■ **Note** Technically, the CollectionViewSource is not a view. It's a helper class that allows you to retrieve a view (using the GetDefaultView() method you've seen in the previous examples) and a factory that can create a view when you need it (as you'll see in this section).

The two most important properties of the CollectionViewSource class are View, which wraps the view object, and Source, which wraps the data source. The CollectionViewSource also adds the SortDescriptions and GroupDescriptions properties, which mirror the identically named view properties you've already learned about. When the CollectionViewSource creates a view, it simply passes the value of these properties to the view.

The CollectionViewSource also includes a Filter event, which you can handle to perform filtering. This filtering works in the same way as the Filter callback that's provided by the view object, except it's defined as an event so you can easily hook up your event handler in XAML.

For example, consider the previous example, which placed products in groups using price ranges. Here's how you would define the converter and CollectionViewSource you need for this example declaratively:

```
<local:PriceRangeProductGrouper x:Key="Price50Grouper" GroupInterval="50"/>
  <CollectionViewSource x:Key="GroupByRangeView">
    <CollectionViewSource.SortDescriptions>
      <component:SortDescription PropertyName="UnitCost" Direction="Ascending"/>
    </CollectionViewSource.SortDescriptions>
    <CollectionViewSource.GroupDescriptions>
      <PropertyGroupDescription PropertyName="UnitCost"
       Converter="{StaticResource Price50Grouper}"/>
  </CollectionViewSource.GroupDescriptions>
</CollectionViewSource>
```

Notice that the SortDescription class isn't one of the WPF namespaces. To use it, you need to add the following namespace alias:

```
xmlns:component="clr-namespace:System.ComponentModel;assembly=WindowsBase"
```

Once you've set up the CollectionViewSource, you can bind to it in your list:

```
<ListBox ItemsSource="{Binding Source={StaticResource GroupByRangeView}}" ... >
```

At first glance, this looks a bit odd. It seems as though the ListBox control is binding to the CollectionViewSource, not the view exposed by the CollectionViewSource (which is stored in the CollectionViewSource.View property). However, WPF data binding makes a special exception for the CollectionViewSource. When you use it in a binding expression, WPF asks the CollectionViewSource to create its view and then binds that view to the appropriate element.

The declarative approach doesn't really save you any work. You still need code that retrieves the data at runtime. The difference is that now your code must pass the data along to the CollectionViewSource rather than supply it directly to the list:

```
ICollection<Product> products = App.StoreDB.GetProducts();
CollectionViewSource viewSource = (CollectionViewSource)
  this.FindResource("GroupByRangeView");
viewSource.Source = products;
```

Alternatively, you could create the products collection as a resource using XAML markup. You could then bind the CollectionViewSource to your products collection declaratively. However, you still need to use code to populate your products collection.

■ **Note** People use a few dubious tricks to create code-free data binding. Sometimes, the data collection is defined and filled using XAML markup (with hard-coded values). In other cases, the code for populating the data object is hidden away in the data object's constructor. Both these approaches are severely impractical. I mention them only because they're often used to create quick, off-the-cuff data binding examples.

Now that you've seen the code-based and markup-based approaches for configuring a view, you're probably wondering which one is the better design decision. Both are equally valid. The choice you make depends on where you want to centralize the details for your data view.

However, the choice becomes more significant if you want to use *multiple* views. In this situation, there's a good case to be made for defining all your views in markup and then using code to swap in the appropriate view.

■ **Tip** Creating multiple views makes sense if your views are dramatically different. (For example, they group on completely different criteria.) In many other cases, it's simpler to modify the sorting or grouping information for the current view.

Filtering, Sorting, and Grouping

As you've already seen, views track the current position in a collection of data objects. This is an important task, and finding (or changing) the current item is the most common reason to use a view.

Views also provide a number of optional features that allow you to manage the entire set of items. In the following sections, you'll see how you can use a view to filter your data items (temporarily hiding those you don't want to see), how you can use it to apply sorting (changing the data item order), and how you can use it to apply grouping (creating subcollections that can be navigated separately).

Filtering Collections

Filtering allows you to show a subset of records that meet specific conditions. When working with a collection as a data source, you set the filter using the Filter property of the view object.

The implementation of the Filter property is a little awkward. It accepts a Predicate delegate that points to a custom filtering method (that you create). Here's an example of how you can connect a view to a method named FilterProduct():

```
ListCollectionView view = (ListCollectionView)
  CollectionViewSource.GetDefaultView(lstProducts.ItemsSource);
view.Filter = new Predicate<object>(FilterProduct);
```

The filtering examines a single data item from the collection and returns true if it should be allowed in the list or false if it should be excluded. When you create the Predicate object, you specify the type of object that it's meant to examine. The awkward part is that the view expects you to use a Predicate<object> instance—you can't use something more useful (such as Predicate<Product>) to save yourself the type casting code.

Here's a simple method that shows products only if they exceed $100:

```
public bool FilterProduct(Object item)
{
    Product product = (Product)item;
    return (product.UnitCost > 100);
}
```

Obviously, it makes little sense to hard-code values in your filter condition. A more realistic application would filter dynamically based on other information, like the user-supplied criteria shown in Figure 21-3.

