CHAPTER 21

Data Views

Now that you've explored the art of converting data, applying styles to the items in a list, and building data templates, you're ready to move on to *data views*, which work behind the scenes to coordinate collections of bound data. Using data views, you can add navigation logic and implement filtering, sorting, and grouping.

What's New In this chapter, you'll come across two more data-binding tweaks that are introduced in WPF 4.5. In the "Grouping and Virtualization" section, you'll learn how virtualization now works even with grouped data. In the "Live Shaping" section, you'll learn how WPF can watch your bound data for changes and update the linked view automatically.

The View Object

When you bind a collection (or a DataTable) to an ItemsControl, a data view is quietly created behind the scenes. This view sits between your data source and the bound control. The data view is a window into your data source. It tracks the current item, and it supports features such as sorting, filtering, and grouping. These features are independent of the data object itself, which means you can bind the same data in different ways in different portions of a window (or different parts of your application). For example, you could bind the same collection of products to two different lists but filter them to show different records.

The view object that's used depends on the type of data object. All views derive from CollectionView, but two specialized implementations derive from CollectionView: ListCollectionView and BindingListCollectionView. Here's how it works:

- If your data source implements IBindingList, a BindingListCollectionView is created. This happens when you bind an ADO.NET DataTable.
- If your data source doesn't implement IBindingList but it implements IList, a ListCollectionView is created. This happens when you bind an ObservableCollection, like the list of products.
- If your data source doesn't implement IBindingList or IList but it implements IEnumerable, you get a basic CollectionView.

■ **Tip** Ideally, you'll avoid the third scenario. The CollectionView offers poor performance for large items and operations that modify the data source (such as insertions and deletions). As you learned in Chapter 19, if you're not binding to an ADO.NET data object, it's almost always easiest to use the ObservableCollection class.

Retrieving a View Object

To get ahold of a view object that's currently in use, you use the static GetDefaultView() method of the System.Windows.Data.CollectionViewSource class. When you call GetDefaultView(), you pass in the data source—the collection or DataTable that you're using. Here's an example that gets the view for the collection of products that's bound to the list:

```
ICollectionView view = CollectionViewSource.GetDefaultView(lstProducts.ItemsSource);
```

The GetDefaultView() method always returns an ICollectionView reference. It's up to you to cast the view object to the appropriate class, such as a ListCollectionView or BindingListCollectionView, depending on the data source.

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

Navigating with a View

One of the simplest things you can do with a view object is determine the number of items in the list (through the Count property) and get a reference to the current data object (CurrentItem) or current position index (CurrentPosition). You can also use a handful of methods to move from one record to another, such as MoveCurrentToFirst(), MoveCurrentToLast(), MoveCurrentToNext(), MoveCurrentToPrevious(), and MoveCurrentToPosition(). So far, you haven't needed these details because all the examples you've seen have used the list to allow the user to move from one record to the next. But if you want to create a record browser application, you might want to supply your own navigation buttons. Figure 21-1 shows one example.

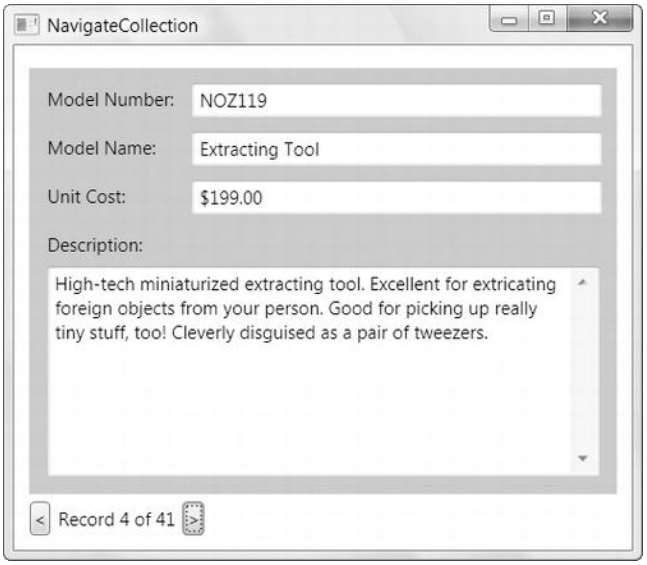

Figure 21-1. *A record browser*

The bound text boxes that show the data for the bound product stay the same. They need only to indicate the appropriate property, as shown here:

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

However, this example doesn't include any list control, so it's up to you to take control of the navigation. To simplify life, you can store a reference to the view as a member variable in your window class:

```
private ListCollectionView view;
```

In this case, the code casts the view to the appropriate view type (ListCollectionView) rather than using the ICollectionView interface. The ICollectionView interface provides most of the same functionality, but it lacks the Count property that gives the total number of items in the collection.

