CHAPTER 7

Configuring Data Binding

Most applications that access data do not simply deal with individual fields, but rather deal with lists and tables of data. Today, the relevant data for an application can come in the form of a traditional database, or it can be in a list contained in memory, data from an XML file, or any number of other formats. Binding your application to a variety of data sources is a crucial skill in the development of applications for today's world.

Once you have bound your data, manipulation of that data allows you to present it in a variety of ways to your application users. In this chapter, you will learn to bind to a variety of data sources, and then in your second lesson use data templates to format the display of your data, as well as sorting grouping and filtering data.

Exam objectives in this chapter:

- Implement data binding.
- Prepare collections of data for display.
- Bind to hierarchical data.
- Create a data template in WPF.

Lessons in this chapter:

Before You Begin

To complete the lessons in this chapter, you must have:

- A computer that meets or exceeds the minimum hardware requirements listed in the "About This Book" section at the beginning of the book.
- Microsoft Visual Studio 2010 Professional Edition installed on your computer.
- An understanding of Microsoft Visual Basic or C# syntax and familiarity with Microsoft .NET Framework 3.5.
- An understanding of Extensible Application Markup Language (XAML).

REAL WORLD

Matthew Stoecker

WPF takes data binding to a level that was unheard of in the Windows Forms world. Now I can create rich user interfaces that have properties bound to just about anything imaginable. Creating the user interface (UI) functionality by which manipulating one control changes the value of another is now simple—no more complicated event handling code. Just set the binding and you're ready to go!

Lesson 1: Binding to Data Sources

In the previous chapter, you saw how to bind a property to another property of an element or object. In this lesson, you learn how to bind properties to a variety of data sources: how to bind to and navigate a list, how to bind to ADO.NET data sources, and how to create a master-detail binding with hierarchical data. Then you learn how to use the *XMLDataSource* and *ObjectDataSource* classes to bind to XML and specialized data.

After this lesson, you will be able to:

- Bind an item control to a list of data.
- Bind a property to a list.
- Navigate a list.
- Bind to an ADO.NET object.
- Bind to hierarchical data.
- Use the *ObjectDataProvider* class to bind to objects and methods.
- Use the *XmlDataProvider* class to bind to XML.

Estimated lesson time: 30 minutes

Binding to a List

Frequently, you want to bind an element to a list of objects rather than just to a single object. There are two scenarios in this case: binding a single element to a list and enabling navigation of that list, and binding a collection (such as the items in *ListBox*) to a list so that the list's elements are all displayed at one time.

Binding an Item Control to a List

One common scenario in binding to a list of data or a collection is to display all items in that collection in a list-based element such as *ListBox*. Item controls have built-in properties that enable data binding. These are described in Table 7-1.

TABLE 7-1 Data-Related Properties of Item Controls

PROPERTY	DESCRIPTION
DisplayMemberPath	Indicates the property of the bound collection that will be used to create the display text for each item.
IsSynchronizedWithCurrentItem	Determines whether the selected item is kept synchronized with the *CurrentItem* property in the *Items* collection. Not strictly a data-related property, but useful for building master-detail views.

ItemsSource	Represents the collection that contains the items that make up the source of the list. You set this property to a *Binding* object that is bound to the appropriate collection.
ItemTemplate	The data template used to create the visual appearance of each item. Data templates will be discussed in Lesson 2 of this chapter, "Manipulating and Displaying Data."

For simple displaying of bound members, you must set the *ItemsSource* property to the collection to which you are binding and set the *DisplayMemberPath* property to the collection member that is to be displayed. More complex displays using data templates are discussed in Lesson 2. The following example demonstrates how to bind a *ListBox* collection to a static resource named *myList* and a display member called *FirstName*:

```
<ListBox Width="200" ItemsSource="{Binding Source={StaticResource myList}}"
   DisplayMemberPath="FirstName" />
```

A more common scenario when working with bound lists, however, is to bind to an object that is defined and filled with data in code. In this case, the best way to bind to the list is to set *DisplayMemberPath* in XAML and then set the *DataContext* property of the element or its container in code. The following example demonstrates how to bind a *ListBox* control to an object called *myCustomers* that is created at run time. The *ListBox* control displays the entries from the *CustomerName* property:

Sample of Visual Basic Code

```
' Code to initialize and fill myCustomers has been omitted
grid1.DataContext = myCustomers;
```

Sample of C# Code

```
// Code to initialize and fill myCustomers has been omitted
grid1.DataContext = myCustomers;
```

Sample of XAML Code

```
<!-- XAML -->

<Grid Name="grid1">
   <ListBox ItemsSource="{Binding}" DisplayMemberPath="CustomerName"
      Margin="92,109,66,53" Name="ListBox1" />
</Grid>
```

Note that in the XAML for this example, the *ItemsSource* property is set to a *Binding* object that has no properties initialized. The *Binding* object binds the *ItemsSource* property of *ListBox*, but because the *Source* property of the *Binding* object is not set, WPF searches upward through the visual tree until it finds a *DataContext* object that has been set. Because *DataContext* for *grid1* has been set in code to *myCustomers*, this then becomes the source for the binding.

Binding a Single Property to a List

You can also bind a single property to a list or collection. The process is much the same as binding an item control to a list except that, initially, only the first item in the list is displayed. You can navigate through the list to display subsequent items in the list. Navigation of lists is discussed in the next section. The following example demonstrates creating the data source object in code and setting the data context for a grid and then, in XAML, binding a *Label* object to the *FirstName* property of the items contained in the data source object:

Sample of Visual Basic Code

```
' Code to initialize and fill myCustomers omitted
grid1.DataContext = myCustomers;
```

Sample of C# Code

```
Code to initialize and fill myCustomers omitted
grid1.DataContext = myCustomers;
```

Sample of XAML Code

```
<!-- XAML -->
<Grid Name="grid1">
   <Label Content="{Binding Path=FirstName}" Height="23"
      Width="100"></Label>
</Grid>
```

Navigating a Collection or List

When individual properties, such as the *Content* property of *Label*, are bound to a collection, they are capable of displaying only a single member of that collection at a time. This is a common pattern seen in data access applications: A window might be designed to display one record at a time and have individual controls display each data column. Because only one record is displayed at a time, it becomes necessary to have a mechanism that enables the user to navigate through the records.

WPF has a built-in navigation mechanism for data and collections. When a collection is bound to a WPF binding, an *ICollectionView* interface is created behind the scenes. The *ICollectionView* interface contains members that manage data currency as well as views, grouping, and sorting, all of which are discussed in Lesson 2 of this chapter. Table 7-2 describes members involved in navigation.

TABLE 7-2 *ICollectionView* Members Involved in Navigation

MEMBER	DESCRIPTION
CurrentItem	This property returns the current item.
CurrentPosition	This property returns the numeric position of the current item.
IsCurrentAfterLast	This property indicates whether the current item is after the last item in the collection.

IsCurrentBeforeFirst	This property indicates whether the current item is before the first item in the collection.
MoveCurrentTo	This method sets the current item to the indicated item.
MoveCurrentToFirst	This method sets the current item to the first item in the collection.
MoveCurrentToLast	This method sets the current item to the last item in the collection.
MoveCurrentToNext	This method sets the current item to the next item in the collection.
MoveCurrentToPosition	This method sets the current item to the item at the designated position.
MoveCurrentToPrevious	This method sets the current item to the previous item.

You can get a reference to *ICollectionView* by calling the *CollectionViewSource.GetDefaultView* method, as shown here:

Sample of Visual Basic Code

```
' This example assumes a collection named myCollection
Dim myView As System.ComponentModel.ICollectionView
myView = CollectionViewSource.GetDefaultView (myCollection)
```

Sample of C# Code

```
// This example assumes a collection named myCollection
System.ComponentModel.ICollectionView myView;
myView = CollectionViewSource.GetDefaultView (myCollection);
```

When calling this method, you must specify the collection or list for which to retrieve the view (which is *myCollection* in the previous example). *CollectionViewSource.Get-DefaultView* returns an *ICollectionView* object that is actually one of three classes, depending on the class of the source collection.

If the source collection implements *IBindingList*, the view returned is a *BindingList-CollectionView* object. If the source collection implements *IList* but not *IBindingList*, the view returned is a *ListCollectionView* object. If the source collection implements *IEnumerable* but not *IList* or *IBindingList*, the view returned is a *CollectionView* object.

For navigational purposes, working with the *ICollectionView* should be sufficient. If you need to access members of the other classes, you must cast the *ICollectionView* object to the correct class.

When an item control or content control is bound to a data source, the default collection view handles currency and navigation by default. The current item in the view is returned by the *CurrentItem* property. This is the item currently displayed in all content controls bound to this view. Navigation backward and forward through the list is accomplished using the *MoveCurrentToNext* and *MoveCurrentToPrevious* methods, as shown here:

Sample of Visual Basic Code

```
Dim myView As System.ComponentModel.ICollectionView
myView = CollectionViewSource.GetDefaultView (myCollection)
' Sets the CurrentItem to the next item in the list
myView.MoveCurrentToNext()
' Sets the CurrentItem to the previous item in the list
myView.MoveCurrentToPrevious()
```

Sample of C# Code

```
System.ComponentModel.ICollectionView myView;
myView = CollectionViewSource.GetDefaultView (myCollection);
// Sets the CurrentItem to the next item in the list
myView.MoveCurrentToNext();
// Sets the CurrentItem to the previous item in the list
myView.MoveCurrentToPrevious();
```

Because item controls typically are bound to all the items in a list, navigation through the list is not usually essential. However, you can use an item control to enable the user to navigate records by setting *IsSynchronizedWithCurrentItem* to *True*, as shown in bold here:

```
<ListBox ItemsSource="{Binding}" DisplayMemberPath="City"
   Margin="92,109,66,53" Name="ListBox1"
   IsSynchronizedWithCurrentItem="True" />
```

When the *IsSynchronizedWithCurrentItem* property is set to *True*, the *SelectedItem* property of the item control always is synchronized with the *CurrentItem* property of the view. Thus, if the user selects an item in an item control, the *CurrentItem* property of the view is set to that item. This change then is reflected in all other controls bound to the same data source. This concept is illustrated in the lab for this lesson.

Binding to ADO.NET Objects

Binding to ADO.NET objects is basically the same as binding to any other collection or list. Because ADO.NET objects usually are initialized in code, the general pattern is to initialize the ADO.NET objects in code and set the data context for the user interface. Bindings in XAML should point to the appropriate ADO.NET object.

Setting *DataContext* to an ADO.NET *DataTable* Object

When setting *DataContext* to an ADO.NET *DataTable* object that contains the data to which you want to bind, set the *ItemsSource* property of any item controls you are binding to that table to an empty *Binding* object and specify the column to be displayed in the *DisplayMemberPath* property. For content controls or other single properties, you should set the appropriate property to a *Binding* object that specifies the appropriate column in the *Path* property. The following example demonstrates the code to initialize an ADO.NET data set and set the *DataContext*, and the XAML to bind a *ListBox* control and a *Label* control to that dataset:

Sample of Visual Basic Code

```
Public Class Window1
   Dim aset As NwindDataSet = New NwindDataSet()
   Dim custAdap As _
      NwindDataSetTableAdapters.CustomersTableAdapter = _
      New NwindDataSetTableAdapters.CustomersTableAdapter()
   Dim ordAdap As NwindDataSetTableAdapters.OrdersTableAdapter _
      = New NwindDataSetTableAdapters.OrdersTableAdapter()
   Public Sub New()
      InitializeComponent()
      custAdap.Fill(aset.Customers)
      OrdAdap.Fill(aset.Orders)
      Grid1.DataContext = aset.Customers
End Sub
```

Sample of C# Code

```
public partial class Window1 : Window
{
   NwindDataSet aset = new NwindDataSet();
   NwindDataSetTableAdapters.CustomersTableAdapter custAdap =
      new NwindDataSetTableAdapters.CustomersTableAdapter();
   NwindDataSetTableAdapters.OrdersTableAdapter ordAdap =
      new NwindDataSetTableAdapters.OrdersTableAdapter();
   public Window1()
   {
      InitializeComponent();
      custAdap.Fill(aset.Customers);
      ordAdap.Fill(aset.Orders);
      Grid1.DataContext = aset.Customers;
   }
}
```

Sample of XAML Code

```
<!-- XAML -->
<Grid Name="Grid1">
   <ListBox ItemsSource="{Binding}" DisplayMemberPath="ContactName"
      Name="listBox1" Width="100" Height="100" VerticalAlignment="Top" />

   <Label Content="{Binding Path=ContactTitle}" Height="23" Width="100"
      Name="label1" VerticalAlignment="Top"></Label>
</Grid>
```

Setting *DataContext* to an ADO.NET *DataSet* Object

You can also set *DataContext* to an ADO.NET *DataSet* object instead of to a *DataTable* object. Because a *DataSet* object is a collection of *DataTable* objects, you must provide the name of the *DataTable* object as part of the *Path* property. Examples are shown here:

Sample of Visual Basic Code

```
' The rest of the code is identical to the previous example
' and has been omitted.
Grid1.DataContext = aset
```

Sample of C# Code

```
// The rest of the code is identical to the previous example
// and has been omitted.
Grid1.DataContext = aset;

<!--XAML-->
<Grid Name="Grid1">
   <ListBox ItemsSource="{Binding Path=Customers}"
      DisplayMemberPath="ContactName" Name="listBox1" Width="100"
      Height="100" VerticalAlignment="Top" />
   <Label Content="{Binding Path=Customers/ContactTitle}" Height="23"
      Width="100" Name="label1" VerticalAlignment="Top"></Label>
</Grid>
```

Using the Visual Studio Designer Tools for ADO.NET

In Visual Studio 2010, substantial design time support has been added for working with ADO.NET in WPF applications. After you have added an ADO.NET data source to your application, you can navigate the available data fields in that data source in the Data Sources window, shown here in Figure 7-1.

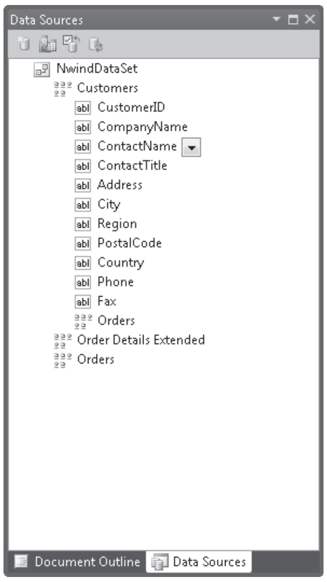


FIGURE 7-1 The Data Sources window.

The Data Sources window enables you to drag fields from your data source and drop them onto the design surface. When a data field is dropped onto the design surface, a bound control is automatically generated along with any required XAML code to create required data resources, such as *CollectionViewSource* objects. You can change the type of bound control that is created by clicking the field in the Data Sources window and selecting the control from the drop-down box. If the control you desire is not listed, you can add it to the default list by selecting Customize.

Binding to Hierarchical Data

When binding to lists of complex objects, you might want to create a master-detail view that enables the user to select one item from an upper-level list and view the details about the selected object. For lists of complex objects, this is as simple as setting the *Path* property of each detail binding to the correct property to display and making sure that the upper-level item controls in the user interface have the *IsSynchronizedWithCurrentItem* property set to *True* so that the selected item is set to the current item and the detail lists are updated automatically. The following example demonstrates a simple master-detail view. Each *Division* object contains a list of *Group* objects, each *Group* object contains a list of *Employee* objects, and all objects in this example have a *Name* property:

```
<Grid DataContext="{Binding Source={StaticResource Divisions}}">
   <StackPanel>
      <Label>Divisions</Label>
      <ListBox ItemsSource="{Binding}" DisplayMemberPath="Name"
         IsSynchronizedWithCurrentItem="True" />
   </StackPanel>

   <StackPanel>
      <Label Content="{Binding Path=Name}" />
      <ListBox ItemsSource="{Binding Path=Groups}" DisplayMemberPath="Name"
         IsSynchronizedWithCurrentItem="True" />
   </StackPanel>

   <StackPanel>
      <Label Content="{Binding Path=Groups/Name}" />
      <ListBox DisplayMemberPath="Name"
         ItemsSource="{Binding Path=Groups/Employees}" />
   </StackPanel>
</Grid>
```

For displaying hierarchical data, see Using Hierarchical Data Templates in Lesson 2.

Binding to Related ADO.NET Tables

When hierarchical data is retrieved from databases, it typically is presented in related tables. In ADO.NET, these relationships between tables are established by *DataRelation* objects, which link a column in one table to a column in another table. In WPF, you can retrieve related records by binding to the data relation. The pattern for creating a binding to a relation is as follows, where *ParentTable* represents a table, *Relation* represents a child relation of that table, and *relation2* represents a child relation of the data returned by *Relation*:

```
{Binding Path=ParentTable/Relation/Relation2}
```

You can bind through child relations only; you cannot bind up through parent relations.

