17

CHAPTER

# Building Your Own Bound Controls

**In This Chapter**

Building class objects

Building user controls

Implementing data binding

In this chapter, I'll show you how to build your own COM components that can be bound to a data source. I'll also cover how to create your own data source, which can be used in place of the ADO Data Control.

## Introducing Data Sources and Consumers

In the ADO world, everything can be classified into one of two groups: data sources and data consumers. A *data source* produces data that can be read by a data consumer. A *data consumer* may update data provided by a data source. The technique used to connect the data source to the data consumer is known as *data binding.* This is the same technique that you use to bind a text box control (data consumer) to the ADO Data Control (data source).

Data sources and data consumers are just COM objects that support a few special properties and events. COM objects can take the form of ActiveX controls, ActiveX DLLs, and ActiveX EXEs. The form you choose depends on how you want your objects to work. The same principles of binding apply equally to all three types of COM components.

## Data sources

A data source is responsible for generating an ADO `Recordset` object, which is accessed by a data consumer. A common example of a data source in Visual Basic is the ADO Data Control. While you might think that the ADO Data Control is very complicated, it really isn't. I'll show you how to build one later in this chapter, in the section called "Building a Data Source".

A data source has two key elements: the `DataSourceBehavior` property and the `GetDataMember` event. When the `DataSourceBehavior` property has a value of `vbDataSource` (1), the object becomes a data source. The `GetDataMember` event will be triggered whenever the data source needs an object pointer to the `Recordset` object. Whenever the current record in the `Recordset` object changes, the data consumer will be notified and it can update its information.

The data source has the option of making more than one recordset available to a data consumer. The `DataMember` property is used to identify the specific recordset that the data consumer wants to access. This value is passed to the object using the `GetDataMember` event. If your data source only returns a single `Recordset` object, then this property can be ignored.

## Data consumers

A data consumer receives data generated by a data source. It doesn't deal with the `Recordset` object directly, but rather it identifies one or more fields of data that it wants to access and the properties that will receive it. Then, as the current record pointer in the `Recordset` moves, the data source will assign the updated data to the specified properties in the data consumer.

Data consumers come in two flavors: simple and complex. A ***simple data consumer*** binds only a single property to a data source, while a ***complex data consumer*** can bind multiple properties to a data source. The type of data consumer is identified by the `DataBindingBehavior` property. A value of `vbSimpleBound` (1) means that the object has one property bound to the data source, while a value of `vbComplexBound` (2) means that the object has multiple properties bound to the data source.

After selecting the type of data consumer, you need to adjust the attributes for each of the properties you want to bind to the data source. This involves using the Procedure Attributes dialog box to mark the property as data bound. If you're not familiar with this tool, see "Setting Property Attributes," later in this chapter.

While a simple data consumer can specify the necessary binding information directly in the object's properties, a complex data consumer can't. It uses the `DataBindings` collection and the `DataBinding` object to hold the definitions. The `DataBinding` object contains all of the properties that would have been used in the object itself had the object been a simple data consumer. Thus, you need one `DataBinding` object for each property you wish to bind. The `DataBindings` collection holds the set of `DataBinding` objects for the data consumer.

# A Brief Introduction to COM Components

Many database applications, which are written as a series of large programs, could benefit from rewriting them to use COM (component object model) components. COM components are ideal for isolating commonly used functions from the programs that use them. You can use them to hold your application logic, such as how to validate a particular data element, or you can use them to hold your database logic, such as how to retrieve a set of information from the database.

By isolating commonly used routines away from your main programs, you make it easier to update your application. Since COM components live in files external to your program's EXE file, you can replace one component without necessarily affecting the others. This will let you replace one small file, rather than replacing all of the program files that make up your application.

## What is a COM component?

A COM component is an object module that contains executable code that can be dynamically loaded into memory at runtime. Communication with a COM component follows a fairly strict set of rules, the details of which a Visual Basic programmer really doesn't have to worry about. The Visual Basic programmer only has to worry about properties, methods, and events that form the interface to the COM component.