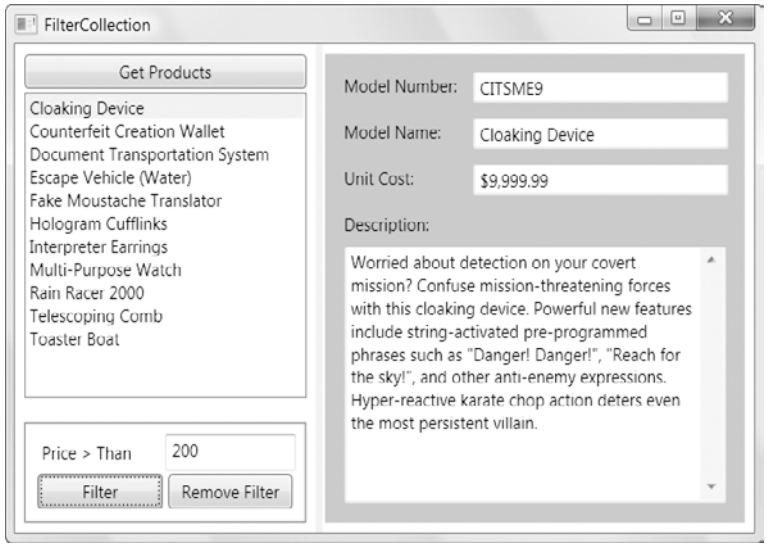

Figure 21-3. *Filtering the product list*

You can use two strategies to make this scenario work. If you use an anonymous delegate, you can define an inline filtering method, which gives you access to any local variables that are in scope in the current method. Here's an example:

```
ListCollectionView view = (ListCollectionView)
  CollectionViewSource.GetDefaultView(lstProducts.ItemsSource);
view.Filter = delegate(object item)
              {
                  Product product = (Product)item;
                  return (product.UnitCost > 100);
              }
```

Although this is a neat, elegant approach, in more complex filtering scenarios you're more likely to use a different strategy and create a dedicated filtering class. That's because in these situations, you often need to filter using several different criteria, and you may want the ability to modify the filtering criteria later.

The filtering class wraps the filtering criteria and the callback method that performs the filtering. Here's an extremely simple filtering class that filters products that fall below a minimum price:

```
public class ProductByPriceFilter
{
    public decimal MinimumPrice
    {
        get; set;
    }

    public ProductByPriceFilter(decimal minimumPrice)
    {
        MinimumPrice = minimumPrice;
    }

    public bool FilterItem(Object item)
    {
        Product product = item as Product;
        if (product != null)
        {
            return (product.UnitCost > MinimumPrice);
        }
        return false;
    }
}
```

Here's the code that creates the ProductByPriceFilterer and uses it to apply minimum price filtering:

```
private void cmdFilter_Click(object sender, RoutedEventArgs e)
{
    decimal minimumPrice;
    if (Decimal.TryParse(txtMinPrice.Text, out minimumPrice))
    {
        ListCollectionView view =
          CollectionViewSource.GetDefaultView(lstProducts.ItemsSource)
          as ListCollectionView;

        if (view != null)
        {
            ProductByPriceFilter filter =
              new ProductByPriceFilter(minimumPrice);
            view.Filter = new Predicate<object>(filter.FilterItem);
        }
    }
}
```

It might occur to you to create different filters for filtering different types of data. For example, you might plan to create (and reuse) a MinMaxFilter, a StringFilter, and so on. However, it's usually more helpful to create a single filtering class for each window where you want to apply filtering. That's because you can't chain more than one filter together.

■ **Note** Of course, you could create a custom implementation that solves this problem—for example, a FilterChain class that wraps a collection of IFilter objects and calls the FilterItem() method of each one to find out whether to exclude an item. However, this extra layer may be more code and complexity than you need.

If you want to modify the filter later without re-creating the ProductByPriceFilter object, you'll need to store a reference to the filter object as a member variable in your window class. You can then modify the filter properties. However, you'll also need to call the Refresh() method of the view object to force the list to be refiltered. Here's some code that adjusts the filter settings whenever the TextChanged event fires in the text box that contains the minimum price:

```
private void txtMinPrice_TextChanged(object sender, TextChangedEventArgs e)
{
    ListCollectionView view =
      CollectionViewSource.GetDefaultView(lstProducts.ItemsSource)
      as ListCollectionView;
    if (view != null)
    {
        decimal minimumPrice;
        if (Decimal.TryParse(txtMinPrice.Text, out minimumPrice) &&
          (filter != null))
        {
            filter.MinimumPrice = minimumPrice;
            view.Refresh();
        }
    }
}
```

■ **Tip** It's a common convention to let the user choose to apply different types of conditions using a series of check boxes. For example, you could create a check box for filtering by price, by name, by model number, and so on. The user can then choose which filter conditions to apply by checking the appropriate check boxes.

Finally, you can completely remove a filter by setting the Filter property to null:

```
view.Filter = null;
```

Filtering the DataTable

Filtering works differently with the DataTable. If you've worked with ADO.NET before, you probably already know that every DataTable works in conjunction with a DataView object (which is, like the DataTable, defined in the System.Data namespace along with the other core ADO.NET data objects). The ADO.NET DataView plays much the same role as the WPF view object. Like a WPF view, it allows you to filter records (by field content using the RowFilter property or by row state using the RowStateFilter property). It also supports sorting through the Sort property. Unlike the WPF view object, the DataView doesn't track the position in a set of data. It also provides additional properties that allow you to lock down editing capabilities (AllowDelete, AllowEdit, and AllowNew).

It's quite possible to change the way a list of data is filtered by retrieving the bound DataView and modifying its properties directly. (Remember, you can get the default DataView from the DataTable.DefaultView property.) However, it would be nicer if you had a way to adjust the filtering through the WPF view object so that you can continue to use the same model.

