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 ➤ 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 ➤ 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/yd64qwr`.

Improving Performance in Large 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 of 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 (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" ... >
```

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.
- **Using grouping.** Grouping automatically configures the list to use pixel-based scrolling rather than item-based scrolling. (Chapter 6 explains the difference when it describes the ScrollViewer control.) When pixel-based scrolling is switched on, virtualization isn't possible—at least not in this release of WPF.
- **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 triggers 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.

Currently, 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

WPF 3.5 SP1 improved the virtualization story with *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.

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.

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. Another option is to implement the 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 or not 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 IDataErrorInfo in your data objects and by writing custom validation rules.

The DataErrorValidationRule

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.

In the Windows Forms world, developers could use the IDataErrorInfo interface (from the System.ComponentModel namespace) to avoid exceptions but still place the validation code in the data class. The IDataErrorInfo interface was originally designed to support grid-based display controls such as the DataGridView, but it also works as an all-purpose solution for reporting errors. Although IDataErrorInfo wasn't supported in the first release of WPF, it is supported in WPF 3.5.

The IDataErrorInfo interface requires two members: a string property named Error and a string indexer. The Error property provides an overall error string that describes the entire object (which could be something as simple as "Invalid Data"). The string indexer accepts a property name and returns the corresponding detailed error information. For example, if you pass "UnitCost" to the string indexer, you might receive a response such as "The UnitCost cannot be negative." The key idea here is that properties are set normally, without any fuss, and the indexer allows the user interface to check for invalid data. The error logic for the entire class is centralized in one place.

Here's a revised version of the Product class that implements IDataErrorInfo. Although you could use IDataErrorInfo to provide validation messages for a range of validation problems, this validation logic checks just one property—ModelNumber—for errors:

```
public class Product : INotifyPropertyChanged, IDataErrorInfo
{
    ...

    private string modelNumber;
    public string ModelNumber
    {
        get { return modelNumber; }
        set {
            modelNumber = value;
            OnPropertyChanged(new PropertyChangedEventArgs("ModelNumber"));
        }
    }

    // Error handling takes place here.
    public string this[string propertyName]
    {
        get
        {
            if (propertyName == "ModelNumber")
            {
                bool valid = true;
                foreach (char c in ModelNumber)
                {
                    if (!Char.IsLetterOrDigit(c))
                    {
                        valid = false;
                        break;
                    }
                }
                if (!valid)
                  return "The ModelNumber can only contain letters and numbers.";
            }
            return null;
        }
    }

    // WPF doesn't use this property.
    public string Error
    {
        get { return null; }
    }
}
```

To tell WPF to use the IDataErrorInfo interface and use it to check for errors when a property is modified, you must add the DataErrorValidationRule to the collection of Binding.ValidationRules, as shown here:

```
<TextBox Margin="5" Grid.Column="1">
  <TextBox.Text>
    <Binding Path="ModelNumber">
      <Binding.ValidationRules>
        <DataErrorValidationRule></DataErrorValidationRule>
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

Incidentally, you can combine both approaches by creating a data object that throws exceptions for some types of errors and uses IDataErrorInfo to report others. You just need to make sure you use both the ExceptionValidationRule and the DataErrorValidationRule.

■ **Tip** .NET provides two shortcuts. Rather than adding the ExceptionValidationRule to the binding, you can set the Binding.ValidatesOnExceptions property to true. Rather than adding the DataErrorValidationRule, you can set the Binding.ValidatesOnDataErrors property to true.

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 make 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 make 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 Different Error 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.

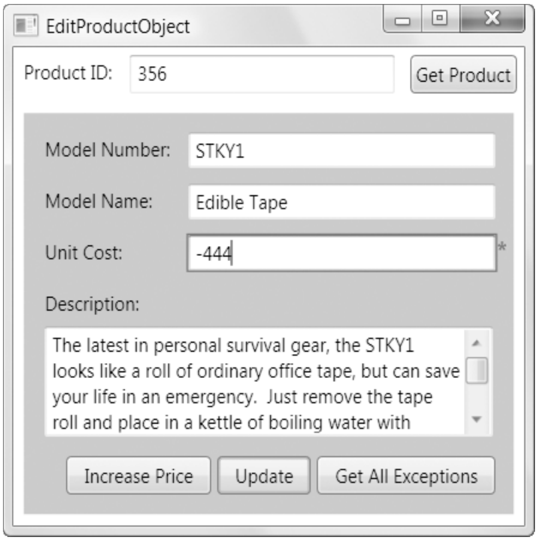
Error templates use the *adorner layer*, which is a drawing layer that exists just above ordinary window content. Using the adorner layer, you can add a visual embellishment to indicate an error without replacing the control template of the control underneath or changing the layout in your window. The 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're able to 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.)



The latest in personal survival gear, the STKY1 looks like a roll of ordinary office tape, but can save your life in an emergency. Just remove the tape roll and place in a kettle of boiling water with

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:

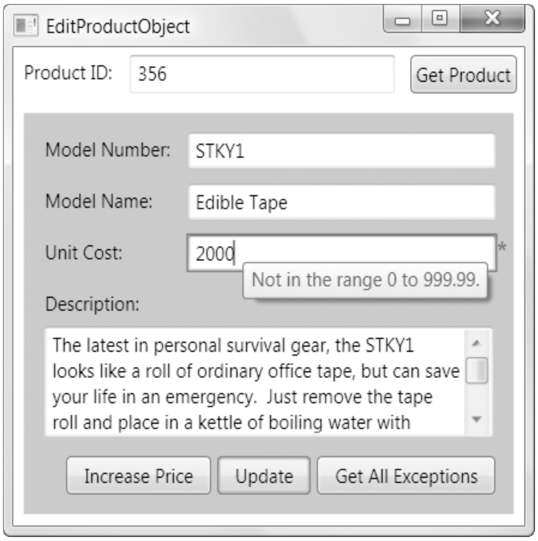
```
ToolTip="{Binding ElementName=adornerPlaceholder,
          Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"
```

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.



The latest in personal survival gear, the STKY1 looks like a roll of ordinary office tape, but can save your life in an emergency. Just remove the tape roll and place in a kettle of boiling water with

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");
```