

Chapter

6

Windows Forms Data Binding

Overview

One of the most powerful aspects of the Windows Forms architecture is data binding. Historically, data binding was used to bind visual views to data stored in databases and other data sources. Some database management systems (DBMSs), such as Microsoft Access, have provided GUI APIs to help developers quickly bind data to visual controls. Each DBMS usually had its own associated API for data binding purposes. Some even had no associated API, which forced developers to provide the implementation from scratch. Binding to other types of data structures, such as arrays, was out of the question. .NET, however, solves all these problems and more. With .NET, a client can bind to almost any data structure, including arrays, collections, database tables, rows, and views.

Data Binding Concepts

For .NET data binding to be possible, there must be providers of data and consumers of data. A provider of data is an object or component that exposes its data to the outside world. A consumer is an object or component that uses the data exposed by a provider with the intent to display or modify that data. With .NET data binding, the minimum requirement to support list-based data binding is for the provider to implement the *IList* interface. The *IList* interface

represents an index-based collection. Data providers and data consumers are discussed in more detail in the following sections.

Data Providers

The following objects implement the *IList* interface, so they are inherently data providers.

Arrays

An array is simply a collection of objects that can be accessed by a numeric index. Arrays can be either single-dimensional or multi-dimensional.

DataSet

The *DataSet* class is a .NET representation of a database or of some other data source. It does not, however, need to actually be connected to a real database. As a matter of fact, it acts as a “disconnected” data source with the ability to track changes and merge new data. When binding to a *DataSet* object, the data consumer is responsible for asking for the particular *DataTable* or *DataRowView* object or objects with which data binding should occur. In some cases, the data consumer would really be binding to the *DataSet* object’s default *DataRowViewManager* instance.

DataTable

A *DataTable* object typically represents a table in a database. However, it may also be used to represent the structure of an XML element or the parent of a collection. A *DataTable* object contains collections of *DataColumn* objects and *DataRow* objects. Complex controls, such as the *DataGrid* control, can be bound to a *DataTable* object with ease. Note that when you bind to a *DataTable* object, you really bind to the table’s default *DataRowView* object.

DataRowView

A *DataView* object is simply a customized view of the data in a *DataTable* object. For example, it might contain all rows sorted by a particular column or all rows that match a certain filter expression. When data binding to a *DataView* object, all controls involved in the data binding process will receive a snapshot of the data at that particular moment during which data binding occurs. Whenever the underlying data changes, the bound controls must have some way of knowing how to refresh themselves. This process will be discussed shortly.

DataViewManager

The *DataViewManager* class represents the default settings of an entire *DataSet* object. Similar to the *DataView* class, it is a snapshot view of the *DataSet* object. The main difference is that the *DataViewManager* class also includes relations.

DataColumn

A *DataColumn* object typically represents, and is analogous to, a column in a database table. However, it may also represent an XML attribute or an attribute-less XML element. You can only simple-bind to a *DataColumn* object. This means that only simple controls, such as a *TextBox* control, can be bound to a *DataColumn* object.

Other .NET Objects

In actuality, any .NET object can support data binding, but you might not automatically reap all of the benefits provided by the .NET architecture for using just any object. Also, when binding to these objects, only the public properties (not public fields) can be bound. Therefore, you must be careful when data binding to data sources exposed by Web services. The public properties of any types returned by a Web service will be converted to public fields in the Web service's client proxy code created by Visual Studio .NET 2003.

Be careful. You can only bind to the public properties, not the public fields, of data source objects.

Data Consumers

A data consumer is an object or component that uses the data exposed by a data provider with the intent to display or modify that data. In Windows Forms, a data consumer is typically a data-bound control. Simple data-bound controls include, but are not limited to, `TextBox`, `Label`, `CheckBox`, and `RadioButton` controls. These controls can display only one data value provided by a data source. On the other hand, the `DataGrid`, `ListBox`, and `ComboBox` controls can display a list of values. These controls are therefore referred to as complex data-bound controls.

Binding and BindingContext

Windows Forms controls support either simple binding or complex binding. Controls that support simple binding include the `TextBox` control. A `TextBox` control can support only one data value at a time. The following example shows how to bind a `TextBox` control with the `FirstName` column of a `Customers DataTable` object:

```
Dim nameTextBox As TextBox = New TextBox()  
Dim dataSet As DataSet = CreateMyDataSet()  
  
nameTextBox.DataBindings.Add("Text", dataSet, "Customers.FirstName")
```

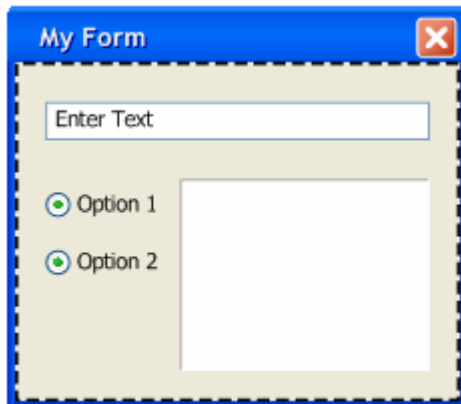
Binding

Every Windows Form control has a `DataBindings` property, which is an instance of `ControlBindingsCollection`. The `ControlBindingsCollection` class represents a collection of `Binding` objects that bind the properties of controls to data source members. Whenever the data source member changes, the control's property is automatically updated to reflect the change, and vice-versa. Different properties of the same control can be bound to different data sources.

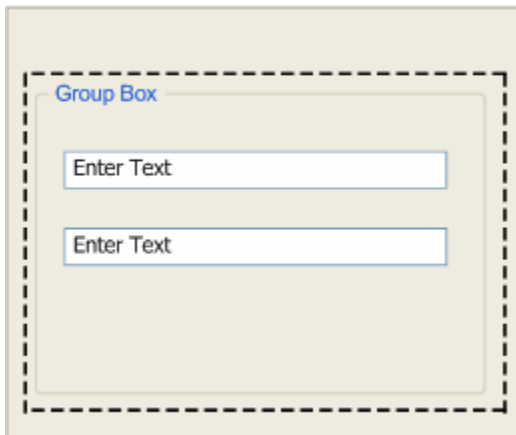
BindingContext

Every container control on a form, including the form itself, contains at least one *BindingContext* object. Actually, all controls derived from *System.Windows.Forms.Control* have the *BindingContext* property, but only container controls really make use of it. Non-container controls simply return the *BindingContext* object of their immediate container. A *BindingContext* object is just an object that provides binding support to multiple data sources. Since more than one data source can be viewed on a form, the *BindingContext* object enables retrieval of any particular data source. Specifically, a *BindingContext* object manages a collection of *BindingManagerBase* objects. *BindingManagerBase* is an abstract class that enables synchronization of data-bound controls that are bound to the same data source. A *BindingContext* object can be visualized as follows (The dashed lines represent the *BindingContext* object):

Form



GroupBox



In the previous figures, the *BindingContext* object basically says, “I will manage and keep track of all controls and their associated data sources and data-bound members. If the current record in the one of the managed data sources changes, I will refresh all controls that I track with the new values.”