The following example demonstrates initializing a data set with two related tables: *Customers* and *Orders*. The *Orders* table has a foreign key called *CustomerID* that relates to the primary key of the *Customers* table, also called *CustomerID*. The name of the relation in the database is *CustomersOrders*. The code shows the initializing of this data set and the setting of *DataContext* to the parent table. The XAML shows how to bind the related table to the data relation to view related records automatically:

Sample of Visual Basic Code

```
Public Class Window1
   Dim aset As NwindDataSet = New NwindDataSet()
   Dim custAdap As _
      NwindDataSetTableAdapters.CustomersTableAdapter = _
         New NwindDataSetTableAdapters.CustomersTableAdapter()
   Dim ordAdap As NwindDataSetTableAdapters.OrdersTableAdapter = _
      New NwindDataSetTableAdapters.OrdersTableAdapter()
   Public Sub New()
      InitializeComponent()
      custAdap.Fill(aset.Customers)
      OrdAdap.Fill(aset.Orders)
      Grid1.DataContext = aset.Customers
End Sub
```

Sample of C# Code

```
public partial class Window1 : Window
{
   NwindDataSet aset = new NwindDataSet();
   NwindDataSetTableAdapters.CustomersTableAdapter custAdap =
      new NwindDataSetTableAdapters.CustomersTableAdapter();
   NwindDataSetTableAdapters.OrdersTableAdapter ordAdap =
      new NwindDataSetTableAdapters.OrdersTableAdapter();
   public Window1()
   {
      InitializeComponent();
      custAdap.Fill(aset.Customers);
      ordAdap.Fill(aset.Orders);
      Grid1.DataContext = aset.Customers;
   }
}
```

Sample of XAML Code

```
<!-- XAML -->
<Grid Name="Grid1">
   <ListBox ItemsSource="{Binding}" DisplayMemberPath="ContactName"
      Name="listBox1" Width="100" Height="100" VerticalAlignment="Top" />
   <ListBox ItemsSource="{Binding Path=CustomersOrders}"
      DisplayMemberPath="OrderID" Height="100" Width="100"
      Name="listBox2" VerticalAlignment="Bottom" />
</Grid>
```

Binding to an Object with *ObjectDataProvider*

The *ObjectDataProvider* class enables you to bind a WPF element or property to a method called on an object. You can specify an object type and a method on that type, then bind to the results of that method call. Table 7-3 lists the important properties of *ObjectDataProvider*.

TABLE 7-3 Important Properties of *ObjectDataProvider*

PROPERTY	DESCRIPTION
ConstructorParameters	Represents the list of parameters to pass to the constructor.
IsAsynchronous	Indicates whether object creation and method calls are performed on the foreground thread or on a background thread.
MethodName	Represents the name of the method of the source object to call.
MethodParameters	Represents the list of parameters to pass to the method specified by the *MethodName* property.
ObjectInstance	Gets or sets the object used as the binding source.
ObjectType	Gets or sets the type of object of which to create an instance.

The following example demonstrates how to create a simple *ObjectDataProvider* object for an object with a method that takes no parameters:

```
<Window.Resources>
  <ObjectDataProvider x:Key="myObjectProvider" ObjectType="{x:Type
      local:myObject}" MethodName="GetCollection" />
</Window.Resources>
```

You can also bind to this *ObjectDataProvider* object, using a *Binding* object, as shown here:

```
<ListBox Name="ListBox1" DisplayMemberPath="ItemID"
   ItemsSource="{Binding Source={StaticResource myObjectProvider}}" />
```

If the method requires parameters, you can provide them in the *MethodParameters* property. Likewise, if the constructor requires parameters, they are provided in the *ConstructorParameters* property. Examples are shown here:

```
<Window.Resources>
   <ObjectDataProvider x:Key="myObjectProvider"
      ObjectType="{x:Type local:myObject}" MethodName="GetCollection">
      <ObjectDataProvider.ConstructorParameters>
         <system:Double>12</system:Double>
      </ObjectDataProvider.ConstructorParameters>
      <ObjectDataProvider.MethodParameters>
         <system:String>Items</system:String>
      </ObjectDataProvider.MethodParameters>
   </ObjectDataProvider>
</Window.Resources>
```

Although you can bind to data presented by *ObjectDataProvider,* you cannot update data; that is, the binding is always going to be read-only.

Binding to XML Using *XmlDataProvider*

The *XmlDataProvider* class enables you to bind WPF elements to data in the XML format. Important properties of the *XmlDataProvider* class are shown in Table 7-4.

TABLE 7-4 Important Properties of *XmlDataProvider*

PROPERTY	DESCRIPTION
Document	Gets or sets the *XmlDocument* object to be used as the binding source.
Source	Gets or sets the Uniform Resource Indicator (URI) of the XML file to be used as the binding source.
XPath	Gets or sets the *XPath* query used to generate the XML data.

The following example demonstrates an *XmlDataProvider* object providing data from a source file called Items.xml:

```
<Window.Resources>
   <XmlDataProvider x:Key="Items" Source="Items.xml" />
</Window.Resources>
```

You can also provide the XML data inline as an XML data island. In this case, you wrap the XML data in *XData* tags, as shown here:

```
<Window.Resources>
   <XmlDataProvider x:Key="Items">
      <x:XData>
         <!--XML Data omitted-->
      </x:XData>
   </XmlDataProvider>
</Window.Resources>
```

You can bind elements to the data provided by *XmlDataProvider* in the same way you would bind to any other data source: namely, using a *Binding* object and specifying the *XmlDataProvider* object in the *Source* property, as shown here:

```
<ListBox ItemsSource="{Binding Source={StaticResource Items}}"
   DisplayMemberPath="ItemName" Name="listBox1" Width="100" Height="100"
   VerticalAlignment="Top" />
```

Using *XPath* with *XmlDataProvider*

You can use *XPath* expressions to filter the results exposed by *XmlDataProvider* or to filter the records displayed in the bound controls. By setting the *XPath* property of *XmlDataProvider* to an *XPath* expression, you can filter the data provided by the source. The following example filters the results exposed by an *XmlDataProvider* object to include only those nodes called *<ExpensiveItems>* in the *<Items>* top-level node:

```
<Window.Resources>
   <XmlDataProvider x:Key="Items" Source="Items.xml"
   XPath="Items/ExpensiveItems" />
</Window.Resources>
```

You can also apply *XPath* expressions in the bound controls. The following example sets the *XPath* property to *Diamond* (shown in bold), which indicates that only data contained in *<Diamond>* tags will be bound:

```
<ListBox ItemsSource="{Binding Source={StaticResource Items}
   XPath=Diamond}" DisplayMemberPath="ItemName" Name="listBox1" Width="100"
   Height="100" VerticalAlignment="Top" />
```

EXAM TIP

There are different methods for binding to different data sources. Be sure you understand how to bind not only to lists of data but also to ADO.NET objects, *ObjectDataProvider* objects, and *XmlDataProvider* objects because each one has its own individual subtleties.

Accessing a Database

In this lab, you create an application to view related data in a database. You use the Northwind sample database and create an application that enables you to choose a contact name, view a list of dates on which orders were placed, and see the details about shipping and which products were ordered. In Lesson 2, you will build on this application.

EXERCISE 1 Adding a Data Source

1. From the top-level folder of the companion CD, copy the Nwind database to a convenient location.
2. In Visual Studio, open a new WPF Application project.
3. In Visual Studio, from the Data menu choose Add New Data Source. The Data Source Configuration Wizard opens, with Database selected in the first page. Click Next to continue.
4. On the Choose A Database Model page, select Dataset and click Next.
5. On the Choose Your Data Connection page, click New Connection to open the Add Connection dialog box. Change Data Source to Microsoft Access Database File, click Continue, and then click Browse to browse to the location where you copied your Nwind database file. After selecting your database file, click Open. In the Add Connection dialog box, click Test Connection to test the connection. If the test is successful, click OK, click OK in the Add Connection dialog box, click Next in the Choose

Your Data Connection dialog box, and then click Yes in the pop-up window. Click Next again.

6. On the Choose Your DataBase Objects page, expand the Tables node and select the Customers table and the Orders table. Expand the Views node, select Order Details Extended, and click Finish. A new class called *NwindDataSet* is added to your solution, and table adapters are created.
7. In Solution Explorer, double-click NwindDataSet.xsd to open the Dataset Designer. Right-click the Dataset Designer; choose Add and then choose Relation to open the Relation window.
8. In the Relation window, set Parent Table to Orders and set Child Table to Order Details Extended. Verify that Key Columns and Foreign Key Columns are set to OrderID. Click OK to create the new data relation.
9. From the Build menu, choose Build Solution to build and save your solution.

EXERCISE 2 Binding to Relational Data

1. In the designer, add the namespace attribute (shown bolded in the following code) to the Window element to import your project. Please note that, depending on the name of your project, you will have to update the namespace of this project.

```
<Window x:Class="Window1"
          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
          Title="Window1" Height="350" Width="523" xmlns:my="clr-namespace:Lesson_1">
</Window.Resources>
```

2. From the Data menu, choose Show Data Sources to open the Data Sources window. Expand the Customers node.
3. Click one of the fields—for example, CompanyName—under the Customers node. Open the drop-down box and select Customize. The Customize Control Binding window opens.
4. In the Customize Control Binding window, set the Data Type drop-down box to String and ensure that the box next to ListBox is selected. Repeat this procedure with the Data Type drop-down box set to DateTime. Click OK.
5. In the Data Sources window, under the Customers node, open the drop-down box for ContactName and then select ListBox.
6. From the Data Sources window, drag the ContactName node to the design surface. A bound list box with an appropriate label is created.
7. In the Data Sources window, within the Customers node, expand the Orders node and open the drop-down box for OrderDate and select ListBox.
8. From the Data Sources Window, drag the OrderDate node to the Design surface to create a bound list box.

9. In the Data Sources window, within the Customers node and within the Orders node, open the Order Details Extended node. Open the drop-down box for ProductName and select ListBox.
10. From the Data Sources window, drag the ProductName node to the design surface to create a bound list box.
11. In the Data Sources window, within the Customers node, under the Orders node, drag the ShipName, ShipAddress, ShipCity, and ShipCountry nodes to the design surface to create bound text boxes.
12. Press F5 to run your application. Note that when a Contact Name is selected in the first list box, the other bound controls automatically sync to the selected item. Similarly, if an order date is selected in the second list box, the items in the third list (for product name) box are synchronized with the selected item.

Lesson Summary

- When binding an item control to a list of data, you must set the *ItemsSource* property to a *Binding* class that specifies the source of the list. *DisplayMemberPath* indicates which property of the list should be displayed in the item control.
- Individual properties can be bound to lists of data. Initially, the first member of any such bound list is displayed, but the list can be navigated through the default *ICollectionView* property retrieved through the *CollectionViewSource* object.
- When binding to hierarchical data, set the *IsSynchronizedWithCurrentItem* property on each item control to *True* to enable automatic UI updates of hierarchical data. You can set a *Binding* class to detail lists by specifying the detail list through the *Path* property.
- You can bind to the value returned by a method through *ObjectDataProvider.*
- You can bind to XML with *XmlDataProvider. XmlDataProvider* requires XML as a file, an *XmlDocument* object, or a data island.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Binding to Data Sources." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE **ANSWERS**

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

1. Which of the following code samples shows a *ListBox* object correctly bound to the *CustomerAddress* field of a data table? The data table is named *Customers* in an ADO.

NET data set named *mySet*. Take both XAML and code into account in the possible answers.

A.

Sample of Visual Basic Code

```
Grid1.DataContext=mySet
```

Sample of C# Code

```
Grid1.DataContext=mySet;
```

Sample of XAML Code

```
<!-- XAML -->
<Grid Name="Grid1">
   <ListBox ItemsSource="{Binding}"
   DisplayMemberPath="CustomerAddress"
   Name="listBox1" Width="100" Height="100" VerticalAlignment="Top" />
</Grid>
```

B.

Sample of Visual Basic Code

```
Grid1.DataContext=mySet
```

Sample of C# Code

```
Grid1.DataContext=mySet;
```

Sample of XAML Code

```
<!-- XAML -->
<Grid Name="Grid1">
<ListBox ItemsSource="{Binding Path=mySet.Customers}"
   DisplayMemberPath="CustomerAddress"
   Name="listBox1" Width="100" Height="100" VerticalAlignment="Top" />
</Grid>
```

C.

Sample of Visual Basic Code

```
Grid1.DataContext=mySet.Customers
```

Sample of C# Code

```
Grid1.DataContext=mySet.Customers;
```

Sample of XAML Code

```
<!-- XAML -->
<Grid Name="Grid1">
<ListBox ItemsSource="{Binding}"
   DisplayMemberPath="CustomerAddress"
   Name="listBox1" Width="100" Height="100" VerticalAlignment="Top" />
</Grid>
```

D.

Sample of Visual Basic Code

```
Grid1.DataContext=mySet.Customers
```

Sample of C# Code

```
Grid1.DataContext=mySet.Customers;
```

Sample of XAML Code

```
<!-- XAML -->
<Grid Name="Grid1">
<ListBox ItemsSource="{Binding}"
   DisplayMemberPath="Customers/CustomerAddress"
   Name="listBox1" Width="100" Height="100" VerticalAlignment="Top" />
</Grid>
```

2. You're binding a *ListBox* object to the Price field in an ADO.NET data table called Details. This table is related to the Orders table through a parent relation called OrdersDetails. The Orders table is related to a table called Customers through a parent relation called CustomersOrders. The *DataContext* property for your list box has been set to the Customers table, which is the parent table of the CustomersOrders relation. Which of the following correctly binds *ListBox* to the *Price* field?

A.

```
<ListBox ItemsSource="{Binding}"
   DisplayMemberPath="Price" Name="listBox1"
   Width="100" Height="100" VerticalAlignment="Top" />
```

B.

```
<ListBox ItemsSource="{Binding Path=OrdersDetails}"
   DisplayMemberPath="Price" Name="listBox1"
   Width="100" Height="100" VerticalAlignment="Top" />
```

C.

```
<ListBox ItemsSource="{Binding Path=CustomersOrders/OrdersDetails}"
   DisplayMemberPath="Price" Name="listBox1"
   Width="100" Height="100" VerticalAlignment="Top" />
```

D.

```
<ListBox ItemsSource="{Binding Path=Customers/CustomersOrders/OrdersDetails}"
   DisplayMemberPath="Price" Name="listBox1"
   Width="100" Height="100" VerticalAlignment="Top" />
```

Lesson 2: Manipulating and Displaying Data

WPF has built-in functionality for sorting, grouping, and filtering data. In addition, it provides unprecedented support for customizing data display through data template technology. In this lesson, you learn to apply sorting and grouping through the *ICollectionView* interface. You learn to create and implement custom filters for data displayed in WPF, and you learn to use data templates to customize the appearance of data in your user interface.

After this lesson, you will be able to:

- Create and implement a data template.
- Sort data through *ICollectionView*.
- Apply grouping to data with *ICollectionView*.
- Use an *IComparer* to create a custom sorting scheme.
- Apply a custom filter to bound data.
- Apply a filter to a bound ADO.NET object.
- Implement and use a *DataTemplateSelector* class.
- Create and use *HierarchicalDataTemplate*.

Estimated lesson time: 30 minutes

Data Templates

So far, you have seen how to bind content controls and item controls to data to display bound data in the user interface. The results, however, have been underwhelming. You have seen how to present data fields as simple text in content controls and lists of text in item controls. You can create a rich data presentation experience, however, by incorporating data templates into your UI design.

A *data template* is a bit of XAML that describes how bound data is displayed. A data template can contain elements that are each bound to a data property, along with additional markup that describes layout, color, and other aspects of appearance. The following example demonstrates a simple data template that describes a *Label* element bound to the *ContactName* property. The *Foreground*, *Background*, *BorderBrush*, and *BorderThickness* properties are also set:

```
<DataTemplate>
   <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
      Background="Yellow" BorderThickness="3" Foreground="Blue" />
</DataTemplate>
```

Figure 7-2 shows a list box displaying a list of bound data. Figure 7-3 shows the same list box displaying the same bound data with the previously cited data template set to the *ListBox.ItemsTemplate* property.

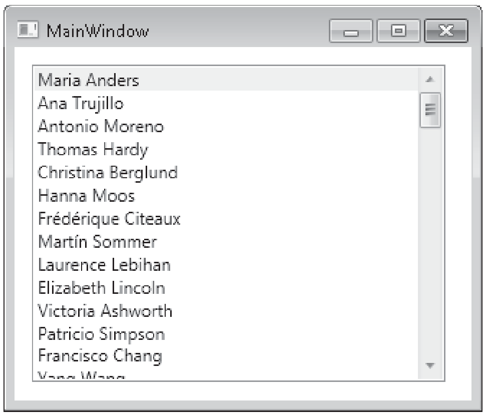


FIGURE 7-2 A bound list box without a data template.

