The first trick is to replace your Binding object with a MultiBinding. Then, you define the arrangement of bound properties using the MultiBinding.StringFormat property. Here's an example that turns Joe and Smith into "Smith, Joe" and displays the result in a TextBlock:

```
<TextBlock>
  <TextBlock.Text>
    <MultiBinding StringFormat="{1}, {0}">
      <Binding Path="FirstName"></Binding>
      <Binding Path="LastName"></Binding>
    </MultiBinding>
  </TextBlock.Text>
</TextBlock>
```

You'll notice in this example that the two fields are used as is in the StringFormat property. Alternatively, you can use format strings to change it. For example, if you were combining a text value and a currency value with a MultiBinding, you might set StringFormat to "{0} costs {1:C}."

If you want to do anything more ambitious with the two source fields than simply stitching them together, you need the help of a value converter. You can use this technique to perform calculations (such as multiplying UnitPrice by UnitsInStock) or apply formatting that takes several details into consideration (such as highlighting all high-priced products in a specific category). However, your value converter must implement IMultiValueConverter rather than IValueConverter.

Here's an example where a MultiBinding uses the UnitCost and UnitsInStock properties from the source object and combines them using a value converter:

```
<TextBlock>Total Stock Value: </TextBlock>
<TextBox>
  <TextBox.Text>
    <MultiBinding Converter="{StaticResource ValueInStockConverter}">
      <Binding Path="UnitCost"></Binding>
      <Binding Path="UnitsInStock"></Binding>
    </MultiBinding>
  </TextBox.Text>
</TextBox>
```

The IMultiValueConverter interface defines similar Convert() and ConvertBack() methods as the IValueConverter interface. The main difference is that you're provided with an array of values rather than a single value. These values are placed in the same order that they're defined in your markup. Thus, in the previous example, you can expect UnitCost to appear first, followed by UnitsInStock.

Here's the code for the ValueInStockConverter:

```
public class ValueInStockConverter : IMultiValueConverter
{
    public object Convert(object[] values, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        // Return the total value of all the items in stock.
        decimal unitCost = (decimal)values[0];
        int unitsInStock = (int)values[1];
        return unitCost * unitsInStock;
    }
```

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

List Controls

String formatting and value converters are all you need to apply flexible formatting to individual bound values. But bound lists need a bit more. Fortunately, WPF provides no shortage of formatting choices. Most of these built into the base ItemsControl class from which all list controls derive, so this is where your list-formatting exploration should start.

As you know, the ItemsControl class defines the basic functionality for controls that wrap a list of items. Those items can be entries in a list, nodes in a tree, commands in a menu, buttons in a toolbar, and so on. Figure 20-4 provides an at-a-glance overview of all the ItemsControl classes in WPF.

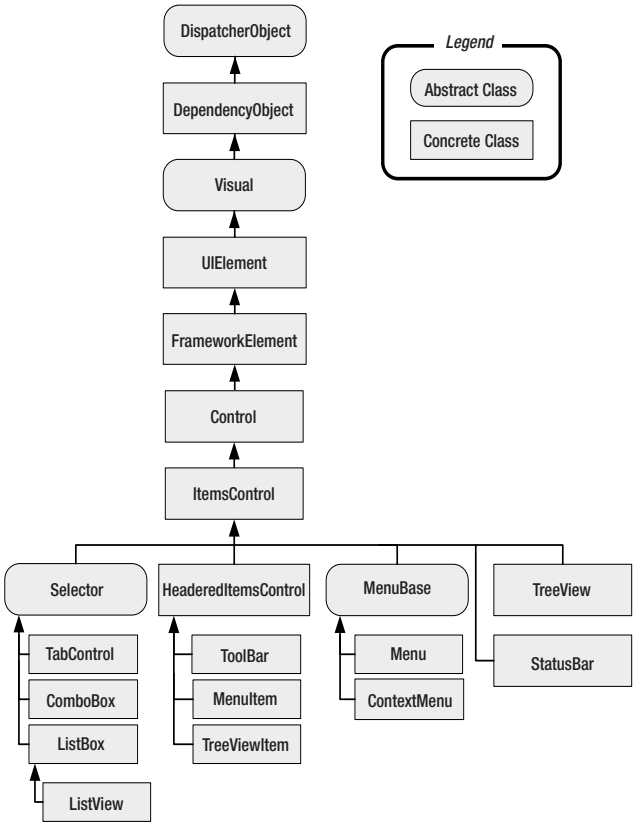

Figure 20-4. *Classes that derive from ItemsControl*

■ **Note** You'll notice that some item wrappers appear in the class hierarchy of classes that derive from ItemsControl. For example, not only will you see the expected Menu and TreeView classes, but you'll also see MenuItem and TreeViewItem. That's because these classes have the ability to contain their own collection of items—that's what gives trees and menus their nested, hierarchical structure. On the other hand, you won't find ComboBoxItem or ListBoxItem in this list, because they don't need to hold a child collection of items and so don't derive from ItemsControl.

The ItemsControl defines the properties that support data binding and two key formatting features: styles and data templates. You'll explore both features in the following sections, and Table 20-3 gives you a quick overview of the properties that support them. The table is loosely organized from more basic features to more advanced ones, and it also mirrors the order you'll explore them in this chapter.

