Applied ADO.NET: Building Data-Driven Solutions

MAHESH CHAND
AND
DAVID TALBOT

Apress™

Applied ADO.NET: Building Data-Driven Solutions

ISBN (pbk): 1-59059-073-2

Printed and bound in the United States of America 12345678910

Technical Reviewer: Philip Pursglove

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Project Manager: Tracy Brown Collins

Development Editor: Philip Pursglove

Copy Editor: Kim Wimpsett

Compositor: Diana Van Winkle, Van Winkle Design Group

Artist and Cover Designer: Kurt Krames

Indexer: Ron Strauss

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 7

Data Binding and Windows Forms Data-Bound Controls

When it comes to developing interactive database applications, it's difficult to resist using data-bound controls. Data-bound controls are easy to use, and they also provide many handy, built-in features. You used `DataGrid`, `ListBox`, and other data-bound controls in the previous chapters. In this chapter, we discuss the basics of data binding, how to use data-bound controls, and how to develop interactive database applications using these controls with a minimal amount of time and effort.

Both Windows Forms and Web Forms provide a rich set of data-bound controls, which help developers build data-driven Windows and Web applications. In this chapter, we concentrate on Windows Forms. Chapter 16 covers data binding in Web Forms.

Understanding Data Binding

So what are data-bound controls? You've already seen the `DataGrid` and `ListBox` controls in the previous chapters. You used these controls to display data from a data source. Data-bound controls are Windows controls that represent and manipulate data in Graphical User Interface (GUI) forms. Both Windows Forms and Web Forms provide a variety of flexible and powerful data-bound controls. These data-bound controls vary from a `TextBox` to a `DataGrid`.

The process of binding a data source's data to GUI controls is called *data binding*. Most of the editable Windows controls provide data binding, either directly or indirectly. These controls contain members that connect directly to a data source, and then the control takes care of displaying the data and other details. For example, to view data in a `DataGrid` control, you just need to set its `DataSource` property to a data source. This data source could be a `DataSet`, `DataView`, array, collection, or other data source. Data-bound controls can display data, and

they are smart enough to display properties (metadata) of the stored data such as data relations.

You can divide data binding into two categories: simple data binding and complex data binding. In *simple data binding*, a control displays data provided by a data feed. In fact, the control itself is not capable of displaying complex data. Setting the `Text` property of a `TextBox` or `Label` control is an example of simple data binding. *Complex data binding*, on the other hand, allows controls to bind to multiple columns and complex data. Binding an entire database table or multiple columns of a database table to a `DataGrid` or a `ListBox` control is an example of complex data binding.

Using the Binding Class

The `Binding` class, defined in the `System.Windows.Forms` namespace, represents simple binding between the data source item and a control.

Constructing a Binding Object

The `Binding` class constructor, which creates an instance of the `Binding` class, takes three arguments: a data-bound control's property name, a data source as an `Object`, and a data member, usually the name of the data source columns as a string. You define the `Binding` class constructor as follows:

```
Public Sub New( _
   ByVal propertyName As String, _
   ByVal dataSource As Object, _
   ByVal dataMember As String _
)
```

In this syntax, `dataSource` can be a `DataSet`, `DataTable`, `DataView`, `DataViewManager`, any class that implements `IList`, and a class object. Listing 7-1 binds the `Employees.FirstName` column of a `DataSet` to the `Text` property of a `TextBox`.

Listing 7-1. Binding a `TextBox` *Using* `Binding`

```
Dim ds As DataSet = New DataSet()
ds = GetDataSet("Employees")
Dim bind1 As Binding
bind1 = New Binding("Text", ds, "Employees.FirstName")
```

```
TextBox1.DataBindings.Add(bind1)
```

Besides the previous two controls, you can perform simple binding on many controls including `Button`, `CheckBox`, `CheckedListBox`, `ComboBox`, `DateTimePicker`, `DomainUpDown`, `GroupBox`, `HscrollBar`, `Label`, `LinkLabel`, `ListBox`, `ListView`, `MonthCalender`, `NumericUpDown`, `PictureBox`, `ProgressBar`, `RadioButton`, `RichTextBox`, `ScrollBar`, `StatusBar`, `TextBox`, `TreeView`, and `VscrollBar`. Listing 7-2 binds the `Text` property of a `ComboBox`, `Label`, and `Button` control with the `DataTable`'s `LastName`, `City`, and `Country` columns (respectively).

Listing 7-2. Binding Multiple Controls Using `Binding`

```
ComboBox1.DataBindings.Add _
    (New Binding("Text", ds, "Employees.LastName"))
TextBox2.DataBindings.Add _
   (New Binding("Text", ds, "Employees.EmployeeID"))
Label4.DataBindings.Add(
    New Binding("Text", ds, "Employees.City"))
Button1.DataBindings.Add(
    New Binding("Text", ds, "Employees.Country"))
```

Understanding the BindingsCollection Class

The `BindingsCollection` class represents a collection of `Binding` objects for a control. You access the `BindingsCollection` class through the control's `DataBindings` property. The `BindingsCollection` class provides members to add, count, and remove `Binding` objects to the collection. Listing 7-1 and Listing 7-2 used the `Add` method of `BindingsCollection` to add a `Binding` object to the collection.

The `BindingsCollection` class has three properties: `Count`, `Item`, and `List`. The `Count` property returns the total number of items in the collection. The `Item` property returns the `Binding` object at a specified index, and the `List` property returns all the items in a collection as an `ArrayList`.

The `Add` method of `BindingsCollection` adds a `Binding` object to the collection. The `Remove` method deletes a `Binding` object from the collection. The `RemoveAt` method removes a `Binding` object at the specified index. The `Clear` method removes all the `Binding` objects from the collection.

Listing 7-3 counts the total number of `Binding` objects associated with a control and removes the `Binding` objects from various controls.

Listing 7-3. Counting and Removing `Binding` *Objects*

```
MessageBox.Show("Total Bindings: " + _
Button1.DataBindings.Count.ToString())
TextBox1.DataBindings.RemoveAt(0)
TextBox2.DataBindings.Clear()
```

NOTE *The* `BindingsCollection` *class is a collection of* `Binding` *objects. The index of* `Binding` *objects in a collection is 0 based, which means the 0th item of the collection is the first item and (n–1)th item in the collection is the nth item.*

Setting Binding Class Members

The `Binding` class provides six properties: `BindingManagerBase`, `BindingMemberInfo`, `Control`, `DataSource`, `IsBinding`, and `PropertyName`.

`BindingManagerBase` represents the `BindingManagerBase` object, which manages the binding between a data source and data-bound controls.

The `BindingMemberInfo` property object is a `BindingMemberInfo` structure that contains the information about the binding. The `BindingMemberInfo` structure has three properties: `BindingField`, `BindingMember`, and `BindingPath`. The `BindingField` property returns the data-bound control's property name. `BindingMember` returns the information used to specify the data-bound control's property name, and `BindingPath` returns the property name, or the period-delimited hierarchy of property names, that precedes the data-bound object's property.

Listing 7-4 reads the bindings available on all the controls and displays their information by using the `BindingMemberInfo` property.

Listing 7-4. Reading All the Bindings of a Form

```
Dim str As String
  Dim curControl As Control
  Dim curBinding As Binding
  For Each curControl In Me.Controls
    For Each curBinding In curControl.DataBindings
      Dim bInfo As BindingMemberInfo = _
      curBinding.BindingMemberInfo
```

```
      str = "Control: " + curControl.Name
      str += ", BindingPath: " + bInfo.BindingPath
      str += ", BindingField: " + bInfo.BindingField
      str += ", BindingMember: " + bInfo.BindingMember
      MessageBox.Show(str)
    Next curBinding
  Next curControl
```

The `Control` and `DataSource` properties return the control and data source that belong to this binding. The `IsBinding` property returns `True` if the binding is active; otherwise it returns `False`. `PropertyName` returns the name of the bound control's property that can be used in data binding. Listing 7-5 displays the `DataSource` and `PropertyName` properties of a `TextBox`.

Listing 7-5. Reading a `TextBox` *Control's Binding Properties*

```
If (TextBox1.DataBindings(0).IsBinding) Then
      Dim ds As DataSet = _
      CType(TextBox1.DataBindings(0).DataSource, DataSet)
      str = "DataSource : " + ds.Tables(0).TableName
      str += ", Property Name: " + _
      TextBox1.DataBindings(0).PropertyName
      MessageBox.Show(str)
End If
```

In addition to the previously discussed properties, the `Binding` class also provides two protected methods: `OnParse` and `OnFormat`. `OnParse` raises the Parse event, and `OnFormat` raises the `Format` event. The `Parse` event occurs when the value of a data-bound control is changing, and the `Format` event occurs when the property of a control is bound to a data value. The event handler for both the `Parse` and `Format` events receives an argument of type `ConvertEventArgs` containing data related to this event, which has two members: `DesiredType` and `Value`. `DesiredType` returns the data type of the desired value, and `Value` gets and sets the value of the `ConvertEventArgs` object.

Now let's say you want to convert text and decimal values for a `Binding` for a `TextBox`. You write the code in Listing 7-6, where you change the `Binding` type and add event handlers for the `Binding` objects for `Format` and `Parse` members.

NOTE *This listing uses the Customers table instead of Employees because the Employees table doesn't have any decimal data. If you want to use the Employees table, you could convert a Date type to a String type.*

Listing 7-6. Adding Format and Parse Event Handlers

```
Dim bind2 As Binding = New Binding _
     ("Text", ds, "customers.custToOrders.OrderAmount")
AddHandler bind1.Format, AddressOf DecimalToCurrencyString
AddHandler bind1.Parse, AddressOf CurrencyStringToDecimal
Private Sub DecimalToCurrencyString(ByVal sender As Object, _
 ByVal cevent As ConvertEventArgs)
    If Not cevent.DesiredType Is GetType(String) Then
      Exit Sub
    End If
    cevent.Value = CType(cevent.Value, Decimal).ToString("c")
End Sub
Private Sub CurrencyStringToDecimal(ByVal sender As Object, _
  ByVal cevent As ConvertEventArgs)
    If Not cevent.DesiredType Is GetType(Decimal) Then
      Exit Sub
    End If
    cevent.Value = Decimal.Parse(cevent.Value.ToString, _
    NumberStyles.Currency, Nothing)
End Sub
```

NOTE *To test this code, you need to create a* `DataSet` *from the Employees table of the Northwind database and use it as a data source when constructing a* `Binding` *object. Also, don't forget to add a reference to the* `System.Globalization` *namespace because the* `NumberStyle` *enumeration is defined in this namespace.*

Understanding the BindingManagerBase Functionality

The `BindingManagerBase` class is an abstract base class. You use its functionality through its two derived classes: `CurrencyManager` and `PropertyManager`.

By default data-bound controls provide neither data synchronization nor the position of the current item. The `BindingManagerBase` object provides the data synchronization in Windows Forms and makes sure that all controls on a form are updated with the correct data.

Question and Answer

Question: What is data synchronization?

Answer: Have you ever developed database applications in Visual Basic 6.0 or Microsoft Foundation Classes (MFC)? In both of those languages, a data-bound control lets you navigate through data from one record to another and update data in the controls available on the form. As you move to the next record, the next row was fetched from the data source and every control was updated with the current row's data. This process is called *data synchronization*.

OK, now let's say a form has three controls: a `TextBox`, a `Label`, and a `PictureBox`. All three controls support data binding from a `DataSet`, which is filled with the data from the Employees table. The `TextBox` control displays `FirstName`, the `Label` control displays `LastName`, and the `PictureBox` control displays Photo properties (columns) of the `DataSet`. All of the controls must be synchronized in order to display the correct first name, last name, and photo for the same employee.

`CurrencyManager` accomplishes this synchronization by maintaining a pointer to the current item for the list. All controls are bound to the current item so they display the information for the same row. When the current item changes, `CurrencyManager` notifies all the bound controls so that they can refresh their data. Furthermore, you can set the `Position` property to specify the row in the `DataSet` or `DataTable` to which the controls point. Figure 7-1 shows the synchronization process.

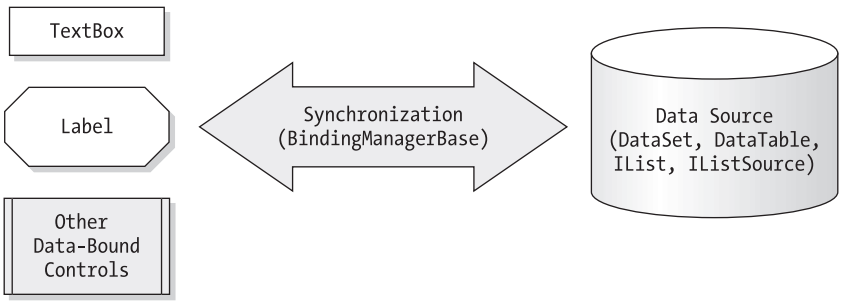

Figure 7-1. Synchronization between a data source and data-bound controls

Using the BindingManagerBase Class Members

As you learned, the `Binding` property returns the collection of binding objects as a `BindingsCollection` object that `BindingManagerBase` manages. Listing 7-7 creates a `BindingManagerBase` object for the form and reads all of the binding controls.