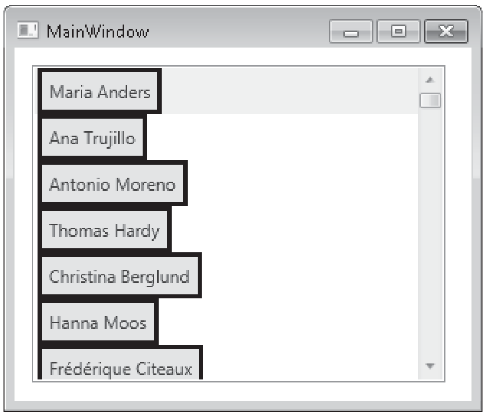


FIGURE 7-3 A bound list box with a data template applied.

When binding a property or list directly to a control, you are limited to binding a single property. With data templates, however, you can bind more than one property in each item, thereby displaying multiple bits of related data together. The following example demonstrates this concept. It is a modification of the template shown in the previous example, but a header is added to each *ContactName* item that consists of the text "Company Name:" and the *CompanyName* value associated with that contact name. Additional properties are set to provide style, and both labels are placed within a *StackPanel* class to facilitate layout:

```
<DataTemplate>
   <StackPanel>
      <Label Background="Purple" Foreground="White" BorderBrush="Red"
         BorderThickness="4">
         <Label.Content>
            <WrapPanel HorizontalAlignment="Stretch">
               <TextBlock>Company Name: </TextBlock>
               <TextBlock Text="{Binding CompanyName}" />
            </WrapPanel>
         </Label.Content>
      </Label>
      <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
```

```
        HorizontalAlignment="Stretch" Background="Yellow"
        BorderThickness="3" Foreground="Blue" />
  </StackPanel>
</DataTemplate>
```

Figure 7-4 shows the results of applying this data template.

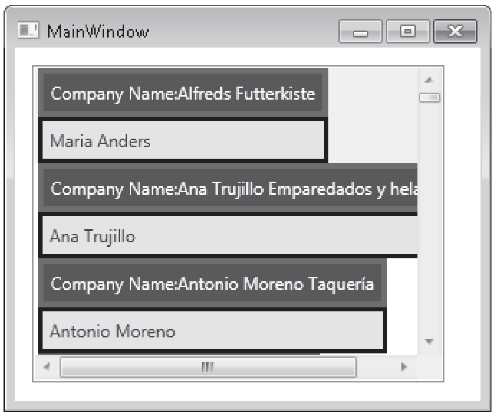

MainWindow
Company Name:Alfreds Futterkiste
Maria Anders
Company Name:Ana Trujillo Emparedados y hela
Ana Trujillo
Company Name:Antonio Moreno Taquería
Antonio Moreno

FIGURE 7-4 A bound list box with a data template that includes a header with related data.

You can apply data templates to content controls as well. Although a content control can display only a single record at a time, it can use all the formatting features of the data template technology.

Setting the Data Template

You set the data template on a control by setting one of two properties. For content controls, you set the *ContentTemplate* property, as shown in bold here:

```
<Label Height="23" HorizontalAlignment="Left" Margin="56,0,0,91"
  Name="label1" VerticalAlignment="Bottom" Width="120">
  <Label.ContentTemplate>
    <DataTemplate>
      <!--Actual data template omitted-->
    </DataTemplate>
  </Label.ContentTemplate>
</Label>
```

For item controls, you set the *ItemsTemplate* property, as shown in bold here:

```
<ListBox ItemsSource="{Binding}" IsSynchronizedWithCurrentItem="True"
  Margin="18,19,205,148" Name="listBox1">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <!--Actual data template omitted-->
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Note that for item controls, the *DisplayMemberPath* and *ItemTemplate* properties are mutually exclusive; you can set one but not the other.

A frequent pattern with data templates is to define them in a resource collection and reference them in your element rather than defining them inline as shown in the previous examples. All that is required to reuse a data template in this manner is to define the template in a resource collection and set a *Key* property for the template, as shown here:

```
<Window.Resources>
   <DataTemplate x:Key="myTemplate">
      <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
         Background="Yellow" BorderThickness="3" Foreground="Blue" />
   </DataTemplate>
</Window.Resources>
```

Then you can set the template by referring to the resource, as shown in bold here:

```
<ListBox ItemTemplate="{StaticResource myTemplate}"
   Name="ListBox1" />
```

EXAM TIP

Although data templates appear to be rather simple, be sure that you have lots of practice implementing and using them. Data templates are an extremely powerful feature of WPF and promise to figure prominently on the exam.

Using Converters to Apply Conditional Formatting in Data Templates

One of the most useful things you can do with converters is to apply conditional formatting to displayed data. For example, suppose you are writing an application that binds to a list of dates on which orders were placed. You might want orders that were placed in the current month to have a different foreground color than the other orders. You can accomplish this by binding the *Foreground* property of the control used to bind the date to the *Date* field and providing a converter that evaluates the date and returns a brush of the appropriate color. The following examples show such a converter, an instance of that converter added to the *Window.Resources* collection, and, finally, the *Foreground* property bound to the *Date* property using the converter to return a *Brush* object. First, here's the converter:

Sample of Visual Basic Code

```
<ValueConversion(GetType(DateTime), GetType(Brush))> _
Public Class DateBrushConverter
   Implements IValueConverter

   Public Function Convert(ByVal value As Object, ByVal targetType As  _
      System.Type, ByVal parameter As Object, ByVal culture As  _
      System.Globalization.CultureInfo) As Object Implements _
      System.Windows.Data.IValueConverter.Convert
       Dim aDate As DateTime
       aDate = CType(value, DateTime)
       If aDate.Month = Now.Month Then
          Return New SolidColorBrush(Colors.Red)
       Else
          Return New SolidColorBrush(Colors.Black)
       End If
```

```
    End Function

    Public Function ConvertBack(ByVal value As Object, ByVal targetType _
       As System.Type, ByVal parameter As Object, ByVal culture As  _
       System.Globalization.CultureInfo) As Object Implements _
         System.Windows.Data.IValueConverter.ConvertBack
        Throw New NotImplementedException()
    End Function
End Class
```

Sample of C# Code

```
[ValueConversion(typeof(DateTime), typeof(Brush))]
public class DateBrushConverter : IValueConverter
{
   public object Convert(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
   {
       DateTime aDate = (DateTime)value;
       if (aDate.Month == DateTime.Now.Month)
          return new SolidColorBrush(Colors.Red);
       else
          return new SolidColorBrush(Colors.Black);
   }

   public object ConvertBack(object value, Type targetType, object
      parameter, System.Globalization.CultureInfo culture)
   {
      throw new NotImplementedException();
   }
}
```

An instance is then added to the *Window.Resources* collection, as shown in bold here:

```
<Window x:Class="WpfApplication5.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:local="clr-namespace:WpfApplication5"
   Title="Window1" Height="300" Width="300">
   <Window.Resources>
       <local:DateBrushConverter
           x:Key="myDateConverter"></local:DateBrushConverter>
   </Window.Resources>
   <!--- the rest of the window is omitted --!>
</Window>
```

Finally, you can bind the *Foreground* property to a date by using the converter, as shown in bold here:

```
<ListBox Margin="60,58,98,104" Name="ListBox1">
   <ListBox.ItemTemplate>
      <DataTemplate>
         <Label Content="{Binding Path=OrderDate}" Foreground="{Binding
            Path=OrderDate, Converter={StaticResource myDateConverter}}" />
      </DataTemplate>
   </ListBox.ItemTemplate>
</ListBox>
```

Using *DataTemplateSelector*

The *DataTemplateSelector* class enables you to assign data templates dynamically to collection objects based on the content of the data. For example, consider that you have two data templates, shown in XAML here:

```
<Window.Resources>
        <DataTemplate x:Key="BasicTemplate">
            <TextBlock Text="{Binding Path=RequiredDate}"/>
        </DataTemplate>
        <DataTemplate x:Key="PriorityTemplate">
            <TextBlock Text="{Binding Path=RequiredDate}" Background="Red"/>
        </DataTemplate>
</Window.Resources>
```

Both templates in this example are quite simple. They each define a text block that displays a field called RequiredDate—in this example, the date an order is required by the customer. However, in the template called PriorityTemplate, the background of the text block is red. Suppose you want your application to examine the value of the RequiredDate field and use PriorityTemplate for orders that are due this month and BasicTemplate for all others. You can implement this logic by using the *DataTemplateSelector* class.

DataTemplateSelector is an abstract class that exposes a single method (which can be overridden) called *SelectTemplate*. *SelectTemplate* returns a *DataTemplate* object and, when overridden, should incorporate the logic required to determine the correct template based on the data. The *SelectTemplate* method accepts two parameters: an *Object* parameter that represents the object bound by the data template, and a *DependencyObject* parameter that represents the container for the bound object. The following example shows a simple implementation of *DataTemplateSelector* that examines a field in a data row and returns a data template based on the value of that field.

Sample of Visual Basic Code

```
Public Class DateDataTemplateSelector
    Inherits DataTemplateSelector
    Public Overrides Function SelectTemplate(ByVal item As Object, ByVal container As
DependencyObject) _
        As DataTemplate
        Dim element As FrameworkElement
        element = TryCast(container, FrameworkElement)
        ' Tests to ensure that both the item and the container are not null, and that
        ' the item is of the expected data type.
        If element IsNot Nothing AndAlso item IsNot Nothing AndAlso TypeOf item Is
System.Data.DataRowView Then
            ' Casts the item as the expected type, in this case a System.Data.
DataRowView
            Dim dateitem As System.Data.DataRowView = CType(item, System.Data.
DataRowView)
            ' Compares the value of the expected field with the current month
            If CType(dateitem("RequiredDate"), Date).Month = DateTime.Now.Month Then
                ' Returns the PriorityTemplate in the case of a match
                Return TryCast(element.FindResource("PriorityTemplate"), DataTemplate)
            Else
                ' Returns the BasicTemplate for other cases
                Return TryCast(element.FindResource("BasicTemplate"), DataTemplate)
            End If
        End If
        Return Nothing
    End Function
End Class
```

Sample of C# Code

```
public class DateDataTemplateSelector : DataTemplateSelector
{
    public override DataTemplate SelectTemplate(object item, DependencyObject container)
    {
     FrameworkElement element;
     System.Data.DataRowView dateitem;
     element = container as FrameworkElement;
     // Tests to ensure that both the item and the container are not null, and that the
     // item is of the expected data type.
     if (element != null && item != null && item is System.Data.DataRowView)
         {
         // Casts the item as the expected type, in this case a System.Data.DataRowView
         dateitem = (System.Data.DataRowView)item;
         // Compares the value of the expected field with the current month
         if (((DateTime)dateitem["RequiredDate"]).Month == DateTime.Now.Month)
             // Returns the PriorityTemplate in the case of a match
             return element.FindResource("PriorityTemplate") as DataTemplate;
         else
             // Returns the BasicTemplate for other cases
             return element.FindResource("BasicTemplate") as DataTemplate;
         }
     return null;
    }
}
```

After you have created your inherited *DataTemplateSelector* class, you can create an instance of it in your Resources section, as shown here:

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:my="clr-namespace:WpfApplication4"
    Title="MainWindow" Height="350" Width="525" >
    <Window.Resources>
        <my:DateDataTemplateSelector x:Key="myTemplateSelector" />
    </Window.Resources>
    <!--  Implementation omitted -->
</Window>
```

After you have defined a *DataTemplateSelector* resource, you can assign the resource to the *ItemTemplateSelector* property of an item control, as shown here:

```
<ListBox ItemTemplateSelector="{StaticResource myTemplateSelector}" />
```

When an item is added to the list box in this example, the *SelectTemplate* method of *DataTemplateSelector* is executed on that item, and the appropriate template is returned and applied to the display of that item.

Using Hierarchical Data Templates

When displaying hierarchical data, such as in a *TreeView* or a *Menu* control, you can use *HierarchicalDataTemplate* to provide formatting and layout for both items in a list and related sub-items. The *HierarchicalDataTemplate* class incorporates all the properties of the *DataTemplate* class and serves as the standard data template for list items. *HierarchicalDataTemplate* also incorporates a number of properties that apply to sub-items, which Table 7-5 summarizes.

TABLE 7-5 Item-Related Properties of *HierarchicalDataTemplate*

PROPERTY	DESCRIPTION
ItemBindingGroup	Gets or sets the *BindingGroup* property that is copied to each child item
ItemContainerStyle	Gets or sets the *Style* property that is applied to the item container for each child item
ItemContainerStyleSelector	Gets or sets custom style-selection logic for a style that can be applied to each item container
ItemsSource	Gets or sets the binding for this data template, which indicates where to find the collection that represents the next level in the data hierarchy
ItemStringFormat	Gets or sets a composite string that specifies how to format the items in the next level in the data hierarchy if they are displayed as strings
ItemTemplate	Gets or sets the data template to apply to the *ItemTemplate* property on a generated *HeaderedItemsControl* control (such as *MenuItem* or *TreeViewItem*) to indicate how to display items from the next level in the data hierarchy
ItemTemplateSelector	Gets or sets *DataTemplateSelector* to apply to the *ItemTemplateSelector* property on a generated *HeaderedItemsControl* control (such as *MenuItem* or *TreeViewItem*) to indicate how to select a template to display items from the next level in the data hierarchy

The *ItemsSource* property is a binding that points to the sub-items to be displayed in the hierarchical display. The *ItemTemplate* property represents the data template for the sub-items. Normally, this is a regular data template, but note that sub-items can have sub-items of their own, in which case you can set this property to another *HierarchicalDataTemplate* class.

If you want to examine the sub-items programmatically and provide different data templates based on the data values, you can set the *ItemTemplateSelector* property to an instance of a class deriving from *DataTemplateSelector*, as described in the previous section.

The following XAML code demonstrates an example of *HierarchicalDatatemplate*. In this example, a window defines two CollectionViewResource objects, which represent an ADO.NET *DataTable* object and a *DataRelation* object that defines the relationship between it and a related table. The user interface of the window contains a *TreeView* control that will display a list of items, each of which will have sub-items. *HierarchicalDataTemplate*, in this example, is applied to the items that are displayed in *ListView*, and the *ItemTemplate* property of *HierarchicalDataView* refers to the DataTemplate class applied to sub-items in *ListView*.

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:my="clr-namespace:WpfApplication4"
    Title="MainWindow" Height="350" Width="525" >
    <Window.Resources>
        <!-- Creates a resource that refers to a DataSet defined in code -->
        <my:NwindDataSet x:Key="NwindDataSet" />
        <!-- Creates a resource that refers to the Customers table of the DataSet -->
        <CollectionViewSource x:Key="CustomersViewSource" Source="{Binding
Path=Customers,
            Source={StaticResource NwindDataSet}}" />
        <!-- Creates a resource that refers to a DataRelation defined by the Customers
table -->
        <CollectionViewSource x:Key="CustomersOrdersViewSource" Source="{Binding
Path=CustomersOrders,
            Source={StaticResource CustomersViewSource}}" />
        <!-- The DataTemplate that is applied to sub-items in the list view -->
        <DataTemplate x:Key="BasicTemplate">
            <TextBlock Text="{Binding Path=RequiredDate}"/>
        </DataTemplate>
        <!-- The HierarchicalDataTemplate that is applied to Listview items -->
        <HierarchicalDataTemplate x:Key="HierarchTemplate"  ItemsSource="{Binding
Path=CustomersOrders}"
            ItemTemplate="BasicTemplate" >
            <TextBlock Foreground="Red" Text="{Binding Path=CompanyName}" />
          </HierarchicalDataTemplate>
    </Window.Resources>
    <Grid DataContext="{StaticResource CustomersViewSource}">
        <TreeView Height="248" HorizontalAlignment="Left" Margin="46,39,0,0"
Name="TreeView1"
            VerticalAlignment="Top" Width="394" ItemsSource="{Binding}"
            ItemTemplate="{StaticResource HierarchTemplate}" />
    </Grid>
</Window>
```

Sorting Data

When presenting data in the user interface, you want to sort it in various ways. Bound data can be sorted through the default *ICollectionView* element for the data list. You saw previously how to obtain a reference to the default collection view through the *CollectionViewSource* object, as shown again here:

Sample of Visual Basic Code

```
Dim myView As System.ComponentModel.ICollectionView
myView = CollectionViewSource.GetDefaultView(myCollection)
```

Sample of C# Code

```
System.ComponentModel.ICollectionView myView;
myView = CollectionViewSource.GetDefaultView(myCollection);
```

ICollectionView exposes a property called *SortDescriptions*, which contains a collection of *SortDescription* objects. *SortDescription* objects describe the column name to sort by and a direction that specifies either an ascending or a descending sort order. The following example demonstrates sorting by a column named LastName in ascending order:

Sample of Visual Basic Code

```
myView.SortDescriptions.Add(New _
   System.ComponentModel.SortDescription("LastName", _
   System.ComponentModel.ListSortDirection.Ascending)
```

Sample of C# Code

```
myView.SortDescriptions.Add(new
   System.ComponentModel.SortDescription("LastName",
   System.ComponentModel.ListSortDirection.Ascending);
```

SortDescription objects are applied to a collection in the order in which they are added to the collection view. Thus, if you wanted to sort first by the LastName column and then by the FirstName column, you first would add a *SortDescription* object that specified sorting by *LastName* and then add a *SortDescription* object that specified sorting by *FirstName*, as shown here:

Sample of Visual Basic Code