By default, only one *BindingContext* object is created for a form, regardless of the number of controls contained on the form.

Here is the syntax for retrieving a data source from the *BindingContext* object:

```
Dim customers As BindingManagerBase = Me.BindingContext(dataSet, "Customers")
```

Here is the syntax for creating a new *BindingContext* object:

```
groupBox1.BindingContext = New BindingContext  
groupBox2.BindingContext = New BindingContext
```

In the previous example, two *BindingContext* objects are created and are assigned to two GroupBox controls. This allows the contained controls in both GroupBox controls to be bound to the same data source, but using two different binding managers.

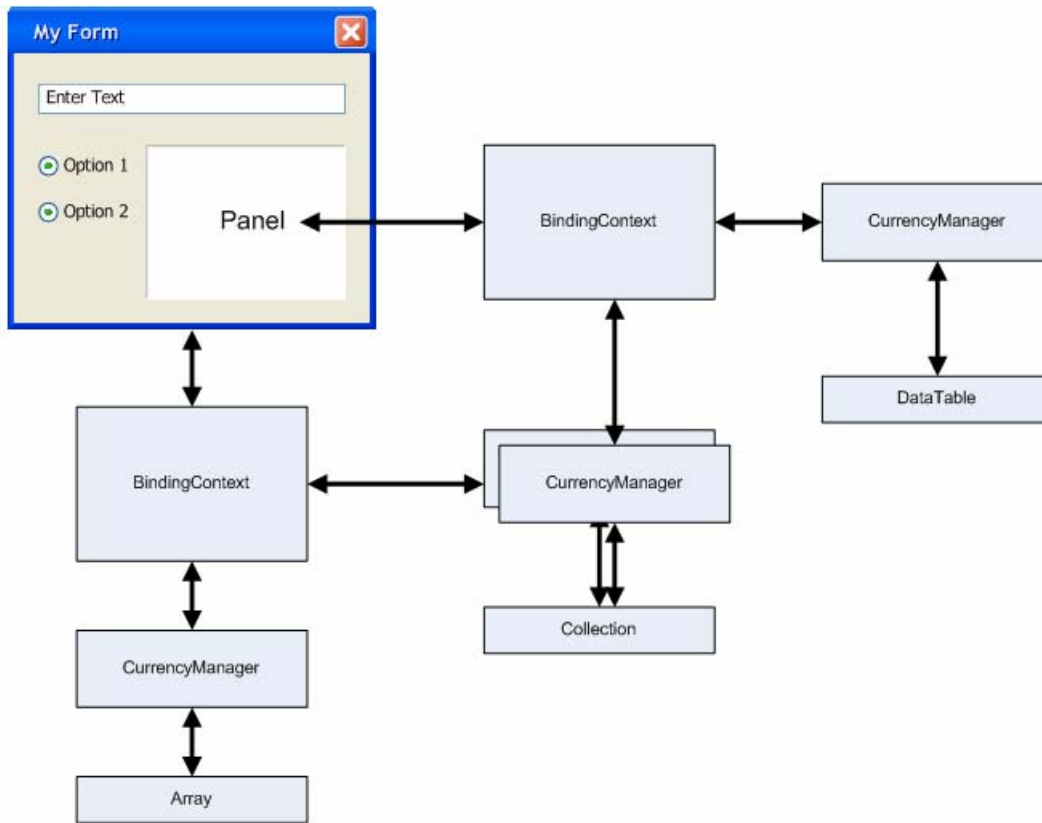
The two .NET Framework classes derived from *BindingManagerBase* are described next.

CurrencyManager

Any data source that is bound to a Windows Forms control is associated with a *CurrencyManager* object. Actually, the true name for *CurrencyManager* should be “concurrency manager” or “current manager.” During the days of ActiveX Data Objects (ADO), the data source object itself kept track of the current record. The problem with this approach was that multiple data consumers could not reuse the same data source object concurrently in an efficient manner. For example, if there were two Grid controls on a form that used ADO to display their data, and if both Grid controls used the current record for highlighting purposes, there would have been no way for each Grid control to highlight a different item at the same time. With .NET, the current record is no longer maintained in the data source object itself, which makes the data source object truly disconnected from the actual data store. The current record is maintained by a *CurrencyManager* object. A *CurrencyManager* object has a one-to-one relationship with a data source.

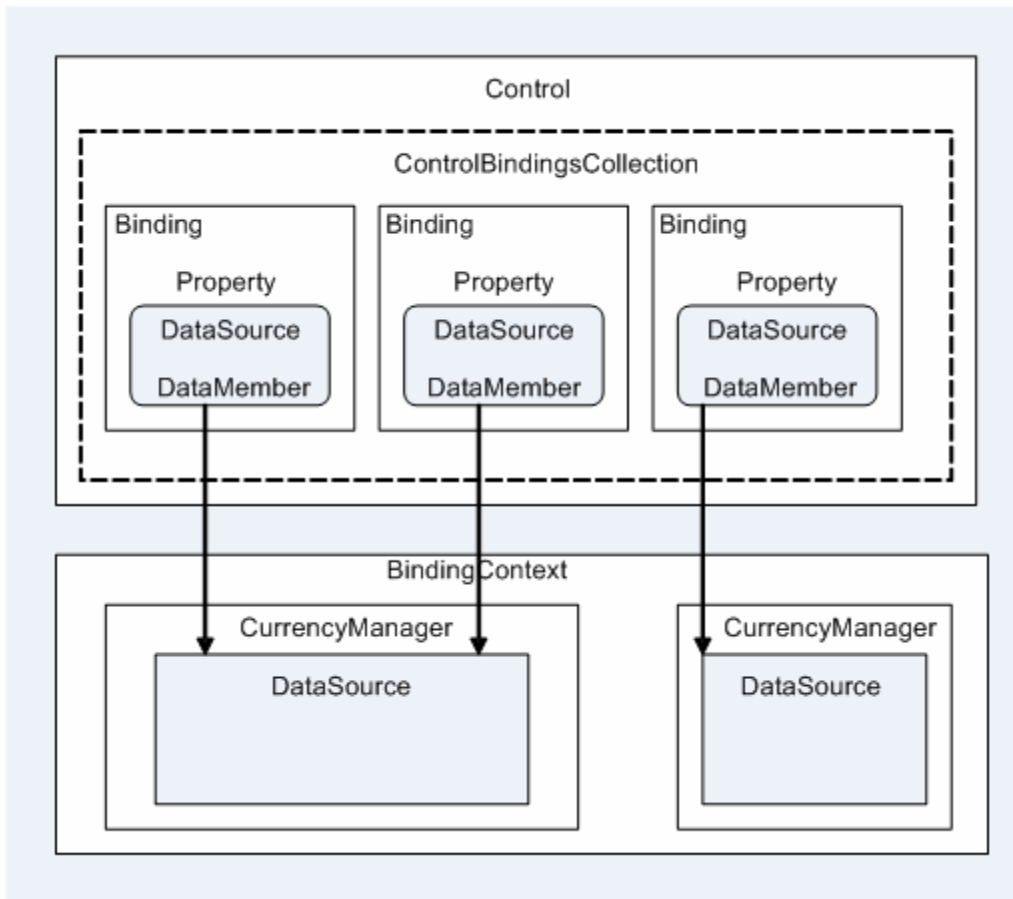
A *CurrencyManager* object is automatically created when a *Binding* object is created if it is the first time that the data source has been bound. (Remember that there is only one *CurrencyManager* object per data source per *BindingContext* object.)