Listing 7-7. Reading All Controls Participating in Data Binding

```
' Get the BindingManagerBase
Dim bindingBase As BindingManagerBase = _
     Me.BindingContext(ds, "Employees")
Dim bindingObj As Binding
 ' Read each Binding object from the collection
For Each bindingObj In bindingBase.Bindings
      MessageBox.Show(bindingObj.Control.Name)
Next bindingObj
```

NOTE *To read a form's controls that are participating in data binding, you must make sure that the form's data source and control's data source are the same.*

The `Count` property returns the total number of rows being managed by `BindingManagerBase`. The `Current` property returns the current object, and the `Position` property represents (both gets and sets) the position in the underlying list to which controls bound to this data source point. We use these properties in the following sample examples.

Table 7-1 describes the BindingManagerBase class methods.

Table 7-1. The BindingManagerBase *Class Methods*

METHOD	DESCRIPTION
AddNew	Adds a new item to the list
CancelCurrentEdit	Cancels the current edit operation
EndCurrentEdit	Ends the current edit operation
GetItemProperties	Returns the list of property descriptions for the data source
RemoveAt	Deletes the row at the specified index
ResumeBinding	Resumes data binding
SuspendBinding	Suspends data binding
GetListName	Protected. Returns the name of the list
OnCurrentChanged	Raises the CurrentChanged event, which occurs when the bound value changes
PullData	Pulls data from the data-bound control into the data source
PushData	Pushes data from data source into the data-bound control
UpdateIsBinding	Updates the binding

Besides the properties and methods discussed previously, the BindingManagerBase class provides two events: CurrentChanged and PositionChanged. The CurrentChanged event occurs when the bound value changes, and the PositionChanged event occurs when the position changes.

Using CurrencyManager and PropertyManager

CurrencyManager manages a list of Binding objects on a form. It's inherited from the BindingManagerBase class. Besides the functionality provided by BindingManagerBase, the CurrencyManager provides two members: a List property and a Refresh method. The List property returns the list of bindings maintained by CurrencyManager as an IList object. To convert an IList to other objects, you need to cast it with the type of the object, which must implement IList. Some of the objects that implement IList are DataView, DataTable, DataSet, Array, ArrayList, and CollectionBase.

You create a CurrencyManager object by using the BindingContext object, which returns either CurrencyManager or PropertyManager, depending on the value of the data source and data members passed to the Item property of BindingContext. If the data source is an object that can only return a single property (instead of a list of objects), the type will be PropertyManager. For example, if you specify a TextBox

control as the data source, PropertyManager will be returned. If the data source is an object that implements IList, IListSource, or IBindingList, such as a DataSet, DataTable, DataView, or an Array, CurrencyManager will be returned.

You can create a CurrencyManager from objects such as a DataView and vice versa. For example, the following code creates a CurrencyManager from a DataView and a DataView from a CurrencyManager:

```
Dim dv As DataView
dv = New DataView(ds.Tables("Customers"))
Dim curManager1 As CurrencyManager = DataGrid1.BindingContext(dv)
Dim list As IList = curManager1.List
Dim dv1 As DataView = CType(curManager1.List, DataView)
Dim curManager2 As CurrencyManager = Me.BindingContext(ds1)
```

Unlike CurrencyManager, PropertyManager doesn't provide any additional members besides the members provided by its base class, BindingManagerBase.

Understanding BindingContext

Each object inherited from the Control class has a BindingContext object attached to it. BindingContext manages the collection of BindingManagerBase objects for that object such as a form. The BindingContext creates the CurrencyManager and PropertyManager objects, which were discussed previously. Normally you use the Form class's BindingContext to create a CurrencyManager and PropertyManager for a form and its controls, which provide data synchronization.

The Item property of BindingContext returns the BindingManagerBase (either CurrencyManager or PropertyManager). The Contains methods returns True if it contains the specified BindingManagerBase.

Besides the Item and Contains members, the BindingContext has three protected methods: Add, Clear, and Remove. The Add method adds a BindingManagerBase to the collection, the Clear method removes all items in the collection, and the Remove method deletes the BindingManagerBase associated with the specified data source.

Building a Record Navigation System

Now let's see data binding in action. In this section, you'll develop an application that provides data synchronization. In this application, you'll build a record navigation system. The controls will display records, and then when you click the

Move Next, Move Last, Move Previous, and Move First buttons, the controls will display the respective records.

To begin, create a Windows application and design a form that looks like Figure 7-2. For this example, you don't have to place the `ReadBindingMemberInfo` and `Remove` controls. Add a `ComboBox` control, two `TextBox` controls, a `ListBox` control, some `Label` controls, and some `Button` controls. The Load Data button loads data to the controls and attaches `Binding` objects to the `BindingContext`. You should also add four buttons with brackets as the text (<<, <, >, >>), which represents the Move First, Move Previous, Move Next, and Move Last records.

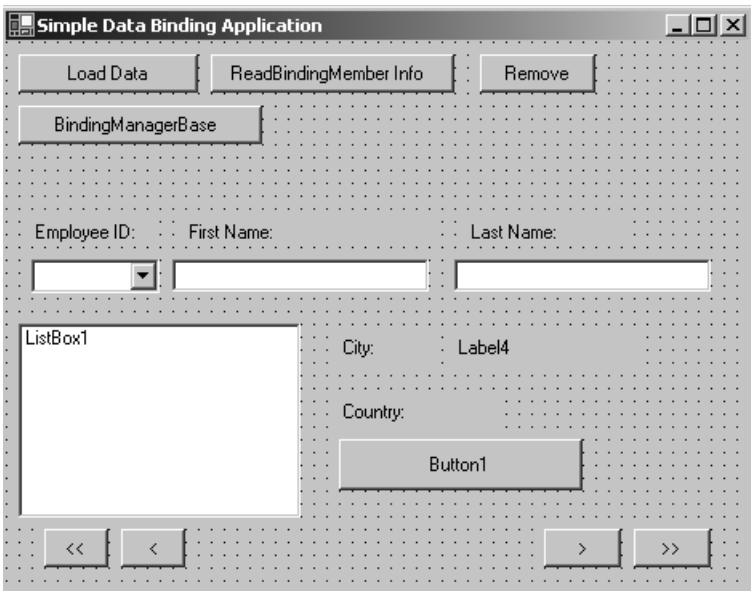

Figure 7-2. Record navigation form

NOTE *You can create your own form, but to save you some time, you can download the code from the Apress (`www.apress.com`) or C# Corner (`www.c-sharpcorner.com`) Web sites. Open the project in Visual Studio .NET (VS .NET) to understand it better.*

As usual, first you add some variables to the project, which shown in Listing 7-8. Don't forget to change your server name; the server name in this example is MCB.

Listing 7-8. Record Navigation System Variables

```
Private ConnectionString As String = "Integrated Security=SSPI;" & _
"Initial Catalog=Northwind;Data Source=MCB;"
Private conn As SqlConnection = Nothing
Private sql As String = Nothing
Private adapter As SqlDataAdapter = Nothing
Private ds As DataSet = Nothing
```

Second, you call the `LoadData` method on the Load Data button click:

```
Private Sub LoadBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles LoadBtn.Click
    LoadData()
End Sub
```

Listing 7-9 shows the `LoadData` and `GetDataSet` methods. The `GetDataSet` method returns a `DataSet` object from the table name passed in the method. The `LoadData` method creates bindings for these controls with different `DataTable` columns.

Listing 7-9. `LoadData` *and* `GetDataSet` *Methods*

```
Private Sub LoadData()
    Dim ds As DataSet = New DataSet()
    ds = GetDataSet("Employees")
    Dim bind1 As Binding
    bind1 = New Binding("Text", ds, "Employees.FirstName")
    TextBox1.DataBindings.Add(bind1)
    TextBox2.DataBindings.Add _
 (New Binding("Text", ds, "Employees.LastName"))
    ComboBox1.DataBindings.Add _
   (New Binding("Text", ds, "Employees.EmployeeID"))
    Label4.DataBindings.Add(New Binding("Text", ds, "Employees.City"))
    Button1.DataBindings.Add(New Binding("Text", ds, "Employees.Country"))
    ListBox1.DataSource = ds.Tables(0).DefaultView
    ListBox1.DisplayMember = "Title"
End Sub
' object based on various parameters.
Public Function GetDataSet(ByVal tableName As String) As DataSet
   sql = "SELECT * FROM " + tableName
    ds = New DataSet(tableName)
```

```
    conn = New SqlConnection()
    conn.ConnectionString = ConnectionString
    adapter = New SqlDataAdapter(sql, conn)
    adapter.Fill(ds, tableName)
    Return ds
End Function
```

The previously discussed steps will load the first row from the Employees table to the controls. Now, the next step is to write code for the move buttons. Listing 7-10 shows the code for all four buttons—Move First, Move Next, Move Previous, and Move Last. As you can see, this code uses the `Position` and `Count` properties of `BindingManagerBase` to set the position of the new record. `BindingContext` and other `Binding` objects manage everything for you under the hood.

Listing 7-10. Move Next, Move Previous, Move First, and Move Last Button Code

```
Private Sub MoveFirstBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MoveFirstBtn.Click
    Me.BindingContext(Me.ds, "Employees").Position = 0
End Sub

Private Sub MovePrevBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MovePrevBtn.Click
    Dim idx As Int32 = _
       Me.BindingContext(Me.ds, "Employees").Position
    Me.BindingContext(Me.ds, "Employees").Position = idx - 1
End Sub

Private Sub MoveNextBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MoveNextBtn.Click
    Dim idx As Int32 = _
       Me.BindingContext(Me.ds, "Employees").Position
    Me.BindingContext(Me.ds, "Employees").Position = idx + 1
End Sub

Private Sub MoveLastBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MoveLastBtn.Click
    Me.BindingContext(Me.ds, "Employees").Position = _
    Me.BindingContext(Me.ds, "Employees").Count - 1
End Sub
```

When you run your application, the first record looks like Figure 7-3. Clicking the Move First, Move Next, Move Previous, and Move Last buttons will navigate you through the first, next, previous, and last records of the table (respectively).

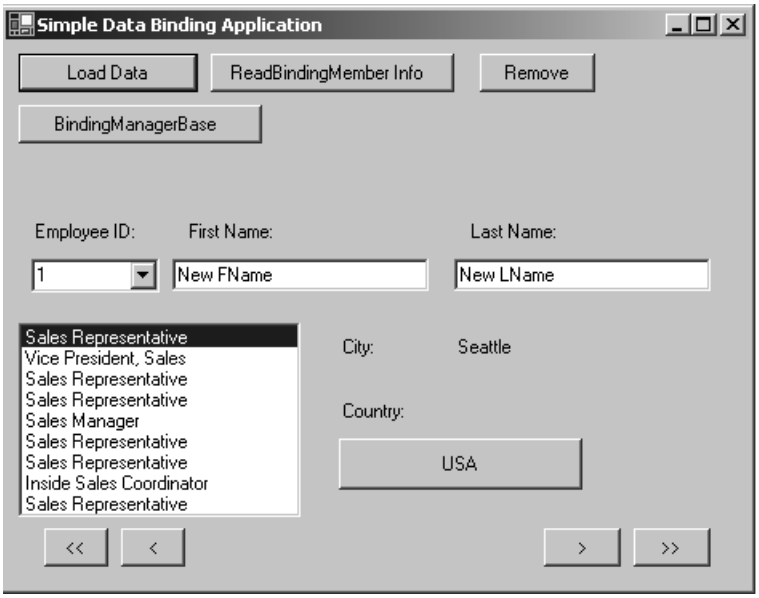

Figure 7-3. Record navigation system in action

Question and Answer

Question: When I click the move buttons, I don't see the pointer in the `ListBox` moving. Why not?

Answer: The `ListBox` control doesn't use the same binding method as simple data-bound controls such as `TextBox` or `Label` controls. We discuss this in the following section.

You just saw how to implement a record navigation system using simple data binding and simple data-bound controls. In the following section, we show you how to build a record navigation system using complex data-bound controls such as `ListBox` and `DataGrid` controls.

Working with Complex Data-Bound Controls

Unlike simple data-bound controls, the complex data-bound controls can display a set of data such as a single column or a collection of columns. The controls such as `DataGrid` are even able to display data from multiple tables of a database. Whereas the simple data-bound controls usually use their `Text` properties for binding, complex data-bound controls use their `DataSource` and `DataMember` properties.

In the following sections, we discuss some of the common complex data-bound controls such as the `ComboBox`, `ListBox`, and `DataGrid`.

The Role of Control Class in Data Binding

The `Control` class is the mother of all Windows controls. This class's basic functionality is required by visual Windows controls that are available from the Toolbox or through other wizards. The `Control` class handles user input through the keyboard, the mouse, and other pointing devices. It also defines the position and size of controls; however, it doesn't implement painting.

