyDataErrors, because it's true by default (unlike the similar ValidatesOnDataErrors property that's used with the IDataErrorInfo interface). However, it's still a good idea to set it explicitly to make your intention to use it clear in the markup.

Incidentally, you can combine both approaches by creating a data object that throws exceptions for some types of errors and uses IDataErrorInfo or INotifyDataErrorInfo to report others. However, keep in mind that these two approaches have a broad difference. When an exception is triggered, the property is not updated in the data object. But when you use the IDataErrorInfo or INotifyDataErrorInfo interface, invalid values are allowed but flagged. The data object is updated, but you can use notifications and the BindingValidationFailed event to inform the user.

Custom Validation Rules

The approach for applying a custom validation rule is similar to applying a custom converter. You define a class that derives from ValidationRule (in the System.Windows.Controls namespace), and you override the Validate() method to perform your validation. If desired, you can add properties that accept other details that you can use to influence your validation (for example, a validation rule that examines text might include a Boolean CaseSensitive property).

Here's a complete validation rule that restricts decimal values to fall between some set minimum and maximum. By default, the minimum is set at 0, and the maximum is the largest number that will fit in the decimal data type, because this validation rule is intended for use with currency values. However, both these details are configurable through properties for maximum flexibility.

```
public class PositivePriceRule : ValidationRule
{
    private decimal min = 0;
    private decimal max = Decimal.MaxValue;

    public decimal Min
    {
        get { return min; }
        set { min = value; }
    }

    public decimal Max
    {
        get { return max; }
        set { max = value; }
    }

    public override ValidationResult Validate(object value,
      CultureInfo cultureInfo)
    {
        decimal price = 0;

        try
        {
            if (((string)value).Length > 0)
                price = Decimal.Parse((string)value, NumberStyles.Any, culture);
        }
```

```
            catch
            {
                return new ValidationResult(false, "Illegal characters.");
            }

            if ((price < Min) || (price > Max))
            {
                return new ValidationResult(false,
                  "Not in the range " + Min + " to " + Max + ".");
            }
            else
            {
                return new ValidationResult(true, null);
            }
        }
    }
```

Notice that the validation logic uses the overloaded version of the Decimal.Parse() method that accepts a value from the NumberStyles enumeration. That's because validation is always performed *before* conversion. If you've applied both the validator and the converter to the same field, you need to be sure that your validation will succeed if there's a currency symbol present. The success or failure of the validation logic is indicated by returning a ValidationResult object. The IsValid property indicates whether the validation succeeded, and if it didn't, the ErrorContent property provides an object that describes the problem. In this example, the error content is set to a string that will be displayed in the user interface, which is the most common approach.

Once you've perfected your validation rule, you're ready to attach it to an element by adding it to the Binding.ValidationRules collection. Here's an example that uses the PositivePriceRule and sets the Maximum at 999.99:

```
<TextBlock Margin="7" Grid.Row="2">Unit Cost:</TextBlock>
  <TextBox Margin="5" Grid.Row="2" Grid.Column="1">
    <TextBox.Text>
      <Binding Path="UnitCost">
        <Binding.ValidationRules>
          <local:PositivePriceRule Max="999.99" />
        </Binding.ValidationRules>
      </Binding>
    </TextBox.Text>
</TextBox>
```

Often, you'll define a separate validation rule object for each element that uses the same type of rule. That's because you might want to adjust the validation properties (such as the minimum and maximum in the PositivePriceRule) separately. If you know that you want to use *exactly* the same validation rule for more than one binding, you can define the validation rule as a resource and simply point to it in each binding using the StaticResource markup extension.

As you've probably gathered, the Binding.ValidationRules collection can take an unlimited number of rules. When the value is committed to the source, WPF checks each validation rule, in order. (Remember, a value in a text box is committed to the source when the text box loses focus, unless you specify otherwise with the UpdateSourceTrigger property.) If all the validation succeeds, WPF then calls the converter (if one exists) and applies the value to the source.

■ **Note** If you add the PositivePriceRule followed by the ExceptionValidationRule, the PositivePriceRule will be evaluated first. It will capture errors that result from an out-of-range value. However, the ExceptionValidationRule will catch type-casting errors that result if you type an entry that can't be cast to a decimal value (such as a sequence of letters).