The following diagram shows the relationship between a Form control, a Panel control, *CurrencyManager* objects, and data sources:



In the previous diagram, the Form control contains the automatically created *BindingContext* object, which contains two *CurrencyManager* objects, one managing an array and the other managing a collection. The *Panel* control contains a newly created *BindingContext* object (remember that only the Form control's *BindingContext* object is created by default), which also contains two *CurrencyManager* objects, one managing the same collection that is bound to the Form control and the other managing a *DataTable* object. Under normal circumstances, only one *CurrencyManager* object would be created for the collection. But since there are two *BindingContext* objects, each must contain its own collection of *CurrencyManager* objects.

The following diagram shows control binding in action:



In the previous diagram, a particular control has three properties that are participating in data binding, as you can note from the three *Binding* objects. These bindings are stored in the control's *ControlBindings* property, which is an instance of *ControlBindingsCollection*. Remember, the *ControlBindingsCollection* class is a collection of *Binding* objects, and a *Binding* object associates the property of a control with a data source member. Whenever the data source member value changes, the control's property is updated, and vice-versa. Two of the bindings are associated with the same data source, while the third one is associated with a different data source. The *CurrencyManager* object ensures that the properties that are associated with the same data source are synchronized.

PropertyManager

The *PropertyManager* class is used to identify and maintain the current property of an object. *PropertyManager* is derived from *BindingManagerBase*, but oddly, most of all of the base properties and methods are overridden to do nothing. For example, setting the *Position* property of the class has no effect. Also, the *AddNew* and *RemoveAt* methods throw an exception of type *NotSupportedException*. Your guess is as good as mine as to why this class was derived from *BindingManagerBase*. As of this writing, the *PropertyManager* class is only used by the PropertyGrid control. The PropertyGrid control uses the current property to raise events, display property descriptions, and invoke the appropriate editors.

The following code shows how to return a *PropertyManager* object from a *BindingContext* object:

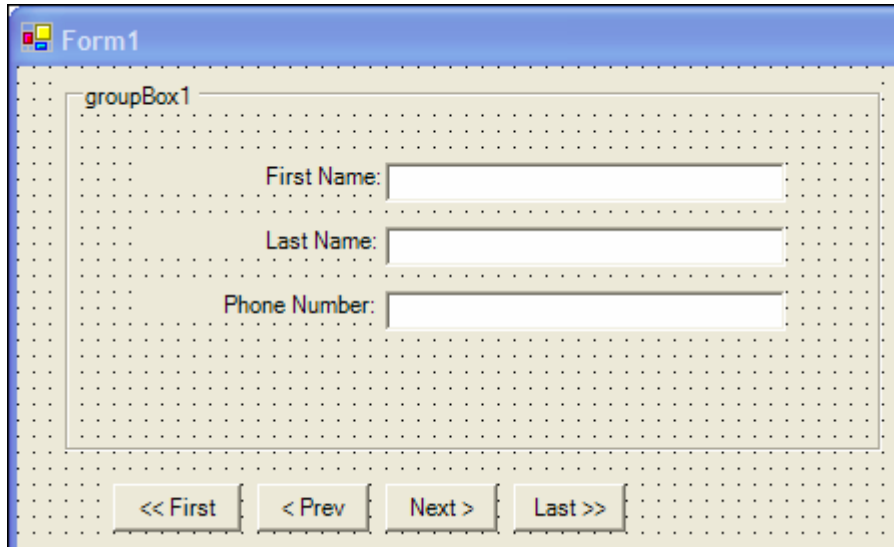
```
Dim singleCustomer As Customer = New Customer
Dim pm As PropertyManager = Me.BindingContext(singleCustomer)
```

Simple Binding Example

With simple binding, the property on a control is bound to a single data source member. The data source will typically be a collection, an array, or a *DataTable* object. If the data source is a collection or an array, binding will occur with a property of an item in the collection or array. If the data source is a *DataTable* object, binding will occur with a *DataColumn* object of the *DataTable* object.

In this example, I will walk through the implementation of binding customer data to controls on a form. First, create a new Windows Application project in Visual Studio .NET 2003. Drag a GroupBox control to the form. Then drag three Label controls and three TextBox controls to the form inside the GroupBox control. Finally, drag four Button controls to the form outside

the GroupBox control. Arrange and label the controls like the ones shown in the following figure:



Name your controls as follows:

Group Box: `_groupBox1`

First Name Label : `_firstNameLabel`

Last Name Label: `_lastNameLabel`

Phone Number Label: `_phoneNumberLabel`

First Name Textbox: `_firstNameTextBox`

Last Name Textbox: `_lastNameTextBox`

Phone Number Textbox: `_phoneNumberTextBox;`

First Button: `_firstButton`

Previous Button: `_previousButton`

Next Button: `_nextButton`

Last Button: `_lastButton`

For each of the four buttons, add an event handler named `Button_Navigate` for the `Click` event. To dynamically associate the event handler with the events, use the following syntax:

```
AddHandler _firstButton.Click, New EventHandler(Me.Button_Navigate)
```

To do this declaratively, ensure that the Button controls are declared using the `WithEvents` keyword, and add a `Handles` clause that specifies each Button control's `Click` event to the declaration of the `Button_Navigate` event handler, as shown here:

```
Private WithEvents _firstButton As Button

Private Sub Button_Navigate(ByVal sender As Object, ByVal e As EventArgs)
Handles _firstButton.Click
End Sub
```

Now, you need to define a `Customer` class. Create a new Visual Basic .NET class file and add the following code to it:

```
Public Class Customer
    Private _firstName As String
    Private _lastName As String
    Private _phoneNumber As String

    Public Sub New(ByVal firstName As String, ByVal lastName As String,
ByVal phoneNumber As String)
        _firstName = firstName
        _lastName = lastName
        _phoneNumber = phoneNumber
    End Sub
```

```

Public ReadOnly Property FirstName() As String
    Get
        Return _firstName
    End Get
End Property

Public ReadOnly Property LastName() As String
    Get
        Return _lastName
    End Get
End Property

Public ReadOnly Property PhoneNumber() As String
    Get
        Return _phoneNumber
    End Get
End Property
End Class

```

For simplicity, this example only stores a customer's name and phone number. In a production application, more properties would likely need to be implemented.

Inside the constructor of the *Form1* class after the call to *InitializeComponent*, initialize an array of customers with some arbitrary values. Declare the array as *ReadOnly* since it will not be reassigned after initialization. Here is the code:

```

Private ReadOnly _customers As Customer()

#Region " Windows Form Designer generated code "

Public Sub New()
    MyBase.New()

    'This call is required by the Windows Form Designer.
    InitializeComponent()

    'Add any initialization after the InitializeComponent() call
    _customers = New Customer() _
    {
        _
        New Customer("James", "Henry", "123-123-1234"), _
        New Customer("Bill", "Gates", "234-234-2345"), _
        New Customer("Tupac", "Shakur", "345-345-3456") _
    }
    ...