When the window first loads, you can get the data, place it in the DataContext of the window, and store a reference to the view:

```
ICollection<Products> products = App.StoreDB.GetProducts();
this.DataContext = products;

view = (ListCollectionView)CollectionViewSource.GetDefaultView(this.DataContext);
view.CurrentChanged += new EventHandler(view_CurrentChanged);
```

The second line does all the magic needed to show your collection of items in the window. It places the whole collection of Product objects in the DataContext. The bound controls on the form will search up the element tree until they find this object. Of course, you want the binding expressions to bind to the current item in the collection, not the collection itself, but WPF is smart enough to figure this out automatically. It automatically supplies them with the current item, so you don't need a stitch of extra code.

The previous example has one additional code statement. It connects an event handler to the CurrentChanged event of the view. When this event fires, you can perform a few useful actions, such as enabling or disabling the previous and next buttons depending on the current position and displaying the current position in a TextBlock at the bottom of the window.

```
private void view_CurrentChanged(object sender, EventArgs e)
{
    lblPosition.Text = "Record " + (view.CurrentPosition + 1).ToString() +
      " of " + view.Count.ToString();
    cmdPrev.IsEnabled = view.CurrentPosition > 0;
    cmdNext.IsEnabled = view.CurrentPosition < view.Count - 1;
}
```

This code seems like a candidate for data binding and triggers. However, the logic is just a bit too complex (partly because you need to add 1 to the index to get the record position number that you want to display).

The final step is to write the logic for the previous and next buttons. Because these buttons are automatically disabled when they don't apply, you don't need to worry about moving before the first item or after the last item.

```
private void cmdNext_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToNext();
}

private void cmdPrev_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToPrevious();
}
```

For an interesting frill, you can add a list control to this form so the user has the option of stepping through the records one at a time with the buttons or using the list to jump directly to a specific item (as shown in Figure 21-2).

Figure 21-2. *A record browser*

In this case, you need a ComboBox that uses the ItemsSource property (to get the full list of products) and uses a binding on the Text property (to show the right item):

```
<ComboBox Name="lstProducts" DisplayMemberPath="ModelName"
 Text="{Binding Path=ModelName}"
 SelectionChanged="lstProducts_SelectionChanged"></ComboBox>
```

When you first retrieve the collection of products, you'll bind the list:

```
lstProducts.ItemsSource = products;
```

This might not have the effect you expect. By default, the selected item in an ItemsControl is not synchronized with the current item in the view. That means that when you make a new selection from the list, you aren't directed to the new record—instead, you end up modifying the ModelName property of the current record. Fortunately, there are two easy approaches to solve the problem.

The brute-force approach is to simply move to the new record whenever an item is selected in the list. Here's the code that does it:

```
private void lstProducts_SelectionChanged(object sender, RoutedEventArgs e)
{
    view.MoveCurrentTo(lstProducts.SelectedItem);
}
```

A simpler solution is to set the ItemsControl.IsSynchronizedWithCurrentItem to true. That way, the currently selected item is automatically synchronized to match the current position of the view with no code required.

USING A LOOKUP LIST FOR EDITING

The ComboBox provides a handy way to edit record values. In the current example, it doesn't make much sense—after all, there's no reason to give one product the same name as another product. However, it's not difficult to think of other scenarios where the ComboBox is a great editing tool.

For example, you might have a field in your database that accepts one of a small set of preset values. In this case, use a ComboBox, and bind it to the appropriate field using a binding expression for the Text property. However, fill the ComboBox with the allowable values by setting its ItemsSource property to point to the list you've defined. And if you want to display the values in the list one way (say, as text) but store them another way (as numeric codes), just add a value converter to your Text property binding.

Another case where a lookup list makes sense is when dealing with related tables. For example, you might want to allow the user to pick the category for a product using a list of all the defined categories. The basic approach is the same: set the Text property to bind to the appropriate field, and fill in the list of options with the ItemsSource property. If you need to convert low-level unique IDs into more meaningful names, use a value converter.