```
myView.SortDescriptions.Add(New _
   System.ComponentModel.SortDescription("LastName", _
   System.ComponentModel.ListSortDirection.Ascending)
myView.SortDescriptions.Add(New _
   System.ComponentModel.SortDescription("FirstName", _
   System.ComponentModel.ListSortDirection.Ascending)
```

Sample of C# Code

```
myView.SortDescriptions.Add(new
   System.ComponentModel.SortDescription("LastName",
   System.ComponentModel.ListSortDirection.Ascending);
myView.SortDescriptions.Add(new
   System.ComponentModel.SortDescription("FirstName",
   System.ComponentModel.ListSortDirection.Ascending);
```

Applying Custom Sorting

If you are binding to a class whose default view is *ListCollectionView* (that is, it implements *IList* but not *IBindingList*), such as *ObservableCollection*, you can implement custom sort orders by setting the *ListCollectionView.CustomSort* property. The *CustomSort* property takes an object that implements the *IComparer* interface. The *IComparer* interface requires a single method called *Compare* that takes two arguments and returns an integer that represents the result of the comparison of those two objects. The following example demonstrates an implementation of the *IComparer* interface that takes two *Employee* objects and sorts by the length of the *LastName* property:

Sample of Visual Basic Code

```
Public Class myComparer
   Implements IComparer

   Public Function Compare(ByVal x As Object, ByVal y As Object) As _
      Integer Implements System.Collections.IComparer.Compare
      Dim empX As Employee = CType(x, Employee)
      Dim empY As Employee = CType(y, Employee)
      Return empX.LastName.Length.CompareTo(empY.LastName.Length)
   End Function
End Class
```

Sample of C# Code

```
public class myComparer : System.Collections.IComparer
{
   public int Compare(object x, object y)
   {
      Employee empX = (Employee)x;
      Employee empY = (Employee)y;
      return empX.LastName.Length.CompareTo(empY.LastName.Length);
   }
}
```

Then you can create an instance of this class to set to the *CustomSort* property of your *ICollectionView* element, as shown here:

Sample of Visual Basic Code

```
Dim myView As ListCollectionView
myView = CType(CollectionViewSource.GetDefaultView(myCollection), _
   ListCollectionView)
myView.CustomSort = New myComparer()
```

Sample of C# Code

```
System.ComponentModel.ICollectionView myView;
myView =
   (ListCollectionView)CollectionViewSource.GetDefaultView(myCollection);
myView.CustomSort = new myComparer();
```

Grouping

ICollectionView also supports grouping data. Grouping data is similar to sorting but takes advantage of the built-in functionality of item controls to provide formatting for different groups, thus enabling the user to distinguish groups visually at run time. You can create a group by adding a *PropertyGroupDescription* object to the *ICollectionView.GroupDescriptions* collection. The following example demonstrates creating groups based on the value of the *EmployeeTitle* property:

Sample of Visual Basic Code

```
Dim myView As System.ComponentModel.ICollectionView
myView = CollectionViewSource.GetDefaultView(myCollection)
myView.GroupDescriptions.Add( _
   New PropertyGroupDescription("EmployeeTitle"))
```

Sample of C# Code

```
System.ComponentModel.ICollectionView myView;
myView = CollectionViewSource.GetDefaultView(myCollection);
myView.GroupDescriptions.Add(
   new PropertyGroupDescription("EmployeeTitle"));
```

At first glance, applying this grouping seems to have the same effect as sorting by *EmployeeTitle*. The difference becomes evident when this data is displayed in an item control with the *GroupStyle* property set. The *GroupStyle* property provides formatting information for grouped items and enables you to set how they are displayed. Table 7-6 explains the properties of the *GroupStyle* object.

TABLE 7-6 Properties of *GroupStyle*

PROPERTY	DESCRIPTION
ContainerStyle	Gets or sets the style applied to the GroupItem object generated for each item
ContainerStyleSelector	Represents an instance of StyleSelector that determines the appropriate style to use for the container
HeaderTemplate	Gets or sets the template used to display the group header
HeaderTemplateSelector	Represents an instance of StyleSelector that determines the appropriate style to use for the header
Panel	Gets or sets a template that creates the panel used to lay out the items

The following example demonstrates setting the *GroupHeader* and *Panel* properties. In this example, the *Panel* property is replaced by a *WrapPanel* element that causes the groups to wrap horizontally, and the *HeaderTemplate* is set to display the header in purple text on a red background:

```
<ListBox ItemsSource="{Binding}" IsSynchronizedWithCurrentItem="True"
   DisplayMemberPath="ContactName" Margin="18,19,25,0" Name="listBox1"
   Height="100" VerticalAlignment="Top">
   <ListBox.GroupStyle>
      <GroupStyle>
         <GroupStyle.HeaderTemplate>
            <DataTemplate>
               <Label Content="{Binding Path=Name}" Foreground="Purple"
                  Background="Red" Padding="4" />
            </DataTemplate>
         </GroupStyle.HeaderTemplate>
         <GroupStyle.Panel>
            <ItemsPanelTemplate>
               <WrapPanel/>
            </ItemsPanelTemplate>
         </GroupStyle.Panel>
      </GroupStyle>
   </ListBox.GroupStyle>
</ListBox>
```

Note that in the data template for the *GroupStyle.HeaderTemplate* property, the *Content* property is bound to the *Name* property. This is not the *Name* property of the objects bound in the *ListBox* control, but, actually, the *Name* property of the *PropertyGroupDescription* element. Thus, when you want the header to reflect the name of the category, you always bind to the *Name* property rather than to the name of the property you are using to group your data.

Creating Custom Grouping

In addition to creating groupings based on the values of object properties, you can create custom groups to arrange your items in an item control. To create a custom group, you must create a class that implements the *IValueConverter* interface, which exposes two methods: *Convert* and *ConvertBack*. For the purposes of creating custom grouping, you need to provide only a "real" implementation for the *Convert* method; the *ConvertBack* method will never be called. The *Convert* method takes an object as a parameter, which will be the object represented by the property named in *PropertyGroupDescription*. In the *Convert* method, examine the value of the object, determine which custom group it belongs to, and then return a value that represents that custom group. The following example is a simple demonstration of this concept. It is designed to examine a value from a *Region* property and determine whether that region is the Isle of Wight. If it is, it returns Isle Of Wight as the group header; otherwise, it returns Everyplace Else.

Sample of Visual Basic Code

```
Public Class RegionConverter
   Implements IValueConverter

   Public Function Convert(ByVal value As Object, ByVal targetType As _
      System.Type, ByVal parameter As Object, ByVal culture As _
      System.Globalization.CultureInfo) As Object Implements _
      System.Windows.Data.IValueConverter.Convert
      If Not value.ToString = "" Then
         Dim region As String = value.ToString
         If region = "Isle of Wight" Then
            Return "Isle of Wight"
         End If
      End If
      Return "Everyplace else"
   End Function

   Public Function ConvertBack(ByVal value As Object, ByVal targetType _
      As System.Type, ByVal parameter As Object, ByVal culture As _
      System.Globalization.CultureInfo) As Object Implements _
      System.Windows.Data.IValueConverter.ConvertBack
      Throw New NotImplementedException()
   End Function
End Class
```

Sample of C# Code

```
public class RegionGrouper : IValueConverter
{
   public object Convert(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
   {
      if (!(value.ToString() == ""))
      {
         string Region = (string)value;
         if (Region == "Isle of Wight")
            return "Isle of Wight";
      }
      return "Everyplace else";
   }

   public object ConvertBack(object value, Type targetType, object
      parameter, System.Globalization.CultureInfo culture)
   {
      throw new NotImplementedException();
   }
}
```

Then you can add your custom *PropertyGroupDescription* parameter by specifying the *Region* property and a new instance of the *RegionGrouper* property, as shown here:

Sample of Visual Basic Code

```
Dim myView As System.ComponentModel.ICollectionView
myView = CollectionViewSource.GetDefaultView(myCollection)
myView.GroupDescriptions.Add(New PropertyGroupDescription("Region", _
   New RegionGrouper()))
```

Sample of C# Code

```
System.ComponentModel.ICollectionView myView;
myView = CollectionViewSource.GetDefaultView(myCollection);
myView.GroupDescriptions.Add(new PropertyGroupDescription("Region",
   new RegionGrouper()));
```

Filtering Data

ICollectionView has built-in functionality that enables you to designate a method to filter the data in your collection view. The *ICollectionView* property accepts a *Predicate* delegate that specifies a Boolean method, which takes the object from the collection as an argument, examines it, and returns *True* if the object is included in the filtered view and *False* if the object is excluded. The following two examples demonstrate setting a simple filter. The first example demonstrates setting the filter on an *ICollectionView* object, and the second example demonstrates the method that implements that filter. These examples involve a collection of items named *myItems*:

Sample of Visual Basic Code

```
Dim myView As System.ComponentModel.ICollectionView
myView = CollectionViewSource.GetDefaultView(myItems)
myView.Filter = New Predicate(Of Object)(AddressOf myFilter)
```

Sample of C# Code

```
System.ComponentModel.ICollectionView myView;
myView = CollectionViewSource.GetDefaultView(myItems);
myView.Filter = new Predicate<object>(myFilter);
```

The next example shows *myFilter*, a simple method that excludes all objects whose *ToString* method returns a string length of eight characters or fewer:

Sample of Visual Basic Code

```
Public Function myFilter(ByVal param As Object) As Boolean
   Return (param.ToString.Length > 8)
End Function
```

Sample of C# Code

```
public bool myFilter(object param)
{
   return (param.ToString().Length > 8);
}
```

Note that the *Predicate* delegate to which the *Filter* property is set must be *Predicate<object>* [*Predicate(Of Object)* in Visual Basic] rather than a strongly typed *Predicate* delegate such as *Predicate<Item>*. You must perform any necessary casting in the method that implements the filter.

Filtering ADO.NET Objects

The filtering method described in the previous section does not work when binding to ADO.NET objects or anytime the underlying *ICollectionView* object is actually a *BindingListCollectionView* object. This is because the data collection bound to in this view actually is connected through an ADO.NET *DataView* layer, which implements its own filtering scheme. However, you can take advantage of the filtering capabilities of *DataView* by setting the *BindingListCollectionView.CustomFilter* property. This property applies a string-based expression directly to the underlying *DataView* layer. The filter expression is a string expression in the following format:

```
<ColumnName> <Operator> <Value>
```

For example, the following filter limits the rows returned to rows whose *Sandwich* property has a value of 'Muffaletta':

```
Sandwich = 'Muffaletta'
```

To set the filter, you would create the following code:

Sample of Visual Basic Code

```
Dim myView As BindingListCollectionView
myView = CType(CollectionViewSource.GetDefaultView(myItems), _
   BindingListCollectionView;
myView.CustomFilter = "Sandwich = 'Muffaletta'"
```

Sample of C# Code

```
BindingListCollectionView myView;
myView =
   (BindingListCollectionView)CollectionViewSource.GetDefaultView(myItems);
myView.CustomFilter = "Sandwich = 'Muffaletta'";
```

String literals for the filter expression must be enclosed in single quotes, as shown in the preceding code. Although the complete rules for filter expressions are beyond the scope of this lesson, they follow the same syntax as comparison expressions for the *DataColumn.Expression* property, and you can view a complete reference in the Microsoft reference topic of that name at *http://msdn2.microsoft.com/en-us/library/system.data.datacolumn.expression(VS.71).aspx*.

✓ Quick Check

- **What is the difference between sorting by a particular property and creating a group based on that property?**

Quick Check Answer

- **When you create a group, you have access to the *GroupStyle* object created for that group. You can set individual templates for the header and container and supply a different layout panel.**

Practice with Data Templates

In this lab, you expand on the application you completed in the previous lab. You add data templates to display greater detail and visual appeal to the *ContactNames* and *ProductNames* lists.

EXERCISE Adding Data Templates

1. Open the completed solution for Lesson 1 of this chapter from either the CD or your completed exercise. You might want to expand the user interface horizontally and increase the width of listbox1 (for contact name) and listbox3 (for product name).
2. In XAML view, delete the following code from the declaration for listBox1:

   ```
   DisplayMemberPath="ContactName"
   ```

3. In XAML view, add the following data template and set it to the *ItemTemplate* property of listBox1:

   ```
   <DataTemplate>
      <StackPanel Orientation="Horizontal">
         <Label Background="Purple" Foreground="White" BorderBrush="Red"
            BorderThickness="4" Width="300">
            <Label.Content>
               <StackPanel HorizontalAlignment="Stretch">
                  <TextBlock>Company Name: </TextBlock>
                  <TextBlock Text="{Binding CompanyName}" />
               </StackPanel>
            </Label.Content>
         </Label>
         <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
            HorizontalAlignment="Stretch" Background="Yellow"
            BorderThickness="3" Foreground="Blue" Width="200" />
      </StackPanel>
   </DataTemplate>
   ```

   This data template does a lot of things. It creates *StackPanel*, in which is bound the *CompanyName* field, and then lays it out so that it appears to the left of the *ContactName* property for that company. It also adds some colorful UI effects.
4. Delete the following code from the declaration for listBox3:

   ```
   DisplayMemberPath="ProductName"
   ```

5. In XAML view, set the following data template to the *ItemTemplate* property of listBox3:

   ```
   <DataTemplate>
      <StackPanel Orientation="Horizontal">
         <Label Background="Yellow" Content="{Binding Path=Quantity}"
            Width="25" />
         <Label Background="AliceBlue" Content="{Binding Path=ProductName}"
            MinWidth="120" />
         <Label Background="Red" Content="{Binding Path=ExtendedPrice}"
            Width="50" />
      </StackPanel>
   </DataTemplate>
   ```

   This data template binds to the *Quantity* and *ExtendedPrice* fields, as well as to the *ProductName* fields, and sets the background color for each field.

6. Press F5 to build and run your application. Note that the display is now more colorful and that additional fields are bound in the *ListBox* elements. Note that although a *Label* object in the data template for listBox3 is bound to the *ExtendedPrice* field, it appears as a regular number instead of being formatted as currency.

Lesson Summary

- Data templates are pieces of XAML markup that specify how bound data should be displayed. They enable you to set the appearance and behavior of bound data as well as specify related fields to be displayed. You can set a data template for an item control by setting the *ItemTemplate* property, and you can set the data template for a content control by setting the *ContentTemplate* property.
- You can apply sorting to data by accessing the default collection view and adding a new sort description to the *SortDescriptions* property. Standard sorting enables you to sort based on a data field value in either ascending or descending mode. You can create custom sorts by creating a class that implements *IComparer* and setting it to the *CustomSort* property.
- You can create groups of data by adding a new *PropertyGroupDescription* parameter to the *GroupDescriptions* property of the default collection view. You can set the header and other formatting for groups by setting the *GroupStyle* property of the item control that binds the data.
- You can specify a delegate to perform filtering by creating a Boolean method that performs the filtering and specifying a *Predicate<object>* delegate that points to that method. This does not work with ADO.NET objects, however, and you must set the *CustomFilter* property to a filter expression to filter ADO.NET records.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Manipulating and Displaying Data." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE **ANSWERS**

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

1. Which of the following code samples correctly demonstrates a data template that binds the *ContactName* field set in *ListBox*? Assume that *DataContext* is set correctly.

   A.

```
<ListBox ItemsSource="{Binding}" name="ListBox1">
   <DataTemplate>
      <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
```

```
         Background="Yellow" BorderThickness="3" Foreground="Blue" />
   </DataTemplate>
</ListBox>
```

B.

```
<ListBox name="ListBox1">
   <ListBox.ItemsSource>
      <DataTemplate>
         <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
            Background="Yellow" BorderThickness="3" Foreground="Blue" />
      </DataTemplate>
   </ListBox.ItemsSource>
</ListBox>
```

C.

```
<ListBox ItemsSource="{Binding}" name="ListBox1">
   <ListBox.ItemTemplate>
      <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
         Background="Yellow" BorderThickness="3" Foreground="Blue" />
   </ListBox.ItemTemplate>
</ListBox>
```

D.

