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 do using 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 has the ability to 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.

CHAPTER 20

■ ■ ■

Formatting Bound Data

In Chapter 19, you learned the essentials of WPF data binding—how to pull information out of an object and display it in a window, with little code required. Along the way, you considered how to make that information editable, how to deal with collections of data objects, and how to perform validation to catch bad edits. However, there's still a lot more to learn.

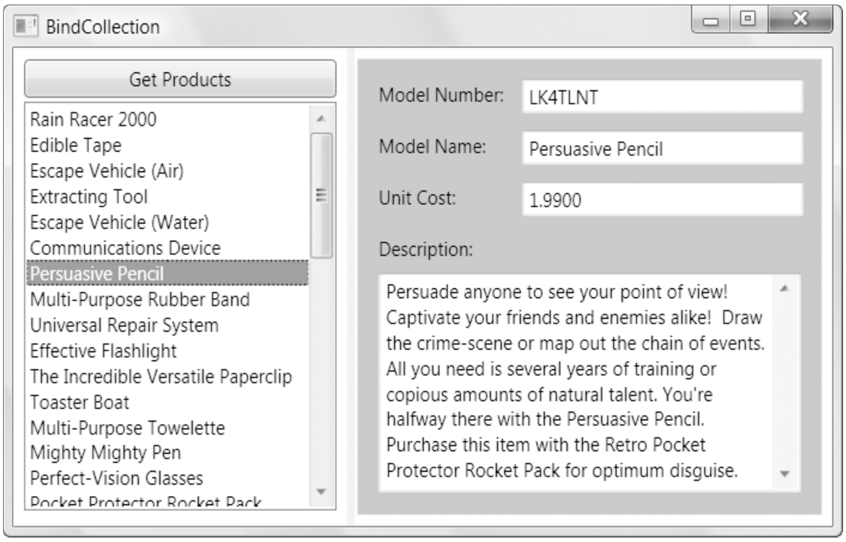
In this chapter, you'll continue your exploration by tackling several subjects that will allow you to build better bound windows. First, you'll look at data conversion, the powerful and extensible system WPF uses to examine values and convert them. As you'll discover, this process of conversion goes far beyond simple conversion, giving you the ability to apply conditional formatting and deal with images, files, and other types of specialized content.

Next, you'll look at how you can format entire lists of data. First you'll review the infrastructure that supports bound lists, starting with the base ItemsControl class. Then you'll learn how to refine the appearance of list with styles, along with triggers that apply alternating formatting and selection highlighting. Finally, you'll use the most powerful formatting tool of all, *data templates*, which let you customize the way each item is shown in an ItemsControl. Data templates are the secret to converting a basic list into a rich data presentation tool complete with custom formatting, picture content, and additional WPF controls.

■ **What's New** WPF 3.5 SP1 added two significant refinements to the data binding system. First, the new Binding.StringFormat property makes it possible to apply .NET format strings without a custom value converter (see the section "The StringFormat Property"). Next, the ItemsControl added an AlternationIndex property and an AlternationCount property that makes it possible to apply alternating row formatting without a custom style selector (see the section "Alternating Item Style").

Data Binding Redux

In most data binding scenarios, you aren't binding to a single object but to an entire collection. Figure 20-1 shows a familiar example—a form with a list of products. When the user selects a product, its details appear on the right.



Model Number: LK4TLNT

Model Name: Persuasive Pencil

Unit Cost: 1.9900

Description:

Persuade anyone to see your point of view! Captivate your friends and enemies alike! Draw the crime-scene or map out the chain of events. All you need is several years of training or copious amounts of natural talent. You're halfway there with the Persuasive Pencil. Purchase this item with the Retro Pocket Protector Rocket Pack for optimum disguise.