```

End Sub

Now bind all TextBox controls to the array, as shown here:

```

_customers = New Customer() _
{
    New Customer("James", "Henry", "123-123-1234"), _
    New Customer("Bill", "Gates", "234-234-2345"), _
    New Customer("Tupac", "Shakur", "345-345-3456") _
}

_firstNameTextBox.DataBindings.Add("Text", _customers, "FirstName")
_lastNameTextBox.DataBindings.Add("Text", _customers, "LastName")
_phoneNumberTextBox.DataBindings.Add("Text", _customers,
"PhoneNumber")

```

Finally, you need to handle the *Click* event raised by the Button controls to provide navigation functionality, as shown here:

```

Private Sub Button_Navigate(ByVal sender As Object, ByVal e As
EventArgs) Handles _firstButton.Click, _lastButton.Click,
_nextButton.Click, _previousButton.Click

    Dim manager As BindingManagerBase =
_groupBox1.BindingContext(_customers)

    If sender Is _firstButton Then
        manager.Position = 0
    ElseIf sender Is _previousButton Then
        manager.Position = manager.Position - 1
    ElseIf sender Is _nextButton Then
        manager.Position = manager.Position + 1
    ElseIf (sender Is _lastButton) Then
        manager.Position = manager.Count - 1
    End If
End Sub

```

The *Button_Navigate* event handler handles the *Click* event for all four navigation buttons. This code first retrieves the *BindingManagerBase* object from the *BindingContext* object of

_groupBox1. The *BindingManagerBase* object is actually an instance of *CurrencyManager*. The code basically asks the *BindingContext* object to “give me the *CurrencyManager* for the *_customers* data source.” The *CurrencyManger* object changes its *Position* property when a button is clicked. As the *Position* property is changed, all bound controls are updated automatically.

Now, to understand the purpose of the *BindingContext* object, add another *GroupBox* control to the form and three more *TextBox* controls to this *GroupBox* control. Name these controls as follows:

Group Box: *_groupBox2*

First Name Label: *_firstNameLabel2*

Last Name Label: *_lastNameLabel2*

Phone Number Label: *_phoneNumberLabel2*

First Name Textbox: *_firstNameTextBox2*

Last Name Textbox: *_lastNameTextBox2*

Phone Number Textbox: *_phoneNumberTextBox2*

First Button: *_firstButton2*

Previous Button: *_previousButton2*

Next Button: *_nextButton2*

Last Button: *_lastButton2*

Attach the *Click* event of each Button control to the same *Button_Navigate* event handler as before. Now data bind the second set of TextBox controls to the array, as shown here:

```

        _customers = New Customer() _
        { _
            New Customer("James", "Henry", "123-123-1234"), _
            New Customer("Bill", "Gates", "234-234-2345"), _
            New Customer("Tupac", "Shakur", "345-345-3456") _
        }

        _firstNameTextBox.DataBindings.Add("Text", _customers, "FirstName")
        _lastNameTextBox.DataBindings.Add("Text", _customers, "LastName")
        _phoneNumberTextBox.DataBindings.Add("Text", _customers,
"PhoneNumber")
        _firstNameTextBox2.DataBindings.Add("Text", _customers,
"FirstName")
        _lastNameTextBox2.DataBindings.Add("Text", _customers, "LastName")
        _phoneNumberTextBox2.DataBindings.Add("Text", _customers,
"PhoneNumber")

```

Before you can actually see the advantage of the *BindingContext* object, you need to ensure that each set of TextBox controls “lives” in its own data binding context. To do this, you need to create a *BindingContext* object for each GroupBox control. Remember that by default, the form automatically creates a single *BindingContext* object for itself and all child controls. Here is the constructor after creating two new *BindingContext* objects:

```

        _customers = New Customer() _
        { _
            New Customer("James", "Henry", "123-123-1234"), _
            New Customer("Bill", "Gates", "234-234-2345"), _
            New Customer("Tupac", "Shakur", "345-345-3456") _
        }

        _groupBox1.BindingContext = new BindingContext
        _groupBox2.BindingContext = new BindingContext

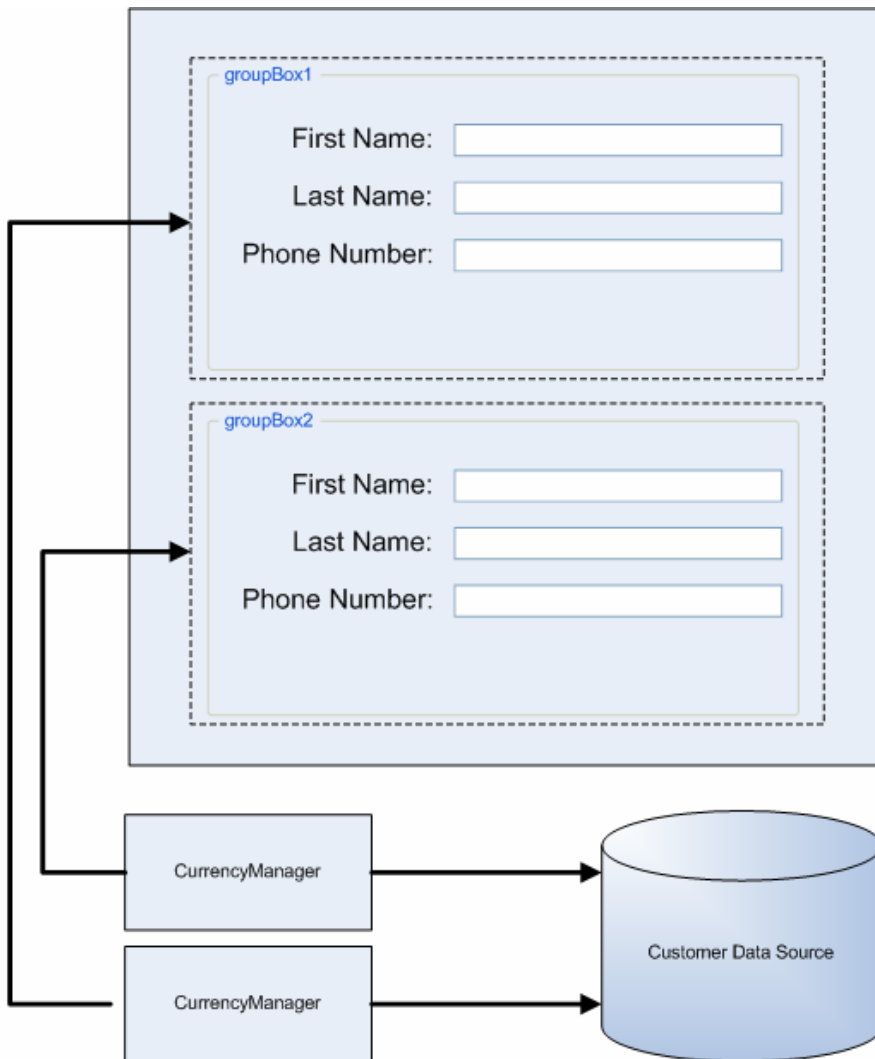
        _firstNameTextBox.DataBindings.Add("Text", _customers, "FirstName")
        _lastNameTextBox.DataBindings.Add("Text", _customers, "LastName")
        _phoneNumberTextBox.DataBindings.Add("Text", _customers,
"PhoneNumber")