It turns out that this is possible, but there are some limitations. Unlike the ListCollectionView, the BindingListCollectionView that's used with the DataTable doesn't support the Filter property. (BindingListCollectionView.CanFilter returns false, and attempting to set the Filter property causes an exception to be thrown.) Instead, the BindingListCollectionView provides a CustomFilter property. The CustomFilter property doesn't do any work of its own—it simply takes the filter string that you specify and uses it to set the underlying DataView.RowFilter property.

The DataView.RowFilter is easy enough to use but a little messy. It takes a string-based filter expression, which is modeled after the snippet of SQL you'd use to construct the WHERE clause in a SELECT query. As a result, you need to follow all the conventions of SQL, such as bracketing string and date values with single quotes ('). And if you want to use multiple conditions, you need to string them all together using the OR and AND keywords.

Here's an example that duplicates the filtering shown in the earlier, collection-based example so that it works with a DataTable of product records:

```
decimal minimumPrice;
if (Decimal.TryParse(txtMinPrice.Text, out minimumPrice))
{
    BindingListCollectionView view =
      CollectionViewSource.GetDefaultView(lstProducts.ItemsSource)
      as BindingListCollectionView;
    if (view != null)
    {
        view.CustomFilter = "UnitCost > " + minimumPrice.ToString();
    }
}
```

Notice that this example takes the roundabout approach of converting the text in the txtMinPrice text box to a decimal value and then back to a string to use for filtering. This requires a bit more work, but it avoids possible injection attacks and errors with invalid characters. If you simply concatenate the text from the txtMinPrice text box to build your filter string, it could contain filter operations (=, <, >) and keywords (AND, OR) that apply completely different filtering than what you intend. This could happen as part of a deliberate attack or because of user error.

Sorting

You can also use a view to implement sorting. The easiest approach is to sort based on the value of one or more properties in each data item. You identify the fields you want to use using System.ComponentModel.SortDescription objects. Each SortDescription identifies the field you want to use for sorting and the sort direction (ascending or descending). You add the SortDescription objects in the order that you want to apply them. For example, you could sort first by category and then by model name.

Here's an example that applies a simple ascending sort by model name:

```
ICollectionView view = CollectionViewSource.GetDefaultView(lstProducts.ItemsSource);
view.SortDescriptions.Add(
  new SortDescription("ModelName", ListSortDirection.Ascending));
```

Because this code uses the ICollectionView interface rather than a specific view class, it works equally well no matter what type of data source you're binding. In the case of a BindingListCollectionView (when binding a DataTable), the SortDescription objects are used to build a sorting string that's applied to the underlying DataView.Sort property.

■ **Note** In the rare case that you have more than one BindingListCollectionView working with the same DataView, both will share the same filtering and sorting settings, because these details are stored in the DataView, not the BindingListCollectionView. If this isn't the behavior you want, you can create more than one DataView to wrap the same DataTable.

As you'd expect, when sorting strings, values are ordered alphabetically. Numbers are ordered numerically. To apply a different sort order, begin by clearing the existing SortDescriptions collection.

You also have the ability to perform a custom sort, but only if you're using the ListCollectionView (not the BindingListCollectionView). The ListCollectionView provides a CustomSort property that accepts an IComparer object that performs the comparison between any two data items and indicates which one should be considered greater than the other. This approach is handy if you need to build a sorting routine that combines properties to get a sorting key. It also makes sense if you have nonstandard sorting rules. For example, you may want to ignore the first few characters of a product code, perform a calculation on a price, convert your field to a different data type or a different representation before sorting, and so on. Here's an example that counts the number of letters in the model name and uses that to determine sort order:

```
public class SortByModelNameLength : IComparer
{
    public int Compare(object x, object y)
    {
        Product productX = (Product)x;
        Product productY = (Product)y;
        return productX.ModelName.Length.CompareTo(productY.ModelName.Length);
    }
}
```

Here's the code that connects the IComparer to a view:

```
ListCollectionView view = (ListCollectionView)
  CollectionViewSource.GetDefaultView(lstProducts.ItemsSource);
view.CustomSort = new SortByModelNameLength();
```

In this example, the IComparer is designed to fit a specific scenario. If you have an IComparer that you need to reuse with similar data in different places, you can generalize it. For example, you could change the SortByModelNameLength class to a SortByTextLength class. When creating a SortByTextLength instance, your code would need to supply the name of the property to use (as a string), and your Compare() method could then use reflection to look it up in the data object.

Grouping

In much the same way that they support sorting, views also allow you to apply grouping. As with sorting, you can group the easy way (based on a single property value) or the hard way (using a custom callback).

To perform grouping, you add System.ComponentModel.PropertyGroupDescription objects to the CollectionView.GroupDescriptions collection. Here's an example that groups products by category name:

```
ICollectionView view = CollectionViewSource.GetDefaultView(lstProducts.ItemsSource);
view.GroupDescriptions.Add(new PropertyGroupDescription("CategoryName"));
```

■ **Note** This example assumes that the Product class has a property named CategoryName. It's more likely that you have a property named Category (which returns a linked Category object) or CategoryID (which identifies the category with a unique ID number). You can still use grouping in these scenarios, but you'll need to add a value converter that examines the grouping information (such as the Category object or CategoryID property) and returns the correct category text to use for the group. You'll see how to use a value converter with grouping in the next example.

This example has one problem. Although your items will now be arranged into separate groups based on their categories, it's difficult to see that any grouping has been applied when you look at the list. In fact, the result is the same as if you simply sorted by category name.

There's actually more taking place—you just can't see it with the default settings. When you use grouping, your list creates a separate GroupItem object for each group, and it adds these GroupItem objects to the list. The GroupItem is a content control, so each GroupItem holds the appropriate container (like ListBoxItem objects) with your actual data. The secret to showing your groups is formatting the GroupItem element so it stands out.