Figure 20-1. Browsing a collection of products

In Chapter 19, you learned to build exactly this sort of form. Here's a quick review of the basic steps:

1. First you need to create the list of items, which you can show in an ItemsControl. Set the DisplayMemberPath to indicate the property (or field) you want to show for each item in the list. This list shows the model name of each item:

   ```
   <ListBox Name="lstProducts" DisplayMemberPath="ModelName"></ListBox>
   ```

2. To fill the list with data, set the ItemsSource property to your collection (or DataTable). Typically, you'll perform this step in code when your window loads or the user clicks a button. In this example, the ItemsControl is bound to an ObservableCollection of Product objects.

   ```
   ObservableCollection<Product> products = App.StoreDB.GetProducts();
   lstProducts.ItemsSource = products;
   ```

3. To show item-specific information, add as many elements as you need, each with a binding expression that identifies the property or field you want to display. In this example, each item in the collection is a Product object. Here's an example that shows the model number of an item by binding to the Product.ModelNumber property:

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

4. The easiest way to connect the item-specific elements to the currently selected item is to wrap them in a single container. Set the DataContext property of the container to refer to the selected item in the list:

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

So far, this is all review. However, what you haven't yet considered is how to tailor the appearance of the data list and the data fields. For example, you don't yet know how to format numeric values, how to create a list that shows multiple pieces of information at once (and arranges these pieces in a pleasing way), and how to deal with nontext content, such as picture data. In this chapter, you'll cover all these tasks as you begin to build more respectable data forms.

Data Conversion

In a basic binding, the information travels from the source to the target without any change. This seems logical, but it's not always the behavior you want. Often, your data source might use a low-level representation that you don't want to display directly in your user interface. For example, you might have numeric codes you want to replace with human-readable strings, numbers that need to be cut down to size, dates that need to be displayed in a long format, and so on. If so, you need a way to convert these values into the right display form. And if you're using a two-way binding, you also need to do the converse—take user-supplied data and convert it to a representation suitable for storage in the appropriate data object.

Fortunately, WPF has two tools that can help you:

- **String formatting.** This feature allows you to convert data that's represented as text—for example, strings that contain dates and numbers—by setting the Binding.StringFormat property. It's a convenient technique that works for at least half of all formatting tasks.
- **Value converters.** This is a far more powerful (and somewhat more complicated) feature that lets you convert any type of source data into any type of object representation, which you can then pass on to the linked control.

In the following sections, you'll consider both approaches.

The StringFormat Property

String formatting is the perfect tool for formatting numbers that need to be displayed as text. For example, consider the UnitCost property from the Product class introduced in the previous chapter. UnitCost is stored as a decimal, and, as a result, when it's displayed in a text box, you'll see values like 3.9900. Not only does this display format show more decimal places than you'd probably like, it also leaves out the currency symbol. A more intuitive representation would be the currency-formatted value $3.99.

The easiest solution is to set the Binding.StringFormat property. WPF will use the format string to convert the raw text to its display value, just before it appears in the control. Just as importantly, WPF will (in most cases) use this string to perform the reverse conversion, taking any edited data and using it to update the bound property.

When setting the Binding.StringFormat property, you use standard .NET format strings, in the form {0:C}. Here, the *0* represents the first value, and the *C* refers to the format string you want to apply—which is, in this case, the standard local-specific currency format, which translates 3.99 to $3.99 on a U.S. computer. The entire expression is wrapped in curly braces.

Here's an example that applies the format string to the UnitCost field so that it's displayed as a currency value:

```
<TextBox Margin="5" Grid.Row="2" Grid.Column="1"
 Text="{Binding Path=UnitCost, StringFormat={}{0:C}}">
</TextBox>
```

You'll notice that the StringFormat value is preceded with the curly braces {}. In full, it's {}{0:C} rather than just {0:C}. The slightly unwieldy pair of braces at the beginning are required to escape the string. Otherwise, the XAML parser can be confused by the curly brace at the beginning of {0:C}.