When you perform validation with the PositivePriceRule, the behavior is the same as when you use the ExceptionValidationRule—the text box is outlined in red, the HasError and Errors properties are set, and the Error event fires. To provide the user with some helpful feedback, you need to add a bit of code or customize the ErrorTemplate. You'll learn how to take care of both approaches in the following sections.

■ **Tip** Custom validation rules can be extremely specific so that they target a specific constraint for a specific property or much more general so that they can be reused in a variety of scenarios. For example, you could easily create a custom validation rule that validates a string using a regular expression you specify, with the help of .NET's System.Text.RegularExpressions.Regex class. Depending on the regular expression you use, you could use this validation rule with a variety of pattern-based text data, such as e-mail addresses, phone numbers, IP addresses, and ZIP codes.

Reacting to Validation Errors

In the previous example, the only indication the user receives about an error is a red outline around the offending text box. To provide more information, you can handle the Error event, which fires whenever an error is stored or cleared. However, you must first be sure you've set the Binding.NotifyOnValidationError property to true:

```
<Binding Path="UnitCost" NotifyOnValidationError="True">
```

The Error event is a routed event that uses bubbling, so you can handle the Error event for multiple controls by attaching an event handler in the parent container, as shown here:

```
<Grid Name="gridProductDetails" Validation.Error="validationError">
```

Here's the code that reacts to this event and displays a message box with the error information. (A less disruptive option would be to show a tooltip or display the error information somewhere else in the window.)

```
private void validationError(object sender, ValidationErrorEventArgs e)
{
    // Check that the error is being added (not cleared).
    if (e.Action == ValidationErrorEventAction.Added)
    {
        MessageBox.Show(e.Error.ErrorContent.ToString());
    }
}
```

The ValidationErrorEventArgs.Error property provides a ValidationError object that bundles together several useful details, including the exception that caused the problem (Exception), the validation rule that

was violated (ValidationRule), the associated Binding object (BindingInError), and any custom information that the ValidationRule object has returned (ErrorContent).

If you're using custom validation rules, you'll almost certainly choose to place the error information in the ValidationError.ErrorContent property. If you're using the ExceptionValidationRule, the ErrorContent property will return the Message property of the corresponding exception. However, there's a catch. If an exception occurs because the data type cannot be cast to the appropriate value, the ErrorContent works as expected and reports the problem. However, if the property setter in the data object throws an exception, this exception is wrapped in a TargetInvocationException, and the ErrorContent provides the text from the TargetInvocationException.Message property, which is the much less helpful warning "Exception has been thrown by the target of an invocation."

Thus, if you're using your property setters to raise exceptions, you'll need to add code that checks the InnerException property of the TargetInvocationException. If it's not null, you can retrieve the original exception object and use its Message property instead of the ValidationError.ErrorContent property.

Getting a List of Errors

At certain times, you might want to get a list of all the outstanding errors in your current window (or a given container in that window). This task is relatively straightforward—all you need to do is walk through the element tree testing the Validation.HasError property of each element.

The following code routine demonstrates an example that specifically searches out invalid data in TextBox objects. It uses recursive code to dig down through the entire element hierarchy. Along the way, the error information is aggregated into a single message that's then displayed to the user.

```
private void cmdOK_Click(object sender, RoutedEventArgs e)
{
    string message;
    if (FormHasErrors(message))
    {
        // Errors still exist.
        MessageBox.Show(message);
    }
    else
    {
        // There are no errors. You can continue on to complete the task
        // (for example, apply the edit to the data source.).
    }
}

private bool FormHasErrors(out string message)
{
    StringBuilder sb = new StringBuilder();
    GetErrors(sb, gridProductDetails);
    message = sb.ToString();
    return message != "";
}

private void GetErrors(StringBuilder sb, DependencyObject obj)
{
    foreach (object child in LogicalTreeHelper.GetChildren(obj))
    {
```

```
            TextBox element = child as TextBox;
            if (element == null) continue;

            if (Validation.GetHasError(element))
            {
                sb.Append(element.Text + " has errors:\r\n");
                foreach (ValidationError error in Validation.GetErrors(element))
                {
                    sb.Append("  " + error.ErrorContent.ToString());
                    sb.Append("\r\n");
                    }
                }
                // Check the children of this object for errors.
                GetErrors(sb, element);
            }
        }
    }
```

In a more complete implementation, the FormHasErrors() method would probably create a collection of objects with error information. The cmdOK_Click() event handler would then be responsible for constructing an appropriate message.