If you count the `Control` class members, you'll find that this class is one of the biggest classes available in the .NET Framework Library. In the following sections, you'll explore some of the data-binding functionality implemented by this class.

The `Control` class provides two important and useful properties, which play a vital role in the data-binding process. These properties are `BindingContext` and `DataBinding`. The `BindingContext` property represents the `BindingContext` attached to a control. As discussed earlier, `BindingContext` returns a single `BindingManagerBase` object for all data-bound controls. The `BindingManagerBase` object provides the synchronization for all data-bound controls. The `Control` class also implements the `DataSourceChanged` event, which raises when the data source of a control is changed. We discuss this event in more detail shortly.

Using the ListControl Class

The `ListControl` class is the base class for `ListBox` and `ComboBox` controls and implements the data-binding functionality. The `ListControl` class provides four data-binding properties: `DataManager`, `DataSource`, `DisplayMember`, and `ValueMember`.

The `DataManager` (read-only) property returns the `CurrencyManager` object associated with a `ListControl` class.

The `DataSource` property (both get and set) represents the data source for a `ListControl` class.

The DisplayMember (both get and set) represents a string, which specifies the property of a data source whose contents you want to display. For example, if you want to display the data of a DataTable's Name column in a ListBox control, you set DisplayMember ="Name".

The ValueMember (both get and set) property represents a string, which specifies the property of the data source from which to draw the value. The default value of this property is an empty string ("").

You'll see how to use these properties in the following samples.

ListControl DataBinding-Related Events

Besides the BindingContextChanged event, the ListControl class implements three data-binding events: OnDataSourceChanged, OnDisplayMemberChanged, and OnValueMemberChanged. The OnDataSourceChanged method raises the DataSourceChanged event. This event occurs when the DataSource property of a ListControl class is changed. The OnDisplayMemberChanged method raises the DisplayMemberChanged event, which occurs when the DisplayMember property of the control changes. The OnValueMemberChanged method raises the ValueMemberChanged event, which occurs when the ValueMember property of the control changes.

These events are useful when your program needs a notification when any of these events occur. Listing 7-11 attaches these events with event handlers. The code also shows the handler methods, which will be called when the event occurs.

Listing 7-11. Adding a ListBox *Control Event Handler*

```
' Bind data with controls
  Private Sub BindListControls()
    ComboBox1.DataSource = ds.Tables(0)
    ComboBox1.DisplayMember = "EmployeeID"
    ListBox1.DataSource = ds.Tables(0)
    ListBox1.DisplayMember = "FirstName"
    ListBox2.DataSource = ds.Tables(0)
    ListBox2.DisplayMember = "LastName"
    ListBox3.DataSource = ds.Tables(0)
    ListBox3.DisplayMember = "Title"
  End Sub
```

```
  Private Sub ComboDataSourceChangedMethod(ByVal sender As Object, _
  ByVal cevent As EventArgs) Handles ListBox1.DataSourceChanged
    MessageBox.Show("Data Source changed")
  End Sub
  Private Sub DisplayMemberChangedMethod(ByVal sender As Object, _
  ByVal cevent As EventArgs) Handles ListBox1.DisplayMemberChanged
    MessageBox.Show("Display Member changed")
  End Sub
  Private Sub ValueMemberChangedMethod(ByVal sender As Object, _
  ByVal cevent As EventArgs) Handles ListBox1.ValueMemberChanged
    MessageBox.Show("Value Member changed")
  End Sub
  Private Sub BindingContextChangedMethod(ByVal sender As Object, _
  ByVal cevent As EventArgs) Handles ListBox1.BindingContextChanged
    MessageBox.Show("Binding Context changed")
  End Sub
```

To raise these events, just change the value of the `ListBox` properties (see Listing 7-12).

Listing 7-12. Raising `ListBox` *Events*

```
Private Sub ListChangedEvents_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles ListChangedEvents.Click
    Dim custDataSet As DataSet = New DataSet()
    sql = "SELECT CustomerID, ContactName, City FROM Customers"
    custDataSet = New DataSet("Customers")
    conn = New SqlConnection()
    conn.ConnectionString = ConnectionString
    adapter = New SqlDataAdapter(sql, conn)
    adapter.Fill(custDataSet, "Customers")
    ListBox1.DataSource = custDataSet.Tables(0)
    ListBox1.DisplayMember = "ContactName"
    ListBox1.ValueMember = "ContactName"
    conn.Close()
    conn.Dispose()
End Sub
```

Data Binding in ComboBox and ListBox Controls

Now you'll learn how to use complex data binding in a `ComboBox` and a `ListBox` control. Unlike simple data-bound controls, complex data-bound controls maintain the default binding synchronization. For instance, if you bind a data source with a `ListBox` and a `ComboBox` control, and then move from one item to another in a control, you can see the selection change in the second control respective to the item you select in the first control.

To prove this theory, you'll create a Windows application with a `ComboBox` and three `ListBox` controls. The final form looks like Figure 7-4.

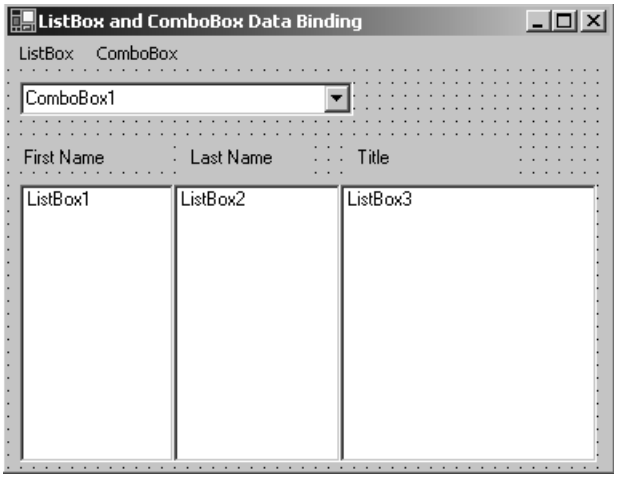

Figure 7-4. `ListBox` *and* `ComboBox` *data-binding form*

As usual, you load data in the `Form_Load` event. Listing 7-13 shows the event handler code, where you call the `FillDataSet` and `BindListControl` methods.

Listing 7-13. The Form Load Event Handler

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
    FillDataSet()
    BindListControls()
End Sub
```

The FillDataSet method simply opens the connection and fills data in a DataSet. Listing 7-14 shows this method.

Listing 7-14. The FillDataSet *Method*

```
' Fill DataSEt
Private Sub FillDataSet()
    ds = New DataSet()
    sql = "SELECT * FROM Employees"
    ds = New DataSet("Employees")
    conn = New SqlConnection()
    conn.ConnectionString = ConnectionString
    adapter = New SqlDataAdapter(sql, conn)
    adapter.Fill(ds, "Employees")
    conn.Close()
    conn.Dispose()
End Sub
```

The BindListControls method is where you bind the ComboBox and ListBox controls. Listing 7-15 shows the BindListControls method. As you can see, this code binds the ComboBox to the EmployeeID column and binds the three ListBox controls to the FirstName, LastName, and Title columns.

Listing 7-15. Binding ListBox *and* ComboBox *Controls*

```
' Bind data with controls
Private Sub BindListControls()
    ComboBox1.DataSource = ds.Tables(0)
    ComboBox1.DisplayMember = "EmployeeID"
    ListBox1.DataSource = ds.Tables(0)
    ListBox1.DisplayMember = "FirstName"
    ListBox2.DataSource = ds.Tables(0)
    ListBox2.DisplayMember = "LastName"
    ListBox3.DataSource = ds.Tables(0)
    ListBox3.DisplayMember = "Title"
End Sub
```

If you run the application and select any record in the ComboBox or ListBox controls, you'll see that the other controls select the correct value. For instance, if you select the sixth record in the ComboBox, all of the ListBox controls reflect this choice (see Figure 7-5).

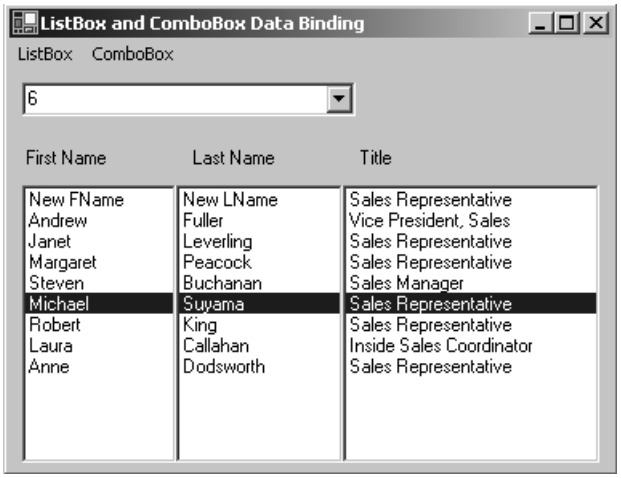

Figure 7-5. Data synchronization in `ComboBox` *and* `ListBox` *controls*

Data Binding in a DataGrid Control

The `DataGrid` control is much more powerful than any other data-bound control. It's also capable of displaying data relations. A `DataGrid` control displays data in a tabular, scrollable, and editable grid. Like other data-bound controls, a `DataGrid` control can display data from various sources with the help of its `DataSource` property. The `DataSource` property can be a `DataTable`, `DataView`, `DataSet`, or `DataViewManager`.

When a `DataSet` or a `DataViewManager` contains data from more than one table, you can specify what table you want to display in the `DataGrid` property by using the `DataMember` property. For example, let's say you have a `DataSet` that contains two tables—`Customers` and `Orders`. By default, if you bind a `DataSet`, it'll display data from both tables. But if you want to display data from the Customers table only, you need to set the `DataMember` property to the table's name. Listing 7-16 sets the `DataSource` and `DataMember` properties of a `DataGrid`.

Listing 7-16. Setting the `DataSource` *and* `DataMember` *Properties of a* `DataGrid` *Control*

```
ds = New DataSet()
sql = "SELECT * FROM Customers"
ds = New DataSet()
adapter = New SqlDataAdapter(sql, conn)
adapter.Fill(ds)
DataGrid1.DataSource = ds
DataGrid1.DataMember = "Customers"
```

You can also set the `DataSource` and `DataMember` properties by using the `DataGrid` control's `SetDataBinding` method. This method takes the first argument as a `dataSource` and the second argument as a `dataMember`. Typically, a data source is a `DataSet`, and the `dataMember` is the name of a table available in the `DataSet`. The following code shows how to call the `SetDataBinding` method of a `DataGrid` control:

```
DataGrid1.SetDataBinding(ds, "Customers")
```

You'll use the `DataGrid` control and its members throughout this chapter.

Deleting Data Binding

Removing data binding from a data-bound control is simple. The following code snippet deletes data binding from a `DataGrid` control:

```
DataGrid1.DataSource = null;
DataGrid1.DataMember = "";
```

The DataGrid: Super Data-Bound Control

The `DataGrid` control is one of the most flexible and versatile controls in Windows Forms. In this section, we discuss some of the `DataGrid` functionality.

The `DataGrid` class represents the `DataGrid` control in Windows Forms. Before writing any code, you'll learn about the `DataGrid` class properties and methods. Figure 7-6 shows a `DataGrid`'s parent items and background, and Figure 7-7 shows some of the `DataGrid` parts.

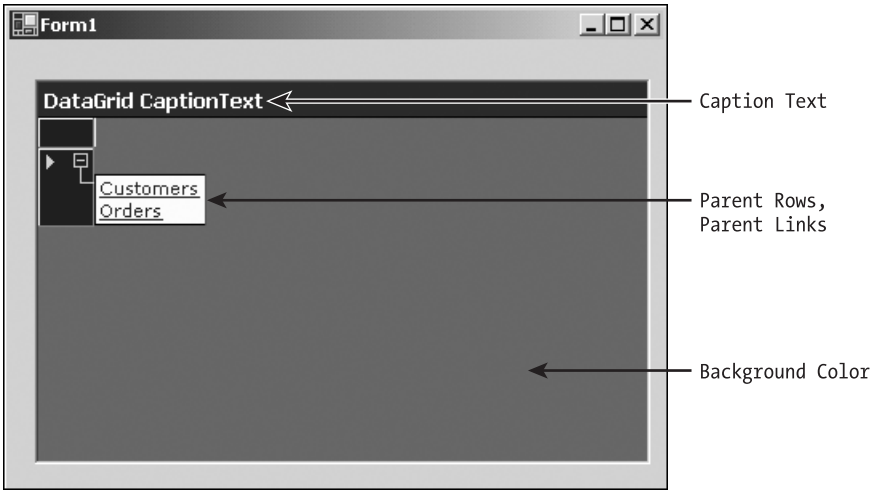

Figure 7-6. The DataGrid*'s parent items and background*

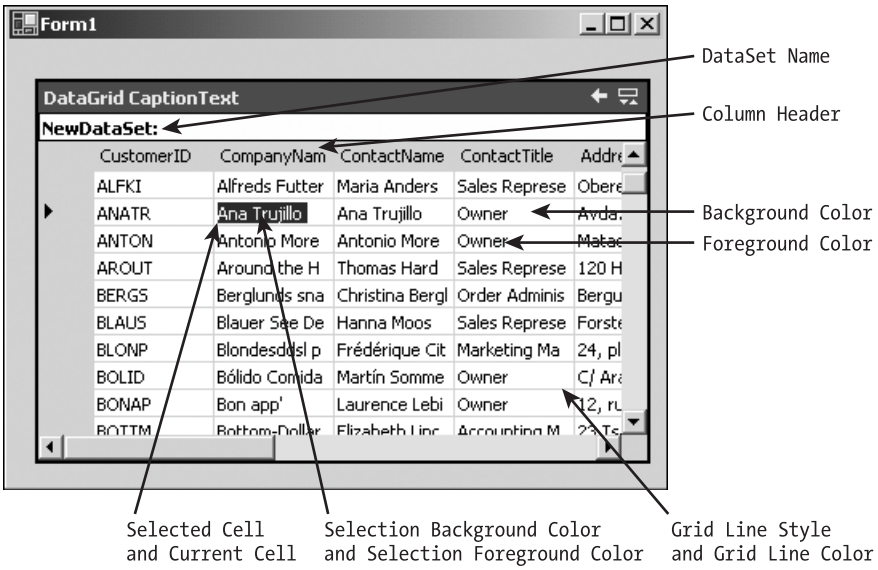

Figure 7-7. The DataGrid*'s parts*

Understanding the DataGrid Class Members

Like all other Windows controls, the `DataGrid` inherits from the `Control` class, which means that the data-binding functionality defined in the `Control` class is available in the `DataGrid` control. Besides the hundreds of members implemented in the `Control` class, the `DataGrid` provides many more members. Table 7-2 describes the `DataGrid` class properties.

Table 7-2. The `DataGrid` *Class Properties*

PROPERTY	DESCRIPTION
`AllowNavigation`	Indicates whether navigation is allowed. True or false. Both get and set.
`AllowSorting`	Indicates whether sorting is allowed. True or false. Both get and set.
`AlternatingBackColor`	Background color of alternative rows.
`BackColor`	Background color of the grid.
`BackgroundColor`	Color of the nonrow area of the grid. This is the background color if the grid has no rows.
`BorderStyle`	Style of the border.
`CaptionBackColor`	Background color of caption.
`CaptionFont`	Font of caption.
`CaptionForeColor`	Foreground color of caption.
`CaptionText`	Caption text.
`CaptionVisible`	Indicates whether caption is visible.
`ColumnHeadersVisible`	Indicates whether column headers are visible.
`CurrentCell`	Returns current selected cell.
`CurrentRowIndex`	Index of the selected row.
`DataMember`	Represents the data sources among multiple data sources. If there's only one data source, such as a `DataTable` or a `DataSet` with a single table, there's no need to set this property. Both get and set.
`DataSource`	Represents the data source such as a `DataSet`, a `DataTable`, or `Ilist`.
`FirstVisibleColumn`	Index of the first visible column.
`FlatMode`	`FlatMode`. Type of `FlatMode` enumeration.
`ForeColor`	Foreground color.
`GridLineColor`	Color of grid lines.

Table 7-2. The `DataGrid` *Class Properties (Continued)*

PROPERTY	DESCRIPTION
`GridLineStyle`	Style of grid lines.
`HeaderBackColor`	Background color of column headers.
`HeaderFont`	Font of column headers.
`HeaderForeColor`	Foreground color of column headers.
`Item`	Value of the specified cell.
`LinkColor`	Color of the text that you can click to navigate to a child table.
`LinkHoverColor`	Link color changes to when the mouse moves over it.
`ParentRowBackColor`	Background color of parent rows. Parent rows are rows that allow you to move to child tables.
`ParentRowForeColor`	Foreground color of parent rows.
`ParentRowLabelStyle`	Label style of parent rows.
`ParentRowsVisible`	Indicates whether parent rows are visible.
`PreferredColumnWidth`	Default width of columns in pixel.
`PreferredRowHeight`	Default height of rows in pixels.
`ReadOnly`	Indicates whether grid is read only.
`RowHeaderVisible`	Indicates whether row header is visible.
`RowHeaderWidth`	Width of row headers.
`SelectionBackColor`	Background color of selected rows.
`SelectionForeColor`	Foreground color of selected rows.
`TableStyles`	Table style. `DataGridTableStyle` type.
`VisibleColumnCount`	Total number of visible columns.
`VisibleRowCount`	Total number of visible rows.
`HorizScrollBar`	Protected. Returns the horizontal scroll bar of the grid.
`VertScrollBar`	Protected. Returns the horizontal scroll bar of the grid.
`ListManager`	Protected. Returns the `CurrencyManager` of the grid.

Table 7-3 describes the `DataGrid` class methods.

Table 7-3. The `DataGrid` *Class Methods*

METHOD	DESCRIPTION
`BeginEdit`	Starts the editing operation
`BeginInit`	Begins the initialization of grid that is used on a form or used by other components
`Collapse`	Collapses children if a grid has parent and child relationship nodes expanded
`EndEdit`	Ends the editing operation
`EndInit`	Ends grid initialization
`Expand`	Expands children if grid has children in a parent/child relation
`GetCurrentCellBounds`	Returns a rectangle that specifies the four corners of the selected cell
`HitTest`	Gets information when clicking on the grid
`IsExpanded`	`True` if node of the specified row is expanded; otherwise false
`IsSelected`	`True` if specified row is selected; otherwise false
`NavigateBack`	Navigates to the table previously displayed in the grid
`NavigateTo`	Navigates to the table specified by the row and relation name
`ResetAlternatingBackColor`	Resets the `AlternatingBackColor` property to the default color
`ResetBackColor`	Resets background color to default
`ResetGridLineColor`	Resets grid lines color to default
`ResetHeaderBackColor`	Resets header background to default
`ResetHeaderFont`	Resets header font to default
`ResetHeaderForeColor`	Resets header foreground color to default
`ResetLinkColor`	Resets link color to default
`ResetSelectionBackColor`	Resets selection background color to default
`ResetSelectionForeColor`	Resets selection foreground color to default
`Select`	Selects a specified row
`SetDataBinding`	Sets the `DataSource` and `DataMember` properties
`UnSelect`	Unselects a specified row

Besides the methods described in Table 7-3, the DataGrid class provides some protected methods (see Table 7-4).

Table 7-4. The DataGrid *Class Protected Methods*

PROTECTED METHOD	DESCRIPTION
CancelEditing	Cancels the current edit operation and rolls back all changes
GridHScrolled	Listens for the horizontal scroll bar's scroll event
GridVScrolled	Listens for the vertical scroll bar's scroll event
OnBackButtonClicked	Listens for the caption's Back button clicked event
OnBorderStyleChanged	Raises the BorderStyleChanged event
OnCaptionVisibleChanged	Raises the CaptionVisibleChanged event
OnDataSourceChanged	Raises the DataSourceChanged event
OnFlatModeChanged	Raises the FlatModeChanged event
OnNavigate	Raises the Navigate event
OnParentRowsLabelStyleChanged	Raises the ParentRowsLabelStyleChanged event
OnParentRowsVisibleChanged	Raises the ParentRowsVisibleChanged event
OnReadOnlyChanged	Raises the ReadOnlyChanged event
OnRowHeaderClick	Raises the RowHeaderClick event
OnScroll	Raises the Scroll event
OnShowParentDetailsButtonClicked	Raises the ShowParentDetailsButtonClick event
ProcessGridKey	Processes keys for grid navigation
ProcessTabKey	Gets a value indicating whether the Tab key should be processed
ResetSelection	Turns off selection for all rows that are selected

Exploring the DataGrid Helper Objects

The DataGrid class comes with 13 helper objects (classes, structures, and enumerations). What do we mean by *helper classes*? Helper classes provide simple methods to access some of the more complicated aspects of the DataGrid class. These helper objects are DataGrid.HitTestInfo, the DataGrid.HitTestType enumeration, DataGridBoolColumn, the DataGridCell structure, DataGridColumnStyle, DataGridColumnStyle.CompModSwitches, the DataGridColumnStype.DataGridColumnHeaderAccessibleObject DataGridLineStyle enumeration, the DataGridParentRowsLabelStyle enumeration, DataGridPreferredColumnWidthTypeConverter,

DataGridTableStyle, DataGridTextBox, and DataGridTextBoxColumn. We discuss some of these objects in the following section. You'll see the rest of them later in this chapter.

Understanding the DataGrid and DataGrid Column Styles

The DataGrid control hides much more functionality in it. Not only can it display data and data relations, it also provides functionality to customize its styles including color, text, caption, and font. The TableStyles property of DataGrid opens the door for formatting a grid and its columns. The GridStyles property returns an object of GridTableStyleCollection, which is a collection of DataGridTableStyle.

DataGridTableStyle represents the style of a DataTable that can be viewed in the grid area of a DataGrid. The GridTableStyles class of DataGridTableStyle represents a collection of DataGridColumnStyle. Figure 7-8 represents the relationship between the DataGrid-related style objects. We discuss these objects in more detail in the following sections.

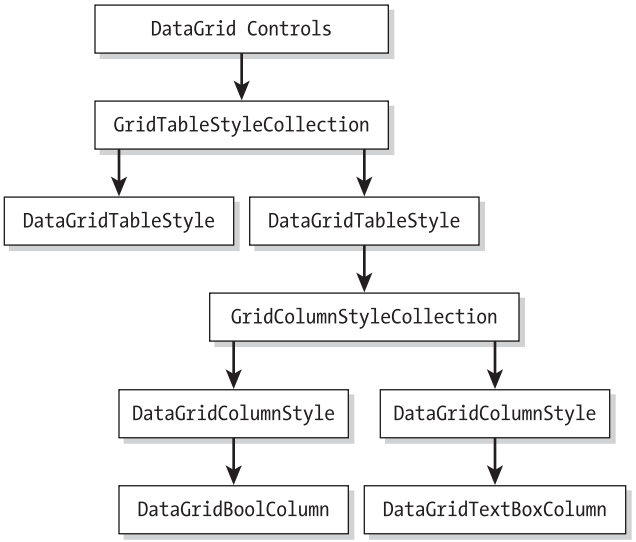

Figure 7-8. DataGrid*-related style objects*

Before you see these objects in action, you'll look at these object classes and their members briefly.

Using the DataGridTableStyle Class

The DataGridTableStyle object customizes the grid style for each DataTable in a DataSet. However, the DataGridTableStyle name is a little misleading. From its name, you would probably think the DataGridTableStyle represents the style of a DataGridTable such as its color, text, and font. Correct? Actually, DataGridTableStyle represents the grid itself. Using DataGridTableStyle, you can set the style and appearance of each DataTable, which is being displayed by the DataGrid. To specify which DataGridTableStyle is used when displaying data from a particular DataTable, set the MappingName to the TableName of a DataTable. For example, if a DataTable's TableName is Customers, you set MappingName to the following:

```
DataGridTableStyle.Mapping="Customers"
```

The DataGridTableStyle class provides similar properties and methods to those in the DataGrid class. Some of these properties are AllowSorting, AlternativeBackColor, BackColor, ColumnHeaderVisible, ForeColor, and GridColumnStyle.

Using the GridColumnStyles Property

The GridColumnStyles property returns a collection of columns available in a DataGridTableStyle as a GridColumnStylesCollection, which is a collection of DataGridColumnStyle objects. By default all columns are available through this property.

Using the GridTableStyleCollection Members

The GridTableStyleCollection is a collection of DataGridTableStyle. The TableStyle property of DataGrid represents and returns a collection of DataGridTableStyle objects as a GridTableStylesCollection object.

TIP DataGridTableStyle *is useful when it comes to managing a* DataGrid*'s style programmatically. One of the real-world usages of* DataGridTableStyle *is when you need to change the column styles of a* DataGrid *or want to move columns from one position to another programmatically.*

Unlike other collection objects, by default the `GridTableStylesCollection` doesn't contain any `DataGridTableStyle` objects. You need to add `DataGridTableStyle` objects to the collection. By default a `DataGrid` displays default settings such as color, text, font, width, and formatting. By default all columns of `DataTable` are displayed.

Constructing and Adding a DataGridStyle

You'll now learn how to create a `DataGridTableStyle` object and add it to the `DataGrid`'s `DataGridTableStyle` collection. Listing 7-17 creates a `DataGridTableStyle`, set its properties, and adds two columns: a Boolean and a text box column. The `DataGridBookColumn` class represents a Boolean column with check boxes, and the `DataGridTextBoxColumn` represents a text box column. (We discuss these classes in the following sections.)

Listing 7-17. Creating and Adding a `DataGridTableStyle`