Incidentally, the {} escape sequence is required only when the StringFormat value begins with a brace. Consider this example, which adds a literal sequence of text before each formatted value:

```
<TextBox Margin="5" Grid.Row="2" Grid.Column="1"
 Text="{Binding Path=UnitCost, StringFormat=The value is {0:C}.}">
</TextBox>
```

This expression converts a value such as 3.99 to "The value is $3.99." Because the first character in the StringFormat value is an ordinary letter, not a brace, the initial escape sequence isn't required. However, this format string works in one direction only. If the user tries to supply an edited value that includes this literal text (such as "The value is $4.25"), the update will fail. On the other hand, if the user performs an edit with the numeric characters only (4.25) or with the numeric characters and the currency symbol ($4.25), the edit will succeed, and the binding expression will convert it to the display text "The value is $4.25" and show that in the text box.

To get the results you want with the StringFormat property, you need the right format string. You can learn about all the format strings that are available in the Visual Studio help. However, Table 20-1 and Table 20-2 show some of the most common options you'll use for numeric and date values, respectively. And here's a binding expression that uses a custom date format string to format the OrderDate property:

```
<TextBlock Text="{Binding Date, StringFormat={}{0:MM/dd/yyyy}}"></TextBlock>
```

Table 20-1. *Format Strings for Numeric Data*

Type	Format String	Example
Currency	C	$1,234.50. Parentheses indicate negative values: ($1,234.50). The currency sign is locale-specific.
Scientific (Exponential)	E	1.234.50E+004.
Percentage	P	45.6%.
Fixed Decimal	F?	Depends on the number of decimal places you set. F3 formats values like 123.400. F0 formats values like 123.

Table 20-2. *Format Strings for Times and Dates*

Type	Format String	Format
Short Date	d	M/d/yyyy For example: 10/30/2008
Long Date	D	dddd, MMMM dd, yyyy For example: Wednesday, January 30, 2008
Long Date and Short Time	f	dddd, MMMM dd, yyyy HH:mm aa For example: Wednesday, January 30, 2008 10:00 AM
Long Date and Long Time	F	dddd, MMMM dd, yyyy HH:mm:ss aa For example: Wednesday, January 30, 2008 10:00:23 AM
ISO Sortable Standard	s	yyyy-MM-dd HH:mm:ss For example: 2008-01-30 10:00:23
Month and Day	M	MMMM dd For example: January 30
General	G	M/d/yyyy HH:mm:ss aa (depends on locale-specific settings) For example: 10/30/2008 10:00:23 AM

The WPF list controls also support string formatting for list items. To use it, you simply set the ItemStringFormat property of the list (which is inherited from the base ItemsControl class). Here's an example with a list of product prices:

```
<ListBox Name="lstProducts" DisplayMemberPath="UnitCost" ItemStringFormat="{0:C}">
</ListBox>
```

The formatting string is automatically passed down to the binding that grabs the text for each item.

Introducing Value Converters

The Binding.StringFormat property is created for simple, standard formatting with numbers and dates. But many data binding scenarios need a more powerful tool, called a *value converter* class.

A value converter plays a straightforward role. It's responsible for converting the source data just before it's displayed in the target and (in the case of a two-way binding) converting the new target value just before it's applied back to the source.

■ **Note** This approach to conversion is similar to the way data binding worked in the world of Windows Forms with the Format and Parse binding events. The difference is that in a Windows Forms application, you could code this logic anywhere—you simply needed to attach both events to the binding. In WPF, this logic must be encapsulated in a value converter class, which makes for easier reuse.

Value converters are an extremely useful piece of the WPF data binding puzzle. They can be used in several useful ways:

- **To format data to a string representation.** For example, you can convert a number to a currency string. This is the most obvious use of value converters, but it's certainly not the only one.
- **To create a specific type of WPF object.** For example, you could read a block of binary data and create a BitmapImage object that can be bound to an Image element.
- **To conditionally alter a property in an element based on the bound data.** For example, you might create a value converter that changes the background color of an element to highlight values in a specific range.

Formatting Strings with a Value Converter

To get a basic idea of how a value converter works, it's worth revisiting the currency-formatting example you looked at in the previous section. Although that example used the Binding.StringFormat property, you can accomplish the same thing—and more—with a value converter. For example, you could round or truncate values (changing 3.99 to 4), use number names (changing 1,000,000 into 1 million), or even add a dealer markup (multiplying 3.99 by 15%). You can also tailor the way that the reverse conversion works to change user-supplied values into the right data values in the bound object.