```

```
        _firstNameTextBox2.DataBindings.Add("Text", _customers,  
"FirstName")  
        _lastNameTextBox2.DataBindings.Add("Text", _customers, "LastName")  
        _phoneNumberTextBox2.DataBindings.Add("Text", _customers,  
"PhoneNumber")
```

Now, each GroupBox control and any child controls have their own context for data binding. Even though the controls contained in both GroupBox controls may bind to the same data source, they will be bound using different *CurrencyManager* objects.

You can visualize how the TextBox controls are data bound and synchronized in the following diagram:



From the previous diagram, you can see that each GroupBox control has its own *CurrencyManager* object for the Customers data source. Therefore, changing the *Position* property on the first *CurrencyManager* object will have no effect on the TextBox controls contained in the second GroupBox control. Similarly, changing the *Position* property on the second *CurrencyManager* object will have no effect on the TextBox controls contained in the

first *GroupBox* control. Therefore, add the following code to the *Button_Navigate* event handler to manipulate the second *CurrencyManager* object:

```

Private Sub Button_Navigate(ByVal sender As Object, ByVal e As
EventArgs) Handles _firstButton.Click, _lastButton.Click,
_nextButton.Click, _previousButton.Click

    Dim manager As BindingManagerBase =
_groupBox1.BindingContext(_customers)
    Dim manager2 As BindingManagerBase =
_groupBox2.BindingContext(_customers)

    If sender Is _firstButton Then
        manager.Position = 0
    ElseIf sender Is _previousButton Then
        manager.Position = manager.Position - 1
    ElseIf sender Is _nextButton Then
        manager.Position = manager.Position + 1
    ElseIf (sender Is _lastButton) Then
        manager.Position = manager.Count - 1
    ElseIf sender Is _firstButton2 Then
        manager2.Position = 0
    ElseIf sender Is _previousButton2 Then
        manager2.Position = manager2.Position - 1
    ElseIf sender Is _nextButton2 Then
        manager2.Position = manager2.Position + 1
    ElseIf sender Is _lastButton2 Then
        manager2.Position = manager2.Count - 1
    End If
End Sub

```

Data Binding Interfaces

.NET provides a standard set of interfaces related to data binding. Each of these interfaces is described next:

ICollection

Any class that implements the *ICollection* interface must support a list of homogenous types. That is, all list items must be of the same type. The first item in the list always determines the type.

Some of the base classes that implement *IList* include *Array*, *ArrayList*, *CollectionBase*, *DataGridView*, and *DataGridViewManager*.

Typed IList

Similar to *IList*, the list must be of homogenous types. However, the type of the items in the list must be known at compile time.

IList and IComponent

When a class implements both *IList* and *IComponent*, the class can be data bound at design time.

IListSource

This interface allows an object to “act” like a list for data binding purposes. The implemented object is not an instance of *IList*, but it should be able to provide one. The *DataSet* and *DataTable* classes both implement this interface. *IListSource* provides a single property and a single method, which are described next:

ContainsListCollection: Indicates whether the collection is a collection of *IList* objects. For the *DataSet* implementation, this property returns *True*, because the *DataSet* class contains a collection of collections. For the *DataTable* implementation, this property returns *False*, because the *DataTable* class contains a collection of objects. In simple terms, implement this property to indicate how deep to go for returning a bindable list.

GetList: Returns the *IList* object that will be data-bound. The *DataSet* class uses this property to return a *DataViewManager* object. The *DataTable* class uses this property to return a *DataView* object.

ITypedList

This interface allows a collection object to expose its items' properties. This interface is useful in situations where the public properties of the collection object should be different from the properties available for data binding. This interface is also necessary during complex binding when a list is empty but you still need to know the property names of the list items. (Remember, the *IList* class alone uses the data type of the first item in the list.) This is useful when columns headers should be created for empty lists.

IBindingList

This interface offers change notification when the list or list items have changed. There is one property, *SupportsChangeNotification*, which indicates whether this interface's *ListChanged* event should be raised. The *ListChangedEventArgs* class contains a property named *ListChangedType* that describes the type of change that occurred. The available *ListChangedType* enumeration members are as follows:

ItemAdded: An item has been added to the list. The index of the new item is the value of the *NewIndex* property of the *ListChangedEventArgs* class.

ItemChanged: An item in the list has been changed. The index of the changed item is the value of the *NewIndex* property of the *ListChangedEventArgs* class.

ItemDeleted: An item has been removed from the list. The index of the deleted item is the value of the *NewIndex* property of the *ListChangedEventArgs* class.

ItemMoved: An item has been moved to another location within the list. The previous index is the value of the *OldIndex* property of the *ListChangedEventArgs* class. The new index is the value of the *NewIndex* property of the *ListChangedEventArgs* class.

PropertyDescriptorAdded: A *PropertyDescriptor* object has been added.

PropertyDescriptorChanged: A *PropertyDescriptor* object has been changed.

PropertyDescriptorDeleted: A *PropertyDescriptor* object has been deleted.

Reset: The list has a lot of changes and controls should refresh themselves.

IEditableObject

This interface supports transaction-like operations. It allows objects to specify when changes should be made permanent. Hence, it allows changes to be rolled back. The *DataGrid* control is one control that opts to call methods of this interface. The following methods are defined in this interface:

BeginEdit: Signals that an edit operation has started. Any changes to the object should be temporarily stored after this method has been called. When implementing this method, be sure that back-to-back calls are non-destructive. That is, the method itself should not cause any changes to any temporary objects.

CancelEdit: Cancels any changes made after the *BeginEdit* call. In other words, all temporary objects can be destroyed when this method is called.

EndEdit: Commits any changes made after the *BeginEdit* call. Once this method is called, changes cannot and should not be rolled back.