Table 20-3. *Formatting-Related Properties of the ItemsControl Class*

Name	Description
ItemsSource	The bound data source (the collection or DataView that you want to display in the list).
DisplayMemberPath	The property that you want to display for each data item. For a more sophisticated representation or to use a combination of properties, use the ItemTemplate instead.
ItemStringFormat	A .NET format string that, if set, will be used to format the text for each item. Usually, this technique is used to convert numeric or date values into a suitable display representation, exactly as the Binding.StringFormat property does.
ItemContainerStyle	A style that allows you to set the properties of the container that wraps each item. The container depends on the type of list (for example, it's ListBoxItem for the ListBox class and ComboBoxItem for the ComboBox class). These wrapper objects are created automatically as the list is filled.
ItemContainerStyleSelector	A StyleSelector that uses code to choose a style for the wrapper of each item in the list. This allows you to give different styles to different items in the list. You must create a custom StyleSelector yourself.
AlternationCount	The number of alternating sets in your data. For example, an AlternationCount of 2 alternates between two different row styles, an AlternationCount of 3 alternates between three different row styles, and so on.

Name	Description
ItemTemplate	A template that extracts the appropriate data out of your bound object and arranges it into the appropriate combination of controls.
ItemTemplateSelector	A DataTemplateSelector that uses code to choose a template for each item in the list. This allows you to give different templates to different items. You must create a custom DataTemplateSelector class yourself.
ItemsPanel	Defines the panel that's created to hold the items of the list. All the item wrappers are added to this container. Usually, a VirtualizingStackPanel is used with a vertical (top-to-bottom) orientation.
GroupStyle	If you're using grouping, this is a style that defines how each group should be formatted. When using grouping, the item wrappers (ListBoxItem, ComboBoxItem, and so on) are added in GroupItem wrappers that represent each group, and these groups are then added to the list. Grouping is demonstrated in Chapter 21.
GroupStyleSelector	A StyleSelector that uses code to choose a style for each group. This allows you to give different styles to different groups. You must create a custom StyleSelector yourself.

The next rung in the ItemsControl inheritance hierarchy is the Selector class, which adds a straightforward set of properties for determining (and setting) a selected item. Not all ItemsControl classes support selection. For example, selection doesn't have any meaning for the ToolBar or Menu, so these classes derive from ItemsControl but not Selector.

The properties that the Selector class adds include SelectedItem (the selected data object), SelectedIndex (the position of the selected item), and SelectedValue (the "value" property of the selected data object, which you designate by setting SelectedValuePath). Notice that the Selector class doesn't provide support for multiple selection—that's added to the ListBox through its SelectionMode and SelectedItems properties (which is essentially all the ListBox class adds to this model).

List Styles

For the rest of this chapter, you'll be concentrating on two features that are provided by all the WPF list controls: styles and data templates.

Out of these two tools, styles are simpler (and less powerful). In many cases, they allow you to add a bit of formatting polish. In the following sections, you'll see how styles let you format list items, apply alternating-row formatting, and apply conditional formatting according to the criteria you specify.

The ItemContainerStyle

In Chapter 11, you learned how styles allow you to reuse formatting with similar elements in different places. Styles play much the same role with lists—they allow you to apply a set of formatting characteristics to each of the individual items.

This is important, because WPF's data binding system generates list item objects automatically. As a result, it's not so easy to apply the formatting you want to individual items. The solution is the ItemContainerStyle property. If the ItemContainerStyle is set, the list control will pass it down to each of its items, as the item is created. In the case of a ListBox control, each item is represented by a ListBoxItem object. (In a ComboBox, it's ComboBoxItem, and so on.) Thus, any style you apply with the ListBox.ItemContainerStyle property is used to set the properties of each ListBoxItem object.

Here's one of the simplest possible effects that you can achieve with the ListBoxItem. It applies a blue-gray background to each item. To make sure the individual items stand apart from each other (rather than having their backgrounds merge together, the style also adds some margin space:

```
<ListBox Name="lstProducts" Margin="5" DisplayMemberPath="ModelName">
  <ListBox.ItemContainerStyle>
    <Style>
      <Setter Property="ListBoxItem.Background" Value="LightSteelBlue" />
      <Setter Property="ListBoxItem.Margin" Value="5" />
      <Setter Property="ListBoxItem.Padding" Value="5" />
    </Style>
  </ListBox.ItemContainerStyle>
</ListBox>
```

On its own, this isn't terribly interesting. However, the style becomes a bit more polished with the addition of triggers. In the following example, a property triggers changes the background color and adds a solid border when the ListBoxItem.IsSelected property becomes true. Figure 20-5 shows the result.

```
<ListBox Name="lstProducts" Margin="5" DisplayMemberPath="ModelName">
  <ListBox.ItemContainerStyle>
    <Style TargetType="{x:Type ListBoxItem}">
      <Setter Property="Background" Value="LightSteelBlue" />
      <Setter Property="Margin" Value="5" />
      <Setter Property="Padding" Value="5" />

      <Style.Triggers>
        <Trigger Property="IsSelected" Value="True">
          <Setter Property="Background" Value="DarkRed" />
          <Setter Property="Foreground" Value="White" />
          <Setter Property="BorderBrush" Value="Black" />
          <Setter Property="BorderThickness" Value="1" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </ListBox.ItemContainerStyle>
</ListBox>
```

For cleaner markup, this style uses the Style.TargetType property so that it can set properties without including the class name in each setter.