Creating a View Declaratively

The previous example used a simple pattern that you'll see throughout this chapter. The code retrieves the view you want to use and then modifies it programmatically. However, you have another choice—you can construct a CollectionViewSource declaratively in XAML markup and then bind the CollectionViewSource to your controls (such as the list).

■ **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.

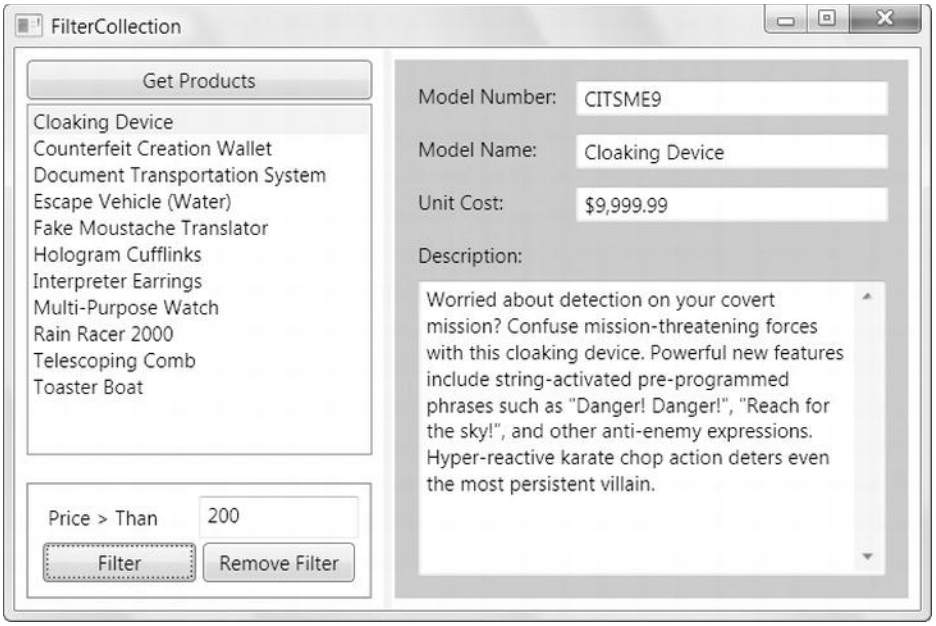

Figure 21-3. *Filtering the product list*

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 can 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.