Recall that COM components come in three flavors: ActiveX DLLs, ActiveX EXEs, and ActiveX Controls. Every Visual Basic programmer is familiar with ActiveX controls, which are placed on their forms to perform various functions. ActiveX DLLs (Dynamic Linking Libraries) and ActiveX EXEs (Executables) are really just a series of one or more Visual Basic `Class` modules that are compiled into a single file.

The key difference between an ActiveX control and ActiveX DLLs and EXEs is that a control has a visible presence that can be included on a Visual Basic form. ActiveX DLLs and EXEs are built using the Visual Basic `Class` modules, while an ActiveX control is built using a `UserControl` module.

## Using class modules

Class modules are used to build both ActiveX DLLs and ActiveX EXEs. Which of these you choose depends on how you plan to use them. The code in an ActiveX DLL runs inside your program's address space and responds quicker to processing requests than an ActiveX EXE. An ActiveX EXE runs independently of your program and need not reside on the same computer. This allows you to spread your processing over multiple computers.

A `Class` module is just a template for an object. It describes the public properties, methods, and events that other programs will use, and it contains the code and local data storage definitions that are used to respond to various processing requests.

*Properties* appear to the user as a variable that is part of the object. They are implemented as either a public module level variable that is visible to the users of the object or as a special routine that is called when you want to return or set a property value. Property values are returned using the `Property Get` statement, which is equivalent to the `Function` statement. The `Property Set` statement is used to assign object values to a property, while the `Property Let` is used to assign values to all other types of variables. While it is legal to use a set of parameters with the `Property` statements, you should make the PropertyGet and Let statements have the same parameter list.

*Methods* are normal functions and subroutines that are used to perform actions using information in the object. These routines must be declared as Public in order to be accessed by the object's user. Otherwise, they may only be called by other routines in the object itself.

*Events* are subroutines external to the object that can be called from within the object. The subroutine header must be included in the `Class` module in order to define the parameters that may be passed to the subroutine, while the actual subroutine will be coded by the object's user and will reside in the user's program.

Tip

**Eventless:** Even though you have the ability to use events in your class module, you shouldn't use them, because they limit the usefulness of your objects.

## Persistable objects

It is often desirable to create objects that have a memory of their property values. This means that while you run a program, you can create an object and save its values so that the next time you run the program, those values will be restored.

Tip

**Queuing for bytes:** When using message queuing, it is important for you to make any objects you send persistent. If you don't, you will lose your data when the receiving program re-creates your object (see Chapter 19 for more information about message queuing).

Persistence is managed though the `PropertyBag` object. When you create an instance of the object for the first time, you must provide initial values for each of the properties. As part of the process of destroying the object, you can save the property values into the `PropertyBag` object. Then when the object is re-created, you can restore the property values saved in the `PropertyBag`, which means that the object has all of the same values that it had before it was destroyed.

Consider for a moment how Visual Basic implements ActiveX controls. While in design mode, VB actually creates an instance of the control and lets you manipulate its properties through the Properties window. Then, when you go to run the program, the design time instance of the control is destroyed, but before it is destroyed, it saves a copy of its properties in the `PropertyBag`.

When you run the program, a run-time instance of the control is created, which reads its property settings from the PropertyBag. All of your design time settings are present in the new instance of the control, so the control can draw itself on the form in the proper location, as dictated by the Left, Top, Width, and Height properties. This also means that the rest of the property values are available so the control should behave as per your design.

## Class module properties

Table 17-1 lists the internal properties for a class object. These properties determine how the object will behave. Note that these properties will not be visible to the user, but merely determine some of the capabilities of your object.