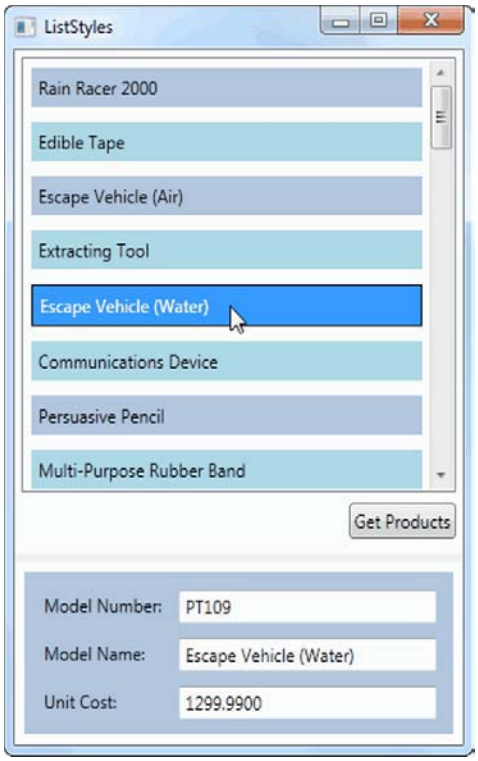

Figure 20-5. *Use style triggers to change the highlighting for the selected item.*

This use of triggers is particularly handy, because the ListBox doesn't provide any other way to apply targeted formatting to the selected item. In other words, if you don't use a style, you're stuck with the standard blue highlighting.

Later in this chapter, when you use data templates to completely revamp data lists, you'll once again rely on the ItemContainerStyle to change the selected item effect.

A ListBox with Check Boxes or Radio Buttons

The ItemContainerStyle is also important if you want to reach deep into a list control and change the control template that its items use. For example, you can use this technique to make every ListBoxItem display a radio button or a check box next to its item text.

Figure 20-6 and Figure 20-7 show two examples—one with a list filled with RadioButton elements (only one of which can be chosen at a time) and one with a list of CheckBox elements. The two solutions are similar, but the list with radio buttons is slightly easier to create.

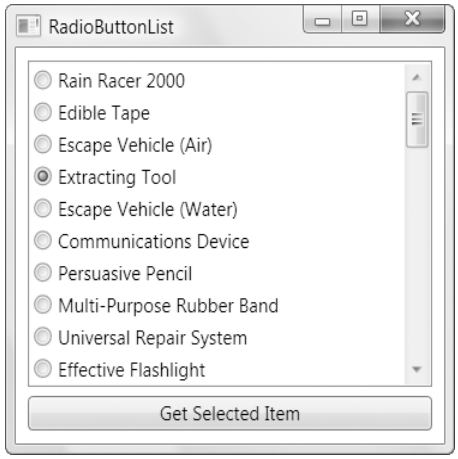

Figure 20-6. *A radio button list using a template*

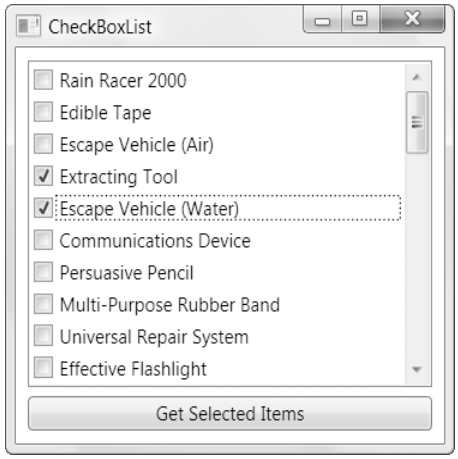

Figure 20-7. *A check box list using a template*

■ **Note** At first glance, using templates to change the ListBoxItem might seem like more work than it's worth. After all, it's easy enough to solve the problem by composition. All you need to do is fill a ScrollViewer with a series of CheckBox objects. However, this implementation doesn't provide the same programming model. There's no easy way to iterate through all the check boxes, and, more important, there's no way to use this implementation with data binding.

The basic technique in this example is to change the control template used as the container for each list item. You don't want to modify the ListBox.Template property, because this provides the template for the ListBox. Instead, you need to modify the ListBoxItem.Template property. Here's the template you need to wrap each item in a RadioButton element:

```
<ControlTemplate TargetType="{x:Type ListBoxItem}">
  <RadioButton Focusable="False" IsChecked="{Binding Path=IsSelected,
RelativeSource={RelativeSource TemplatedParent},Mode=TwoWay}">
    <ContentPresenter></ContentPresenter>
  </RadioButton>
</ControlTemplate>
```

This works because a RadioButton is a content control and can contain any content. Although you could use a binding expression to get the content, it's far more flexible to use the ContentPresenter element, as shown here. The ContentPresenter grabs whatever would ordinarily appear in the item, which might be property text (if you're using the ListBox.DisplayMemberPath property) or a more complex representation of the data (if you're using the ListBox.ItemTemplate property).

The real trick is the binding expression for the RadioButton.IsChecked property. This expression retrieves the value of the ListBoxItem.IsSelected property using the Binding.RelativeSource property. That way, when you click a RadioButton to select it, the corresponding ListBoxItem is marked as selected. At the same time, all other items are deselected. This binding expression also works in the other direction, which means you can set the selection in code and the right RadioButton will be filled in.