The following example illustrates the *IEditableObject* class with an implementation of a *Customer* class.

```
Imports System.ComponentModel

Public Class Customer
    Implements IEditableObject

    Private _transactionStarted As Boolean = False
    Private _firstName, _originalFirstName As String
    Private _lastName, _originalLastName As String
    Private _phoneNumber, _originalPhoneNumber As String

    Public Property FirstName() As String
        Get
            Return _firstName
        End Get
        Set(ByVal Value As String)
            _firstName = Value
        End Set
    End Property

    Public Property LastName() As String
        Get
            Return _lastName
        End Get
        Set(ByVal Value As String)
            _lastName = Value
        End Set
    End Property

    Public Property PhoneNumber() As String
        Get
            Return _phoneNumber
        End Get
        Set(ByVal Value As String)
            _phoneNumber = Value
        End Set
    End Property

    Public Sub BeginEdit() Implements IEditableObject.BeginEdit
        If Not _transactionStarted Then
            _transactionStarted = True

            _originalFirstName = _firstName
            _originalLastName = _lastName
            _originalPhoneNumber = _phoneNumber
        End If
    End Sub

    Public Sub CancelEdit() Implements IEditableObject.CancelEdit
```

```
    If _transactionStarted Then
        _transactionStarted = False
        _firstName = _originalFirstName
        _lastName = _originalLastName
        _phoneNumber = _originalPhoneNumber
    End If
End Sub

Public Sub EndEdit() Implements IEditableObject.EndEdit
    If _transactionStarted Then
        _transactionStarted = False
        _originalFirstName = ""
        _originalLastName = ""
        _originalPhoneNumber = ""
    End If
End Sub
End Class
```

IDataErrorInfo

This interface offers custom error information to which controls can bind. During data binding, this allows controls to retrieve specific error information from the data source itself. For example, if a particular column in a *DataTable* object has an *Integer* type, setting a field to a string value for this column will cause the data source to return an appropriate error. This interface provides the following two properties:

Error: Returns an error message indicating what is wrong.

Item: An indexer that gets the error message for the specified column name or property name.

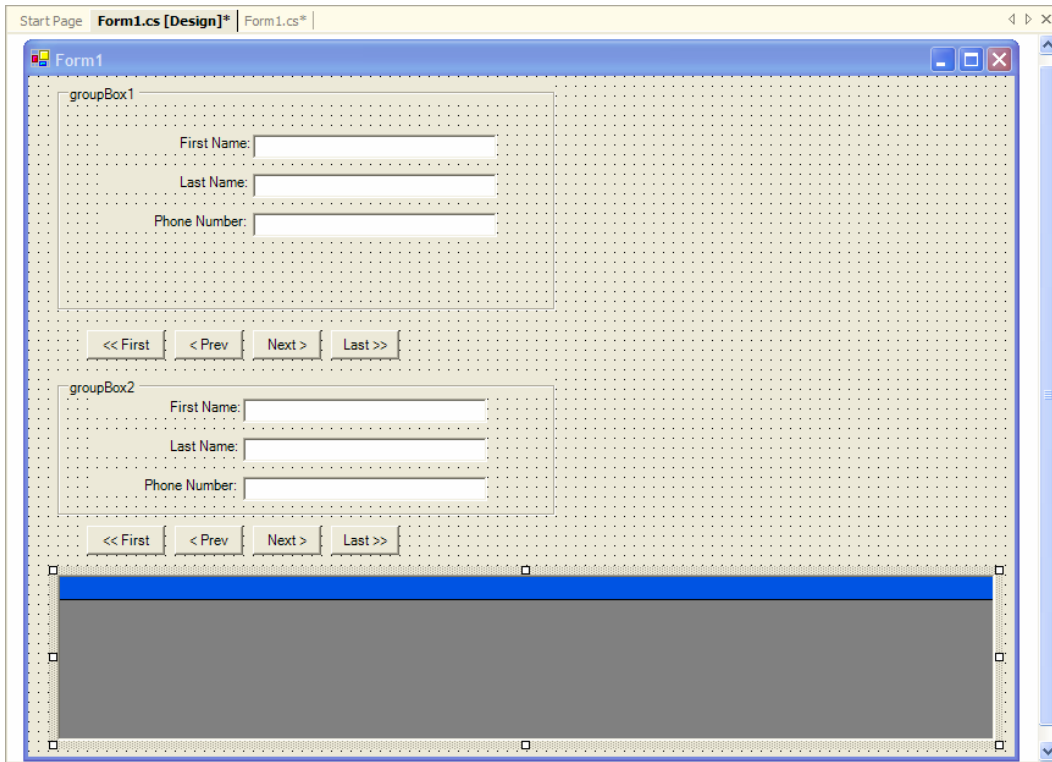
Complex Binding Example

In one of the previous sections, you learned how to implement simple binding. I discussed how to bind public properties of controls to properties of objects and columns of *DataTable* objects, as well as how to synchronize the data. But there are also situations where an entire collection

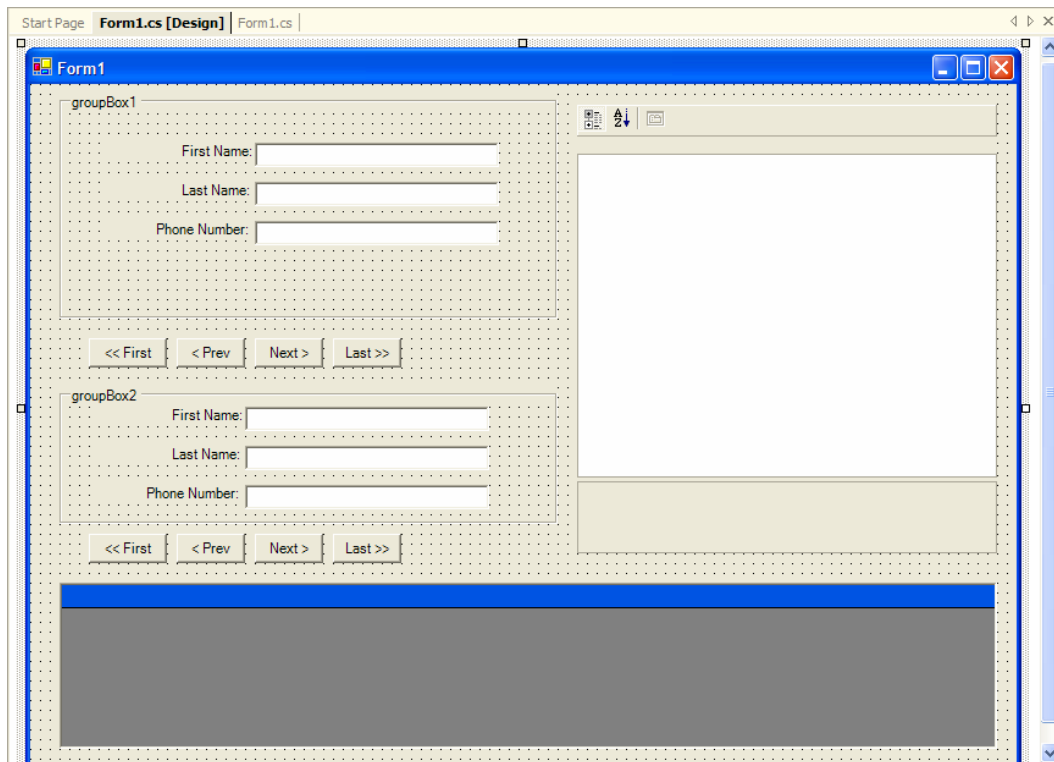
of data needs to be bound, such as when viewing a list of software bugs. Typical controls that support such complex data binding include the DataGrid, ListBox, ComboBox, and ErrorProvider controls.