```
<ListBox ItemsSource="{Binding}" name="ListBox1">
   <ListBox.ItemTemplate>
      <DataTemplate>
         <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
            Background="Yellow" BorderThickness="3" Foreground="Blue" />
      </DataTemplate>
   </ListBox.ItemTemplate>
</ListBox>
```

2. Which of the following code samples correctly sets the filter on the *CollectionView* object *myView*?

A.

Sample of Visual Basic Code

```
myView.Filter = New Predicate(Of Object)(AddressOf myFilter)

Public Function myFilter(ByVal param As Object) As Boolean
   Return (param.ToString.Length > 8)
End Function
```

Sample of C# Code

```
myView.Filter = new Predicate<object>(myFilter);

public bool myFilter(object param)
{
   return (param.ToString().Length > 8);
}
```

B.

Sample of Visual Basic Code

```
myView.Filter = New Predicate(Of Object)(AddressOf myFilter)
Public Function myFilter(ByVal param As Object) As Object
   Return (param.ToString.Length > 8)
End Function
```

Sample of C# Code

```
myView.Filter = new Predicate<object>(myFilter);

public object myFilter(object param)
{
   return (param.ToString().Length > 8);
}
```

C.

Sample of Visual Basic Code

```
myView.CustomFilter = New Predicate(Of Object)(AddressOf myFilter)

Public Function myFilter(ByVal param As Object) As Boolean
   Return (param.ToString.Length > 8)
End Function
```

Sample of C# Code

```
myView.CustomFilter = new Predicate<object>(myFilter);

public bool myFilter(object param)
{
   return (param.ToString().Length > 8);
}
```

D.

Sample of Visual Basic Code

```
myView.CustomFilter = New Predicate(Of Object)(AddressOf myFilter)
Public Function myFilter(ByVal param As Object) As Object
   Return (param.ToString.Length > 8)
End Function
```

Sample of C# Code

```
myView.CustomFilter = new Predicate<object>(myFilter);

public object myFilter(object param)
{
   return (param.ToString().Length > 8);
}
```

Case Scenarios

In the following case scenarios, you apply what you've learned about how to use controls to design user interfaces. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario 1: Getting Information from the Field

Field agents are diligently scouring the world trying to locate new supplies of "stuff" for your company's ongoing research efforts. They need to relay regular reports to us in a specific format, which will be logged into our central database. Each of the agents is equipped with a laptop and a satellite Internet connection, but direct connections to the company database are impossible. They also have a report generator that outputs XML files and e-mails them to the mail-handling program, which stores them as raw XML in a queue where they can be vetted by support staff before being saved in the database. You are in charge of creating an application to help these support staff handle the data processing.

Technical Requirements

- XML files can be read, but they must not be altered in any way.
- The user must be able to edit the information before saving it into the central database.

Answer the following question for your manager:

- What is the ideal data access strategy for our application?

Case Scenario 2: Viewing Customer Data

One of your company's clients has contracted with you to create an application to help maximize his business potential. He wants to view customer records that meet any of several specific sets of criteria: customers who buy big-ticket items, customers who buy a lot of small items, customers who buy only in April, and so on. And he wants to view them all at once, but he wants them to be arranged in groups by these criteria. He does not want to see any records that do not match any of the criteria groups. All the information required by the application is available in the database he is supplying.

Technical Requirements

- The application should display only records that fall into one of the criteria groups.
- The records should be arranged by criteria group.

Answer the following question for your manager:

- How can you design the application to achieve the grouping the client wants, with minimal development time?

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

- **Practice 1** Create a Really Simple Syndication (RSS) reader that downloads and formats RSS data by using *XmlDataProvider*.
- **Practice 2** Create an application that accesses the Northwind database and enables the user to filter on the *CompanyName* element.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just the content covered in this chapter, or you can test yourself on all the 70-511 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO **PRACTICE TESTS**

For details about all the practice test options available, see the "How to Use the Practice Tests" section in this book's Introduction.

CHAPTER 8

Working with Data Grids and Validating User Input

The display of data in a grid form has been a constant meme in data display ever since the introduction of spreadsheets and, to this day, remains one of the most popular formats for presenting data to the user. In Lesson 1, "Implementing Data-bound Controls in Windows Forms," you learn to use the *DataGridView* control for viewing data in Windows Forms applications and the *DataGrid* control for viewing data in Windows Presentation Foundation (WPF) applications. In Lesson 2, "Validating User Input," you learn how to validate user input at the field level in both Windows Forms and WPF and implement *IDataErrorInfo* for WPF applications.

Exam objectives in this chapter:

- Implement data validation.
- Implement data-bound controls.

Lessons in this chapter:

Before You Begin

To complete the lessons in this chapter, you must have:

- A computer that meets or exceeds the minimum hardware requirements listed in the "About This Book" section at the beginning of the book.
- Microsoft Visual Studio 2010 Professional installed on your computer.
- An understanding of Microsoft Visual Basic or C# syntax and familiarity with Microsoft .NET Framework 4.0.
- An understanding of Extensible Application Markup Language (XAML).

REAL WORLD

Matthew Stoecker

For simple display of data tables, nothing beats *DataGrid*. A glaring hole in previous versions of WPF development technology was the lack of a *DataGrid* control, but with the release of Visual Studio 2010, this highly flexible and useful control is now available for use in your applications.

Lesson 1: Implementing Data-bound Controls in Windows Forms

Binding data is one of the fundamental tasks in developing a user interface. In this lesson, you learn how to bind Windows Forms controls to data and navigate a bound data source. You can bind simple controls, such as *TextBox* or *Label*, to a single element of data, or you can bind complex controls, such as *DataGridView* and *ComboBox*, to multiple elements of data. You also learn how to use a *BindingSource* component and a *DataNavigator* component to navigate data in your user interface. Additionally, you learn to use the *DataGridView* control for viewing complex data in Windows Forms applications and the *DataGrid* control for WPF applications.

After this lesson, you will be able to:

- Use a simple data-bound control to display a single data element on a Windows Form.
- Implement complex data binding to integrate data from multiple sources.
- Navigate forward and backward through records in a data set in Windows Forms.
- Enhance navigation through a data set by using the *DataNavigator* component.
- Define a data source by using a *DataConnector* component.

Estimated lesson time: 45 minutes

Binding Controls to Data

Binding controls to data is the process of displaying data in Windows Forms controls and creating a connection between the control and the underlying data source.

Simple *data binding* describes the process of displaying a single element of data in a control—for example, a *TextBox* control displaying the value from a single column in a table, such as a company name.

Complex data binding describes the process of binding a control to more than one source of data. For example, consider a combo box that displays a list of category names. What if the table you are displaying has only a category ID, such as the Products and Categories tables in the Northwind sample database? You can use complex data binding to display the value from a column in one *DataTable* source based on a foreign key value in another *DataTable* source.

Simple Data Binding

Simple data binding is the process of binding a single element of data to a single control property, such as a *TextBox* control displaying the *ProductName* column from a table (in its *Text* property).

The following code shows how to bind the *ProductName* column from a data table to a text box named *TextBox1*:

Sample of Visual Basic Code

```
TextBox1.DataBindings.Add("Text", productsBindingSource, "ProductName")
```

Sample of C# Code

```
TextBox1.DataBindings.Add("Text", productsBindingSource, "ProductName");
```

Complex Data Binding

Complex data binding is binding more than one element of data to more than one property of a control—for example, a *DataGridView* control that displays an entire table, or a *List* control that displays multiple columns of data.

Controls that enable complex data binding typically contain a *DataSource* property and a *DataMember* property. The *DataSource* property is usually a *BindingSource* or *DataSet* object. The *DataMember* property is typically the table or collection to actually display.

The following code shows how to bind *DataGridView* to the Northwind Customers table, using a *BindingSource* component:

Sample of Visual Basic Code

```
Dim customersBindingSource As New BindingSource(NorthwindDataSet1, "Customers")
DataGridView1.DataSource = customersBindingSource
```

Sample of C# Code

```
BindingSource customersBindingSource = new BindingSource(northwindDataSet1,
"Customers");
DataGridView1.DataSource = customersBindingSource;
```

The following code shows how to bind *DataGridView* to the Northwind Customers table, using a data set:

Sample of Visual Basic Code

```
DataGridView1.DataSource = NorthwindDataSet1
DataGridView1.DataMember = "Customers"
```

Sample of C# Code

```
DataGridView1.DataSource = northwindDataSet1;
DataGridView1.DataMember = "Customers";
```

Navigating Records in a Data Set

You can use a *BindingNavigator* component to navigate the records in a data source. Assign the *BindingNavigator.BindingSource* property a valid *BindingSource* component, and you can use *BindingNavigator* to move back and forth through the records in that data source.

BindingNavigator uses the navigational methods available on the *BindingSource* component to navigate records. For example, *MoveNext* and *MovePrevious* methods are available on the *BindingSource* component.

Defining a Data Source Using a *BindingSource* Component

The *BindingSource* component contains the information that a control needs to bind to a binding source by passing it a reference to a *DataTable* component in *DataSet*. By binding to *BindingSource* instead of to *DataSet*, you can easily redirect your application to another source of data without having to redirect all the data-binding code to point to the new data source.

The following code shows how to create a *BindingSource* component and assign it a reference to the Northwind Customers table:

Sample of Visual Basic Code

```
customersBindingSource = New BindingSource(NorthwindDataSet1, "Customers")
```

Sample of C# Code

```
customersBindingSource = new BindingSource(northwindDataSet1, "Customers");
```

Quick Check

1. **What is the difference between simple and complex binding?**
2. **How do you navigate back and forth through the records in a data table?**

Quick Check Answers

1. **Simple binding binds an individual bit of data (such as a field or column) to a single property of the control to bind to, whereas complex binding binds multiple bits of data to multiple properties of a control.**
2. **Navigate by calling the *Move* methods of a *BindingSource* component or by displaying a data set in the *DataGridView* control.**

Displaying Data in *DataGridView*

To display a data set in a *DataGridView* control or, more specifically, to display a data table in *DataGridView*, set the *DataSource* property of *DataGridView* to *DataSet* and set the *DataMember* property of *DataGridView* to the name of the data table. For example, the following code displays the Northwind Customers table in *DataGridView*:

Sample of Visual Basic Code

```
DataGridView1.DataSource = NorthwindDataSet1
DataGridView1.DataMember = "Customers"
```

Sample of C# Code

```
DataGridView1.DataSource = northwindDataSet1;
DataGridView1.DataMember = "Customers";
```

You can also set a *DataGridView* control to display a data set by using the smart tag available on a *DataGridView* control: this is accomplished by selecting the data set in the Choose Data Source combo box available on the smart tag. The Choose Data Source command enables you to select a data set and data table to display from the *DataSet* list already defined in your project, or you can create a new data set to display by selecting Add Project Data Source on the smart tag, which starts the Data Source Configuration wizard.

Configuring *DataGridView* Columns

There are six built-in types of columns you can use in *DataGridView*, as outlined in Table 8-1. When adding columns to *DataGridView*, select the type of column based on the data you plan to display in it.

TABLE 8-1 *DataGridView* Column Types

COLUMN TYPE	DESCRIPTION
DataGridViewTextBoxColumn	Use this column type to display text and numeric values. A data-bound *DataGridView* control automatically generates this type of column when binding to strings and numeric values.
DataGridViewCheckBoxColumn	Use this column to display Boolean values. *DataGridView* automatically generates this type of column when binding to Boolean values.
DataGridViewImageColumn	Use this column to display images. *DataGridView* automatically generates this type of column when binding to *Image* and *Icon* objects. You can also bind a *DataGridViewImage* column to a byte array.
DataGridViewButtonColumn	Use this column to provide users with a button control.
DataGridViewComboBoxColumn	Use this column type to present lists of choices. This would typically be used for lookups to other tables.
DataGridViewLinkColumn	Use this column type to display links to other data.
Custom Column	If none of the preceding column types provides the specific functionality you require, you can always create a custom column type. To create a custom column, define your class to inherit from *DataGridViewColumn* or any class with a base class of *DataGridViewColumn*. (For example, inherit from *DataGridViewTextBoxColumn* to extend the functionality of that type.)

Adding Tables and Columns to *DataGridView*

To display a table in *DataGridView*, define the columns that make up the schema of the table and add them to *DataGridView*. You can add columns to *DataGridView* with designers using the Add Column dialog box or programmatically in code.

First, decide which type of column to use (refer to Table 8-1) and then use one of the following procedures to add the column to your *DataGridView* control.

Adding Columns to *DataGridView* Using the Designer

To add columns to *DataGridView* in the designer, follow these steps:

1. Select *DataGridView* on your form.
2. Open the smart tag of *DataGridView*.
3. Select Add Column.
4. In the Add Column dialog box, define the column by setting the appropriate values in the dialog box.

Adding Columns to *DataGridView* Programmatically

To add columns to *DataGridView* in code, create an instance of the appropriate type of column, define the column by setting the appropriate properties, and then add the column to the *DataGridView.Columns* collection.

For example, the following code sample creates a new text box column named ProductName:

Sample of Visual Basic Code

```
Dim ProductNameColumn As New DataGridViewTextBoxColumn
ProductNameColumn.Name = "ProductName"
ProductNameColumn.HeaderText = "Product Name"
ProductNameColumn.ValueType = System.Type.GetType("System.String")
DataGridView1.Columns.Add(ProductNameColumn)
```

Sample of C# Code

```
DataGridViewTextBoxColumn ProductNameColumn = new DataGridViewTextBoxColumn();
ProductNameColumn.Name = "ProductName";
ProductNameColumn.HeaderText = "Product Name";
ProductNameColumn.ValueType = System.Type.GetType("System.String");
DataGridView1.Columns.Add(ProductNameColumn); .
```

Deleting Columns in *DataGridView*

You can delete columns in *DataGridView* by using the designer in Visual Studio or programmatically in code.

Deleting Columns in *DataGridView* Using the Designer

To delete columns in a *DataGridView* using the designer, complete the following steps:

1. Select the *DataGridView* on your form.
2. Open the smart tag for the *DataGridView*.
3. Select Edit Columns.
4. In the Edit Columns dialog box, select the column you want to remove from the *DataGridView*.
5. Click the Remove button.

Deleting Columns in a *DataGridView* Programmatically

To delete columns in *DataGridView* in code, call the *Remove* method and provide the name of the column you want to delete. For example, the following code example deletes a column named ProductName from *DataGridView1*:

Sample of Visual Basic Code

```
DataGridView1.Columns.Remove("ProductName")
```

Sample of C# Code

```
DataGridView1.Columns.Remove["ProductName"];
```

Determining the Clicked Cell in *DataGridView*

To determine the clicked cell, use the *DataGridView.CurrentCell* property. *CurrentCell* provides a reference to the currently selected cell and provides properties to access the value of the data in the cell, as well as the row and column index of the cell's current location in *DataGridView*. For example:

Sample of Visual Basic Code

```
Dim CurrentCellValue As String
CurrentCellValue = CustomersDataGridView.CurrentCell.Value.ToString
```

Sample of C# Code

```
String CurrentCellValue;
CurrentCellValue = CustomersDataGridView.CurrentCell.Value.ToString();
```

Validating Input in the *DataGridView* Control

To validate input in an individual cell in a *DataGridView* control, handle the *DataGridView .CellValidating* event and cancel the edit if the value fails validation. The *CellValidating* event is raised when a cell loses focus. Add code to the event handler for the *CellValidating* event to verify that the values in specific columns conform to your business rules and application logic. The event arguments contain the proposed value in the cell, as well as the row and column index of the cell being edited.

For example, the following code validates that the ProductName column does not contain an empty string (use this sample for a *DataGridView* control that is not bound to data):

Sample of Visual Basic Code

```
If DataGridView1.Columns(e.ColumnIndex).Name = "ProductName" Then
    If e.FormattedValue.ToString = "" Then
        dataGridView1.Rows(e.RowIndex).ErrorText = "Product Name is a required field"
        e.Cancel = True
    Else
        dataGridView1.Rows(e.RowIndex).ErrorText = ""
    End If
End If
```

Sample of C# Code

```
if (DataGridView1.Columns[e.ColumnIndex].Name == "ProductName")
{
     if (e.FormattedValue.ToString() == "")
     {
         DataGridView1.Rows[e.RowIndex].ErrorText = "Product Name is a required field";
         e.Cancel = true;
     }
    else
    {
        DataGridView1.Rows[e.RowIndex].ErrorText = "";
    }
}
```

The following code also validates that the ProductName column does not contain an empty string. Use this example for a *DataGridView* control that *is* bound to data. The difference from the preceding example is shown in bold.

Sample of Visual Basic Code

```
If DataGridView1.Columns(e.ColumnIndex).DataPropertyName = "ProductName" Then
    If e.FormattedValue.ToString = "" Then
        dataGridView1.Rows(e.RowIndex).ErrorText = "Product Name is a required field"
        e.Cancel = True
    Else
        dataGridView1.Rows(e.RowIndex).ErrorText = ""
    End If
End If
```

Sample of C# Code

```
if (DataGridView1.Columns[e.ColumnIndex].DataPropertyName == "ProductName")
{
     if (e.FormattedValue.ToString() == "")
     {
         DataGridView1.Rows[e.RowIndex].ErrorText = "Product Name is a required field";
         e.Cancel = true;
     }
    else
    {
        DataGridView1.Rows[e.RowIndex].ErrorText = "";
    }
}
```

Format a *DataGridView* Control by Using Custom Painting

To format a *DataGridView* control by using custom painting, you can handle the *CellPainting* event and insert your own custom painting code. When you handle the *CellPainting* event, *DataGridViewCellPaintingEventArgs* provides access to many properties that simplify custom painting. When you handle the *CellPainting* event, be sure to set *e.Handled* to *True* so the grid does not call its own cell painting routine.

To paint all cells *LightSkyBlue*, place the following code in the *CellPainting* event handler:

Sample of Visual Basic Code

```
' Paint the cell background color LightSkyBlue
e.Graphics.FillRectangle(Brushes.LightSkyBlue, e.CellBounds)