```
Private Sub AddDataGridStyleMethod()
    ' Create a new DataGrudTableStyle
    Dim dgTableStyle As New DataGridTableStyle()
    dgTableStyle.MappingName = "Customers"
    dgTableStyle.BackColor = Color.Gray
    dgTableStyle.ForeColor = Color.Wheat
    dgTableStyle.AlternatingBackColor = Color.AliceBlue
    dgTableStyle.GridLineStyle = DataGridLineStyle.None
    ' Add some columns to the style
    Dim boolCol As New DataGridBoolColumn()
    boolCol.MappingName = "boolCol"
    boolCol.HeaderText = "boolCol Text"
    boolCol.Width = 100
    ' Add column to GridColumnStyle
    dgTableStyle.GridColumnStyles.Add(boolCol)
    ' Text column
    Dim TextCol As New DataGridTextBoxColumn()
    TextCol.MappingName = "Name"
    TextCol.HeaderText = "Name Text"
    TextCol.Width = 200
    ' Add column to GridColumnStyle
    dgTableStyle.GridColumnStyles.Add(TextCol)
    ' Add DataGridTableStyle to the collection
    DataGrid1.TableStyles.Add(dgTableStyle)
End Sub
```

You can even create a DataGridTableStyle from a CurrencyManager. Listing 7-18 creates a DataGridTableStyle from a CurrencyManager and adds it the collection.

Listing 7-18. Creating a DataGridTableStyle from CurrencyManager

```
Private Sub CreateNewDGTableStyle()
    Dim curManager As CurrencyManager
    Dim newTableStyle As DataGridTableStyle
    curManager = CType _
    (Me.BindingContext(ds, "Customers"), CurrencyManager)
    newTableStyle = New DataGridTableStyle(curManager)
    DataGrid1.TableStyles.Add(newTableStyle)
End Sub
```

Using the DataGridColumnStyle Class

The DataGridColumnStyle represents the style of a column. You can attach a DataGridColumnStyle to each column of a DataGrid. The DataGrid can contain different types of columns such as a check box or a text box. As you saw earlier, a DataGridTableStyle contains a collection of DataGridColumnStyle objects, which can be accessed through the GridColumnStyles property of DataGridTableStyle. This object is pretty useful and allows many formatting- and style-related members. Table 7-5 describes the DataGridColumnStyle properties.

Table 7-5. The DataGridColumnStyle *Properties*

PROPERTY	DESCRIPTION
Alignment	Alignment of text in a column. Both get and set.
DataGridTableStyle	Returns the DataGridTableStyle object associated with the column.
HeaderText	Text of the column header. Both get and set.
MappingName	Name used to map the column style to a data member. Both get and set.
NullText	You can set the column text when the column has null values using this property. Both get and set.
PropertyDescriptor	PropertyDescriptor object that determines the attributes of data displayed by the column. Both get and set.
ReadOnly	Indicates if column is read only. Both get and set.
Width	Width of the column. Both get and set.

Besides the methods described in Table 7-6, the DataGridColumnStyle class provides a method, ResetHeaderText, which resets the header text to its default value of null.

Table 7-6. The DataGridColumnStyle *Methods*

METHOD	DESCRIPTION
Abort	Aborts the edit operation.
BeginUpdate	Suspends the painting operation of the column until the EndUpdate method is called.
CheckValidDataSource	If a column is not mapped to a valid property of a data source, this throws an exception.
ColumnStartEditing	Informs DataGrid that the user has start editing the column.
Commit	Completes the editing operation.
ConcedeFocus	Notifies a column that it must relinquish the focus to the control it's hosting.
Edit	Prepares the cell for editing a value.
EndUpdate	Resumes the painting of columns suspended by calling the BeginUpdate method.
EnterNullValue	Enters a DBNullValue into the column.
GetColumnValueAtRow	Returns the value in the specified row.
GetMinimumHeight	Returns the minimum height of a row.
GetPreferedHeight	Returns the height used for automatically resizing columns.
GetPreferedSize	Automatic size.
Invalidate	Redraws the column.
SetColumnValueAtRow	Sets a value in the specified row.
SetDataGrid	Sets the DataGrid to which this column belongs.
SetDataGridInColumn	Sets the DataGrid for the column.
UpdateUI	Updates the value of a row.

Using the DataGridBoolColumn Class

A DataGrid can contain different types of columns such as a check box or a text box. By default all columns are in a simple grid format. The DataGridBoolColumn class represents a Boolean column of a DataGrid. Each cell of a Boolean column contains a check box, which can be checked (true) or unchecked (false). The DataGridBoolColumn class is inherited from the DataGridColumnStyle class. Besides the functionality provided by the DataGridColumnStyle, it provides its own members. Table 7-7 describes the DataGridBoolColumn class properties.

Table 7-7. The `DataGridBoolColumn` *Properties*

PROPERTY	DESCRIPTION
`AllowNull`	Represents whether null values are allowed in this column or not (both get and set)
`FalseValue`	Represents the actual value of column when the value of column is set to `False` (both get and set)
`NullValue`	The actual value used when setting the value of the column to `Value` (both get and set)
`TrueValue`	Represents the actual value of column when the value of column is set to `True` (both get and set)

Listing 7-19 creates a new `DataGridBoolColum` and sets its properties.

Listing 7-19. Creating a `DataGridBoolColumn`

```
Dim dgCol As DataGridBoolColumn
    dgCol = CType(dtGrid.TableStyles _
 ("Customers").GridColumnStyles("Current"), DataGridBoolColumn)
    dgCol.TrueValue = True
    dgCol.FalseValue = False
    dgCol.NullValue = Convert.DBNull
```

Setting DataGrid Sorting and Navigation

By default, navigation and sorting is on in a `DataGrid`. If a `DataGrid` is filled with data and you click the `DataGrid` header, it sorts data in ascending or descending order, depending on the current state. In other words, if the data is sorted in ascending order, right-clicking the header will sort it in descending order—and vice versa. You can activate or deactivate sorting programmatically using the `AllowSorting` property, which is a Boolean type. The following code shows how to set the `AllowSorting` property:

```
' Allow Sorting
If (allowSortingCheckBox.Checked) Then
  dtGrid.AllowSorting = True
Else
  dtGrid.AllowSorting = False
End If
```

Like the AllowSorting property, the AllowNavigation property enables or disables navigation. Setting the property to True indicates that navigation in a DataGrid is allowed and setting it to False means that navigation is not allowed. When you change the AllowNavigation property, the AllowNavigationChanged event is fired. Perhaps you notice in the previous samples that if a DataSet had more than one database table, there were links to each table? When you click a table link, the DataGrid opens that table. If AllowNavigation is False, then no links to child tables display. Listing 7-20 uses AllowNavigation and also handles the AllowNavigationChanged event.

Listing 7-20. AllowNavigation *in Action*

```
' Change navigation using AllowNavigation property
  Private Sub NavigationMenu_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles NavigationMenu.Click
    ' Change navigation. If its true, change it to false and
    ' vice versa
    If dtGrid.AllowNavigation = True Then
      dtGrid.AllowNavigation = False
    Else
      dtGrid.AllowNavigation = True
    End If
  End Sub

  Private Sub AllowNavigationEvent(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles dtGrid.AllowNavigationChanged
    Dim nav As Boolean = dtGrid.AllowNavigation
    Dim str As String = "AllowNavigationChanged event fired. "
    If (nav) Then
      str = str + "Navigation is allowed"
      NavigationMenu.Checked = True
    Else
      str = str + "Navigation is not allowed"
      NavigationMenu.Checked = False
    End If
    MessageBox.Show(str, "AllowNavigation")
  End Sub
```

Setting DataGrid Coloring and Font Styles

As mentioned, the DataGrid provides properties to set the foreground and background color of almost every part of a DataGrid such as headers, grid lines, and so on. The DataGrid also provides font properties to set the font of the DataGrid. Listing 7-21 sets Font and Color properties of a DataGrid.

Listing 7-21. Using Some of the DataGrid*'s* Color *and* Font *Properties*

```
' Setting DataGrid's Color and Font properties
dtGrid.BackColor = Color.Beige
dtGrid.ForeColor = Color.Black
dtGrid.BackgroundColor = Color.Red
dtGrid.SelectionBackColor = Color.Blue
dtGrid.SelectionForeColor = Color.Yellow
dtGrid.GridLineColor = Color.Blue
dtGrid.HeaderBackColor = Color.Black
dtGrid.HeaderForeColor = Color.Gold
'dtGrid.AlternatingBackColor = Color.AliceBlue
dtGrid.LinkColor = Color.Pink
dtGrid.HeaderFont = New Font("Verdana", FontStyle.Bold)
dtGrid.Font = New Font("Verdana", 8, FontStyle.Regular)
```

TIP *You can customize a* DataGrid *and allow the user to select a color and font for each part of the* DataGrid *at runtime as well as at design-time using the Properties window.*

Setting Caption Properties

You just saw the Font property of the DataGrid itself. The DataGrid also provides properties to set the caption's fonts and color. For example, Listing 7-22 sets the font, background color, and foreground color of caption of the DataGrid.

Listing 7-22. The `DataGrid`*'s Caption Properties*

```
dtGrid.CaptionText = "Customized DataGrid"
dtGrid.CaptionBackColor = System.Drawing.Color.Green
dtGrid.CaptionForeColor = System.Drawing.Color.Yellow
dtGrid.CaptionFont = New Font("Verdana", 10, FontStyle.Bold)
```

Seeing DataGridTableStyle and DataGridColumnStyle in Action

A common use of `DataGridColumnStyle` is changing the positions of a `DataGrid`'s columns programmatically. In the following sections, you'll see some common usages of `DataGridTableStyle` and `DataGridColumnStyle`.

As mentioned, you'll see some real-world uses of data-bound controls in this chapter. Specifically, you'll learn how to add check box and text box columns to a `DataGrid`. You also know that the `GridColumnStyles` property returns a collection of `DataGridTableStyle` as an object of `GridColumnStyleCollection`. Using `GridColumnStyleCollection` you can add and remove column styles to a collection. This is what you'll use to add new columns to a collection and attach them to a `DataGridTableStyle`.

To start this application, create a Windows application and define a `DataSet` variable as private:

```
Private ds As DataSet = Nothing
```

On the form's `Load` event, you call the `CreateDataSet`, `DataGrid.SetDataBinding`, and `FillDataGrid` methods (see Listing 7-23).

Listing 7-23. Form's `Load` *Method*

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
    ' Create in memory DataSet. You can even create
    ' a DataSet from a database
    CreateDataSet()
    ' Bind DataSet to DataGrid
    dtGrid.SetDataBinding(ds, "Employees")
    ' Fill data in DataGrid
    FillDataGrid()
  End Sub
```

The CreateDataSet method in Listing 7-24 simply creates a DataSet by creating a DataTable and adding three columns: EmployeeID (integer), Name (string), and StillWorking (Boolean). This method also adds four rows to the DataTable and adds the DataTable to a DataSet.

Listing 7-24. The CreateDataSet *Method*

```
' Create a DataSet with two tables and populate it.
  Private Sub CreateDataSet()
    ' Create a DataSet, add a DataTable
    ' Add DataTable to DataSet
    ds = New DataSet("ds")
    Dim EmployeeTable As DataTable = New DataTable("Employees")
    ' Create DataColumn objects and add to the DataTAable
    Dim dtType As System.Type
    dtType = System.Type.GetType("System.Int32")
    Dim EmpIDCol As DataColumn = _
    New DataColumn("EmployeeID", dtType)
    Dim EmpNameCol As DataColumn = New DataColumn("Name")
    dtType = System.Type.GetType("System.Boolean")
    Dim EmpStatusCol As DataColumn = New DataColumn("StillWorking", dtType)
    EmployeeTable.Columns.Add(EmpIDCol)
    EmployeeTable.Columns.Add(EmpNameCol)
    EmployeeTable.Columns.Add(EmpStatusCol)
    ' Add first records
    Dim row As DataRow = EmployeeTable.NewRow()
    row("EmployeeID") = 1001
    row("Name") = "Jay Leno"
    row("StillWorking") = False
    EmployeeTable.Rows.Add(row)
    ' Add second records
    row = EmployeeTable.NewRow()
    row("EmployeeID") = 1002
    row("Name") = "Peter Kurten"
    row("StillWorking") = True
    EmployeeTable.Rows.Add(row)
    ' Add third records
    row = EmployeeTable.NewRow()
    row("EmployeeID") = 1003
    row("Name") = "Mockes Pope"
    row("StillWorking") = False
    EmployeeTable.Rows.Add(row)
```

```
    ' Add fourth records
    row = EmployeeTable.NewRow()
    row("EmployeeID") = 1004
    row("Name") = "Rock Kalson"
    row("StillWorking") = True
    EmployeeTable.Rows.Add(row)
    ' Add the tables to the DataSet
    ds.Tables.Add(EmployeeTable)
  End Sub
```

In Listing 7-25, the `FillDataSet` method creates a `DataGridTableStyle` and sets its properties. After that, it creates two `DataGridTextBoxColumn`s and one `DataGridBoolColumn` and sets their properties. Also, it makes sure that the `MappingName` of the columns matches with the name of the columns of the `DataTable`. After creating each column, you add these methods to the column collection by using the `DataGrid.GridColumnStyles.Add` method. Finally, you add `DataGridTableStyle` to the `DataGrid` by using the `DataGrid.TableStyles.Add` method. After doing so, the `DataGrid` should have a new style with a check box and two text box columns.