To complete this template, you need to set the RadioButton.Focusable property to false. Otherwise, you'll be able to tab to the currently selected ListBoxItem (which is focusable) and then into the RadioButton itself, which doesn't make much sense.

To set the ListBoxItem.Template property, you need a style rule that can dig down to the right level. Thanks to the ItemContainerStyle property, this part is easy:

```
<Window.Resources>
  <Style x:Key="RadioButtonListStyle" TargetType="{x:Type ListBox}">
    <Setter Property="ItemContainerStyle">
      <Setter.Value>
        <Style TargetType="{x:Type ListBoxItem}" >
          <Setter Property="Margin" Value="2" />
          <Setter Property="Template">
            <Setter.Value>
              <ControlTemplate TargetType="{x:Type ListBoxItem}">
                <RadioButton Focusable="False"
                 IsChecked="{Binding Path=IsSelected, Mode=TwoWay,
                             RelativeSource={RelativeSource TemplatedParent} }">
                  <ContentPresenter></ContentPresenter>
                </RadioButton>
              </ControlTemplate>
            </Setter.Value>
          </Setter>
        </Style>
      </Setter.Value>
    </Setter>
  </Style>
</Window.Resources>
```

Although you could set the ListBox.ItemContainerStyle property directly, this example factors it out one more level. The style that sets the ListBoxItem.Control template is wrapped in another style that applies this style to the ListBox.ItemContainerStyle property. This makes the template reusable, allowing you to connect it to as many ListBox objects as you want.

```
<ListBox Style="{StaticResource RadioButtonListStyle}" Name="lstProducts"
 DisplayMemberPath="ModelName">
```

You could also use the same style to adjust other properties of the ListBox.

Creating a ListBox that shows check boxes is just as easy. In fact, you have to make only two changes. First, replace the RadioButton element with an identical CheckBox element. Then, change the ListBox.SelectionMode property to allow simple multiple selection. Now, the user can check as many or as few items as desired.

Here's the style rule that transforms an ordinary ListBox into a list of check boxes:

```
<Style x:Key="CheckBoxListStyle" TargetType="{x:Type ListBox}">
  <Setter Property="SelectionMode" Value="Multiple"></Setter>
  <Setter Property="ItemContainerStyle">
    <Setter.Value>
      <Style TargetType="{x:Type ListBoxItem}" >
        <Setter Property="Margin" Value="2" />
        <Setter Property="Template">
          <Setter.Value>
            <ControlTemplate TargetType="{x:Type ListBoxItem}">
              <CheckBox Focusable="False"
               IsChecked="{Binding Path=IsSelected, Mode=TwoWay,
                           RelativeSource={RelativeSource TemplatedParent} }">
                <ContentPresenter></ContentPresenter>
              </CheckBox>
            </ControlTemplate>
          </Setter.Value>
        </Setter>
      </Style>
    </Setter.Value>
  </Setter>
</Style>
```

Alternating Item Style

One common way to format a list is to use alternating row formatting—in other words, a set of formatting characteristics that distinguishes every second item in a list. Often, alternating rows are given subtly different backgrounds colors so that the rows are clearly separated, as shown in Figure 20-8.

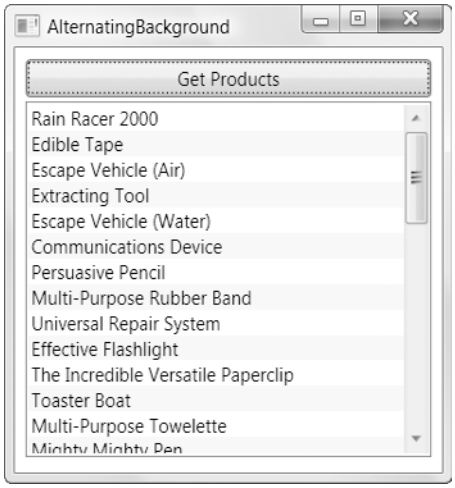

Figure 20-8. *Alternating-row highlighting*

WPF has built-in support for alternating items through two properties: AlternationCount and AlternationIndex.

AlternationCount is the number of items that form a sequence, after which the style alternates. By default, AlternationCount is set to 0, and alternating formatting isn't used. If you set AlternationCount to 1, the list will alternate after every item, which allows you to apply the even-odd formatting pattern shown in Figure 20-8.

Every ListBoxItem is given an AlternationIndex, which allows you to determine how it's numbered in the sequence of alternating items. Assuming you've set AlternationCount to 2, the first ListBoxItem gets an AlternationIndex of 0, the second gets an AlternationIndex of 1, the third gets an AlternationIndex of 0, the fourth gets an AlternationIndex of 1, and so on. The trick is to use a trigger in your ItemContainerStyle that checks the AlternationIndex value and varies the formatting accordingly.