' Draw the contents of the cell
If Not (e.Value Is Nothing) Then
    e.Graphics.DrawString(e.Value.ToString, e.CellStyle.Font, _
        Brushes.Black, e.CellBounds.X, e.CellBounds.Y)
End If
e.Handled = True
```

Sample of C# Code

```
// Paint the cell background color LightSkyBlue
e.Graphics.FillRectangle(Brushes.LightSkyBlue, e.CellBounds);
// Draw the contents of the cell
if (e.Value != null)
{
    e.Graphics.DrawString(e.Value.ToString(), e.CellStyle.Font,
        Brushes.Black, e.CellBounds.X, e.CellBounds.Y);
}
e.Handled = true;
```

Quick Check

1. What properties do you set on a *DataGridView* control to display a specific data table?
2. How do you determine what cell is clicked in a *DataGridView* control?

Quick Check Answers

1. Set the *DataSource* property to the *DataSet* and the *DataMember* properties to the name of the data table.
2. Inspect the *DataGridView.CurrentCell* property.

Using *DataGrid* in WPF Applications

The WPF *DataGrid* control is analogous to the Windows Forms *DataGridView* control in that it is designed to display tables of data in a cohesive format. You can add columns manually to your *DataGrid* control, or it will automatically create appropriate columns for your data based on the schema of the data source.

Binding Data Sources to a *DataGrid* Control

A *DataGrid* control can be bound to a data source that implements the *IEnumerable* interface. By default, it will automatically create columns for the properties of the bound object and will create rows that display the values of those properties for each object in the data source. You can set the data source for a *DataGrid* control by setting the *ItemsSource* property in code, as shown here:

Sample of Visual Basic Code

```
' Refers to a Binding already present in code
DataGrid1.ItemsSource = myCustomersBinding
```

Sample of C# Code

```
// Refers to a DataTableView already present in code
dataGrid1.ItemsSource = myCustomersBinding;
```

Or, more commonly, you will set the data source in XAML by setting the *ItemsSource* property, as shown here:

```
<Window.Resources>
        <my:NwindDataSet x:Key="NwindDataSet" />
        <CollectionViewSource x:Key="CustomersViewSource" Source="{Binding
Path=Customers,
            Source={StaticResource NwindDataSet}}" />
</Window.Resources>
    <Grid DataContext="{StaticResource CustomersViewSource}">
        <DataGrid ItemsSource="{Binding}"
            <!-- Additional implementation omitted -->
        </DataGrid>
    </Grid>
```

For changes in the collection to be updated automatically in the *DataGrid* control, the collection to which the *DataGrid* control is bound must implement *INotifyCollectionChanged*, such as *ObservableCollection*. For changes to property values to be reflected in real time, the bound properties must implement *INotifyPropertyChanged*.

Binding a *DataGrid* Control to a Data Source in Your Project

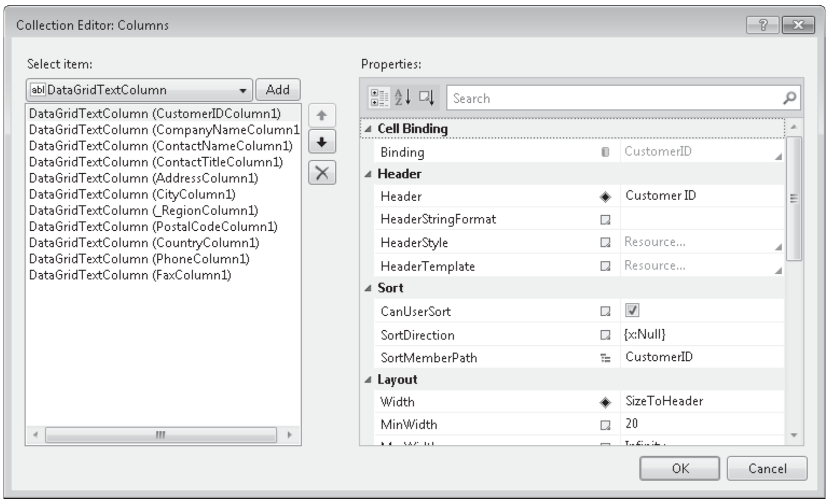
If you have already added a data source, such as an ADO.NET database, to your project, you can easily create a bound *DataGrid* control in your application through the Data Sources window.

To create a bound data grid through the Data Sources window:

1. If the Data Sources window is not already visible, choose Show Data Sources from the Data menu. The Data Sources window opens.
2. In the Data Sources window, each available data table should have a grid symbol next to it. If a grid symbol is not shown, highlight the data table, click the drop-down menu, and choose DataGrid.
3. With the grid symbol showing next to the data table, drag the data table from the Data Sources window to the design surface of the form. A *DataGrid* control representing the data table is added to your project.

Using *DataGrid* Columns

The columns displayed in the *DataGrid* control are very configurable and can be manipulated at design time through the Columns Collection editor in Visual Studio, shown in Figure 8-1. To make the Columns Collection editor appear, click the ellipses (...) to the right of the *Columns* property in the Property grid.



```
DataGridTextColumn (CustomerIDColumn1)
DataGridTextColumn (CompanyNameColumn1
DataGridTextColumn (ContactNameColumn1)
DataGridTextColumn (ContactTitleColumn1)
DataGridTextColumn (AddressColumn1)
DataGridTextColumn (CityColumn1)
DataGridTextColumn (_RegionColumn1)
DataGridTextColumn (PostalCodeColumn1)
DataGridTextColumn (CountryColumn1)
DataGridTextColumn (PhoneColumn1)
DataGridTextColumn (FaxColumn1)
```

Cell Binding	
Binding	CustomerID
Header	
Header	Customer ID
HeaderStringFormat	
HeaderStyle	Resource...
HeaderTemplate	Resource...
Sort	
CanUserSort	✓
SortDirection	{x:Null}
SortMemberPath	CustomerID
Layout	
Width	SizeToHeader
MinWidth	20

FIGURE 8-1 The Collection Editor.

Using the Collection editor, you can add and remove *DataGrid* columns, as well as change the properties of individual columns.

When a *DataGrid* control is bound to a data source, appropriate columns are automatically generated for the data set. Depending on the data type of the bound property, types of columns are generated as shown in Table 8-2.

TABLE 8-2 DataGrid Column Types

DATA TYPE	GENERATED COLUMN TYPE
Boolean	DataGridCheckBoxColumn
Enum	DataGridComboBoxColumn
String	DataGridTextBoxColumn
Uri	DataGridHyperlinkColumn

If you want to provide custom functionality during column auto-generation, handle the *DataGrid.AutoGeneratingColumn* property. To turn automatic column generation off and set all columns manually, set the *AutoGenerateColumns* property to *False*.

Using *DataGridTemplateColumn*

For other data types, or to create your own custom data columns, you can use a column of type *DataGridTemplateColumn*, which is a customizable column that enables you to define three *DataTemplate* objects that govern the appearance and behavior of the column. The *HeaderTemplate* property of *DataGridTemplateColumn* determines the data template that will be used for the column header. The *CellTemplate* property determines the data template that will be used for cells when data is being displayed, and the *CellEditingTemplate* property indicates the data template used when data in the cell is being edited. These templates should be defined in your Windows or Application resources as shown here:

```
<Window.Resources>
     <my:NwindDataSet x:Key="NwindDataSet" />
     <CollectionViewSource x:Key="CustomersViewSource"
         Source="{Binding Path=Customers, Source={StaticResource NwindDataSet}}" />
     <DataTemplate x:Key="myHeaderTemplate">
         <Label Background="Purple" Foreground="Yellow" Content="{Binding}"/>
     </DataTemplate>
     <DataTemplate x:Key="myCellTemplate">
         <Button Foreground="Red" Content="{Binding Path=CompanyName}"/>
     </DataTemplate>
     <DataTemplate x:Key="myCellEditingTemplate">
         <TextBox Background="Red" Text="{Binding Path=CompanyName}"/>
     </DataTemplate>
 </Window.Resources>
```

In this example, *HeaderTemplate* defines a purple label with yellow text. This will be seen only in the *DataGrid* header, and the empty binding the *Content* property is set to ensure that the *Content* property will be bound to the *Header* property of the column.

CellTemplate in this example creates a *Button* control that displays the bound CompanyName in red as the content of the button. *CellEditingTemplate* displays a text box with the background in red, also bound to the same CompanyName. *CellEditingTemplate* is displayed whenever *BeginEditCommand* is processed for that cell—for example, when the cell is selected and the F2 button is pressed, and *CellTemplate* will be displayed when editing is committed or cancelled.

In addition to providing templates, you can instead set *TemplateSelector* controls to provide dynamic selection of the template to use for the header, cell, and cell editing. You can provide these through the *HeaderTemplateSelector*, *CellTemplateSelector*, and *CellEditingTemplateSelector* properties, respectively. See Chapter 7, "Configuring Data Binding," for further information about using *TemplateSelector* controls.

Using Row Details

In some cases, you might want to display some data about a row only when that row is selected. For example, if a data table has several columns, you might want to display only a subset in the grid itself, and then display other columns in a specialized format only when a row of data is being inspected. You can access this functionality through the *RowDetailsTemplate* property, which enables you to create a data template that will display additional data beneath a row in the data grid when that row is selected. Consider the following example:

```
<DataTemplate x:Key="myRowDetailsTemplate">
    <StackPanel DataContext="{StaticResource OrdersViewSource}"
        Orientation="Horizontal">
        <Label Content="Order Date" Background="Red" />
        <DatePicker Text="{Binding Path=OrderDate}" />
        <Label Content="Required Date" Background="Red" />
        <DatePicker Text="{Binding Path=RequiredDate}" />
    </StackPanel>
</DataTemplate>
```

This example demonstrates a data template to be used for a *RowDetailsTemplate* control. It defines a *StackPanel* control that contains four other controls: two labels and two *DatePicker* controls that are bound to the *OrderDate* and *RequiredDate* fields. At run time, the values of these bound fields will be displayed next to the labels underneath the selected row. By default, row details are visible only for the selected row. You can make them visible for all rows by setting the *RowDetailsVisibilityMode* property to *Visible*, or you can always hide them by setting the *RowDetailsVisibilityMode* to *Collapsed*.

You can also define a template selector to use for row details by setting the *RowDetailsTemplateSelector* property.

Working with *DataGridView*

In this practice, you work with data in a *DataGridView* control.

EXERCISE Working with *DataGridView*

First, you create a Windows Forms application and see how to manipulate the definition as well as the columns and data in a *DataGridView* control.

1. Create a Windows Forms application and name it **DataGridViewExample**.

2. Open the Data Sources window (on the Data menu, select Show Data Sources).
3. Click Add New Data Source to start the Data Source Configuration Wizard.
4. Leave the default of Database and click Next twice.
5. Select (or create) a connection to the Northwind sample database by browsing to the database file and selecting it; click Next. When prompted, click Yes and then Next again.
6. Expand the Tables node. Select the Customers table and then click Finish.
7. Drag the Customers node from the Data Sources window onto the form.

   At this point, you can actually run the application, and the form appears with the Customers table loaded into a *DataGridView* control.
8. Drag two *Button* controls onto the form below *DataGridView* and set the following properties:
   - Button1:
     - *Name* = AddColumnButton
     - *Text* = Add Column
   - Button2:
     - *Name* = DeleteColumnButton
     - *Text* = Delete Column
9. Double-click the Add Column button to create the button-click event handler and to open the form in code view.
10. Add the following code to *Form1*. The additional code in the *Form1_Load* event creates a new column on the data table, and the code in the *AddColumnButton_Click* event handler adds a new column to *DataGridView*:

    Sample of Visual Basic Code

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles MyBase.Load
    'TODO: This line of code loads data into the
    'NwindDataSet.Customers table. You can move, or remove
    'it, As needed.
    Me.CustomersTableAdapter.Fill(Me.NwindDataSet.Customers)

    ' Add a new column to the Customers DataTable
    ' to be used to demonstrate adding and removing
    ' columns in a DataGridView in the methods below
    Dim Location As New DataColumn("Location")
    Location.Expression = "City + ', ' + Country"
    NwindDataSet.Customers.Columns.Add(Location)
End Sub

Private Sub AddColumnButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles AddColumnButton.Click
    Dim LocationColumn As New DataGridViewTextBoxColumn
    LocationColumn.Name = "LocationColumn"
    LocationColumn.HeaderText = "Location"
    LocationColumn.DataPropertyName = "Location"
    CustomersDataGridView.Columns.Add(LocationColumn)
End Sub
```

Sample of C# Code

```
private void Form1_Load(object sender, EventArgs e)
{
    // TODO: This line of code loads data into the
    // nwindDataSet.Customers table. You can move, or
    // remove it, as needed.
    this.customersTableAdapter.Fill(this.nwindDataSet.Customers);
    // Add a new column to the Customers DataTable
    // to be used to demonstrate adding and removing
    // columns in a DataGridView in the methods below
    DataColumn Location = new DataColumn("Location");
    Location.Expression = "City + ', ' + Country";
    nwindDataSet.Customers.Columns.Add(Location);
}

private void AddColumnButton_Click(object sender, EventArgs e)
{
    DataGridViewTextBoxColumn LocationColumn = new
        DataGridViewTextBoxColumn();
    LocationColumn.Name = "LocationColumn";
    LocationColumn.HeaderText = "Location";
    LocationColumn.DataPropertyName = "Location";
    customersDataGridView.Columns.Add(LocationColumn);
}
```

11. In the designer, double-click the Delete Column button to create the *DeleteColumnButton_Click* event handler. Add the following code to the *DeleteColumnButton_Click* event handler:

Sample of Visual Basic Code

```
Try
    CustomersDataGridView.Columns.Remove("LocationColumn")
Catch ex As Exception
    MessageBox.Show(ex.Message)
End Try
```

Sample of C# Code

```
try
{
    customersDataGridView.Columns.Remove("LocationColumn");
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

12. Drag another *Button* control onto the form and set the following properties:

- *Name* = GetClickedCellButton
- *Text* = Get Clicked Cell

13. Drag a *Label* control onto the form and place it next to the Get Clicked Cell button.

14. Double-click the Get Clicked Cell button and add the following code to the *GetClickedCellButton_Click* event handler:

Sample of Visual Basic Code

```
Dim CurrentCellInfo As String
CurrentCellInfo = _
    CustomersDataGridView.CurrentCell.Value.ToString & _
    Environment.NewLine
CurrentCellInfo += "Column: " & _
    CustomersDataGridView.CurrentCell.OwningColumn.DataPropertyName & _
    Environment.NewLine
CurrentCellInfo += "Column Index: " & CustomersDataGridView.CurrentCell.
ColumnIndex.ToString & _
    Environment.NewLine
CurrentCellInfo += "Row Index: " & _
    CustomersDataGridView.CurrentCell.RowIndex.ToString

Label1.Text = CurrentCellInfo
```

Sample of C# Code

```
string CurrentCellInfo;
CurrentCellInfo = customersDataGridView.CurrentCell.Value.ToString() +
    Environment.NewLine;
CurrentCellInfo += "Column: " +
    customersDataGridView.CurrentCell.OwningColumn.DataPropertyName +
    Environment.NewLine;
CurrentCellInfo += "Column Index: " + customersDataGridView.CurrentCell.
ColumnIndex.ToString() +
    Environment.NewLine;
CurrentCellInfo += "Row Index: " +
    customersDataGridView.CurrentCell.RowIndex.ToString();

label1.Text = CurrentCellInfo;
```

15. Create an event handler for the *CustomersDataGridView.CellValidating* event. (Select the *CustomersDataGridView* control on the form, click the Events icon in the Properties window, and double-click the *CellValidating* event.)

16. Add the following code to the *CellValidating* event handler:

Sample of Visual Basic Code

```
If CustomersDataGridView.Columns(e.ColumnIndex).DataPropertyName = _
    "ContactName" Then
    If e.FormattedValue.ToString = "" Then
        CustomersDataGridView.Rows(e.RowIndex).ErrorText = _
            "ContactName is a required field"
        e.Cancel = True
    Else
```

```
        CustomersDataGridView.Rows(e.RowIndex).ErrorText = ""
    End If
End If
```

Sample of C# Code

```
if (customersDataGridView.Columns[e.ColumnIndex].DataPropertyName ==
    "ContactName")
{
    if (e.FormattedValue.ToString() == "")
    {
        customersDataGridView.Rows[e.RowIndex].ErrorText =
        "ContactName is a required field";
        e.Cancel = true;
    }
    else
    {
        customersDataGridView.Rows[e.RowIndex].ErrorText = "";
    }
}
```

17. Drag another *Button* control onto the form and set the following properties:
    - *Name* = ApplyStyleButton
    - *Text* = Apply Style
18. Double-click the Apply Style button and add the following code to the *ApplyStyleButton_Click* event handler:

    Sample of Visual Basic Code

```
CustomersDataGridView.AlternatingRowsDefaultCellStyle.BackColor = _
    Color.LightGray
```

Sample of C# Code