Listing 7-25. The `FillDataGrid` *Method*

```
Private Sub FillDataGrid()
    ' Create a DataGridTableStyle and set its properties
    Dim dgTableStyle As DataGridTableStyle = New DataGridTableStyle()
    dgTableStyle.MappingName = "Employees"
    dgTableStyle.AlternatingBackColor = Color.Gray
    dgTableStyle.BackColor = Color.Black
    dgTableStyle.AllowSorting = True
    dgTableStyle.ForeColor = Color.White
    ' Create a DataGridColumnStyle. Add it to DataGridTableStyle
    Dim dgTextCol As DataGridColumnStyle = New DataGridTextBoxColumn()
    dgTextCol.MappingName = "Name"
    dgTextCol.HeaderText = "Employee Name"
    dgTextCol.Width = 100
    dgTableStyle.GridColumnStyles.Add(dgTextCol)
    ' Get PropertyDescriptorCollection by calling GetItemProperties
    Dim pdc As PropertyDescriptorCollection = Me.BindingContext _
    (ds, "Employees").GetItemProperties()
    'Create a DataGrodTextBoxColu
    Dim dgIntCol As DataGridTextBoxColumn = _
    New DataGridTextBoxColumn(pdc("EmployeeID"), "i", True)
    dgIntCol.MappingName = "EmployeeID"
```

```
    dgIntCol.HeaderText = "Employee ID"
    dgIntCol.Width = 100
    dgTableStyle.GridColumnStyles.Add(dgIntCol)
    ' Add CheckBox column using DataGridCoolColumn
    Dim dgBoolCol As DataGridColumnStyle = New DataGridBoolColumn()
    dgBoolCol.MappingName = "StillWorking"
    dgBoolCol.HeaderText = "Boolean Column"
    dgBoolCol.Width = 100
    dgTableStyle.GridColumnStyles.Add(dgBoolCol)
    ' Add table style to DataGrid
    dtGrid.TableStyles.Add(dgTableStyle)
  End Sub
```

Seeing HitTest in Action

You can use a *hit test* to get information about a point where a user clicks a control. There are many real-world usages of a hit test. For example, say you want to display two pop-up menus when a user right-clicks a certain area on a `DataGrid`. One area is on the `DataGrid` column header; this right-click pop-up menu will have options such as Sort Ascending, Sort Descending, Hide, and Find. As you can pretty guess from these names, the sort menu items will sort a column's data in ascending and descending order, the Hide menu item will hide (or delete) a column, and the Find menu item will search for a keyword in the selected column.

The second pop-up menu will pop up when you right-click any grid's cell. This menu will have options such as Move First, Move Previous, Move Next, and Move Last that will allow you to move to the first, previous, next, and last rows of a `DataGrid`.

Now, using only these two cases, you can find out what `DataGrid` part is processing the hit test action (in other words, which one is being clicked by the user). The `HitTest` method of `DataGrid` performs a hit test action.

Using the DataGrid.HitTestInfo Class

The `HitTest` method takes a point and returns the `DataGrid.HitTestInfo` object, which determines the part of a `DataGrid` clicked by the user. It's useful when you're designing a custom grid and want to do different things when user clicks different parts of the `DataGrid`.

The `DataGrid.HitTestInfo` class has three properties: `Column`, `Row`, and `Type`. The `Column` and `Row` properties represent the number of the column and row that the user has clicked. The `Type` property specifies the part of the `DataGrid` that is clicked.

The DataGrid.HitTestType enumeration is used as the Type property, which is defined in Table 7-8.

Table 7-8. The DaaGrid.HitTestType *Enumeration*

MEMBER	DESCRIPTION
Caption	Returns True if the caption was clicked.
Cell	Returns True if a cell was clicked.
ColumnHeader	Returns True if a column header was clicked.
ColumnResize	Represents the column border, the line between column headers.
None	Returns True if the background area was clicked.
ParentRow	The parent row displays information about the parent table of the currently displayed child table.
RowHeader	Returns True if the row header was clicked.
RowResize	Returns True if the line between rows.

You can also check the Type property against the combination of DataGrid.HitTestType enumeration members. Listing 7-26 is the mouse down event handler of a DataGrid, which tracks almost every portion of a DataGrid and generates a message when you right-click a DataGrid. Simply copy this code, right-click the DataGrid, and see it in action.

Listing 7-26. Seeing HitTest *in Action*

```
' DataGrid Mouse down event handler
  Private Sub dtGrid_MouseDown(ByVal sender As Object, _
  ByVal e As System.Windows.Forms.MouseEventArgs) Handles dtGrid.MouseDown
    Dim grid As DataGrid = CType(sender, DataGrid)
    Dim hti As DataGrid.HitTestInfo
    ' When right mouse button was clicked
    If (e.Button = MouseButtons.Right) Then
      hti = grid.HitTest(e.X, e.Y)
      Select Case hti.Type
        Case DataGrid.HitTestType.None
          MessageBox.Show("Background")
        Case DataGrid.HitTestType.Cell
          MessageBox.Show("Cell - Row:" & hti.Row & ", Col: " & hti.Column)
        Case DataGrid.HitTestType.ColumnHeader
          MessageBox.Show("Column header " & hti.Column)
        Case DataGrid.HitTestType.RowHeader
```

```
            MessageBox.Show("Row header " & hti.Row)
          Case DataGrid.HitTestType.ColumnResize
            MessageBox.Show("Column seperater " & hti.Column)
          Case DataGrid.HitTestType.RowResize
            MessageBox.Show("Row seperater " & hti.Row)
          Case DataGrid.HitTestType.Caption
            MessageBox.Show("Caption")
          Case DataGrid.HitTestType.ParentRows
            MessageBox.Show("Parent row")
        End Select
      End If
End Sub
```

Reshuffling DataGrid Columns

How about reshuffling or moving `DataGrid` columns? Reshuffling a `DataGrid`'s columns is a simple trick. You need to find which column you want to reshuffle. You can do this by using the column name or column index. In this sample, you'll use the column index.

How about reading information about a `DataGridTableStyle` and its columns? The following code reads information about a grid's tables and their names:

```
Dim gridStyle As DataGridTableStyle
For Each gridStyle In DataGrid1.TableStyles
  infoStr = "Table Name: " + gridStyle.MappingName
  Dim colStyle As DataGridColumnStyle
  For Each colStyle In gridStyle.GridColumnStyles
    infoStr = "Column: " + colStyle.MappingName
  Next
Next
```

Let's see this in action. Create a Windows application, add a `DataGrid` control, two `Button` controls, two `Label` controls, two `TextBox` controls, and a `ListBox` control. Next, set their properties and positions. The final form looks like Figure 7-9. As you can see, to exchange two columns, you enter column index in both text boxes and click the Exchange Columns button.

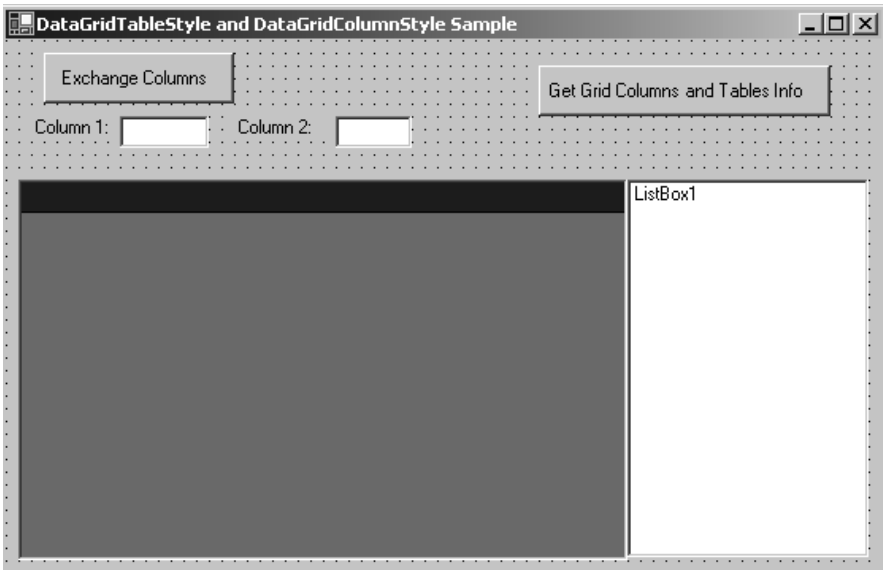

Figure 7-9. Column reshuffling form

Now let's write the code. As usual, you first define some variables:

```
' Developer defined variables
  Private conn As SqlConnection = Nothing
  Private ConnectionString As String = "Integrated Security=SSPI;" & _
     "Initial Catalog=Northwind;Data Source=MCB;"
  Private sql As String = Nothing
  Private ds As DataSet = Nothing
  Private adapter As SqlDataAdapter = Nothing
```

Next, add a new method called `FillDataGrid`, which fills the `DataGrid` from the Customers table of the Northwind database. You call the `FillDataGrid` method from the form's `Load` event handler (see Listing 7-27). You can also see from the `FillDataGrid` method, this code adds `DataGridTableStyle` to each `DataTable` in a `DataSet`.

CAUTION *What if you don't add DataGridTableStyle? By default, the* `DataGrid` *doesn't have any* `DataGridTableStyle` *and uses the default* `DataGridTableStyle`. *To make this program work, you must add it manually.*

Listing 7-27. The `FillDataGrid` *and* `Form_Load` *Methods*

```
  Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
    FillDataGrid()
  End Sub
Private Sub FillDataGrid()
    sql = "SELECT * FROM Customers"
    conn = New SqlConnection(connectionString)
    adapter = New SqlDataAdapter(sql, conn)
    ds = New DataSet("Customers")
    adapter.Fill(ds, "Customers")
    DataGrid1.DataSource = ds.Tables(0).DefaultView
    ' By default there is no DataGridTableStyle object.
    ' Add all DataSet table's style to the DataGrid
    Dim dTable As DataTable
    For Each dTable In ds.Tables
      Dim dgStyle As DataGridTableStyle = New DataGridTableStyle()
      dgStyle.MappingName = dTable.TableName
      DataGrid1.TableStyles.Add(dgStyle)
    Next
    ' DataGrid settings
    DataGrid1.CaptionText = "DataGrid Customization"
    DataGrid1.HeaderFont = New Font("Verdana", 12)
  End Sub
```

Now you write code on the Exchange Columns button click event handler (see Listing 7-28). As you can see, you need to make sure that the text boxes aren't empty. After that you call the `ReshuffleColumns` method, which actually moves the columns from one position to another.

Listing 7-28. Exchanging the Button Click Handler

```
Private Sub ExchangeColsBtn_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles ExchangeColsBtn.Click
  If (TextBox1.Text.Length < 1) Then
    MessageBox.Show("Enter a number between 0 to 19")
    TextBox1.Focus()
    Return
  ElseIf (TextBox2.Text.Length < 1) Then
    MessageBox.Show("Enter a number between 0 to 19")
    TextBox1.Focus()
    Return
  End If
  ' Get column 1 and column 2 indexes
   Dim col1 As Integer = Convert.ToInt32(TextBox1.Text)
   Dim col2 As Integer = Convert.ToInt32(TextBox2.Text)
   ' Exchange columns
   ReshuffleColumns(col1, col2, "Customers", DataGrid1)
End Sub
```

As mentioned earlier, moving column positions in a grid involves resetting a `DataGridTableStyle`. As you can see from Listing 7-29, you read the current `DataGridTableStyle` and create a new `DataGridTableStyle`. Next, you copy the entire current `DataGridTableStyle` including the two columns that you want to exchange and then change positions of these columns. Next, you remove the current `DataGridTableStyle` from the `DataGrid` and apply the new `DataGridTableStyle` by using the `DataGrid.TableStyles.Remove` and `DataGrid.TableStyles.Add` methods.