For example, the ListBox control shown here gives alternate items a slightly different background color (unless the item is selected, in which case the higher-priority trigger for ListBoxItem.IsSelected wins out):

```
<ListBox Name="lstProducts" Margin="5" DisplayMemberPath="ModelName"
 AlternationCount="2">
  <ListBox.ItemContainerStyle>
    <Style TargetType="{x:Type ListBoxItem}">
      <Setter Property="Background" Value="LightSteelBlue" />
      <Setter Property="Margin" Value="5" />
      <Setter Property="Padding" Value="5" />
        <Style.Triggers>
          <Trigger Property="ItemsControl.AlternationIndex" Value="1">
            <Setter Property="Background" Value="LightBlue" />
          </Trigger>
```

```
          <Trigger Property="IsSelected" Value="True">
            <Setter Property="Background" Value="DarkRed" />
            <Setter Property="Foreground" Value="White" />
            <Setter Property="BorderBrush" Value="Black" />
            <Setter Property="BorderThickness" Value="1" />
          </Trigger>
        </Style.Triggers>
      </Style>
    </ListBox.ItemContainerStyle>
</ListBox>
```

You'll notice that AlternationIndex is an attached property that's defined by the ListBox class (or, technically, the ItemsControl class that it derives from). It's not defined in the ListBoxItem class, so when you use it in a style trigger, you need to specify the class name.

Interestingly, alternating items don't need to be every second item. Instead, you can create more complex alternating formatting that alternates in a sequence of three or more. For example, to use three groups, set AlternationCount to 3, and write triggers for any of the three possible AlternationIndex values (0, 1, or 2). In the list, items 1, 4, 7, 10, and so on, will have an AlternationIndex of 0. Items 2, 5, 8, 11, and so on, get an AlternationIndex of 1. And finally, items 3, 6, 9, 12, and so on, get an AltenationIndex of 2.

Style Selectors

You've now seen how to vary the style based on the selection state of the item or its position in the list. However, you might want to use a number of other conditions—criteria that depend on your data rather than the ListBoxItem container that holds it.

To deal with this situation, you need a way to give different items completely different styles. Unfortunately, there's no way to do this declaratively. Instead, you need to build a specialized class that derives from StyleSelector. This class has the responsibility of examining each data item and choosing the appropriate style. This work is performed in the SelectStyle() method, which you must override.

Here's a rudimentary selector that chooses between two styles:

```
public class ProductByCategoryStyleSelector : StyleSelector
{
    public override Style SelectStyle(object item, DependencyObject container)
    {
        Product product = (Product)item;
        Window window = Application.Current.MainWindow;

        if (product.CategoryName == "Travel")
        {
            return (Style)window.FindResource("TravelProductStyle");
        }
        else
        {
            return (Style)window.FindResource("DefaultProductStyle");
        }
    }
}
```

In this example, products that are in the Travel category get one style, while all other products get another. In this example, both the styles you want to use must be defined in the Resources collection of the window, with the key names TravelProductStyle and DefaultProductStyle.

This style selector works, but it's not perfect. One problem is that your code depends on details that are in the markup, which means there's a dependency that isn't enforced at compile time and could easily be disrupted (for example, if you give your styles the wrong resource keys). The other problem is that this style selector hard-codes the value it's looking for (in this case, the category name), which limits reuse.

A better idea is to create a style selector that uses one or more properties to allow you to specify some of these details, such as the criteria you're using to evaluate your data items and the styles you want to use. The following style selector is still quite simple but extremely flexible. It's able to examine any data object, look for a given property, and compare that property against another value to choose between two styles. The property, property value, and styles are all specified as properties. The SelectStyle() method uses reflection to find the right property in a manner similar to the way data bindings work when digging out bound values.

Here's the complete code:

```
public class SingleCriteriaHighlightStyleSelector : StyleSelector
{
    public Style DefaultStyle
    {
        get; set;
    }

    public Style HighlightStyle
    {
        get; set;
    }

    public string PropertyToEvaluate
    {
        get; set;
    }

    public string PropertyValueToHighlight
    {
        get; set;
    }

    public override Style SelectStyle(object item,
      DependencyObject container)
    {
        Product product = (Product)item;

        // Use reflection to get the property to check.
        Type type = product.GetType();
        PropertyInfo property = type.GetProperty(PropertyToEvaluate);
```

```
        // Decide if this product should be highlighted
        // based on the property value.
        if (property.GetValue(product, null).ToString() == PropertyValueToHighlight)
        {
            return HighlightStyle;
        }
        else
        {
            return DefaultStyle;
        }
    }
}
```

To make this work, you'll need to create the two styles you want to use, and you'll need to create and initialize an instance of the SingleCriteriaHighlightStyleSelector.

Here are two similar styles, which are distinguished only by the background color and the use of bold formatting:

```
<Window.Resources>
  <Style x:Key="DefaultStyle" TargetType="{x:Type ListBoxItem}">
    <Setter Property="Background" Value="LightYellow" />
    <Setter Property="Padding" Value="2" />
  </Style>

  <Style x:Key="HighlightStyle" TargetType="{x:Type ListBoxItem}">
    <Setter Property="Background" Value="LightSteelBlue" />
    <Setter Property="FontWeight" Value="Bold" />
    <Setter Property="Padding" Value="2" />
  </Style>
</Window.Resources>
```

When you create the SingleCriteriaHighlightStyleSelector, you point it to these two styles. You can also create the SingleCriteriaHighlightStyleSelector as a resource (which is useful if you want to reuse it in more than one place), or you can define it inline in your list control, as in this example:

```
<ListBox Name="lstProducts" HorizontalContentAlignment="Stretch">
  <ListBox.ItemContainerStyleSelector>
    <local:SingleCriteriaHighlightStyleSelector
      DefaultStyle="{StaticResource DefaultStyle}"
      HighlightStyle="{StaticResource HighlightStyle}"
      PropertyToEvaluate="CategoryName"
      PropertyValueToHighlight="Travel"
    >
    </local:SingleCriteriaHighlightStyleSelector>
  </ListBox.ItemContainerStyleSelector>
</ListBox>
```

Here, the SingleCriteriaHighlightStyleSelector looks for a Category property in the bound data item and uses the HighlightStyle if it contains the text *Travel*. Figure 20-9 shows the result.

