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 a remarkably powerful tool 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.

### 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—at 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 by using the template and styling features of the ListBox. However, the View property is a useful abstraction. Here are some of its advantages:

Reusability 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 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.