Listing 7-29. The `ReshuffleColumns` *Method*

```
Private Sub ReshuffleColumns(ByVal col1 As Integer, _
ByVal col2 As Integer, ByVal mapName As String, ByVal grid As DataGrid)
  Dim existingTableStyle As DataGridTableStyle = grid.TableStyles(mapName)
  Dim counter As Integer = existingTableStyle.GridColumnStyles.Count
  Dim NewTableStyle As DataGridTableStyle = New DataGridTableStyle()
  NewTableStyle.MappingName = mapName
  Dim i As Integer
  For i = 0 To counter - 1 Step +1
    If i <> col1 And col1 < col2 Then
      NewTableStyle.GridColumnStyles.Add _
      (existingTableStyle.GridColumnStyles(i))
    End If
```

```
    If i = col2 Then
      NewTableStyle.GridColumnStyles.Add _
      (existingTableStyle.GridColumnStyles(col1))
    End If
    If i <> col1 And col1 > col2 Then
      NewTableStyle.GridColumnStyles.Add _
      (existingTableStyle.GridColumnStyles(i))
    End If
  Next
  ' Remove the existing table style and add new style
  grid.TableStyles.Remove(existingTableStyle)
  grid.TableStyles.Add(NewTableStyle)
End Sub
```

Reading information about a `DataGrid`'s columns using a `DataGridColumnStyle` is simple. You just read the `GridColumnStyleCollection` using the `GridColumnStyles` property of `DataGridTableStyle`. Listing 7-30 reads a `DataGrid`'s column styles and adds them to the `ListBox` control.

Listing 7-30. Getting a `DataGrid` *Columns' Style Using* `DataGridColumnStyle`

```
Private Sub GetInfoBtn_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles GetInfoBtn.Click
  Dim infoStr As String = "Visible Rows: " + _
  DataGrid1.VisibleRowCount.ToString()
  ListBox1.Items.Add(infoStr)
  infoStr = "Visible Cols: " + _
  DataGrid1.VisibleColumnCount.ToString()
  ListBox1.Items.Add(infoStr)
  infoStr = "Total Rows: " + _
  ds.Tables(0).Rows.Count.ToString()
  ListBox1.Items.Add(infoStr)
  infoStr = "Total Cols: " + _
  ds.Tables(0).Columns.Count.ToString()
  ListBox1.Items.Add(infoStr)
  ' Get all table styles in the Grid and Column Styles
  ' which returns table and column names
  Dim gridStyle As DataGridTableStyle
```

```
  For Each gridStyle In DataGrid1.TableStyles
    infoStr = "Table Name: " + gridStyle.MappingName
    ListBox1.Items.Add(infoStr)
    Dim colStyle As DataGridColumnStyle
    For Each colStyle In gridStyle.GridColumnStyles
      infoStr = "Column: " + colStyle.MappingName
      ListBox1.Items.Add(infoStr)
    Next
  Next
End Sub
```

Now run the application and enter **1** in the Column 1 box and enter **2** in the Column 2 box and then click the Exchange Columns buttons. You'll see both columns switched their positions. Now if you click the Get Grid Columns and Tables Info button, the output looks like Figure 7-10.

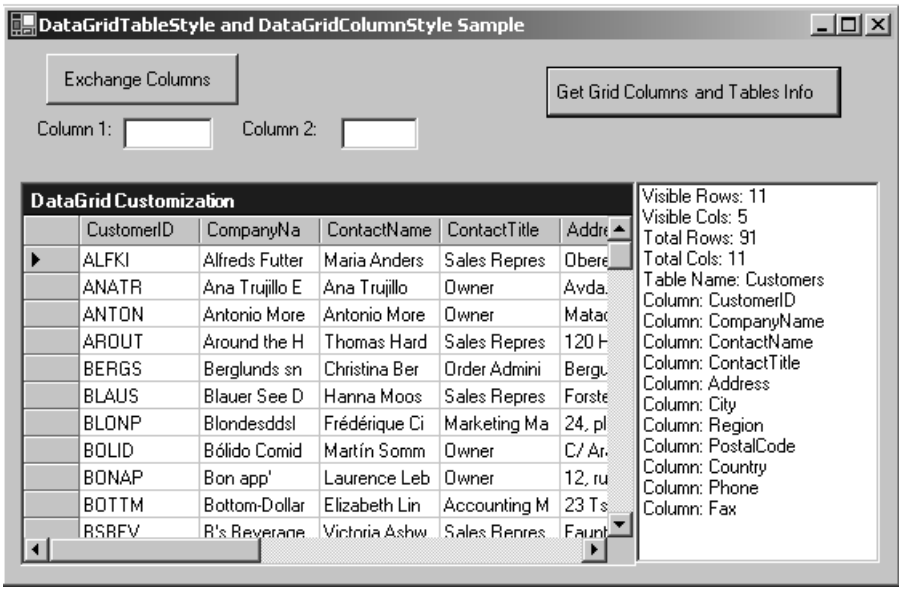

Figure 7-10. Getting a `DataGrid` *control's column styles*

Getting a Column Header Name

Listing 7-31 returns the column name when a user right-clicks a `DataGrid` column header.

Listing 7-31. Getting a `DataGrid` *Column Header Name*

```
  Private Sub DataGrid1_MouseDown(ByVal sender As Object, _
  ByVal e As System.Windows.Forms.MouseEventArgs) Handles DataGrid1.MouseDown
    Dim str As String = Nothing
    Dim pt As Point = New Point(e.X, e.Y)
    Dim hti As DataGrid.HitTestInfo = DataGrid1.HitTest(pt)
    ' If right mouse button clicked
    If e.Button = MouseButtons.Right Then
      If hti.Type = DataGrid.HitTestType.ColumnHeader Then
        Dim gridStyle As DataGridTableStyle = _
        DataGrid1.TableStyles("Customers")
        str = gridStyle.GridColumnStyles(hti.Column).MappingName.ToString()
        MessageBox.Show("Column Header " + str)
      End If
    End If
    ' If left mouse button clicked
    If e.Button = MouseButtons.Left Then
      If hti.Type = DataGrid.HitTestType.Cell Then
        str = "Column: " + hti.Column.ToString()
        str += ", Row: " + hti.Row.ToString()
        MessageBox.Show(str)
      End If
    End If
End Sub
```

Hiding a DataGrid's Columns

Now you'll learn a few more uses of a `DataGrid` control. Hiding a `DataGrid` column is simply a job of finding the right column and setting its `Width` property to `0`. For an example, see the `TotalDataGrid` sample that comes with the downloads from `www.apress.com`.

To make your program look better, you'll create a right-click pop-up menu, as shown in Figure 7-11.

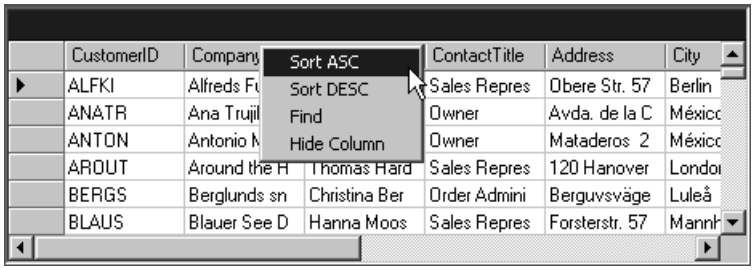

Figure 7-11. Pop-up menu on `DataGrid` *right-click menu*

To create a pop-up menu, you declare a `ContextMenu` and four `MenuItem` objects as `sortAscMenu`, `sortDescMenu`, `findMenu`, and `hideMenu`. You also define two more variables to store the current `DataGridColumnStyle` and column name as follows:

```
Private curColName As String = Nothing
Private curColumnStyle As DataGridColumnStyle
```

If `DataGrid.HitTestType` is `ColumnHeader`, you add menu items and get the current column, as shown in Listing 7-32. In this listing, you simply store the current `DataGridColumnStyle` and the name of the column.

Listing 7-32. Getting the Current `DataGridColumnStyle`

```
Case DataGrid.HitTestType.ColumnHeader
          ' Add context menus
          popUpMenu = New ContextMenu()
          popUpMenu.MenuItems.Add("Sort ASC")
          popUpMenu.MenuItems.Add("Sort DESC")
          popUpMenu.MenuItems.Add("Find")
          popUpMenu.MenuItems.Add("Hide Column")
          Me.ContextMenu = popUpMenu
          Me.BackColor = Color.Sienna
          sortAscMenu = Me.ContextMenu.MenuItems(0)
          sortDescMenu = Me.ContextMenu.MenuItems(1)
          findMenu = Me.ContextMenu.MenuItems(2)
          hideMenu = Me.ContextMenu.MenuItems(3)
          ' Find the Column header name
          Dim gridStyle As DataGridTableStyle = _
          dtGrid.TableStyles("Customers")
          curColName = gridStyle.GridColumnStyles _
          (hti.Column).MappingName.ToString()
         curColumnStyle = gridStyle.GridColumnStyles(hti.Column)
```

Finally, you write the Find menu button click event handler and set `curColumnStyle.Width` to `0`:

```
Private Sub hideMenuHandler(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles hideMenu.Click
    curColumnStyle.Width = 0
  End Sub
```

Implementing Custom Sorting in a DataGrid

By default a `DataGrid` provides you with sorting options when you click a `DataGrid` column. But there may be occasions when you don't want to use the default behavior and instead want to implement your own custom sorting.

In Figure 7-11, you saw the Sort ASC and Sort DESC menu options. As you probably remember from Chapter 3 and Chapter 4, sorting is easy to implement in a `DataView`. To sort a `DataView`, you simply set the `Sort` property of the `DataView` to the column name and to ASC or DESC for ascending and descending sorting, respectively. Listing 7-33 shows the Sort ASC and Sort DESC menu event handler code.

Listing 7-33. Sorting a `DataGrid` *Control's Columns*

```
Private Sub SortAscMenuHandler(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles sortAscMenu.Click
    Dim dv As DataView = ds.Tables("Customers").DefaultView
    dv.Sort = curColName + " ASC"
    dtGrid.DataSource = dv
  End Sub
  Private Sub SortDescMenuHandler(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles sortDescMenu.Click
    Dim dv As DataView = ds.Tables("Customers").DefaultView
    dv.Sort = curColName + " DESC"
    dtGrid.DataSource = dv
End Sub
```

Building a DataGrid Record Navigation System

Move methods are one of the features of the ADO recordset that don't appear in ADO.NET. A recordset provides `MoveFirst`, `MoveNext`, `MovePrevious`, and `MoveLast` methods to move to the first, next, previous and last record in a recordset (respectively). In this example, you'll implement move functionality in a `DataGrid` control.

Listing 7-34 implements move functionality in a custom recordset class called `CustRecordSet.vb`. (We already discussed how you can use `BindingContext` to move the current pointer from one position to another.) In this code, `CreateRecordSet` simply fills and binds a `DataSet` to the grid. The `FirstRecord`, `PrevRecord`, `NextRecord`, and `LastRecord` methods set the current position of the pointer to the first row, current row –1, current row +1, and the last row (respectively).

NOTE *In this example, the table name is Customers. You may want to customize the name so it can work for any database table.*

Listing 7-34. `CustRecordSet.vb`

```
Imports System.Data.SqlClient

Public Class CustRecordSet
  Private dataAdapter As SqlDataAdapter = Nothing
  Private dataSet As DataSet = Nothing
  Private dtGrid As DataGrid = Nothing
  Private frm As Form = Nothing
  Private mapName As String = Nothing

  Public Sub CreateRecordSet(ByVal conn As SqlConnection, _
  ByVal sql As String, ByVal grid As DataGrid, ByVal curForm As Form, _
  ByVal tableName As String)
    Me.dataAdapter = New SqlDataAdapter(sql, conn)
    Me.dataSet = New DataSet("Customers")
    Me.dataAdapter.Fill(Me.dataSet, "Customers")
    dtGrid = grid
    frm = curForm
    mapName = tableName
    dtGrid.DataSource = Me.dataSet
    dtGrid.DataMember = "Customers"
  End Sub
```

```
  Public Sub FirstRecord()
    If frm.BindingContext(Me.dataSet, mapName) Is Nothing Then
      Return
    End If
    frm.BindingContext(Me.dataSet, mapName).Position = 0
  End Sub
  Public Sub PrevRecord()
    If frm.BindingContext(Me.dataSet, mapName) Is Nothing Then
      Return
    End If
    frm.BindingContext(Me.dataSet, mapName).Position -= 1
  End Sub
  Public Sub NextRecord()
    If frm.BindingContext(Me.dataSet, mapName) Is Nothing Then
      Return
    End If
    frm.BindingContext(Me.dataSet, mapName).Position += 1

  End Sub
  Public Sub LastRecord()
    If frm.BindingContext(Me.dataSet, mapName) Is Nothing Then
      Return
    End If
    frm.BindingContext(Me.dataSet, mapName).Position = _
    frm.BindingContext(Me.dataSet, mapName).Count - 1
  End Sub

End Class
```

Now create a Windows application and add a `DataGrid` control and four Button controls (Move First, Move Next, Move Previous, and Move Last). The form's `Load` event calls `FillDataSet`, which creates a new `CustRecordSet` object and calls its `CreateRecordSet` method, which in turn fills data in a `DataGrid` control and binds a `DataSet` with the `DataGrid` control (see Listing 7-35).