Showing a DifferenError Indicator

To get the most out of WPF validation, you'll want to create your own error template that flags errors in an appropriate way. At first glance, this seems like a fairly low-level way to go about reporting an error—after all, a standard control template gives you the ability to customize the composition of a control in minute detail. However, an error template isn't like an ordinary control template.

Error templates use the *adorner layer*, which is a drawing layer that exists just above ordinary window content. Using the adorner layer, you can add a visual embellishment to indicate an error without replacing the control template of the control underneath or changing the layout in your window. The standard error template for a text box works by adding a red Border element that floats just above the corresponding text box (which remains unchanged underneath). You can use an error template to add other details such as images, text, or some other sort of graphical detail that draws attention to the problem.

The following markup shows an example. It defines an error template that uses a green border and adds an asterisk next to the control with the invalid input. The template is wrapped in a style rule so that it's automatically applied to all the text boxes in the current window:

```
<Style TargetType="{x:Type TextBox}">
  <Setter Property="Validation.ErrorTemplate">
    <Setter.Value>
      <ControlTemplate>
        <DockPanel LastChildFill="True">
          <TextBlock DockPanel.Dock="Right" Foreground="Red"
           FontSize="14" FontWeight="Bold">*</TextBlock>
          <Border BorderBrush="Green" BorderThickness="1">
            <AdornedElementPlaceholder></AdornedElementPlaceholder>
          </Border>
        </DockPanel>
      </ControlTemplate>
```

```
    </Setter.Value>
  </Setter>
</Style>
```

The AdornedElementPlaceholder is the glue that makes this technique work. It represents the control itself, which exists in the element layer. By using the AdornedElementPlaceholder, you can arrange your content in relation to the text box underneath.

As a result, the border in this example is placed directly overtop of the text box, no matter what its dimensions are. The asterisk in this example is placed just to the right (as shown in Figure 19-5). Best of all, the new error template content is superimposed on top of the existing content without triggering any change in the layout of the original window. (In fact, if you're careless and include too much content in the adorner layer, you'll end up overwriting other portions of the window.)

Figure 19-5. *Flagging an error with an error template*

■ **Tip** If you want your error template to appear superimposed over the element (rather than positioned around it), you can place both your content and the AdornerElementPlaceholder in the same cell of a Grid. Alternatively, you can leave out the AdornerElementPlaceholder altogether, but then you lose the ability to position your content precisely in relation to the element underneath.

This error template still suffers from one problem—it doesn't provide any additional information about the error. To show these details, you need to extract them using data binding. One good approach is to take the error content of the first error and use it for tooltip text of your error indicator. Here's a template that does exactly that:

```
<ControlTemplate>
  <DockPanel LastChildFill="True">
    <TextBlock DockPanel.Dock="Right"
     Foreground="Red" FontSize="14" FontWeight="Bold"
     ToolTip="{Binding ElementName=adornerPlaceholder,
```

```
                Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"
    >*</TextBlock>
    <Border BorderBrush="Green" BorderThickness="1">
      <AdornedElementPlaceholder Name="adornerPlaceholder">
      </AdornedElementPlaceholder>
    </Border>
  </DockPanel>
</ControlTemplate>
```

The Path of the binding expression is a little convoluted and bears closer examination. The source of this binding expression is the AdornedElementPlaceholder, which is defined in the control template:

```
ToolTip="{Binding ElementName=adornerPlaceholder, ...
```

The AdornedElementPlaceholder class provides a reference to the element underneath (in this case, the TextBox object with the error) through a property named AdornedElement:

```
ToolTip="{Binding ElementName=adornerPlaceholder,
          Path=AdornedElement ...
```

To retrieve the actual error, you need to check the Validation.Errors property of this element. However, you need to wrap the Validation.Errors property in parentheses to indicate that it's an attached property, rather than a property of the TextBox class:

```
ToolTip="{Binding ElementName=adornerPlaceholder,
          Path=AdornedElement.(Validation.Errors) ...
```

Finally, you need to use an indexer to retrieve the first ValidationError object from the collection and then extract its Error content property:

```
ToolTip="{Binding ElementName=adornerPlaceholder,
          Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"
```

Now you can see the error message when you move the mouse over the asterisk.