All complex data bound controls expose two important properties: *DataSource* and *DataMember*. The *DataSource* property can be any type derived from the interfaces discussed previously. The *DataMember* property is a string value that represents either the table name or a public property with which binding should occur. For example, if the *DataSource* property is a *DataSet* object, the *DataMember* property should specify the table with which binding should occur; if the *DataSource* property is a collection, the *DataMember* property should be *Nothing*; and if the *DataSource* property is an object that exposes a collection to be bound through one of the object's public properties, the *DataMember* property should be the name of that property.

This example utilizes the DataGrid control to bind to the array of customers used in the previous simple binding example. First, drag a DataGrid control from the Toolbox to the form that you created in the simple binding example. The DataGrid control will display the entire list of customers (which is only three items in this example). This example uses the row navigation events of the DataGrid control to change the current item in the first GroupBox control. You will have to rearrange the controls and resize the form, as shown here:



Also, go ahead and drag a PropertyGrid control to the form. The PropertyGrid control is not added to the Toolbox by default, so right-click the Toolbox, click “Add/Remove Items...” navigate to the .NET Framework Components tab, and select the PropertyGrid control. The PropertyGrid control will be synchronized with the current item in the list, displaying that item’s properties. Arrange the PropertyGrid control on the form as follows:



Name your controls as follows:

DataGrid: `_dataGrid`

PropertyGrid: `_propertyGrid`

Now, using the code from the simple binding example, add these two statements to the constructor of the form:

```
_dataGrid.DataSource = _customers  
_propertyGrid.DataBindings.Add("SelectedObject",  
_groupBox1.BindingContext[_customers], "Current")
```

Here is a breakdown of what is happening with this code. First, the *DataSource* property of the DataGrid control is set to an instance representing the collection of customers. Since this is the collection with which binding should occur, there is no need to set the *DataMember* property. Next, the PropertyGrid control is synchronized with the current customer of the first GroupBox control. The PropertyGrid control exposes a property, *SelectedObject*, which is used to display all public browsable properties of the selected object.

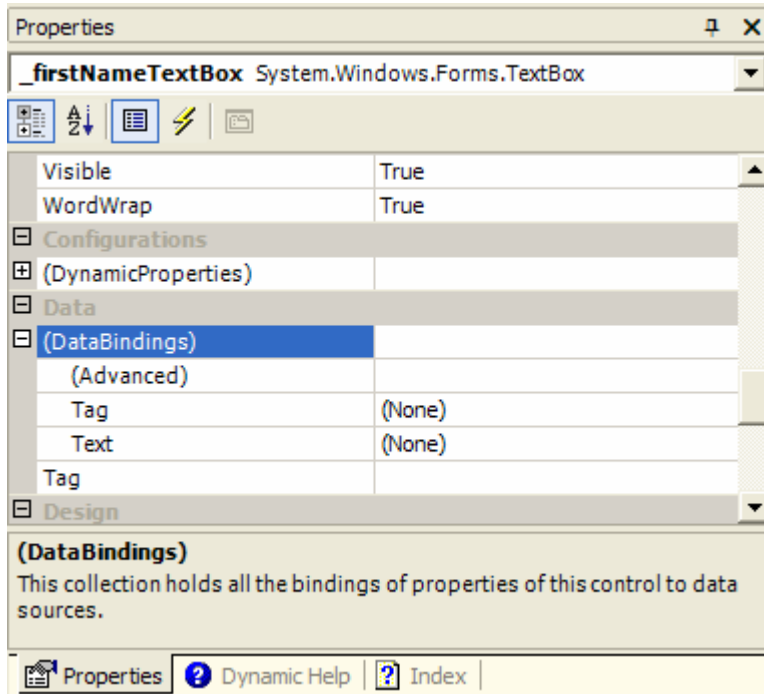
Now compile and run the sample. Notice that by clicking the navigation buttons of the first GroupBox control, the PropertyGrid control automatically updates its display for the new current object. It does this with only one line of code. But there is one small problem: selecting different rows of the DataGrid control does not cause navigation in the first GroupBox control as you probably expected. By now, you should already know the cause of the problem. It's the *BindingContext* class. Since the code does not explicitly assign a *BindingContext* object to the DataGrid control, the DataGrid control uses the form's default *BindingContext* object. And in this example, the form's default *BindingContext* object isn't managing any data bindings. To get around this problem, assign the *BindingContext* object of *_groupBox1* to the *BindingContext* property of the DataGrid control, as shown here:

```
_dataGrid.DataSource = _customers  
_dataGrid.BindingContext = _groupBox1.BindingContext  
_propertyGrid.DataBindings.Add("SelectedObject",  
    _groupBox1.BindingContext(_customers), "Current")
```

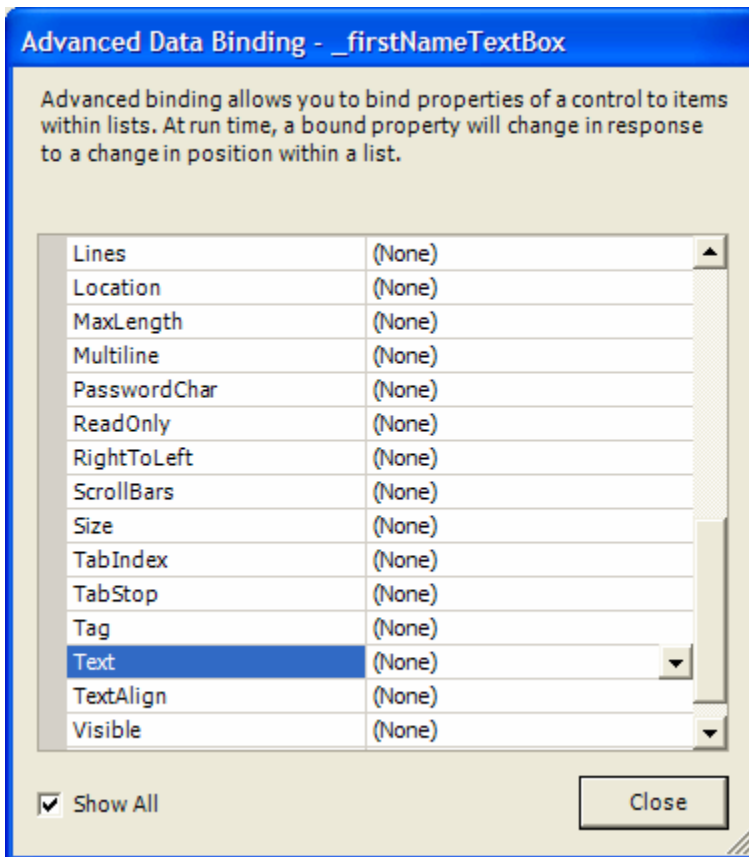
Now if you compile and run the code, navigation should work as expected.