Listing 7-35. Creating a Custom Recordset

```
' form load
  Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
    FillDataGrid()
  End Sub
  ' Fill DataGrid
  Private Sub FillDataGrid()
    sql = "SELECT * FROM Customers"
    conn = New SqlConnection(connectionString)
    recordSet = New CustRecordSet()
    recordSet.CreateRecordSet(conn, sql, DataGrid1, Me, "Customers")
  End Sub
```

Now on the button click event handlers, simply call `CustRecordSet`'s `FirstRecord`, `PrevRecord`, `NextRecord`, and `LastRecord` methods, as shown in Listing 7-36.

Listing 7-36. Moving Record Button Click Event Handlers

```
' Move First button click
  Private Sub MoveFirstBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MoveFirstBtn.Click
    recordSet.FirstRecord()
  End Sub
  ' Move Previous button click
  Private Sub MovePrevBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MovePrevBtn.Click
    recordSet.PrevRecord()
  End Sub
  ' Move next button click
  Private Sub MoveNextBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MoveNextBtn.Click
    recordSet.NextRecord()
  End Sub
  ' Move last button click
  Private Sub MoveLastBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MoveLastBtn.Click
    recordSet.LastRecord()
End Sub
```

The final application looks like Figure 7-12.

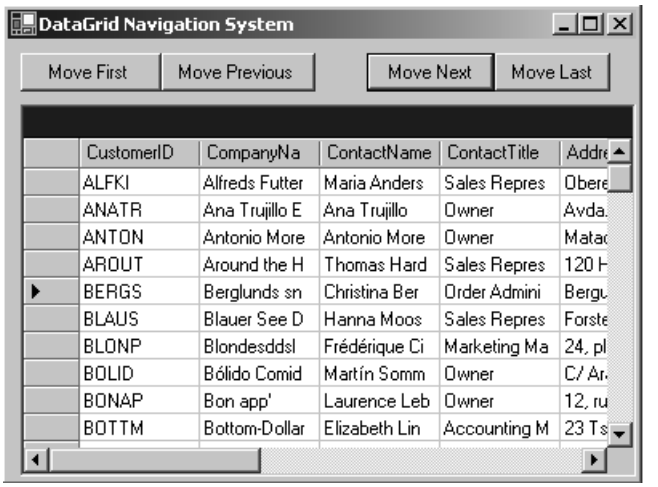

Figure 7-12. `DataGrid` *navigation system*

TIP *You can implement the same functionality on a* `DataGrid` *control's right-click menu by adding four menu items that allow users to move to the first, next, previous, and last records of a* `DataGrid`*. You can even develop your own* `DataGrid` *component with sorting, searching, and navigating features.*

Implementing Search in a DataGrid

You just saw how to implement custom sorting in a `DataGrid` control. After sorting, searching is one more basic requirement of database-driven applications. There are two methods to implement search in a `DataGrid` control:

- Using the `SELECT` statement
- Using a `DataSet` and `DataView`

Searching Using the SELECT Statement

You already used search functionality in a connected environment using the SQL `SELECT` statement. Do you remember using a `SELECT` statement with a `WHERE` clause? In a `WHERE` clause, you passed the criteria specifying the data for which you're looking. If you want to search in multiple tables, you construct a `JOIN` query with `WHERE` clause. You can even search for a keyword using the `SELECT...LIKE` statement, which was discussed in "The DataView in Connected Environments" section of Chapter 4.

You use the `SELECT` statement in a `DataAdapter`, which reads data based on the `SELECT` statement and the criteria passed in it. However, using this method for searching may not be useful when you're searching data frequently—especially when you're searching data in a `DataGrid`. We suggest not using this method when searching data in an isolated application and there's no other application updating the data. Why? The main reason is that every time you change the `SELECT` statement, you need to create a new `DataAdapter` and fill the `DataSet` when you change the `SELECT` statement. This method is useful when there are multiple applications updating the data simultaneously and you want to search in the latest updated data.

Searching Using a DataTable and DataView

The `DataTable` and `DataView` objects provide members that can filter data based on criteria. (See "The DataView" section in Chapter 3 for more information.) You can simply create a `DataView` from a `DataSet`, set a `DataView`'s `RowFilter` property to the search criteria, and then bind the `DataView` to a `DataGrid`, which will display the filtered records.

TIP *Using the same method, you can implement a Search or Find feature in a* `DataGrid` *control. You can also provide a Search option on a* `DataGrid` *control's header so that you know on which column a user has clicked.*

The new application looks like Figure 7-13. Obviously, the Search button searches the column entered in the Column Name text box for a value entered in the Value text box.

NOTE *If you search for a string, use a singe quote (') before and after the string.*

We discuss the `Save` method functionality in the following section.

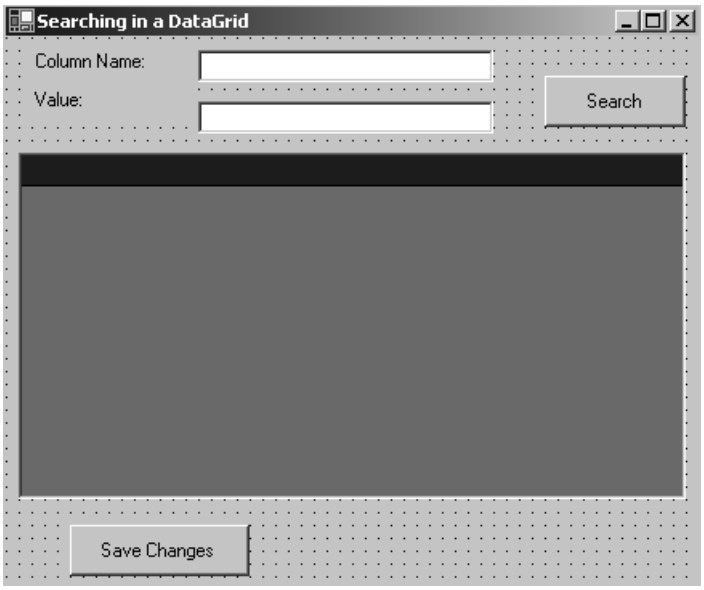

Figure 7-13. Implementing search functionality in a `DataGrid` *control*

After creating a Windows application and adding controls to the form, define following variables:

```
' Developer defined variables
  Private conn As SqlConnection = Nothing
  Private connectionString As String = _
   "Integrated Security=SSPI;Initial Catalog=Northwind;Data Source=MCB;"
  Private sql As String = Nothing
  Private searchView As DataView = Nothing
  Dim adapter As SqlDataAdapter = Nothing
  Dim ds As DataSet = Nothing
```

Now add the FillDataGrid method, which fills the DataGrid and creates a DataView called searchView (see Listing 7-37).

Listing 7-37. FillDataGrid *Method*

```
' Fill DataGrid
  Private Sub FillDataGrid()
    sql = "SELECT * FROM Orders"
    conn = New SqlConnection(connectionString)
    adapter = New SqlDataAdapter(sql, conn)
    ds = New DataSet("Orders")
    adapter.Fill(ds, "Orders")
    DataGrid1.DataSource = ds.Tables("Orders")
    searchView = New DataView(ds.Tables("Orders"))
    Dim cmdBuilder As SqlCommandBuilder = _
    New SqlCommandBuilder(adapter)
    ' Disconnect. Otherwise you would get
    ' Access violations when try multiple operations
    conn.Close()
    conn.Dispose()
 End Sub
```

Now, the next step is to set a RowFilter of searchView based on the values entered in the Column Name and Value text fields. Listing 7-38 shows the code for the Search button. As you can see, the code sets the RowFilter of searchView and binds it to the DataGrid to display the filtered data.

Listing 7-38. Seach Button Click Event Handler

```
Private Sub SearchBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles SearchBtn.Click
    If (TextBox1.Text.Equals(String.Empty)) Then
      MessageBox.Show("Enter a column name")
      TextBox1.Focus()
      Return
    End If
    If (TextBox2.Text.Equals(String.Empty)) Then
      MessageBox.Show("Enter a value")
      TextBox1.Focus()
      Return
    End If
    ' Construct a row filter and apply on the DataView
```

```
    Dim str As String = TextBox1.Text + "=" + TextBox2.Text
    searchView.RowFilter = str
    ' Set DataView as DataSource of DataGrid
    DataGrid1.DataSource = searchView
End Sub
```

At this time, if you run the application, the data from the Orders table is filled in the `DataGrid`. If you enter **EmployeeID** in the Column Name text box and **6** in the Value field and then click the Search button, the filtered data looks like Figure 7-14.

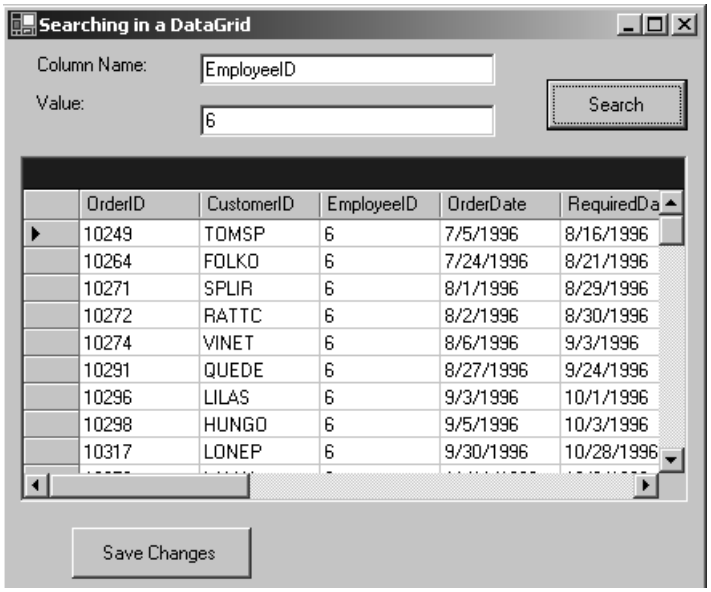

Figure 7-14. Filtered data after searching

Inserting, Updating, and Deleting Data through DataGrids

As you learned earlier, the `DataGrid` control is one of the most powerful, flexible, and versatile controls available in Windows Forms. It has an almost unlimited number of properties and methods. You can add new records, update records, and delete existing records on a `DataGrid` with little effort, and you can easily save the affected data in a database.

When a DataGrid control is in edit mode (the default mode), you can simply add a new record by clicking the last row of the grid and editing the column values. You can update data by changing the existing value of cells. You can delete a row by simply selecting a row and clicking the Delete button.

In the previous example, you used a Save Changes button on a form (see Figure 7-14). Now just write the code in Listing 7-39 on the Save Changes button click to save the data.

Listing 7-39. Saving Updated Data in a Data Source from a DataGrid *Control*

```
Private Sub SaveBtn_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles SaveBtn.Click

    Dim changeDS As DataSet = New DataSet()
    ' Data is modified
    If (ds.HasChanges(DataRowState.Modified)) Then
      changeDS = ds.GetChanges(DataRowState.Modified)
      Dim changedRecords As Integer
      changedRecords = adapter.Update(changeDS, "Orders")
      If (changedRecords > 0) Then
        MessageBox.Show(changedRecords.ToString() & _
        " records modified.")
      End If
    End If
    ' Data is deleted
    If (ds.HasChanges(DataRowState.Deleted)) Then
      changeDS = ds.GetChanges(DataRowState.Deleted)
      Dim changedRecords As Integer
      changedRecords = adapter.Update(changeDS, "Orders")
      If (changedRecords > 0) Then
        MessageBox.Show(changedRecords.ToString() & _
        " records deleted.")
      End If
    End If
    ' Data is added
    If (ds.HasChanges(DataRowState.Added)) Then
      changeDS = ds.GetChanges(DataRowState.Added)
      Dim changedRecords As Integer
      changedRecords = adapter.Update(changeDS, "Orders")
      If (changedRecords > 0) Then
        MessageBox.Show(changedRecords.ToString() & _
        " records added.")
```

```
      End If
    End If
    ds.AcceptChanges()
    DataGrid1.Refresh()
End Sub
```

As you can see from Listing 7-39, you simply get the modified, deleted, and updated changes in a new `DataSet` by calling the `DataSet.GetChanges` method and save the changes by calling the `DataAdapter.Update` method. In the end, you accept the changes by calling `DataSet.AcceptChanges` and refresh the `DataGrid` control by calling the `DataGrid.Refresh` method.

Summary

Data-bound controls are definitely one of the greatest additions to GUI applications. In this chapter, we discussed the basics of data binding and how data binding works in Windows Forms data-bound controls and ADO.NET. We discussed some essential objects that participate in the data-binding phenomena, including `Binding`, `BindingContext`, `BindingsCollection`, `BindingManagerBase`, `PropertyManager`, `CurrencyManager`, and `BindingContext`.

After discussing basics of data binding and how to use these objects, you learned about some data-bound controls and how to bind data using the data-binding mechanism. You also saw some practical usage of data binding and data-bound controls; specifically, you created a record navigation system with a `DataGrid` control. Some of the examples discussed in this chapter included changing `DataGrid` styles programmatically, binding data sources to various data-bound controls, building a record navigation application, and implementing search, add, update, and delete record features in a `DataGrid`.

The next chapter covers constraints and data relations in more detail.