You could use a style that applies formatting to all the GroupItem objects in a list. However, you probably want more than just formatting—for example, you might want to display a group header, which requires the help of a template. Fortunately, the ItemsControl class makes both tasks easy through its ItemsControl.GroupStyle property, which provides a collection of GroupStyle objects. Despite the name, GroupStyle class is not a style. It's simply a convenient package that wraps a few useful settings for configuring your GroupItem objects. Table 21-1 lists the properties of the GroupStyle class.

Table 21-1. *GroupStyle Properties*

Name	Description
ContainerStyle	Sets the style that's applied to the GroupItem that's generated for each group.
ContainerStyleSelector	Instead of using ContainerStyle, you can use ContainerStyleSelector to supply a class that chooses the right style to use, based on the group.
HeaderTemplate	Allows you to create a template for displaying content at the beginning of each group.
HeaderTemplateSelector	Instead of using HeaderTemplate, you can use HeaderTemplateSelector to supply a class that chooses the right header template to use, based on the group.
Panel	Allows you to change the template that's used to hold groups. For example, you could use a WrapPanel instead of the standard StackPanel to create a list that tiles groups from left to right and then down.

In this example, all you need is a header before each group. You can use this to create the effect shown in Figure 21-4.

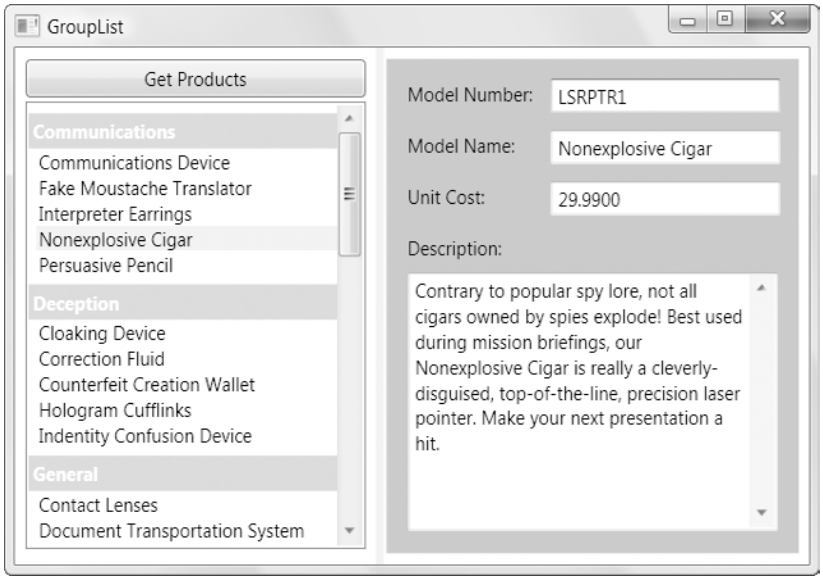

Figure 21-4. *Grouping the product list*

To add a group header, you need to set the GroupStyle.HeaderTemplate. You can fill this property with an ordinary data template, like the ones you saw in Chapter 20. You can use any combination of elements and data binding expressions inside your template.

However, there's one trick. When you write your binding expression, you aren't binding against the data object from your list (in this case, the Product object). Instead, you're binding against the PropertyGroupDescription object for that group. That means if you want to display the field value for that group (as shown in Figure 21-4), you need to bind the PropertyGroupDescription.Name property rather than Product.CategoryName.

Here's the complete template:

```
<ListBox Name="lstProducts" DisplayMemberPath="ModelName">
  <ListBox.GroupStyle>
    <GroupStyle>
      <GroupStyle.HeaderTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding Path=Name}" FontWeight="Bold"
           Foreground="White" Background="LightGreen"
           Margin="0,5,0,0" Padding="3"/>
        </DataTemplate>
      </GroupStyle.HeaderTemplate>
    </GroupStyle>
  </ListBox.GroupStyle>
</ListBox>
```

■ **Tip** The ListBox.GroupStyle property is actually a collection of GroupStyle objects. This allows you to add multiple levels of grouping. To do so, you need to add more than one PropertyGroupDescription (in the order that you want your grouping and subgrouping applied) and then add a matching GroupStyle object to format each level.

You'll probably want to use grouping in conjunction with sorting. If you want to sort your groups, just make sure that the first SortDescription you use sorts based on the grouping field. The following code sorts the categories alphabetically by category name and then sorts each product within the category alphabetically by model name.

```
view.SortDescriptions.Add(new SortDescription("CategoryName",
  ListSortDirection.Ascending));
view.SortDescriptions.Add(new SortDescription("ModelName",
  ListSortDirection.Ascending));
```

One limitation with the simple grouping approach you see here is that it requires a field with duplicate values in order to perform its grouping. The previous example works because many products share the same category and have duplicate values for the CategoryName property. However, this approach doesn't work as well if you try to group by another piece of information, such as the UnitCost field. In this situation, you'll end up with a separate group for each product.

This problem has a solution. You can create a class that examines some piece of information and places it into a conceptual group for display purposes. This technique is commonly used to group data objects using numeric or date information that falls into specific ranges. For example, you could create a

group for products that are less than $50, another for products that fall between $50 and $100, and so on. Figure 21-5 shows this example.