```
customersDataGridView.AlternatingRowsDefaultCellStyle.BackColor =
    Color.LightGray;
```

19. Run the application.
20. Click the Add Column button and then scroll to the end of the columns to verify that the new Location column is there.
21. Click the Delete Column button and verify that the Location column is deleted from *DataGridView*.
22. Select any cell in the grid and then click the Get Clicked Cell button. The *Label* control displays the contents of the cell, the name of the column the cell is in, and the column and row index of the cell.
23. Finally, click the Apply Style button. The *AlternatingRowsDefaultCellStyle* style is set up to display alternating rows with a light gray background.

Lesson Summary

- *DataGridView* is the preferred control for displaying tabular data such as a data table in a Windows Forms application. In a WPF application, the *DataGrid* control is used.
- You can add columns to and remove columns from a *DataGridView* control in the designer by using the Add Column and Edit Column dialog boxes available from the smart tag of *DataGridView*.
- The *DataGridView.CurrentCell* property provides access to the currently selected cell in a *DataGridView* control.
- *DataGridView* raises a *CellValidating* event through which you can add code that verifies that the value in a column conforms to your business rules and application logic.
- You can format the look of *DataGridView* by using styles and custom painting.
- You can create custom columns in a data grid by setting the *HeaderTemplate*, *CellTemplate*, and *CellEditingTemplate* properties of *DataGridTemplateColumn*.
- You can provide customized row details by setting the *RowDetailsTemplate* property of *DataGrid*.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Implementing Data-bound Controls in Windows Forms." The questions are also available on the companion CD if you prefer to review them in electronic form.

***NOTE* ANSWERS**

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

1. What is the best way to determine which cell a user clicks in *DataGridView*?
   - **A.** Use the column and row index of the selected cell.
   - **B.** Use the *DataGridView.CurrentCell* property.
   - **C.** Use the cursor position's *x* and *y* coordinates.
   - **D.** Use the currently selected column and row in the bound *DataTable* control to determine the clicked cell.
2. What is the preferred method of validating input in *DataGridView*?
   - **A.** By adding validation code to the *CellPainting* event handler
   - **B** By adding validation code to the *DataGridView.CellClick* event handler
   - **C.** By adding validation code to the *DataGridView.CellValidating* event handler
   - **D.** By adding code to the *DataGridView* partial class file

3. What is the best way to display a Boolean value in *DataGridView*?

   A. Configure *DataGridViewTextBoxColumn* and display *True* or *False*.

   B. Configure *DataGridViewCheckBoxColumn* to display a check box that is selected or cleared.

   C. Configure *DataGridViewButtonColumn* to display a button that indicates pressed or not pressed.

   D. Configure a custom column to display *Yes* or *No*.

Lesson 2: Validating User Input

In most applications, the user enters information for the application through the user interface. Data validation ensures that all data entered by a user falls within acceptable parameters before proceeding with program execution. For example, you might have a field in which a user enters a ZIP code as part of an address. Using validation, you could verify that the field contained five and only five characters, all of which were numeric, before proceeding. Validating user input reduces the chance of an input error and makes your application more robust.

In this lesson, you learn how to use events to validate user input and direct the focus on your forms. You learn how to use field-level validation, which validates entries as they are made, and form-level validation, which validates all the entries on a form at once. You learn how to use control properties to help restrict input and use the *ErrorProvider* component to provide informative error messages to your users.

After this lesson, you will be able to:

- Explain the difference between form-level and field-level validation.
- Direct the focus using control methods and events.
- Implement form-level validation for your form.
- Implement field-level validation for your form.

Estimated lesson time: 30 minutes

You can choose between two types of validation for user input: form-level validation and field-level validation. Form-level validation verifies data after the user has filled all the fields. For example, a user might be directed to fill in a name, address, and phone number on a form and then click OK. With form-level validation, all the fields on the form would be validated when the user clicked OK.

Field-level validation, alternatively, verifies the data in each field as the field is filled in. For example, if a user fills in a field that holds a phone number, field-level validation could verify that the number contains a valid area code before moving to the next field. As each digit is entered, you could also use control events to verify that only numbers are entered.

Field-Level Validation

You might want to validate data as it is entered into each field. Field-level validation gives the developer control over user input as it occurs. In this section, you learn how to use control events to validate user input and how to use some properties of the *TextBox* control to help restrict input to appropriate parameters.

Using *TextBox* Properties

The most commonly used control for user input is *TextBox*. Several properties of the *TextBox* control enable you to restrict the values of user input that those properties will accept. Some of these properties include:

- *MaxLength*
- *PasswordChar*
- *ReadOnly*
- *MultiLine*

Setting the *MaxLength* Property

The *MaxLength* property limits the number of characters that can be entered into a text box. If the user attempts to exceed the number returned by *MaxLength*, the text box will accept no further input, and the system beeps to alert the user. This property is useful for text boxes that always contain data of the same length, such as a ZIP code field.

Using the *PasswordChar* Property

The *PasswordChar* property enables you to hide user input at run time. For example, if you set the *PasswordChar* property to an asterisk (*), the text box will display an asterisk for each character, regardless of user input. This type of behavior is commonly seen in password logon boxes.

Although the password character is most commonly an asterisk, you can set it to any valid character. Thus, you could display all semicolons or ampersands, for example. The value of the *Text* property will always be set to the value the user enters, regardless of the password character.

Setting the *ReadOnly* Property

The *ReadOnly* property determines whether a user can edit the value displayed in a text box. If *ReadOnly* is set to *True*, the text cannot be changed by user input. If *ReadOnly* is set to *False*, the user can edit the value normally.

Using the *MultiLine* Property

The *MultiLine* property determines whether a text box can accept multiple lines. When set to *True*, the user can enter multiple lines in the text box, each separated by a carriage return. The individual lines are stored as an array of strings in the *TextBox.Lines* collection and can be accessed by their index.

Using Events in Field-Level Validation

Field-level keyboard events enable you to validate user input immediately. Controls that can receive keyboard input raise three keyboard events. They are:

- *KeyDown*
- *KeyPress*
- *KeyUp*

KeyDown and *KeyUp*

The *KeyDown* and *KeyUp* events are raised when a key is pressed and when a key is released, respectively. The control that has the focus raises the event. When these events are raised, they package information about which key or combination of keys has been pressed or released in an instance of *KeyEventArgs*, a class that describes the key combination pressed. A method that handles the *KeyDown* or *KeyUp* event must include a *KeyEventArgs* parameter in its signature.

The *KeyUp* and *KeyDown* events are most commonly used for determining whether the Alt, Ctrl, or Shift key has been pressed. This information is exposed through properties in the *KeyEventArgs* reference that is passed to the handler. The *KeyEventArgs* properties—Alt, Control, and Shift—are properties that return a Boolean value, which indicates whether those keys are down. A value of *True* is returned if the corresponding key is down, and *False* is returned if the key is up. The following example demonstrates a *KeyUp* event handler that checks whether the Alt key is pressed.

Sample of Visual Basic Code

```
Private Sub TextBox1_KeyUp(ByVal sender As Object, ByVal e As _
   System.Windows.Forms.KeyEventArgs) Handles TextBox1.KeyUp
   If e.Alt = True Then
      MessageBox.Show("The ALT key still is down")
   End If
End Sub
```

Sample of Visual C# Code

```
private void textBox1_KeyUp(object sender,
   System.Windows.Forms.KeyEventArgs e)
{
   if (e.Alt == true)
   MessageBox.Show("The ALT key is still down");
}
```

You can also use the *KeyEventArgs.KeyCode* property to examine the actual key that triggered the event. This property returns a *Key* value that represents the key that was pressed (in the case of a *KeyDown* event) or released (in the case of a *KeyUp* event). The following example shows a simple event handler that displays a message box containing a string representation of the key that was pressed.

Sample of Visual Basic Code

```
Private Sub TextBox1_KeyDown(ByVal sender As Object, ByVal e As _
   System.Windows.Forms.KeyEventArgs) Handles TextBox1.KeyDown
   MessageBox.Show(e.KeyCode.ToString())
End Sub
```

Sample of Visual C# Code

```
private void textBox1_KeyDown(object sender,
   System.Windows.Forms.KeyEventArgs e)
{
   MessageBox.Show(e.KeyCode.ToString());
}
```

KeyPress

When a user presses a key that has a corresponding ASCII value, the *KeyPress* event is raised. These keys include any alphabetic and numeric characters (alphanumeric a–z, A–Z, and 0–9), as well as some special keyboard characters such as the Enter and Backspace keys. If a key or key combination does not produce an ASCII value, it will not raise the *KeyPress* event. Examples of keys that do not raise this event include Ctrl, Alt, and the function keys.

This event is most useful for intercepting keystrokes and evaluating them. When this event is raised, an instance of *KeyPressEventArgs* is passed as a parameter to the event handler. The *KeyPressEventArgs.KeyChar* property contains the ASCII character represented by the keystroke that raised the event. If you want to make sure the key pressed is a numeric key, for example, you can evaluate the *KeyChar* property in your *KeyPress* event handler.

Validating Characters

The *Char* data type contains several *Shared* (static) methods that are useful for validating characters trapped by the *KeyPress* event. These methods include:

- *Char.IsDigit*
- *Char.IsLetter*
- *Char.IsLetterOrDigit*
- *Char.IsPunctuation*
- *Char.IsLower*
- *Char.IsUpper*

Each of these methods evaluates a character and returns a Boolean value; they are fairly self-explanatory. The *Char.IsDigit* function returns *True* if a character is a numeric digit, *False* if it is not. The *Char.IsLower* function returns *True* if a character is a lowercase letter, otherwise it returns *False*. The other methods behave similarly. The following example uses the *Char.IsNumber* method to test whether the key pressed is a numeric key:

Sample of Visual Basic Code

```
Private Sub TextBox1_KeyPress (ByVal sender as Object, ByVal e As _
   System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
   If Char.IsDigit(e.KeyChar) = True Then
      MessageBox.Show("You pressed a number key")
   End If
End Sub
```

Sample of Visual C# Code

```
private void textBox1_KeyPress (object sender,
   System.Windows.Forms.KeyPressEventArgs e)
{
   if (Char.IsDigit(e.KeyChar) == true)
      MessageBox.Show("You pressed a number key");
}
```

Handling the Focus

Focus is the ability of an object to receive user input through the mouse or keyboard. Although you can have several controls on your form, only one can have the focus at any given time. The control that has the focus is always on the active form of the application.

Every control implements the *Focus* method. This method sets the focus to the control that called it. The *Focus* method returns a Boolean value that indicates whether the control was successful in setting the focus. Controls that are disabled or invisible cannot receive the focus. You can determine whether a control can receive the focus by checking the *CanFocus* property, which returns *True* if the control can receive the focus and *False* if the control cannot.

Sample of Visual Basic Code

```
' This example checks to see if TextBox1 can receive the focus and
' sets the focus to it if it can.
If TextBox1.CanFocus = True Then
   TextBox1.Focus()
End If
```

Sample of Visual C# Code

```
// This example checks to see if textBox1 can receive the focus and
// sets the focus to it if it can.
if (textBox1.CanFocus == true)
   textBox1.Focus();
```

Focus events occur in the following order:

1. *Enter*
2. *GotFocus*
3. *Leave*
4. *Validating*
5. *Validated*
6. *LostFocus*

The *Enter* and *Leave* events are raised when the focus arrives at a control and when the focus leaves a control, respectively. *GotFocus* and *LostFocus* are raised when a control first obtains the focus and when the focus has been lost from the control, respectively. Although you can use these events for field-level validation, the *Validating* and *Validated* events are more suited to that task.

The *Validating* and *Validated* Events

The easiest way to validate data is to use the *Validating* event, which occurs before a control loses the focus. This event is raised only when the *CausesValidation* property of the control that is about to receive the focus is set to *True*. Thus, if you want to use the *Validating* event to validate data entered in your control, the *CausesValidation* of the next control in the tab order should be set to *True*. To use *Validating* events, the *CausesValidation* property of the control to be validated must also be set to *True*. By default, the *CausesValidation* property of all controls is set to *True* when controls are created at design time. Controls such as Help buttons are typically the only kind of controls that have *CausesValidation* set to *False*.

The *Validating* event enables you to perform sophisticated validation on your controls. You could, for example, implement an event handler that tests whether the value entered corresponds to a very specific format. Another possible use is an event handler that doesn't allow the focus to leave the control until a value has been entered.

The *Validating* event includes an instance of the *CancelEventArgs* class. This class contains a single property, *Cancel*. If the input in your control does not fall within required parameters, you can use the *Cancel* property within your event handler to cancel the *Validating* event and return the focus to the control.

The *Validated* event fires after a control has been validated successfully. You can use this event to perform any actions based upon the validated input.

The following example demonstrates a handler for the *Validating* event. This method requires an entry in TextBox1 before it will allow the focus to move to the next control.

Sample of Visual Basic Code

```
Private Sub TextBox1_Validating(ByVal sender As Object, ByVal e As _
   System.ComponentModel.CancelEventArgs) Handles TextBox1.Validating
   ' Checks the value of TextBox1
   If TextBox1.Text = "" Then
      ' Resets the focus if there is no entry in TextBox1
      e.Cancel = True
   End If
End Sub
```

Sample of Visual C# Code

```
private void textBox1_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   // Checks the value of textBox1
   if (textBox1.Text == "")
      // Resets the focus if there is no entry in TextBox1
      e.Cancel = true;
}
```

To use the *Validating* event of a text box:

1. Add a text box to a form.
2. Create an event handler to handle the *Validating* event of the text box. In the event handler, set the *e.Cancel* property to *True* to cancel validating and return the focus to the text box.
3. Set the *CausesValidation* property to *False* for any controls for which you do not want the *Validating* event to fire.

Form-Level Validation

Form-level validation is the process of validating all the fields on a form at once. A central procedure implements form-level validation and is usually called when the user is ready to proceed to another step. A more advanced method of form-level validation is implementing a form-level keyboard handler.

The following example demonstrates how to create a form-level validation method. The sample tests that all the text boxes on a form have received input when a button called *btnValidate* is pressed, and resets the focus to the first text box it encounters without input.

Sample of Visual Basic Code

```
Private Sub btnValidate_Click(ByVal sender As System.Object, ByVal e _
   As System.EventArgs) Handles btnValidate.Click
   Dim aControl As System.Windows.Forms.Control
   ' Loops through each control on the form
   For Each aControl In Me.Controls
      ' Checks to see if the control being considered is a Textbox and
      ' if it contains an empty string
      If TypeOf aControl Is TextBox AndAlso aControl.Text = "" Then
      ' If a textbox is found to contain an empty string, it is
         ' given the focus and the method is exited.
         aControl.Focus()
         Exit Sub
      End If
   Next
End Sub
```

Sample of Visual C# Code

```
private void btnValidate_Click(object sender, System.EventArgs e)
   {
   // Loops through each control on the form
   foreach (System.Windows.Forms.Control aControl in this.Controls)
   {
      // Checks to see if the control being considered is a Textbox and
      // if it contains an empty string
      if (aControl is System.Windows.Forms.TextBox & aControl.Text ==
         "")
      {
         // If a textbox is found to contain an empty string, it is
         // given the focus and the method is exited.
         aControl.Focus();
```

```
            return;
        }
    }
}
```

Form-Level Keyboard Handler

A keyboard handler is a somewhat more sophisticated technique for form-level validation. A centralized keyboard handler enables you to manage data input for all fields on a form. For example, you could create a method that enables command buttons only after appropriate input has been entered into each field and that performs specific actions with each keystroke.

The *KeyPress*, *KeyDown*, and *KeyUp* events implement a form-level keyboard handler. If a form has no visible and enabled controls, it will automatically raise keyboard events. If there are any controls on the form, however, the form will not automatically raise these events. For the form to raise these events, the *KeyPreview* property of the form must be set to *True*. When set to *True*, the form will raise keystroke events before the control that has the focus. For example, assume there is a *KeyPress* handler for the form and a *KeyPress* handler for a text box on that form, and that the *KeyPreview* property of the form is set to *True*. When a key is pressed, the form will raise the *KeyPress* event first, and the form's *KeyPress* event handler will execute first. When execution has completed, the text box's *KeyPress* event handler will execute.

Providing User Feedback

When invalid input is entered in a field, the user should be alerted and given an opportunity to correct the error. There are many ways to inform the user of an input error. If the error is obvious and self-explanatory, an audio cue can alert the user to the problem. You can use the *Beep* method to produce an attention-getting sound.

Sample of Visual Basic Code

```
' This line causes an audible beep
Beep()
```

Sample of C# Code

```
// This line causes an audible beep
System.Console.Beep();
```

Other ways to draw a user's attention to an error include changing the *BackColor* or *ForeColor* property of a control. For example, a text box with invalid input might have its *BackColor* changed to red.

If more detailed messages are required, you can use the *MessageBox.Show* method. This method displays a small, modal dialog box that contains an informative message. Because it is displayed modally, it halts program execution and is impossible for the user to ignore. The following shows how to call the *MessageBox.Show* method, which includes an informative message:

Sample of Visual Basic Code

```
MessageBox.Show("That value is not valid for this control")
```

Sample of Visual C# Code