To create a value converter, you need to take four steps:

1. Create a class that implements IValueConverter.
2. Add the ValueConversion attribute to the class declaration, and specify the destination and target data types.
3. Implement a Convert() method that changes data from its original format to its display format.
4. Implement a ConvertBack() method that does the reverse and changes a value from display format to its native format.

Figure 20-2 shows how it works.

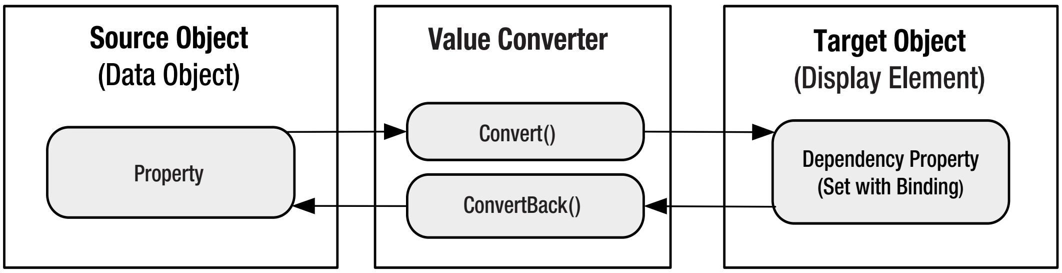


Figure 20-2. Converting bound data

In the case of the decimal-to-currency conversion, you can use the Decimal.ToString() method to get the formatted string representation you want. You simply need to specify the currency format string "C", as shown here:

```
string currencyText = decimalPrice.ToString("C");
```

This code uses the culture settings that apply to the current thread. A computer that's configured for the English (United States) region runs with a locale of en-US and displays currencies with the dollar sign ($). A computer that's configured for another local might display a different currency symbol. (This is the same way that the {0:C} format string works when applied with the Binding.StringFormat property.) If this isn't the result you want (for example, you always want the dollar sign to appear), you can specify a culture using the overload of the ToString() method shown here:

```
CultureInfo culture = new CultureInfo("en-US");
string currencyText = decimalPrice.ToString("C", culture);
```

Converting from the display format back to the number you want is a little trickier. The Parse() and TryParse() methods of the Decimal type are logical choices to do the work, but ordinarily they can't handle strings that include currency symbols. The solution is to use an overloaded version of the Parse() or TryParse() method that accepts a System.Globalization.NumberStyles value. If you supply NumberStyles.Any, you'll be able to successfully strip out the currency symbol, if it exists.

Here's the complete code for the value converter that deals with price values like the Product.UnitCost property:

```
[ValueConversion(typeof(decimal), typeof(string))]
public class PriceConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
      CultureInfo culture)
    {
        decimal price = (decimal)value;
        return price.ToString("C", culture);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
      CultureInfo culture)
    {
        string price = value.ToString(culture);
        decimal result;
        if (Decimal.TryParse(price, NumberStyles.Any, culture, out result))
        {
            return result;
        }
        return value;
    }
}
```

To put this converter into action, you need to begin by mapping your project namespace to an XML namespace prefix you can use in your markup. Here's an example that uses the namespace prefix local and assumes your value converter is in the namespace DataBinding:

```
xmlns:local="clr-namespace:DataBinding"
```

Typically, you'll add this attribute to the <Window> tag that holds all your markup.

Now, you simply need to create an instance of the PriceConverter class and assign it to the Converter property of your binding. To do this, you need the more long-winded syntax shown here:

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

In many cases, the same converter is used for multiple bindings. In this case, it doesn't make sense to create an instance of the converter for each binding. Instead, create one converter object in the Resources collection, as shown here:

```
<Window.Resources>
  <local:PriceConverter x:Key="PriceConverter"></local:PriceConverter>
</Window.Resources>
```

Then, you can point to it in your binding using a StaticResource reference, as described in Chapter 10:

```
<TextBox Margin="5" Grid.Row="2" Grid.Column="1"
 Text="{Binding Path=UnitCost, Converter={StaticResource PriceConverter}}">
</TextBox>
```

Creating Objects with a Value Converter

Value converters are indispensable when you need to bridge the gap between the way data is stored in your classes and the way it's displayed in a window. For example, imagine you have picture data stored as a byte array in a field in a database. You could convert the binary data into a System.Windows.Media.Imaging.BitmapImage object and store that as part of your data object. However, this design might not be appropriate.