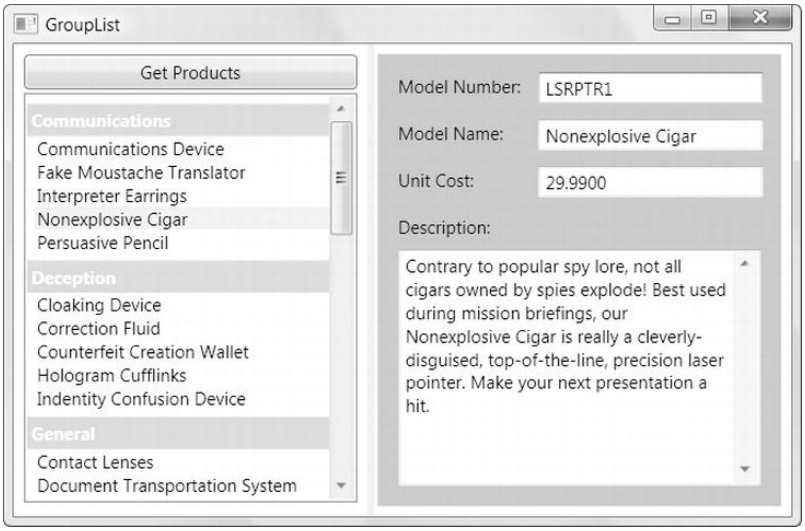

Figure 21-4. *Grouping the product list*

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));
```

Grouping in Ranges

One limitation with the simple grouping approach you see here is that it requires a field with duplicate values 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.

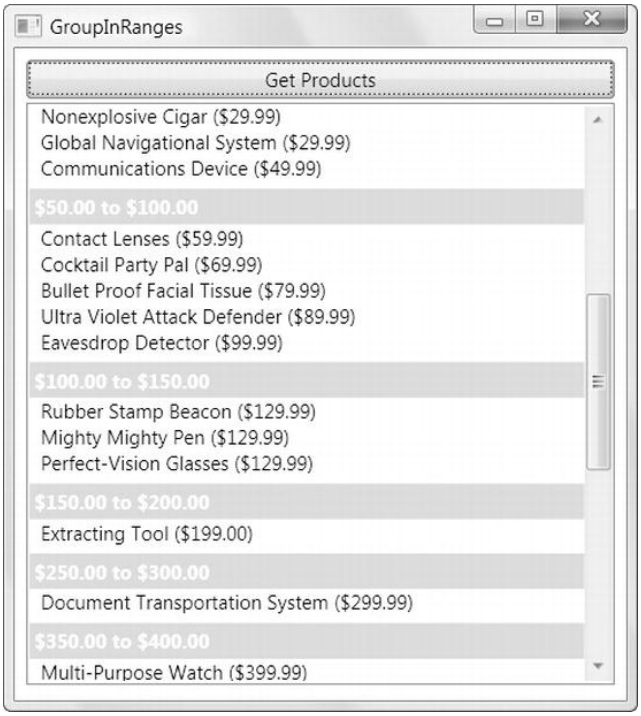

Figure 21-5. *Grouping in ranges*

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));
```

Grouping and Virtualization

In Chapter 19, you learned about virtualization, a feature that reduces the memory overhead of a control and increases its speed when you're binding extremely long lists. However, even if your control supports virtualization, it won't use it when you enable virtualization. WPF corrects this oversight with a new VirtualizingStackPanel.IsVirtualizingWhenGrouping property. Set it to true, and your grouped list gets the same performance-boosting virtualization as an ungrouped list:

```
<ListBox VirtualizingStackPanel.IsVirtualizingWhenGrouping="True" ...>
```

However, you should still be cautious about using grouping and long lists together, because the work involved to group your data can cause a significant slowdown. For that reason, you'll want to profile your application before you implement this design.

Live Shaping

If you make changes to the filtering, sorting, or grouping of a view that's already in use, you need to call ICollectionViewSource.Refresh() method to refresh the view and make sure the right items appears in the list. You've already seen an example that uses this technique: the price-filtering text box that triggers a refresh whenever the user modifies the minimum price limit.

However, some changes are a little trickier to catch. It's easy enough to remember to refresh a view when you're changing that view, but what if a code routine somewhere in your application changes your *data*? For example, imagine if an edit reduces the price of a product below the minimum that's required by your view's filter condition. Technically, this should cause the record to disappear from the current view, but unless you remember to force a refresh, you won't see any change.

WPF 4.5 introduces a feature called *live shaping* that fills this gap. Essentially, live shaping watches for changes in specific properties. If it detects a change (like a lowered price on a Product object), and it determines that the change affects the current view, it triggers a refresh.

To use live shaping, you need to meet three criteria:

- Your data object must implement INotifyPropertyChanged. It uses this interface to signal when its properties change. The current Product object does that already.
- Your collection must implement ICollectionViewLiveShaping. The standard ListCollectionView and BindingListCollectionView classes both implement ICollectionViewLiveShaping, which means you can use live shaping with any of the examples you've seen in this chapter.
- You must explicitly enable live shaping. You do this by setting several properties on the ListCollectionView or BindingListCollectionView object.

The last point is the most important. Live shaping adds extra overhead, so you need to opt-it to this feature when you need it. You do that with three separate properties: IsLiveFiltering, IsLiveSorting, and IsLiveGrouping. Each one enables live shaping for a different view feature. For example, if you set IsLiveFiltering to true but do not set the other two properties, the collection will check for changes that affect the currently set filtering conditions, but it will ignore changes that could affect the sorting or grouping of the list.

After you've enabled live shaping, you also need to tell the collection what properties to monitor. You do that by adding the property name (as a string) to one of three collection properties:

LiveFilteringProperties, LiveSortingProperties, or LiveGroupingProperties. Once again, this design is intended to ensure the best possible performance, by ignoring the properties that aren't important.

For example, consider the price-filtering product example. In this case, it makes sense to turn on IsLiveFiltering and monitor the Product.UnitCost property for changes, because that's the only property that can affect the filtering of the list. (Changes to other properties, like Description or ModelNumber, don't affect whether a product is filtered, and so they aren't important.) To use live shaping in this example, you'd add this code:

```
ListCollectionView view =
 CollectionViewSource.GetDefaultView(lstProducts.ItemsSource) as ListCollectionView;

view.IsLiveFiltering = true;
view.LiveFilteringProperties.Add("UnitCost");
```

Now try editing a record and reducing the price below the filter condition. The Product object reports the change, and the ListCollectionView notices it, reevaluates the condition, and then refreshes the view. The final result is that the low-priced record disappears automatically.

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.