```
MessageBox.Show("That value is not valid for this control");
```

The *ErrorProvider* Component

The *ErrorProvider* component provides an easy way to communicate validation errors to your users. It enables you to set an error message for each control on your form when the input is not valid. When an error message is set, an icon indicating the error will appear next to the control, and the error message text will be shown as a tool tip when the mouse hovers over the affected control. The *ErrorProvider* component can be found in the Windows Forms tab of the Toolbox.

Displaying an Error

To cause an error condition to be displayed next to a control, use the *SetError* method of the *ErrorProvider* component. The *SetError* method requires the name of the control to be set and the text to be provided. It is invoked as shown here:

Sample of Visual Basic Code

```
' This example assumes the existence of a control named nameTextBox and
' an ErrorProvider named myErrorProvider
myErrorProvider.SetError(nameTextBox, "Name cannot be left blank!")
```

Sample of Visual C# Code

```
// This example assumes the existence of a control named nameTextBox and
// an ErrorProvider named myErrorProvider
myErrorProvider.SetError(nameTextBox, "Name cannot be left blank!");
```

When this line of code is executed, an icon will be displayed next to the *nameTextBox* control, and the specified text will appear in a tool tip box when the mouse hovers over the control.

You can also set an error at design time. In the Properties window, you will find that when you add an *ErrorProvider* control to your form, each control has a new property called *Error on x* where *x* is the name of the *ErrorProvider* component. You can set this property to a value at design time in the Properties window. If a value is set for the error, the control will immediately show an error at run time.

Different properties of the *ErrorProvider* component affect how the information is displayed to the user. The *Icon* property controls which icon is displayed next to the control. You might want to have multiple error providers on a single form: one that reports errors and one that reports warnings. You could use different icons for each to provide visual cues to the user. Another property is the *BlinkStyle* property. This property determines whether the error icon blinks when displayed. The *BlinkRate* property determines how rapidly the icon blinks.

To create a validation handler that uses the *ErrorProvider* component:

1. Create your form and add an *ErrorProvider* component to it.

   The *ErrorProvider* component appears in the component tray.
2. Set the *CausesValidation* property of the control for which you want to provide errors to *True* if it is not already true.
3. In the event handler for that control's *Validating* event, test the value. If an error condition occurs, use the *SetError* method to set the error to be displayed. The following example demonstrates a validation handler for a text box named pswordTextBox and an error provider named myErrorProvider:

Sample of Visual Basic Code

```
Private Sub pswordTextBox_Validating(ByVal sender as Object, _
   ByVal e As System.ComponentModel.CancelEventArgs) Handles _
   pswordTextBox.Validating
   ' Validate the entry
   If pswordTextBox.Text = "" Then
      ' Set the error for an invalid entry
      myErrorProvider.SetError(pswordTextBox, _
         "Password cannot be blank!")
   Else
      ' Clear the error for a valid entry-no error will be displayed
      myErrorProvider.SetError(pswordTextBox, "")
   End If
End Sub
```

Sample of Visual C# Code

```
private void pswordTextBox_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   // Validate the entry
   if (pswordTextBox.Text == "")
      // Set the error for an invalid entry
      myErrorProvider.SetError(pswordTextBox,
         "Password cannot be blank!");
   else
      // Clear the error for a valid entry-no error will be displayed
      myErrorProvider.SetError(pswordTextBox, "");
}
```

Implementing *IDataErrorInfo* in WPF Applications

Validation for WPF applications was covered extensively in Chapter 6, "Working with Data Binding." By adding *ValidationRule* objects to the *Binding.ValidationRules* collection, you can validate user input data. Implementing *IDataErrorInfo* in your custom classes, however, enables you to build validation into your data objects to provide an additional level of validation control.

IDataErrorInfo consists of two members: *Error*, which provides an error message that indicates the problem with the object, and *Item*, which provides an error message that indicates what the problem is with a property of the object. The value of the *Error* property can be accessed only by directly querying the property value and is frequently not implemented. However, the *Item* property provides information to the *Validation.Errors* collection in WPF and can be used by built-in validation mechanisms.

Implementing *IDataErrorInfo*

Typically, the bulk of implementation of *IDataErrorInfo* is found in the *Item* property. Implementation of the *Item* property requires you to provide validation for each property that requires validation. Consider the following example:

Sample of Visual Basic Code

```
Default Public ReadOnly Property Item(ByVal columnName As String) _
    As String Implements System.ComponentModel.IDataErrorInfo.Item
    Get
        Dim result As String = Nothing
        If columnName = "FirstName" Then
            If Not FirstName = "John" Then
                result = "First name must equal John"
            End If
        End If
        If columnName = "LastName" Then
            If Not LastName = "Smith" Then
                result = "Last name must equal Smith"
            End If
        End If
        Return result
    End Get
End Property
```

Sample of C# Code

```
    public string this[string columnName]
    {
        get
        {
           string result = null;
           if (columnName == "FirstName")
           {
               if (!(FirstName == "John"))
                   result = "First Name must be John";
           }
           if (columnName == "LastName")
           {
               if (!(LastName == "Smith"))
                   result = "Last Name must be Smith";
           }
           return result;
        }
    }
```

In this rather useless example, the *Item* property provides validation for two properties: *FirstName* and *LastName*, in which the *FirstName* property must be set to "John" and the *LastName* property must be set to "Smith". The name of the column being validated by the method is passed to the method in the *columnName* parameter, and you can test which column is being validated. You can then validate the value of the property in question. In this example, if the property being tested is not set to an appropriate value, an error string is returned. Otherwise, a null string is returned.

After *IDataErrorInfo* is implemented, the WPF Validation apparatus can be used to validate values for the object's properties. A control bound to an object implementing *IDataErrorInfo* should set *ValidatesonDataErrors* to *True*, as shown here:

```
<TextBox.Text>
    <Binding Source="{StaticResource TheCollection}"
        Path="LastName" UpdateSourceTrigger="LostFocus"
        ValidatesOnDataErrors="True"/>
</TextBox.Text>
```

Also note in this example that the *UpdateSourceTrigger* property is set to *LostFocus*. This also determines when the bound content in the control is validated. When the control loses focus, the *Item* method of *IDataErrorInfo* is executed. You can validate the bound content every time it changes by setting this property to *PropertyChanged*, but that tends to be processor-intensive.

If a validation error is found and an error string is returned by the *Item* method, it is added to the *Validation.Errors* collection and can be accessed at run time. You can use a *Style* element to set the tool tip on the error string at run time, as shown here:

```
        <Style TargetType="{x:Type TextBox}">
            <Style.Triggers>
                <Trigger Property="Validation.HasError" Value="true">
                    <Setter Property="ToolTip"
                Value="{Binding RelativeSource={RelativeSource Self},
                       Path=(Validation.Errors)[0].ErrorContent}"/>
                </Trigger>
            </Style.Triggers>
        </Style>
```

In this example, *Style* is applied to controls of type *TextBox*, and it invokes a trigger when the *Validation.HasError* property that sets the *ToolTip* property to the error string for this control is true. That string will be visible when the mouse hovers over the control and will remain until the error is cleared.

You can set the *Validation.ErrorTemplate* attached property to provide a different template for the control when a validation error is present. The *Validation.ErrorTemplate* attached property takes a *ControlTemplate* value. This lesson's practice demonstrates creating an *ErrorTemplate* attached property.

Implementing *IDataErrorInfo*

In this practice, you implement *IDataErrorInfo* in a simple class and then bind it to controls in the user interface. You create a style that automatically sets *ToolTip* to an error string and creates an *ErrorTemplate* attached property that alters the appearance of the control when an error is present.

EXERCISE Implementing *IDataErrorInfo*

1. In Visual Studio, create a new WPF application named **Lesson 2**.
2. In the Code editor for the *MainWindow* class, add the following class.

   Note that it contains an empty implementation for *IDataErrorInfo*; this is the implementation that is automatically generated by Visual Studio.

   Sample of Visual Basic Code

```
Public Class Customer
    Implements System.ComponentModel.IDataErrorInfo
    Private mFirstName As String = ""
    Public Property FirstName As String
        Get
            Return mFirstName
        End Get
        Set(ByVal value As String)
            mFirstName = value
        End Set
    End Property
    Private mLastName As String = ""
    Public Property LastName As String
        Get
            Return mLastName
        End Get
        Set(ByVal value As String)
            mLastName = value
        End Set
    End Property
    Public ReadOnly Property [Error] As String _
        Implements System.ComponentModel.IDataErrorInfo.Error
        Get

        End Get
    End Property

    Default Public ReadOnly Property Item(ByVal columnName As String) As String _
        Implements System.ComponentModel.IDataErrorInfo.Item
        Get

        End Get
    End Property
End Class
```

Sample of C# Code

```
    public class Customer : System.ComponentModel.IDataErrorInfo
    {
        string mFirstName = "";
        public string FirstName
        {
            get
            {
                return mFirstName;
            }
            set
            {
                mFirstName = value;
            }
        }
        string mLastName = "";
        public string LastName
        {
            get
            {
                return mLastName;
            }
            set
            {
                mLastName = value;
            }
        }

        public string Error
        {
            get { }
        }

        public string this[string columnName]
        {
            get { }
        }
    }
```

3. In the *Error* property, add the following line to the getter.

   Sample of Visual Basic Code

   ```
   Throw New NotImplementedException
   ```

   Sample of C# Code

   ```
   throw new NotImplementedException();
   ```

4. In the *Item* property, add the following implementation:

   ```
   Get
       Dim Result As String = Nothing
       Select Case columnName
           Case "FirstName"
               If Not FirstName.Length > 1 Then
                   Result = "First Name must be at least 2 characters long"
   ```

```
            ElseIf FirstName.Length > 10 Then
                Result = "First Name cannot be longer than 10 characters"
            End If
        Case "LastName"
            If Not LastName.Length > 1 Then
                Result = "Last Name must be at least 2 characters long"
            End If
    End Select
    Return Result
End Get
```

Sample of C# Code

```
get
{
    string result = null;
    switch (columnName)
    {
        case "FirstName":
            if (!(FirstName.Length > 1))
                result = "First Name must be at least 2 characters long";
            else if (FirstName.Length > 10)
                result =
                    "First Name cannot be longer than 10 characters";
            break;
        case "LastName":
            if (!(LastName.Length > 1))
                result = "Last Name must be at least 2 characters long";
            break;
    }
    return result;
}
```

5. In the XAML designer, add the following line to the Window declaration to import the project namespace.

   Note that if your project is named anything other than Lesson 2, you will have to modify this line accordingly.

   ```
   xmlns:my="clr-namespace:Lesson_2"
   ```

6. In the XAML designer, add a Windows.Resources section as follows to create an instance of the *Customer* class as a resource for your application.

   ```
       <Window.Resources>
           <my:Customer x:Key="myCustomer" />
       </Window.Resources>
   ```

7. In the XAML designer, add the following XAML code to the Grid declaration.

   Note that the *Text* property of each text box is bound to a property of the myCustomer resource and that the *UpdateSourceTrigger* and the *ValidatesOnDataErrors* properties of the binding are set to *LostFocus* and *True*, respectively.

   ```
           <TextBox Height="23" HorizontalAlignment="Left"
               Margin="61,32,0,0" Name="TextBox1" VerticalAlignment="Top" Width="120"
   ```

```
>
            <TextBox.Text>
                <Binding Source="{StaticResource myCustomer}" Path="FirstName"
                    UpdateSourceTrigger="LostFocus" ValidatesOnDataErrors="True"/>
            </TextBox.Text>
        </TextBox>
        <TextBox Height="23" HorizontalAlignment="Left" Margin="61,81,0,0"
            Name="TextBox2" VerticalAlignment="Top" Width="120" >
            <TextBox.Text>
                <Binding Source="{StaticResource myCustomer}" Path="LastName"
                    UpdateSourceTrigger="LostFocus" ValidatesOnDataErrors="True"/>
            </TextBox.Text>
        </TextBox>
```

8. In the XAML designer, add the following style to the Windows.Resources section.

This style has a target type of *Textbox*, so it is applied automatically to both *Textbox* controls in this application and includes a trigger that fires when the *Validaton.HasError* property for the bound control is *True*. When the trigger fires, it invokes a setter that sets the *ToolTip* property for *Textbox* to the *ErrorContent* component of the validation error, which will display the string returned by the *Item* method in the form of a tool tip whenever the mouse hovers over the control.

```
<Style TargetType="{x:Type TextBox}">
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="true">
            <Setter Property="ToolTip"
        Value="{Binding RelativeSource={RelativeSource Self},
               Path=(Validation.Errors)[0].ErrorContent}"/>
       </Trigger>
    </Style.Triggers>
</Style>
```

9. Add the following setter to the trigger in the style described previously.

This setter sets the *ErrorTemplate* property for the control when the *Validation.HasError* property is true. The template described here outlines the text box with the error in purple and displays red exclamation points as well.

```
<Setter Property="Validation.ErrorTemplate">
    <Setter.Value>
        <ControlTemplate>
            <DockPanel LastChildFill="True">
                <TextBlock DockPanel.Dock="Right" Foreground="Red"
                    FontSize="16pt">!!!</TextBlock>
                <Border BorderBrush="Purple" BorderThickness="3">
                    <AdornedElementPlaceholder />
                </Border>
            </DockPanel>
        </ControlTemplate>
    </Setter.Value>
</Setter>
```

10. Press F5 to build and run your application.

Lesson Summary

- Form-level validation validates all fields on a form simultaneously. Field-level validation validates each field as data is entered and provides a finer level of control over validation.
- The *TextBox* control contains several design-time properties that restrict the values users can enter.
- Keyboard events enable you to validate keystrokes; they are raised by the control that has the focus and is receiving input. The form will also raise these events if the *KeyPreview* property is set to *True*.
- The *Char* structure contains several static methods that are useful for validating character input.
- The *Validating* event occurs before the control loses focus and should be used to validate user input. This event occurs only when the *CausesValidation* property of the control that is about to receive the focus is true. To keep the focus from moving away from the control in the *Validating* event handler, set the *CancelEventArgs.Cancel* property to *True* in the event handler.
- The *ErrorProvider* component enables you to set an error for a control at run time that displays a visual cue and an informative message to the user. To display an error at run time, use the *ErrorProvider.SetError* method.
- *IDataErrorInfo* is an interface in the *System.ComponentModel* namespace that can be implemented in classes to provide validation in WPF applications. Error strings returned by the *Item* property are automatically set to the *ErrorContent* property of the first *Validation.Errors* item for the binding in question.
- You can provide a different template for a control to be displayed when there is a validation error by setting the *ErrorTemplate* attached property of that control.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, “Validating User Input.” The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE **ANSWERS**

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the “Answers” section at the end of the book.

1. How can you set the *ErrorContent* string of a *Validation.Errors* object for a specific control from a class that implements *IDataErrorInfo*?
   - **A.** Return the error string as the return value of the *Error* property. It will automatically be set to the *ErrorContent* string of the correct *Validation.Errors* object when an error occurs in a control bound to this object.
   - **B.** Return the error string as the return value of the *Item* property. It will automatically be set to the *ErrorContent* string of the correct *Validation.Errors* object when an error occurs in a control bound to this object.
   - **C.** Set the *ErrorContent* string directly in the *Error* property.
   - **D.** Set the *ErrorContent* string directly in the *Item* property.
2. Which of the following represents the correct order in which focus events are raised?
   - **A.** *Enter*, *GotFocus*, *Validating*, *Validated*, *Leave*, *LostFocus*
   - **B.** *Enter*, *GotFocus*, *Validating*, *Validated*, *LostFocus*, *Leave*
   - **C.** *Enter*, *GotFocus*, *Leave*, *Validating*, *Validated*, *LostFocus*
   - **D.** *Enter*, *GotFocus*, *LostFocus*, *Leave*, *Validating*, *Validated*

Case Scenario

In the following case scenarios, you apply what you've learned about how to use controls to design user interfaces. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario: The Writer Completeness Chart

You have been contracted by a firm that manages a large group of technical writers, each one working on a different book. The firm has asked you to develop an application to help manage the status and schedule of its writers. You are building a database and associated application for this purpose. Your task is to build the user interface for an overview screen.

The technical requirements are as follows:

- The application should be built using WPF.
- The database will provide a data row about each writer with the following information: Name, Project Title, Projected Number of Chapters, Completed Number of Chapters.
- The application should display the percentage of the project done, expressed as the completed number of chapters divided by the projected number of chapters, and it should be displayed graphically as a pie chart.
- Information for all writers should be visible at once in the application so that it can be browsed easily.

Answer the following question for your manager:

- What is the best strategy for meeting these requirements?

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

- **Practice 1** Implement the project described in the case scenario.
- **Practice 2** Create a *DataGridTemplateColumn* property that accepts a date as data type and displays the date as a text box when not editing and as a date picker when editing.
- **Practice 3** Create a Windows Forms application to enter data for the Northwind Customers table with appropriate field-level validation on each control.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just the content covered in this chapter, or you can test yourself on all the 70-511 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO **PRACTICE TESTS**

For details about all the practice test options available, see the "How to Use the Practice Tests" section in this book's Introduction.