For example, you might need the flexibility to create more than one object representation of your image, possibly because your data library is used in both WPF applications and Windows Forms applications (which use the System.Drawing.Bitmap class instead). In this case, it makes sense to store the raw binary data in your data object and convert it to a WPF BitmapImage object using a value converter. (To bind it to a form in a Windows Forms application, you'd use the Format and Parse events of the System.Windows.Forms.Binding class.)

■ **Tip** To convert a block of binary data into an image, you must first create a BitmapImage object and read the image data into a MemoryStream. Then, you can call the BitmapImage.BeginInit() method, set its StreamSource property to point to your MemoryStream, and call EndInit() to finish loading the image.

The Products table from the Store database doesn't include binary picture data, but it does include a ProductImage field that stores the file name of an associated product image. In this case, there's even more reason to delay creating the image object. First, the image might not be available depending on where the application's running. Second, there's no point in incurring the extra memory overhead storing the image unless it's going to be displayed.

The ProductImage field includes the file name but not the full path of an image file, which gives you the flexibility to put the image files in any suitable location. The value converter has the task of creating a URI that points to the image file based on the ProductImage field and the directory you want to use. The directory is stored using a custom property named ImageDirectory, which defaults to the current directory.

Here's the complete code for the ImagePathConverter that performs the conversion:

```
public class ImagePathConverter : IValueConverter
{
    private string imageDirectory = Directory.GetCurrentDirectory();
    public string ImageDirectory
    {
        get { return imageDirectory; }
        set { imageDirectory = value; }
    }

    public object Convert(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        string imagePath = Path.Combine(ImageDirectory,
          (string)value);
        return new BitmapImage(new Uri(imagePath));
    }

    public object ConvertBack(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        throw new NotSupportedException();
    }
}
```

To use this converter, begin by adding it to the Resources. In this example, the ImageDirectory property is not set, which means the ImagePathConverter defaults to the current application directory:

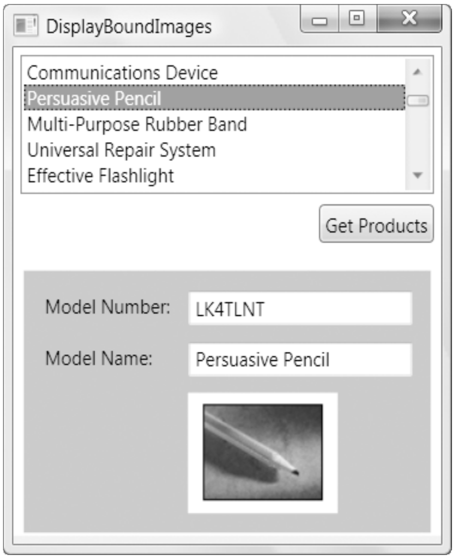
```
<Window.Resources>
  <local:ImagePathConverter x:Key="ImagePathConverter"></local:ImagePathConverter>
</Window.Resources>
```

Now it's easy to create a binding expression that uses this value converter:

```
<Image Margin="5" Grid.Row="2" Grid.Column="1" Stretch="None"
 HorizontalAlignment="Left" Source=
 "{Binding Path=ProductImagePath, Converter={StaticResource ImagePathConverter}}">
</Image>
```

This works because the Image.Source property expects an ImageSource object, and the BitmapImage class derives from ImageSource.

Figure 20-3 shows the result.