Advanced Data Binding

As you learned from the previous sections, all controls on a form will contain a *DataBindings* property. Here is a view of the property browser displaying the *DataBindings* property of a TextBox control:



By default, only the *Text* and *Tag* properties are shown when the *DataBindings* property is expanded. The *Tag* property of a control is used to provide custom data associated with the control. You may add additional properties to this expanded list by choosing them from the Advanced Data Bindings dialog box, which you can access by clicking the ellipsis next to “Advanced.” This dialog box is shown here:



For advanced data binding to work, your form must contain a design-time data source component. You can provide a design-time data source component by dragging a DataSet component from the Toolbox.

Once you have a data source component, you simply associate each property of a control you want bound in the “Advanced Data Binding” dialog box with the data source component. As you associate each property with the data source, the property will be added along with the *Text* and *Tag* properties beneath “(DataBindings)” in the property browser.

When using advanced data binding, you must be sure that properties are not bound twice. If you use the Advanced Data Binding dialog box to bind a control’s property, and then use the

control's *DataBindings* property programmatically to bind the same property, a runtime error will occur.

Dynamic Properties

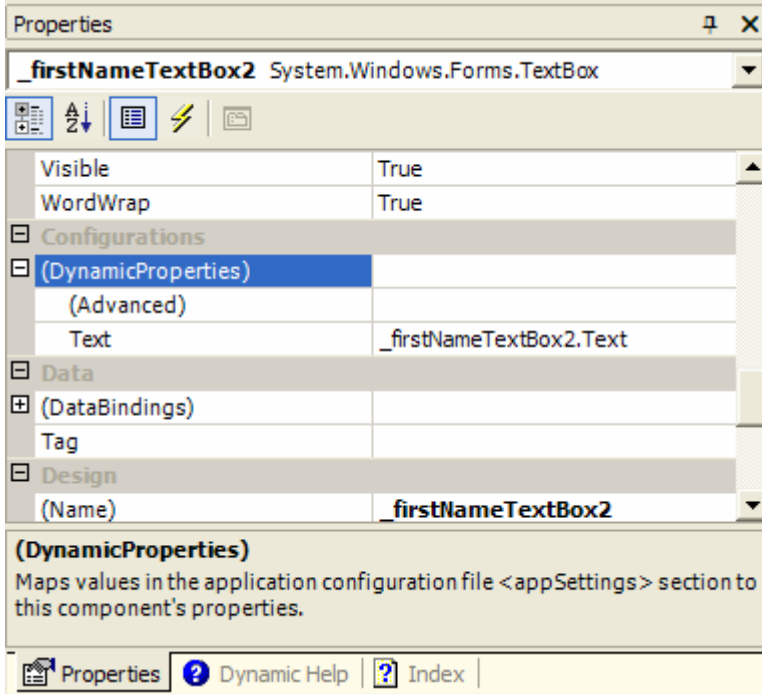
By default, any properties set on a control in the designer are persisted either in code or in a resource file. If the *Localizable* property of the parent form is set to *True*, a control's properties are persisted in a resource file. Otherwise, they are persisted in code.

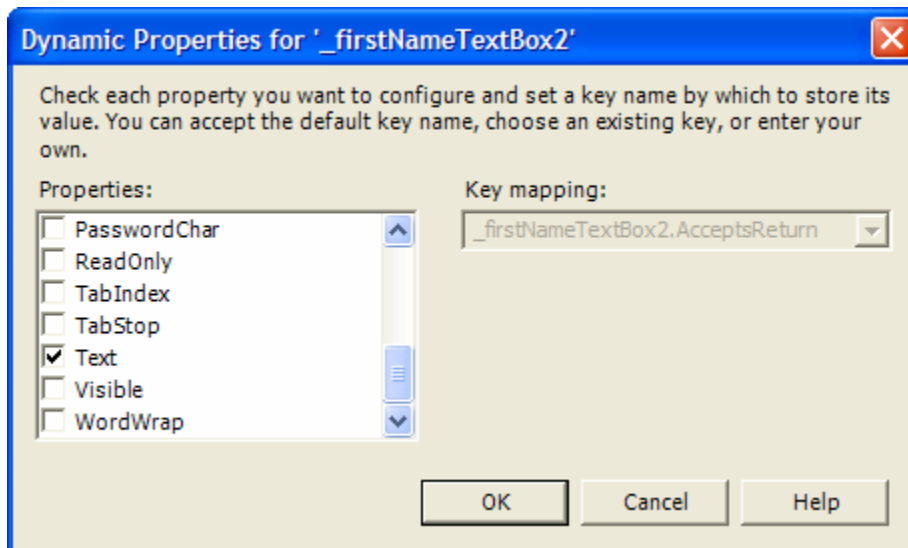
There may be situations, however, where certain properties should be customized by the user or some other customization application. These scenarios include customizing the *BackColor* property of a form or the *FlatStyle* property of a button. In situations like these, it is common to implement configuration files. And with Windows Forms, this capability is built in.

Every Windows Forms application will expect to read from configuration files that conform to a standard naming convention. The format is *MyApp.exe.config*. For example, if your application is named *MyApp.exe*, then your configuration file should be named *MyApp.exe.config*, and it should be placed in the same directory as the application itself. Note that you should not add a file like *MyApp.exe.config* file to a Windows Application project, because it might get regenerated. If you need a configuration file to use during development, then you should use *App.config*. Set the Build Action property of the *App.config* file to *None*. The compiler uses this file to regenerate *MyApp.exe.config* in the appropriate runtime directory.

Every Windows Forms control has a design-time property named *DynamicProperties*. This property is a collection of name-value pairs that map key names to property names. These key names are stored in the application's configuration file. Property values can then be persisted in the configuration file and retrieved at run time during form initialization. In code, each property will be associated with a key name. This key name is used to read the property's

value. Here are snapshots of the DynamicProperties section in the property browser and the Dynamic Properties dialog box for the *Text* property of a TextBox control:





As more properties are selected, indicated by checked check boxes, these properties will be added to the `DynamicProperties` section in the property browser. The Key mapping on the right contains a list of all keys that have been added and that map to values that can be cast to a checked property's type. In other words, you can specify more than one property to use the same key, but the value of that key must be able to be cast to the types of all properties that use the same key; otherwise, the key won't appear in the Key mapping list. This is determined by the first property that is configured. For example, the `ReadOnly` property will have an available key mapping of all `Boolean` properties that have been configured. If the `ReadOnly` property is checked, the `Text` property will then have an available key mapping of all `Boolean` properties and all `String` properties that have been configured.

Summary

This chapter discussed the concepts and the architecture behind Windows Forms data binding. It discussed the relationship between controls and the `BindingContext` class and covered the interfaces related to data binding. It also provided examples of the two types of data binding:

simple binding and complex binding. Simple binding involves binding a single property of a control to a single column or property of a data source. Complex binding involves binding a control to a collection of objects and properties or tables and columns of a data source.

Most importantly, you learned that the synchronization of a data source with its controls is loosely coupled through the *CurrencyManager* class. Data no longer remembers its current position. This allows for multiple bindings and synchronization on the same data.

Lastly, this chapter discussed dynamic properties, their use, and how they are persisted in application configuration files. This concept allows the user to be in more control over the user interface, if desired.

You should now have a better understanding of data binding in Windows Forms. And remember, data no longer remembers its current position.