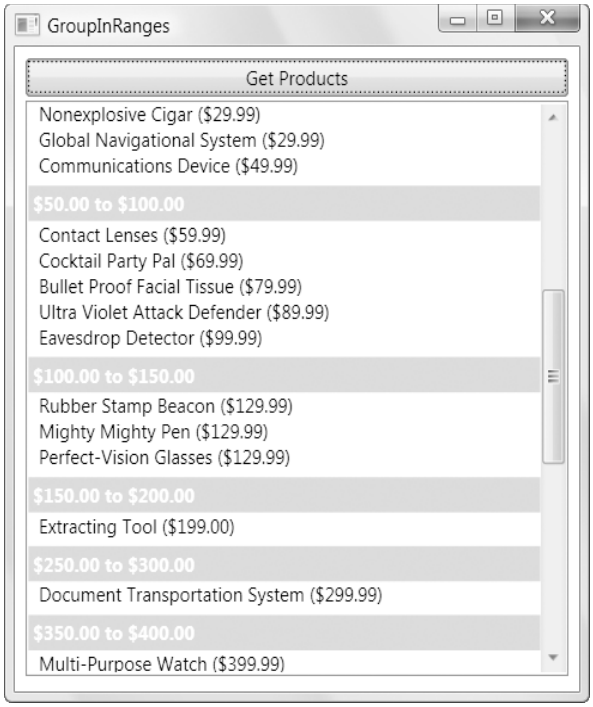

Figure 21-5. *Grouping in ranges*

To create this solution, you need to supply a value converter that examines a field in your data source (or multiple fields if you implement IMultiValueConverter) and returns the group header. As long as you use the same group header for multiple data objects, these objects are placed into the same logical group.

The following code shows the converter that creates the price ranges shown in Figure 21-5. It's designed to have some flexibility—namely, you can specify the size of the grouping ranges. (In Figure 21-5, the group range is 50 units big.)

```
public class PriceRangeProductGrouper : IValueConverter
{
    public int GroupInterval
    {
        get; set;
    }

    public object Convert(object value, Type targetType, object parameter,
      CultureInfo culture)
    {
        decimal price = (decimal)value;
```

```
        if (price < GroupInterval)
        {
            return String.Format(culture, "Less than {0:C}", GroupInterval);
        }
        else
        {
            int interval = (int)price / GroupInterval;
            int lowerLimit = interval * GroupInterval;
            int upperLimit = (interval + 1) * GroupInterval;
            return String.Format(culture, "{0:C} to {1:C}", lowerLimit, upperLimit);
        }
    }

    public object ConvertBack(object value, Type targetType, object parameter,
      CultureInfo culture)
    {
        throw new NotSupportedException("This converter is for grouping only.");
    }
}
```

To make this class even more flexible so that it can be used with other fields, you could add other properties that allow you to set the fixed part of the header text and a format string to use when converting the numeric values to header text. (The current code assumes the numbers should be treated as currencies, so 50 becomes $50.00 in the header.)

Here's the code that uses the converter to apply the range grouping. Note that the products must first be sorted by price, or you'll end up grouping them based on where they fall in the list.

```
ICollectionView view =
  CollectionViewSource.GetDefaultView(lstProducts.ItemsSource);
view.SortDescriptions.Add(new SortDescription("UnitCost",
  ListSortDirection.Ascending));

PriceRangeProductGrouper grouper = new PriceRangeProductGrouper();
grouper.GroupInterval = 50;
view.GroupDescriptions.Add(new PropertyGroupDescription("UnitCost", grouper));
```

This example does its work in code, but you can also create the converter and the view declaratively by placing them both in the Resources collection of the window. You'll see how this works in the next section.

The Last Word

Views are the final piece in the data binding puzzle. They're an invaluable extra layer that sits between your data and the elements that display it, allowing you to manage your position in a collection and giving you the flexibility to implement filtering, sorting, and grouping. In every data binding scenario, there's a view at work. The only difference is whether it's acting behind the scenes or whether you're explicitly taking control of it with code.

You've now considered all the key principles of data binding (and a bit more besides). In the following chapter, you look at three controls that give you still more options for presenting and editing bound data: the ListView, TreeView, and DataGrid.

CHAPTER 22

Lists, Grids, and Trees

So far, you've learned a wide range of techniques and tricks for using WPF data binding to display information in the form you need. Along the way, you've seen many examples that revolve around the lowly ListBox control.

Thanks to the extensibility provided by styles, data templates, and control templates, even the ListBox (and its similarly equipped sibling, the ComboBox) can serve as remarkably powerful tools for displaying data in a variety of ways. However, some types of data presentation would be difficult to implement with the ListBox alone. Fortunately, WPF has a few rich data controls that fill in the blanks, including the following:

- **ListView.** The ListView derives from the plain-vanilla ListBox. It adds support for column-based display and the ability to switch quickly between different "views," or display modes, without requiring you to rebind the data and rebuild the list.
- **TreeView.** The TreeView is a hierarchical container, which means you can create a multilayered data display. For example, you could create a TreeView that shows category groups in its first level and shows the related products under each category node.
- **DataGrid.** The DataGrid is WPF's most full-featured data display tool. It divides your data into a grid of columns and rows, like the ListView, but has additional formatting features (such as the ability to freeze columns and style individual rows), and it supports in-place data editing.

In this chapter, you'll look at these three key controls.

What's New Early versions of WPF lacked a professional grid control for editing data. Fortunately, the powerful DataGrid joined the control library in .NET 3.5 SP1.

The ListView

The ListView is a specialized list class that's designed for displaying different *views* of the same data. The ListView is particularly useful if you need to build a multicolumn view that displays several pieces of information about each data item.

The ListView derives from the ListBox class and extends it with a single detail: the View property. The View property is yet another extensibility point for creating rich list displays. If you don't set the View property, the ListView behaves just like its lesser-powered ancestor, the ListBox. However, the ListView becomes much more interesting when you supply a view object that indicates how data items should be formatted and styled.