Alternatively, you might want to show the error message in a tooltip for the Border or TextBox itself so that the error message appears when the user moves the mouse over any portion of the control. You can perform this trick without the help of a custom error template—all you need is a trigger on the TextBox control that reacts when Validation.HasError becomes true and applies the tooltip with the error message. Here's an example:

```
<Style TargetType="{x:Type TextBox}">
  ...
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="True">
      <Setter Property="ToolTip"
       Value="{Binding RelativeSource={RelativeSource Self},
       Path=(Validation.Errors)[0].ErrorContent}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Figure 19-6 shows the result.

Figure 19-6. *Turning a validation error message into a tooltip*

Validating Multiple Values

The approaches you've seen so far allow you to validate individual values. However, there are many situations where you need to perform validation that incorporates two or more bound values. For example, a Project object isn't valid if its StartDate falls after its EndDate. An Order object shouldn't have a Status of Shipped and a ShipDate of null. A Product shouldn't have a ManufacturingCost greater than the RetailPrice. And so on.

There are various ways to design your application to deal with these limitations. In some cases, it makes sense to build a smarter user interface. (For example, if some fields aren't appropriate based on the information on other fields, you may choose to disable them.) In other situations, you'll build this logic into the data class itself. (However, this won't work if the data is valid in some situations but just not acceptable in a particular editing task.) And lastly, you can use *binding groups* to create custom validation rules that apply this sort of rule through WPF's data binding system.

The basic idea behind binding groups is simple. You create a custom validation rule that derives from the ValidationRule class, as you saw earlier. But instead of applying that validation rule to a single binding expression, you attach it to the container that holds all your bound controls. (Typically, this is the same container that has the DataContext set with the data object.) WPF then uses that to validate the entire data object when an edit is committed, which is known as *item-level validation*.

For example, the following markup creates a binding group for a Grid by setting the BindingGroup property (which all elements include). It then adds a single validation rule, named NoBlankProductRule. The rule automatically applies to the bound Product object that's stored in the Grid.DataContext property.

```
<Grid Name="gridProductDetails"
 DataContext="{Binding ElementName=lstProducts, Path=SelectedItem}">

  <Grid.BindingGroup>
    <BindingGroup x:Name="productBindingGroup">
```

```
      <BindingGroup.ValidationRules>
        <local:NoBlankProductRule></local:NoBlankProductRule>
      </BindingGroup.ValidationRules>
    </BindingGroup>
  </Grid.BindingGroup>

  <TextBlock Margin="7">Model Number:</TextBlock>
  <TextBox Margin="5" Grid.Column="1" Text="{Binding Path=ModelNumber}">
  </TextBox>

  ...
</Grid>
```

In the validation rules you've seen so far, the Validate() method receives a single value to inspect. But when using binding groups, the Validate() method receives a BindingGroup object instead. This BindingGroup wraps your bound data object (in this case, a Product).

Here's how the Validate() method begins in the NoBlankProductRule class:

```
public override ValidationResult Validate(object value, CultureInfo cultureInfo)
{
    BindingGroup bindingGroup = (BindingGroup)value;
    Product product = (Product)bindingGroup.Items[0];
    ...
}
```

You'll notice that the code retrieves the first object from the BindingGroup.Items collection. In this example, there is just a single data object. But it is possible (albeit less common) to create binding groups that apply to different data objects. In this case, you receive a collection with all the data objects.

■ **Note** To create a binding group that applies to more than one data object, you must set the BindingGroup.Name property to give your binding group a descriptive name. You then set the BindingGroupName property in your binding expressions to match:

```
Text="{Binding Path=ModelNumber, BindingGroupName=MyBindingGroup}"
```

This way, each binding expression explicitly opts in to the binding group, and you can use the same binding group with expressions that work on different data objects.

There's another unexpected difference in the way the Validate() method works with a binding group. By default, the data object you receive is for the original object, with none of the new changes applied. To get the new values you want to validate, you need to call the BindingGroup.GetValue() method and pass in your data object and the property name:

```
string newModelName = (string)bindingGroup.GetValue(product, "ModelName");
```

This design makes a fair bit of sense. By holding off on actually applying the new value to the data object, WPF ensures that the change won't trigger other updates or synchronization tasks in your application before they make sense.

Here's the complete code for the NoBlankProductRule:

```
public class NoBlankProductRule : ValidationRule
{
    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        BindingGroup bindingGroup = (BindingGroup)value;

        // This product has the original values.
        Product product = (Product)bindingGroup.Items[0];

        // Check the new values.
        string newModelName = (string)bindingGroup.GetValue(product,
          "ModelName");
        string newModelNumber = (string)bindingGroup.GetValue(product,
          "ModelNumber");

        if ((newModelName == "") && (newModelNumber == ""))
        {
            return new ValidationResult(false,
              "A product requires a ModelName or ModelNumber.");
        }
        else
        {
            return new ValidationResult(true, null);
        }
    }
}
```

When using item-level validation, you'll usually need to create a tightly coupled validation rule like this one. That's because the logic doesn't usually generalize that easily (in other words, it's unlikely to apply to a similar but slightly different case with another data object). You also need to use the specific property name when calling GetValue(). As a result, the validation rules you create for item-level validation probably won't be as neat, streamlined, and reusable as those you create for validating individual values.

As it stands, the current example isn't quite finished. Binding groups use a transactional editing system, which means that it's up to you to officially commit the edit before your validation logic runs. The easiest way to do this is to call the BindingGroup.CommitEdit() method. You can use an event handler that runs when a button is clicked or when an editing control loses focus, as shown here:

```
<Grid Name="gridProductDetails" TextBox.LostFocus="txt_LostFocus"
 DataContext="{Binding ElementName=lstProducts, Path=SelectedItem}">
```

And here's the event handling code:

```
private void txt_LostFocus(object sender, RoutedEventArgs e)
{
    productBindingGroup.CommitEdit();
}
```

If validation fails, the entire Grid is considered invalid and is outlined with a thin red border. As with edit controls like the TextBox, you can change the Grid's appearance by modifying its Validation.ErrorTemplate.

■ **Note** Item-level validation works more seamlessly with the DataGrid control you'll explore in Chapter 22. It handles the transactional aspects of editing, triggering field navigation when the user moves from one cell to another, and calling BindingGroup.CommitEdit() when the user moves from one row to another.

Data Providers

In most of the examples you've seen, the top-level data source has been supplied by programmatically setting the DataContext of an element or the ItemsSource property of a list control. In general, this is the most flexible approach, particularly if your data object is constructed by another class (such as StoreDB). However, you have other options.

One technique is to define your data object as a resource in your window (or some other container). This works well if you can construct your object declaratively, but it makes less sense if you need to connect to an outside data store (such as a database) at runtime. However, some developers still use this approach (often in a bid to avoid writing event handling code). The basic idea is to create a wrapper object that fetches the data you need in its constructor. For example, you could create a resource section like this:

```
<Window.Resources>
  <ProductListSource x:Key="products"></ProductListSource>
</Window.Resources>
```

Here, ProductListSource is a class that derives from ObservableCollection<Products>. Thus, it has the ability to store a list of products. It also has some basic logic in the constructor that calls StoreDB.GetProducts() to fill itself.

Now, other elements can use this in their binding:

```
<ListBox ItemsSource="{StaticResource products}" ... >
```

This approach seems tempting at first, but it's a bit risky. When you add error handling, you'll need to place it in the ProductListSource class. You may even need to show a message explaining the problem to the user. As you can see, this approach mingles the data model, the data access code, and the user interface code in a single muddle, so it doesn't make much sense when you need to access outside resources (files, databases, and so on).

Data providers are, in some ways, an extension of this model. A data provider gives you the ability to bind directly to an object that you define in the resources section of your markup. However, instead of binding directly to the data object itself, you bind to a data provider that's able to retrieve or construct that object. This approach makes sense if the data provider is full-featured—for example, if it can raise events when exceptions occur and provides properties that allow you to configure other details about its operation. Unfortunately, the data providers that are included in WPF aren't yet up to this standard. They're too limited to be worth the trouble in a situation with external data (for example, when fetching the information from a database or a file). They may make sense in simpler scenarios—for example, you could use a data provider to glue together some controls that supply input to a class that calculates a result. However, they add relatively little in this situation except the ability to reduce event handling code in favor of markup.

All data providers derive from the System.Windows.Data.DataSourceProvider class. Currently, WPF provides just two data providers:

- *ObjectDataProvider*, which gets information by calling a method in another class
- *XmlDataProvider*, which gets information directly from an XML file

The goal of both of these objects is to allow you to instantiate your data object in XAML, without resorting to event handling code.

■ **Note** There's still one more option: you can explicitly create a view object as a resource in your XAML, bind your controls to the view, and fill your view with data in code. This option is primarily useful if you want to customize the view by applying sorting and filtering, although it's also preferred by some developers as a matter of taste. In Chapter 21, you'll learn how to use views.

The ObjectDataProvider

The ObjectDataProvider allows you to get information from another class in your application. It adds the following features:

- It can create the object you need and pass parameters to the constructor.
- It can call a method in that object and pass method parameters to it.
- It can create the data object asynchronously. (In other words, it can wait until after the window is loaded and then perform the work in the background.)

For example, here's a basic ObjectDataProvider that creates an instance of the StoreDB class, calls its GetProducts() method, and makes the data available to the rest of your window:

```
<Window.Resources>
  <ObjectDataProvider x:Key="productsProvider" ObjectType="{x:Type local:StoreDB}"
    MethodName="GetProducts"></ObjectDataProvider>
</Window.Resources>
```

You can now create a binding that gets the source from the ObjectDataProvider:

```
<ListBox Name="lstProducts" DisplayMemberPath="ModelName"
 ItemsSource="{Binding Source={StaticResource productsProvider}}"></ListBox>
```

This tag looks like it binds to the ObjectDataProvider, but the ObjectDataProvider is intelligent enough to know you really want to bind to the product list that it returns from the GetProducts() method.

■ **Note** The ObjectDataProvider, like all data providers, is designed to retrieve data but not update it. In other words, there's no way to force the ObjectDataProvider to call a different method in the StoreDB class to trigger an update. This is just one example of how the data provider classes in WPF are less mature than other implementations in other frameworks, such as the data source controls in ASP.NET.

Error Handling

As written, this example has a giant limitation. When you create this window, the XAML parser creates the window and calls the GetProducts() method so it can set up the binding. Everything runs smoothly if the GetProducts() method returns the data you want, but the result isn't as nice if an unhandled exception is

thrown (for example, if the database is too busy or isn't reachable). At this point, the exception bubbles up from the InitializeComponent() call in the window constructor. The code that's showing this window needs to catch this error, which is conceptually confusing. And there's no way to continue and show the window—even if you catch the exception in the constructor, the rest of the window won't be initialized properly.

Unfortunately, there's no easy way to solve this problem. The ObjectDataProvider class includes an IsInitialLoadEnabled property that you can set to false to prevent it from calling GetProducts() when the window is first created. If you set this, you can call Refresh() later to trigger the call. Unfortunately, if you use this technique, your binding expression will fail, because the list won't be able to retrieve its data source. (This is unlike most data binding errors, which fail silently without raising an exception.)

So, what's the solution? You can construct the ObjectDataProvider programmatically, although you'll lose the benefit of declarative binding, which is the reason you probably used the ObjectDataProvider in the first place. Another solution is to configure the ObjectDataProvider to perform its work asynchronously, as described in the next section. In this situation, exceptions cause a silent failure (although a trace message will still be displayed in the Debug window detailing the error).

Asynchronous Support

Most developers will find that there aren't many reasons for using the ObjectDataProvider. Usually, it's easier to simply bind directly to your data object and add the tiny bit of code that calls the class that queries the data (such as StoreDB). However, there is one reason that you might use the ObjectDataProvider—to take advantage of its support for asynchronous data querying.

```
<ObjectDataProvider IsAsynchronous="True" ... >
```

It's deceptively simple. As long as you set the ObjectDataProvider.IsAsynchronous property to true, the ObjectDataProvider performs its work on a background thread. As a result, your interface isn't tied up while the work is underway. Once the data object has been constructed and returned from the method, the ObjectDataProvider makes it available to all bound elements.

■ **Tip** If you don't want to use the ObjectDataProvider, you can still launch your data access code asynchronously. The trick is to use WPF's support for multithreaded applications. One useful tool is the BackgroundWorker component that's described in Chapter 31. When you use the BackgroundWorker, you gain the benefit of optional cancellation support and progress reporting. However, incorporating the BackgroundWorker into your user interface is more work than simply setting the ObjectDataProvider.IsAsynchronous property.

ASYNCHRONOUS DATA BINDINGS

WPF also provides asynchronous support through the IsAsync property of each Binding object. However, this feature is far less useful than the asynchronous support in the ObjectDataProvider. When you set Binding.IsAsync to true, WPF retrieves the bound property from the data object asynchronously. However, the data object itself is still created synchronously.

For example, imagine you create an asynchronous binding for the StoreDB example that looks like this:

```
<TextBox Text="{Binding Path=ModelNumber, IsAsync=True}" />
```

Even though you're using an asynchronous binding, you'll still be forced to wait while your code queries the database. Once the product collection is created, the binding will query the Product.ModelNumber property of the current product object asynchronously. This behavior has little benefit, because the property procedures in the Product class take a trivial amount of time to execute. In fact, all well-designed data objects are built out of lightweight properties such as this, which is one reason that the WPF team had serious reservations about providing the Binding.IsAsync property at all!

The only way to take advantage of Binding.IsAsync is to build a specialized class that includes time-consuming logic in a property get procedure. For example, consider an analysis application that binds to a data model. This data object might include a piece of information that's calculated using a time-consuming algorithm. You could bind to this property using an asynchronous binding but bind to all the other properties with synchronous bindings. That way, some information will appear immediately in your application, and the additional information will appear once it's ready.

WPF also includes a priority binding feature that builds on asynchronous bindings. Priority binding allows you to supply several asynchronous bindings in a prioritized list. The highest-priority binding is preferred, but if it's still being evaluated, a lower-priority binding is used instead. Here's an example:

```
<TextBox>
  <TextBox.Text>
    <PriorityBinding>
      <Binding Path="SlowSpeedProperty" IsAsync="True" />
      <Binding Path="MediumSpeedProperty" IsAsync="True" />
      <Binding Path="FastSpeedProperty" />
    </PriorityBinding>
  </TextBox.Text>
</TextBox>
```

This assumes that the current data context contains an object with three properties named SlowSpeedProperty, MediumSpeedProperty, and FastSpeedProperty. The bindings are placed in their order of importance. As a result, SlowSpeedProperty is always used to set the text, if it's available. But if the first binding is still in the midst of reading SlowSpeedProperty (in other words, there is time-consuming logic in the property get procedure), MediumSpeedProperty is used instead. If that's not available, FastSpeedProperty is used. For this approach to work, you must make all the binding asynchronous, except the fastest, lowest-priority binding at the end of the list. This binding can be asynchronous (in which case the text box will appear empty until the value is retrieved) or synchronous (in which case the window won't be frozen until the synchronous binding has finished its work).

The XmlDataProvider

The XmlDataProvider provides a quick and straightforward way to extract XML data from a separate file, web location, or application resource and make it available to the elements in your application. The XmlDataProvider is designed to be read-only (in other words, it doesn't provide the ability to commit changes), and it isn't able to deal with XML data that may come from other sources (such as a database record, a web service message, and so on). As a result, it's a fairly specific tool.

If you've used .NET to work with XML in the past, you already know that .NET provides a rich set of libraries for reading, writing, and manipulating XML. You can use streamlined reader and writer classes that allow you to step through XML files and handle each element with custom code, you can use XPath or the DOM to hunt for specific bits of content, and you can use serializer classes to convert entire objects to and from an XML representation. Each of these approaches has advantages and disadvantages, but all of them are more powerful than the XmlDataProvider.

If you foresee needing the ability to modify XML or to convert XML data into an object representation that you can work with in your code, you're better off using the extensive XML support that already exists in .NET. The fact that your data is stored in an XML representation then becomes a low-level detail that's irrelevant to the way you construct your user interface. (Your user interface can simply bind to data objects, as in the database-backed examples you've seen in this chapter.) However, if you absolutely must have a quick way to extract XML content and your requirements are relatively light, the XmlDataProvider is a reasonable choice.

To use the XmlDataProvider, you begin by defining it and pointing it to the appropriate file by setting the Source property.