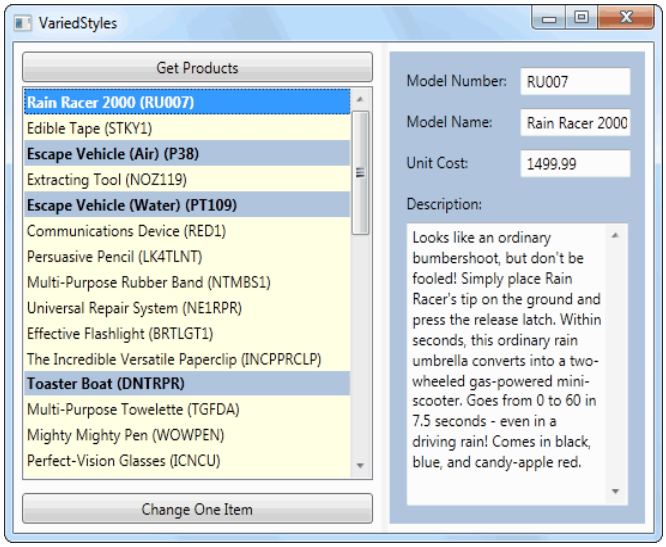

Figure 20-9. *A list with two item styles*

The style selection process is performed once, when you first bind the list. This is a problem if you're displaying editable data and it's possible for an edit to move the data item from one style category to another. In this situation, you need to force WPF to reapply the styles, and there's no graceful way to do it. The brute-force approach is to remove the style selector by setting the ItemContainerStyleSelector property to null and then to reassign it:

```
StyleSelector selector = lstProducts.ItemContainerStyleSelector;
lstProducts.ItemContainerStyleSelector = null;
lstProducts.ItemContainerStyleSelector = selector;
```

You may choose to run this code automatically in response to certain changes by handling events such as PropertyChanged (which is raised by all classes that implement INotifyPropertyChanged, including Product), DataTable.RowChanged (if you're using the ADO.NET data objects), and, more generically, Binding.SourceUpdated (which fires only when Binding.NotifyOnSourceUpdated is true). When you reassign the style selector, WPF examines and updates every item in the list—a process that's quick for small or medium-size lists.

Data Templates

Styles give you some basic formatting abilities, but they don't address the most significant limitation of the lists you've seen so far: no matter how you tweak the ListBoxItem, it's only a ListBoxItem, not a more capable combination of elements. And because each ListBoxItem supports just a single bound field (as set through the DisplayMemberPath property), there's no room to make a rich list that incorporates multiple fields or images.

However, WPF does have another tool that can break out of this rather limiting box, allowing you to use a combination of properties from the bound object and lay them out in a specific way or to display a visual representation that's more sophisticated than a simple string. That tool is the data template.

A *data template* is a chunk of XAML markup that defines how a bound data object should be displayed. Two types of controls support data templates:

- Content controls support data templates through the ContentTemplate property. The content template is used to display whatever you've placed in the Content property.
- List controls (controls that derive from ItemsControl) support data templates through the ItemTemplate property. This template is used to display each item from the collection (or each row from a DataTable) that you've supplied as the ItemsSource.

The list-based template feature is actually based on content control templates. That's because each item in a list is wrapped by a content control, such as ListBoxItem for the ListBox, ComboBoxItem for the ComboBox, and so on. Whatever template you specify for the ItemTemplate property of the list is used as the ContentTemplate of each item in the list.

So, what can you put inside a data template? It's actually quite simple. A data template is an ordinary block of XAML markup. Like any other block of XAML markup, the template can include any combination of elements. It should also include one or more data binding expressions that pull out the information that you want to display. (After all, if you don't include any data binding expressions, each item in the list will appear the same, which isn't very helpful.)

The best way to see how a data template works is to start with a basic list that doesn't use them. For example, consider this list box, which was shown previously:

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

You can get the same effect with this list box that uses a data template:

```
<ListBox Name="lstProducts">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding Path=ModelName}"></TextBlock>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

When the list is bound to the collection of products (by setting the ItemsSource property), a single ListBoxItem is created for each Product. The ListBoxItem.Content property is set to the appropriate Product object, and the ListBoxItem.ContentTemplate is set to the data template shown earlier, which extracts the value from the Product.ModelName property and displays it in a TextBlock.

So far, the results are underwhelming. But now that you've switched to a data template, there's no limit to how you can creatively present your data. Here's an example that wraps each item in a rounded border, shows two pieces of information, and uses bold formatting to highlight the model number:

```
<ListBox Name="lstProducts" HorizontalContentAlignment="Stretch">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Border Margin="5" BorderThickness="1" BorderBrush="SteelBlue"
       CornerRadius="4">
```

```
        <Grid Margin="3">
          <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
          </Grid.RowDefinitions>
          <TextBlock FontWeight="Bold"
           Text="{Binding Path=ModelNumber}"></TextBlock>
          <TextBlock Grid.Row="1"
           Text="{Binding Path=ModelName}"></TextBlock>
        </Grid>
      </Border>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

When this list is bound, a separate Border object is created for each product. Inside the Border element is a Grid with two pieces of information, as shown in Figure 20-10.