Technically, the View property points to an instance of any class that derives from ViewBase (which is an abstract class). The ViewBase class is surprisingly simple; in fact, it's little more than a package that binds together two styles. One style applies to the ListView control (and is referenced by the DefaultStyleKey property), and the other style applies to the items in the ListView (and is referenced by the ItemContainerDefaultStyleKey property). The DefaultStyleKey and ItemContainerDefaultStyleKey properties don't actually provide the style; instead, they return a ResourceKey object that points to it.

At this point, you might wonder why you need a View property—after all, the ListBox already offers powerful data template and styling features (as do all classes that derive from ItemsControl). Ambitious developers can rework the visual appearance of the ListBox by supplying a different data template, layout panel, and control template.

In truth, you don't need a ListView class with a View property in order to create customizable multicolumned lists. In fact, you could achieve much the same thing on your own using the template and styling features of the ListBox. However, the View property is a useful abstraction. Here are some of its advantages:

- **Reusable views.** The ListView separates all the view-specific details into one object. That makes it easier to create views that are data-independent and can be used on more than one list.
- **Multiple views.** The separation between the ListView control and the View objects also makes it easier to switch between multiple views with the same list. (For example, you use this technique in Windows Explorer to get a different perspective on your files and folders.) You could build the same feature by dynamically changing templates and styles, but it's easier to have just one object that encapsulates all the view details.
- **Better organization.** The view object wraps two styles: one for the root ListView control and one that applies to the individual items in the list. Because these styles are packaged together, it's clear that these two pieces are related and may share certain details and interdependencies. For example, this makes a lot of sense for a column-based ListView, because it needs to keep its column headers and column data lined up.

Using this model, there's a great potential to create a number of useful prebuilt views that all developers can use. Unfortunately, WPF currently includes just one view object: the GridView. Although you can use the GridView is extremely useful for creating multicolumn lists, you'll need to create your own custom view if you have other needs. The following sections show you how to do both.

■ **Note** The GridView is a good choice if you want to show a configurable data display, and you want a grid-styled view to be one of the user's options. But if you want a grid that supports advanced styling, selection, or editing, you'll need to step up to the full-fledged DataGrid control described later in this chapter.

Creating Columns with the GridView

The GridView is a class that derives from ViewBase and represents a list view with multiple columns. You define those columns by adding GridViewColumn objects to the GridView.Columns collection.

Both GridView and GridViewColumn provide a small set of useful methods that you can use to customize the appearance of your list. To create the simplest, most straightforward list (which resembles the details view in Windows Explorer), you need to set just two properties for each GridViewColumn: Header and DisplayMemberBinding. The Header property supplies the text that's placed at the top of the column. The DisplayMemberBinding property contains a binding that extracts the piece of information you want to display from each data item.

Figure 22-1 shows a straightforward example with three columns of information about a product.

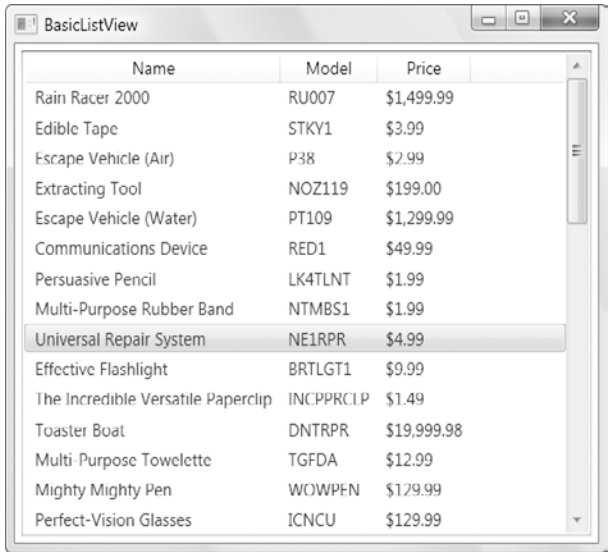

Figure 22-1. *A grid-based ListView*

Here's the markup that defines the three columns used in this example:

```
<ListView Margin="5" Name="lstProducts">
  <ListView.View>
    <GridView>
      <GridView.Columns>
        <GridViewColumn Header="Name"
```

```
         DisplayMemberBinding="{Binding Path=ModelName}" />
        <GridViewColumn Header="Model"
         DisplayMemberBinding="{Binding Path=ModelNumber}" />
        <GridViewColumn Header="Price" DisplayMemberBinding=
"{Binding Path=UnitCost, StringFormat={}{0:C}}" />
      </GridView.Columns>
    </GridView>
  </ListView.View>
</ListView>
```

This example has a few important points worth noticing. First, none of the columns has a hard-coded size. Instead, the GridView sizes its columns just large enough to fit the widest visible item (or the column header, if it's wider), which makes a lot of sense in the flow layout world of WPF. (Of course, this leaves you in a bit of trouble if you have huge columns values. In this case, you may choose to wrap your text, as described in the upcoming "Cell Templates" section.)

Also, notice how the DisplayMemberBinding property is set using a full-fledged binding expression, which supports all the tricks you learned about in Chapter 20, including string formatting and value converters.

Resizing Columns

Initially, the GridView makes each column just wide enough to fit the largest visible value. However, you can easily resize any column by clicking and dragging the edge of the column header. Or, you can double-click the edge of the column header to force the GridViewColumn to resize itself based on whatever content is currently visible. For example, if you scroll down the list and find an item that's truncated because it's wider than the column, just double-click the right edge of that column's header. The column will automatically expand itself to fit.