```
<XmlDataProvider x:Key="productsProvider" Source="store.xml"></XmlDataProvider>
```

You can also set the Source programmatically (which is important if you aren't sure what the file name is that you need to use). By default, the XmlDataProvider loads the XML content asynchronously, unless you explicitly set XmlDataProvider.IsAsynchronous to false.

Here's a portion of the simple XML file used in this example. It wraps the entire document in a top-level Products element and places each product in a separate Product element. The individual properties for each product are provided as nested elements.

```
<Products>
  <Product>
    <ProductID>355</ProductID>
    <CategoryID>16</CategoryID>
    <ModelNumber>RU007</ModelNumber>
    <ModelName>Rain Racer 2000</ModelName>
    <ProductImage>image.gif</ProductImage>
    <UnitCost>1499.99</UnitCost>
    <Description>Looks like an ordinary bumbershoot ... </Description>
  </Product>
  <Product>
    <ProductID>356</ProductID>
    <CategoryID>20</CategoryID>
    <ModelNumber>STKY1</ModelNumber>
    <ModelName>Edible Tape</ModelName>
    <ProductImage>image.gif</ProductImage>
    <UnitCost>3.99</UnitCost>
    <Description>The latest in personal survival gear ... </Description>
  </Product>
  ...
</Products>
```

To pull information from your XML, you use XPath expressions. XPath is a powerful standard that allows you to retrieve the portions of a document that interest you. Although a full discussion of XPath is beyond the scope of this book, it's easy to sketch out the essentials.

XPath uses a pathlike notation. For example, the path / identifies the root of an XML document, and /Products identifies a root element named <Products>. The path /Products/Product selects every <Product> element inside the <Products> element.

When using XPath with the XmlDataProvider, your first task is to identify the root node. In this case, that means selecting the <Products> element that contains all the data. (If you wanted to focus on a specific section of the XML document, you would use a different top-level element.)

```
<XmlDataProvider x:Key="productsProvider" Source="/store.xml"
 XPath="/Products"></XmlDataProvider>
```

The next step is to bind your list. When working the XmlDataProvider, you use the Binding.XPath property instead of the Binding.Path property. This gives you the flexibility to dig into your XML as deeply as you need.

Here's the markup that pulls out all the <Product> elements:

```
<ListBox Name="lstProducts" Margin="5" DisplayMemberPath="ModelName"
 ItemsSource="{Binding Source={StaticResource products}, XPath=Product}" ></ListBox>
```

When setting the XPath property in a binding, you need to remember that your expression is relative to the current position in the XML document. For that reason, you don't need to supply the full path /Products/Product in the list binding. Instead, you can simply use the relative path Product, which starts from the <Products> node that was selected by the XmlDataProvider.

Finally, you need to wire up each of the elements that displays the product details. Once again, the XPath expression you write is evaluated relative to the current node (which will be the <Product> element for the current product). Here's an example that binds to the <ModelNumber> element:

```
<TextBox Text="{Binding XPath=ModelNumber}"></TextBox>
```

Once you make these changes, you'll be left with an XML-based example that's nearly identical to the object-based bindings you've seen so far. The only difference is that all the data is treated as ordinary text. To convert it to a different data type or a different representation, you'll need to use a value converter.

The Last Word

This chapter took a thorough look at data binding. You learned how to create data binding expressions that draw information from custom objects and how to push changes back to the source. You also learned how to use change notification, bind entire collections, and bind to the ADO.NET DataSet.

In many ways, WPF data binding is designed to be an all-purpose solution for automating the way that elements interact and for mapping the object model of an application to its user interface. Although WPF applications are still new, those that exist today use data binding much more frequently and thoroughly than their Windows Forms counterparts. In WPF, data binding is much more than an optional frill, and every professional WPF developer needs to master it.

You haven't reached the end of your data exploration yet. You still have several topics to tackle. In the following chapters, you'll build on the data binding basics you've learned here and tackle these new topics:

- *Data formatting:* You've learned how to get your data but not necessarily how to make it look good. In Chapter 20, you'll learn to format numbers and dates, and you'll go far further with styles and data templates that allow you to customize the way records are shown in a list.
- *Data views:* In every application that uses data binding, there's a data view at work. Often, you can ignore this piece of background plumbing. But if you take a closer look, you can use it to write navigation logic and apply filtering and sorting. Chapter 21 shows the way.
- *Advanced data controls*: For richer data display options, WPF gives you the ListView, TreeView, and DataGrid. All three support data binding with remarkable flexibility. Chapter 22 describes them all.