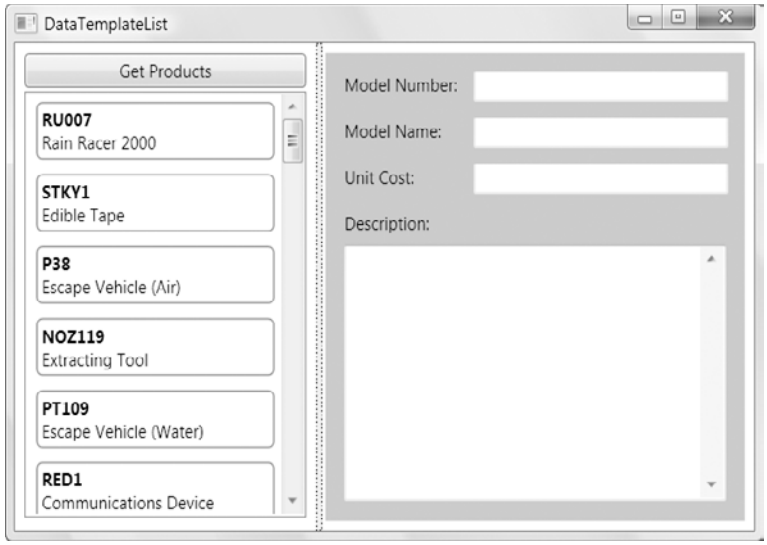

Figure 20-10. *A list that uses a data template*

■ **Tip** When using Grid objects to lay out individual items in a list, you may want to use the SharedSizeGroup property described in Chapter 3. You can apply the SharedSizeGroup property (with a descriptive group name) to individual rows or columns to ensure that those rows and columns are made the same size for every item. Chapter 22 includes an example that uses this approach to build a rich view for the ListView that combines text and image content.

Separating and Reusing Templates

Like styles, templates are often declared as a window or application resource rather than defined in the list where you use them. This separation is often clearer, especially if you use long, complex templates or multiple templates in the same control (as described in the next section). It also gives you the ability to reuse your templates in more than one list or content control if you want to present your data the same way in different places in your user interface.

To make this work, all you need to do is to define your data template in a resources collection and give it a key name. Here's an example that extracts the template shown in the previous example:

```
<Window.Resources>
  <DataTemplate x:Key="ProductDataTemplate">
    <Border Margin="5" BorderThickness="1" BorderBrush="SteelBlue"
     CornerRadius="4">
      <Grid Margin="3">
        <Grid.RowDefinitions>
          <RowDefinition></RowDefinition>
          <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <TextBlock FontWeight="Bold"
         Text="{Binding Path=ModelNumber}"></TextBlock>
        <TextBlock Grid.Row="1"
         Text="{Binding Path=ModelName}"></TextBlock>
      </Grid>
    </Border>
  </DataTemplate>
</Window.Resources>
```

Now you can use your data template using a StaticResource reference:

```
<ListBox Name="lstProducts" HorizontalContentAlignment="Stretch"
 ItemTemplate="{StaticResource ProductDataTemplate}"></ListBox>
```

You can use another interesting trick if you want to reuse the same data template in different types of controls automatically. You can set the DataTemplate.DataType property to identify the type of bound data for which your template should be used. For example, you could alter the previous example by removing the key and specifying that this template is intended for bound Product objects, no matter where they appear:

```
<Window.Resources>
  <DataTemplate DataType="{x:Type local:Product}">
  </DataTemplate>
</Window.Resources>
```

This assumes you've defined an XML namespace prefix named *local* and mapped it your project namespace.

Now this template will be used with any list or content control in this window that's bound to Product objects. You don't need to specify the ItemTemplate setting.

■ **Note** Data templates don't require data binding. In other words, you don't need to use the ItemsSource property to fill a template list. In the previous examples, you're free to add Product objects declaratively (in your XAML markup) or programmatically (by calling the ListBox.Items.Add() method). In both cases, the data template works in the same way.

More Advanced Templates

Data templates can be remarkably self-sufficient. Along with basic elements such as the TextBlock and data binding expressions, they can also use more sophisticated controls, attach event handlers, convert data to different representations, use animations, and so on.

It's worth considering a couple of quick examples that show how powerful data templates are. First, you can use value converter objects in your data binding to convert your data to a more useful representation. The following example uses the ImagePathConverter demonstrated earlier to show the image for each product in the list:

```
<Window.Resources>
  <local:ImagePathConverter x:Key="ImagePathConverter"></local:ImagePathConverter>
  <DataTemplate x:Key="ProductTemplate">
    <Border Margin="5" BorderThickness="1" BorderBrush="SteelBlue"
     CornerRadius="4">
      <Grid Margin="3">
        <Grid.RowDefinitions>
          <RowDefinition></RowDefinition>
          <RowDefinition></RowDefinition>
          <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <TextBlock FontWeight="Bold" Text="{Binding Path=ModelNumber}"></TextBlock>
        <TextBlock Grid.Row="1" Text="{Binding Path=ModelName}"></TextBlock>
        <Image Grid.Row="2" Grid.RowSpan="2" Source=
"{Binding Path=ProductImagePath, Converter={StaticResource ImagePathConverter}}">
        </Image>
      </Grid>
    </Border>
  </DataTemplate>
</Window.Resources>
```

Although this markup doesn't involve anything exotic, the result is a much more interesting list (see Figure 20-11).