For more micromanaged control over column size, you can set a specific width when you declare the column:

```
<GridViewColumn Width="300" ... />
```

This simply determines the initial size of the column. It doesn't prevent the user from resizing the column using either of the techniques described previously. Unfortunately, the GridViewColumn class doesn't define properties like MaxWidth and MinWidth, so there's no way to constrain how a column can be resized. Your only option is to supply a new template for the GridViewColumn's header if you want to disable resizing altogether.

■ **Note** The user can also reorder columns by dragging a header to a new position.

Cell Templates

The GridViewColumn.DisplayMemberBinding property isn't the only option for showing data in a cell. Your other choice is the CellTemplate property, which takes a data template. This is exactly like the data templates you learned about in Chapter 20, except it applies to just one column. If you're ambitious, you can give each column its own data template.

Cell templates are a key piece of the puzzle when customizing the GridView. One feature that they allow is text wrapping. Ordinarily, the text in a column is wrapped in a single-line TextBlock. However, it's easy to change this detail using a data template of your own devising:

```
<GridViewColumn Header="Description" Width="300">
  <GridViewColumn.CellTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding Path=Description}" TextWrapping="Wrap"></TextBlock>
    </DataTemplate>
  </GridViewColumn.CellTemplate>
</GridViewColumn>
```

Notice that in order for the wrapping to have an effect, you need to constrain the width of the column using the Width property. If the user resizes the column, the text will be rewrapped to fit. You *don't* want to constrain the width of the TextBlock, because that would ensure that your text is limited to a single specific size, no matter how wide or narrow the column becomes.

The only limitation in this example is that the data template needs to bind explicitly to the property you want to display. For that reason, you can't create a template that enables wrapping and reuse it for every piece of content you want to wrap. Instead, you need to create a separate template for each field. This isn't a problem in this simple example, but it's annoying if you create a more complex template that you would like to apply to other lists (for example, a template that converts data to an image and displays it in an Image element, or a template that uses a TextBox control to allow editing). There's no easy way to reuse any template on multiple columns; instead, you'll be forced to cut and paste the template, and then modify the binding.

■ **Note** It would be nice if you could create a data template that uses the DisplayMemberBinding property. That way, you could use DisplayMemberBinding to extract the specific property you want and use CellTemplate to format that content into the correct visual representation. Unfortunately, this just isn't possible. If you set both DisplayMember and CellTemplate, the GridViewColumn uses the DisplayMember property to set the content for the cell and ignores the template altogether.

Data templates aren't limited to tweaking the properties of a TextBlock. You can also use date templates to supply completely different elements. For example, the following column uses a data template to show an image. The ProductImagePath converter (shown in Chapter 20) helps by loading the corresponding image file from the file system.

```
<GridViewColumn Header="Picture" >
  <GridViewColumn.CellTemplate>
    <DataTemplate>
      <Image Source=
"{Binding Path=ProductImagePath,Converter={StaticResource ImagePathConverter}}">
      </Image>
    </DataTemplate>
  </GridViewColumn.CellTemplate>
</GridViewColumn>
```

Figure 22-2 shows a ListView that uses both templates to show wrapped text and a product image.

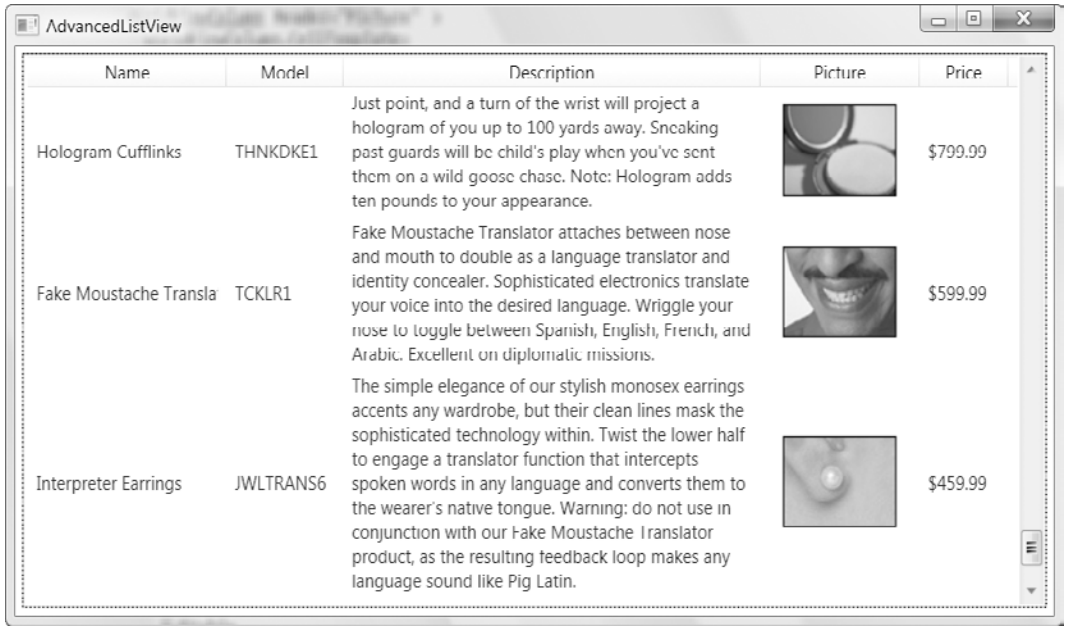

Figure 22-2. *Columns that use templates*

■ **Tip** When creating a data template, you have the choice of defining it inline (as in the previous two examples) or referring to a resource that's defined elsewhere. Because column templates can't be reused for different fields, it's usually clearest to define them inline.