Model Number: LK4TLNT

Model Name: Persuasive Pencil

Figure 20-3. *Displaying bound images*

You might improve this example in a couple of ways. First, attempting to create a BitmapImage that points to a nonexistent file causes an exception, which you'll receive when setting the DataContext, ItemsSource, or Source property. Alternatively, you can add properties to the ImagePathConverter class that allow you to configure this behavior. For example, you might introduce a Boolean SuppressExceptions property. If set to true, you could catch exceptions in the Convert() method and then return the Binding.DoNothing value (which tells WPF to temporarily act as though no data binding is set). Or, you could add a DefaultImage property that takes a placeholder BitmapImage. The ImagePathConverter could then return the default image if an exception occurs.

You'll also notice that this converter supports only one-way conversion. That's because it's not possible to change the BitmapImage object and use that to update the image path. However, you could take an alternate approach. Rather than return a BitmapImage from the ImagePathConverter, you could simply return the fully qualified URI from the Convert() method, as shown here:

```
return new Uri(imagePath);
```

This works just as successfully, because the Image element uses a type converter to translate the Uri to the ImageSource object it really wants. If you take this approach, you could then allow the user to choose a new file path (perhaps using a TextBox that's set with the help of the OpenFileDialog class). You could then extract the file name in the ConvertBack() method and use that to update the image path that's stored in your data object.

Applying Conditional Formatting

Some of the most interesting value converters aren't designed to format data for presentation. Instead, they're intended to format some other appearance-related aspect of an element based on a data rule.

For example, imagine you want to flag high-priced items by giving them a different background color. You can easily encapsulate this logic with the following value converter:

```
public class PriceToBackgroundConverter : IValueConverter
{
    public decimal MinimumPriceToHighlight
    {
        get; set;
    }

    public Brush HighlightBrush
    {
        get; set;
    }

    public Brush DefaultBrush
    {
        get; set;
    }

    public object Convert(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        decimal price = (decimal)value;
        if (price >= MinimumPriceToHighlight)
            return HighlightBrush;
        else
            return DefaultBrush;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        throw new NotSupportedException();
    }
}
```

Once again, the value converter is carefully designed with reusability in mind. Rather than hard-coding the color highlights in the converter, they're specified in the XAML by the code that *uses* the converter:

```
<local:PriceToBackgroundConverter x:Key="PriceToBackgroundConverter"
  DefaultBrush="{x:Null}" HighlightBrush="Orange" MinimumPriceToHighlight="50">
</local:PriceToBackgroundConverter>
```

Brushes are used instead of colors so that you can create more advanced highlight effects using gradients and background images. And if you want to keep the standard, transparent background (so the background of the parent elements is used), just set the DefaultBrush or HighlightBrush property to null, as shown here.

Now all that's left is to use this converter to set the background of some element, like the Border that contains all the other elements:

```
<Border Background=
 "{Binding Path=UnitCost, Converter={StaticResource PriceToBackgroundConverter}}"
 ... >
```

Other Ways to Apply Conditional Formatting

Using a custom value converter is only one of the ways to apply conditional formatting based on your data object. You can also use data triggers in a style, style selector, and template selector, all of which are described in this chapter. Each one of these approaches has its own advantages and disadvantages.

The value converter approach works best when you need to set a single property in an element based on the bound data object. It's easy, and it's automatically synchronized. If you make changes to the bound data object, the linked property is changed immediately.

Data triggers are similarly straightforward, but they support only extremely simple logic that tests for equality. For example, a data trigger can apply formatting that applies to products in a specific category, but it can't apply formatting that kicks in when the price is greater than a specific minimum value. The key advantage of data triggers is that you can use them to apply certain types of formatting and selection effects without writing any code.

Style selectors and template selectors are the most powerful options. They allow you to change multiple properties in the target element at once and change the way items are presented in the list. However, they introduce additional complexity. Also, you need to add code that reapplies your styles and templates if the bound data changes.

Evaluating Multiple Properties

So far, you've used binding expressions to convert one piece of source data into a single, formatted result. And although you can't change the second part of this equation (the result), you *can* create a binding that evaluates or combines the information in more than one source property, with a little craftiness.