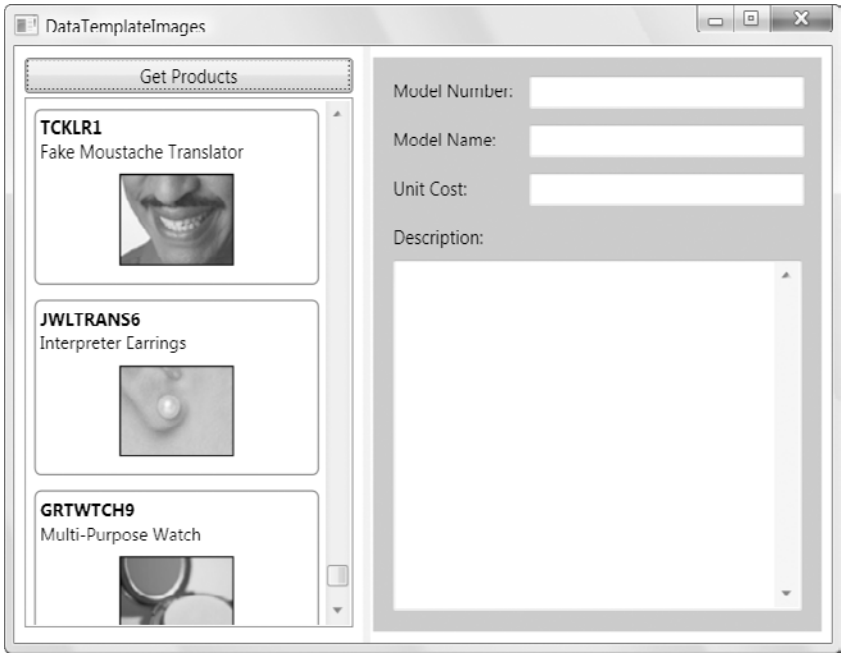

Figure 20-11. *A list with image content*

Another useful technique is to place controls directly inside a template. For example, Figure 20-12 shows a list of categories. Next to each category is a View button that you can use to launch another window with just the matching products in that category.

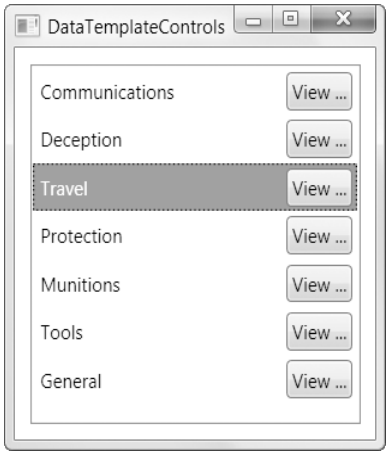

Figure 20-12. *A list with button controls*

The trick in this example is handling the button clicks. Obviously, all of the buttons will be linked to the same event handler, which you define inside the template. However, you need to determine which item was clicked from the list. One solution is to store some extra identifying information in the Tag property of the button, as shown here:

```
<DataTemplate>
  <Grid Margin="3">
    <Grid.ColumnDefinitions>
      <ColumnDefinition></ColumnDefinition>
      <ColumnDefinition Width="Auto"></ColumnDefinition>
    </Grid.ColumnDefinitions>

    <TextBlock Text="{Binding Path=CategoryName}"></TextBlock>
    <Button Grid.Column="2" HorizontalAlignment="Right" Padding="2"
      Click="cmdView_Clicked" Tag="{Binding Path=CategoryID}">View ...</Button>
  </Grid>
</DataTemplate>
```

You can then retrieve the Tag property in the cmdView_Clicked event handler:

```
private void cmdView_Clicked(object sender, RoutedEventArgs e)
{
    Button cmd = (Button)sender;
    int categoryID = (int)cmd.Tag;
    ...
}
```

You can use this information to take another action. For example, you might launch another window that shows products and pass the CategoryID value to that window, which can then use filtering to show only the products in that category. (One easy way to implement filtering is with data views, as described in Chapter 21.)

If you want all the information about the selected data item, you can grab the entire data object by leaving out the Path property when you define the binding:

```
<Button HorizontalAlignment="Right" Padding="1"
  Click="cmdView_Clicked" Tag="{Binding}">View ...</Button>
```

Now your event handler will receive the Product object (if you're binding a collection of Products). If you're binding to a DataTable, you'll receive a DataRowView object instead, which you can use to retrieve all the field values exactly as you would with a DataRow object.

Passing the entire object has another advantage: it makes it easier to update the list selection. In the current example, it's possible to click a button in any item, regardless of whether that item is currently selected. This is potentially confusing, because the user could select one item and click the View button of another item. When the user returns to the list window, the first item remains selected even though the second item was the one that was used by the previous operation. To remove the possibility for confusion, it's a good idea to move the selection to the new list item when the View button is clicked, as shown here:

```
Button cmd = (Button)sender;
Product product = (Product)cmd.Tag;
lstCategories.SelectedItem = product;
```