As you learned in Chapter 20, you can vary templates so that different data items get different templates. To do this, you need to create a template selector that chooses the appropriate template based on the properties of the data object at that position. To use this feature, create your selector, and use it to set the GridViewColumn.CellTemplateSelector property. For a full template selector example, see Chapter 20.

Customizing Column Headers

So far, you've seen how to customize the appearance of the values in every cell. However, you haven't done anything to fine-tune the column headers. If the standard gray boxes don't excite you, you'll be happy to find out that you can change the content and appearance of the column headers just as easily as the column values. In fact, you can use several approaches.

If you want to keep the gray column header boxes but you want to fill them with your own content, you can simply set the GridViewColumn.Header property. The previous examples have the Header property using ordinary text, but you can supply an element instead. Use a StackPanel that wraps a TextBlock and Image to create a fancy header that combines text and image content.

If you want to fill the column headers with your own content, but you don't want to specify this content separately for each column, you can use the GridViewColumn.HeaderTemplate property to define a data template. This data template binds to whatever object you've specified in the GridViewColumn.Header property and presents it accordingly.

If you want to reformat a specific column header, you can use the GridViewColumn.HeaderContainerStyle property to supply a style. If you want to reformat all the column headers in the same way, use the GridView.ColumnHeaderContainerStyle property instead.

If you want to completely change the appearance of the header (for example, replacing the gray box with a rounded blue border), you can supply a completely new control template for the header. Use GridViewColumn.HeaderTemplate to change a specific column, or use GridView.ColumnHeaderTemplate to change them all in the same way. You can even use a template selector to choose the correct template for a given header by setting the GridViewColumn.HeaderTemplateSelector or GridView.ColumnHeaderTemplateSelector property.

Creating a Custom View

If the GridView doesn't meet your needs, you can create your own view to extend the ListView's capabilities. Unfortunately, it's far from straightforward.

To understand the problem, you need to know a little more about the way a view works. Views do their work by overriding two protected properties: DefaultStyleKey and ItemContainerDefaultKeyStyle. Each property returns a specialized object called a ResourceKey, which points to a style that you've defined in XAML. The DefaultStyleKey property points to the style that should be applied to configure the overall ListView. The ItemContainer.DefaultKeyStyle property points to the style that should be used to configure each ListViewItem in the ListView. Although these styles are free to tweak any property, they usually do their work by replacing the ControlTemplate that's used for the ListView and the DataTemplate that's used for each ListViewItem.

Here's where the problems occur. The DataTemplate you use to display items is defined in XAML markup. Imagine you want to create a ListView that shows a tiled image for each item. This is easy enough using a DataTemplate—you simply need to bind the Source property of an Image to the correct property of your data object. But how do you know which data object the user will supply? If you hard-code property names as part of your view, you'll limit its usefulness, making it impossible to reuse your

custom view in other scenarios. The alternative—forcing the user to supply the DataTemplate—means you can't pack as much functionality into the view, so reusing it won't be as useful.

■ **Tip** Before you begin creating a custom view, consider whether you could get the same result by simply using the right DataTemplate with a ListBox or a ListView/GridView combination.

So why go to all the effort of designing a custom view if you can already get all the functionality you need by restyling the ListView (or even the ListBox)? The primary reason is if you want a list that can dynamically change views. For example, you might want a product list that can be viewed in different modes, depending on the user's selection. You could implement this by dynamically swapping in different DataTemplate objects (and this is a reasonable approach), but often a view needs to change both the DataTemplate of the ListViewItem and the layout or overall appearance of the ListView itself. A view helps clarify the relationship between these details in your source code.

The following example shows you how to create a grid that can be switched seamlessly from one view to another. The grid begins in the familiar column-separated view but also supports two tiled image views, as shown in Figure 22-3 and Figure 22-4.

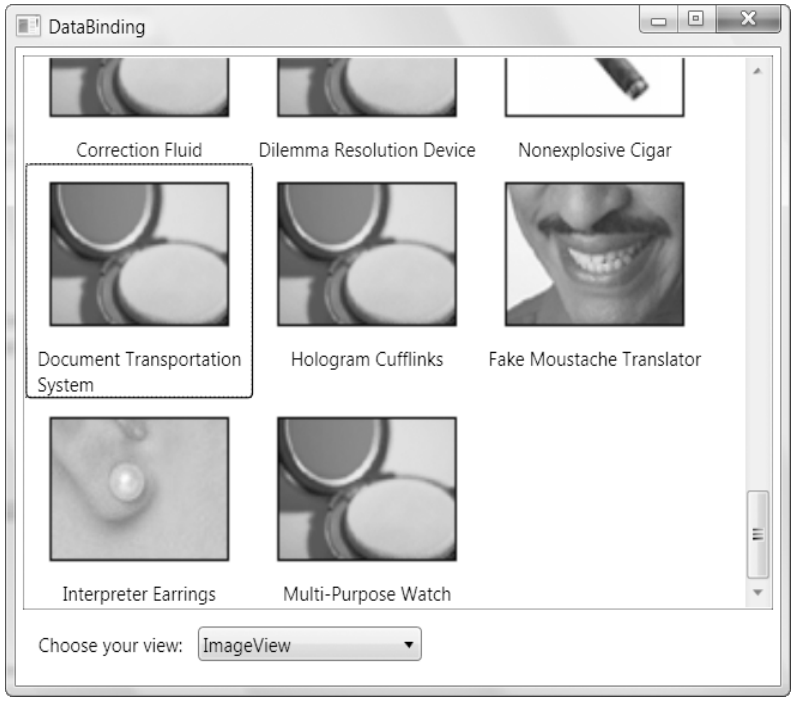

Figure 22-3. *An image view*