Table 17-1
**Properties for a Class Module**

| *Property* | *Description* |
|---|---|
| DataBindingBehavior | An enumerated data type which can be vbSimpleBound (1), meaning that only a single property can be bound; vbComplexBound (2), meaning that multiple properties can be bound; or vbNone (0), meaning that the object isn't a data consumer. |
| DataSourceBehavior | An enumerated data type which can be vbDataSource (1), meaning the object will act as a data source; or vbNone (0), meaning the object isn't a data source. |
| Instancing | An enumerated data type which describes how an object will be reused. A value of Private (0) means that even though your object may have Public members, no programs outside your current project can access it. PublicNotCreatable (1) means that other programs can use this object but can't create it. MultiUse (5) means that other programs can create and use this object. GlobalMultiUse (6) is identical to MultiUse, but the properties and methods of this class can be used without explicitly creating an instance of the object first. |
| Persistable | An enumerated data type that allows you to keep property values between instances. If you set this property to Persistable, additional events will be available in your class module to initialize, save, and restore property values. NotPersistable implies that the object's properties will be reinitialized each time an instance of the object is created. |

## Class module property routines

When you set the `DataBindingBehavior` property to `vbComplexBound` (2), the four property routines described below will automatically be added to your class object to handle the binding issues.

### Public Property Get DataSource() As DataSource

The `DataSource` `Propety Get` routine is used to return the current data source.

### Public Property Set DataSource(ByVal objDataSource As DataSource)

The `DataSource` `Property Set` routine is used to assign a new value to data source, where `objDataSource` is an object reference to a data source.

### Public Property Get DataMember() As DataMember

The `DataMember` `Property Get` routine is used to return the current data member.

### Public Property Let DataMember(ByVal DataMember As DataMember)

The `DataMember` `Property Let` routine is used to assign a new data member to the object, where `DataMember` is an object reference to a data member.

## Class module events

By default, a `Class` module has two events: `Initialize` and `Terminate`, which are called when the object is created and destroyed. However, if you make the object a data source, the `GetDataMember` event is also included.

### Event GetDataMember(DataMember As String, Data As Object)

The `GetDataMember` event is triggered when a user requests a recordset by specifying a value for `DataMember`. Note that this event is present only when the `DataSourceBehavior` property is set to `vbDataSource`.

`DataMember` is a `String` value that contains the name of the data member that the event should return.

`Data` is an object reference that you must return containing a `Recordset` object associated with the `DataMember` value specified.

### Event Initialize()

The `Initialize` event is triggered when a new instance of the object is created. You should initialize your local variables and allocate any module level objects that your object will be using.

### Event InitProperties()

The `InitProperties` event is triggered when a new instance of the object is created. You should use this routine to initialize properties, rather than the `Initialize` event, to avoid conflicts that may occur when using the `Initialize` event and the `ReadProperties` event.

### Event ReadProperties(PropBag As PropertyBag)

The `ReadProperties` event is triggered when an old instance of an object needs to restore its properties, where `PropBag` is an object reference to a `PropertyBag` object, which is used to store the property values.

### Event Terminate()

The `Terminate` event is triggered when the object is about to be destroyed. You should set all of the object variables that still exist to `Nothing`, so that you can reclaim the memory and other resources they are using. If you don't do this, the objects will remain in memory until the user program for an ActiveX DLL or the ActiveX EXE ends.

### Event WriteProperties(PropBag As PropertyBag)

The `WriteProperties` event is triggered before a persistent object is destroyed. In this event, you must use the `PropertyBag` object to save the values of your properties so that later they may be restored using the `ReadProperties` event. `PropBag` is an object reference to a `PropertyBag` object, which is used to store the property values.

## The PropertyBag object

The `PropertyBag` object contains the information that needs to be saved between instances of an object. It works with the `InitProperties`, `ReadProperties`, and `WriteProperties` events in the `Class` module and `UserControl` module. The sole property for the `PropertyBag` object is `Contents`, which is a `Byte()` containing the data that is stored in the `PropertyBag`.

### Function ReadProperty(Name As String, [DefaultValue]) As Variant

The `ReadProperty` method is used to retrieve a property value from the property bag.

`Name` is a `String` containing the name of the property value stored in the property bag. `DefaultValue` is a `Variant` containing the default value of the property.

### Sub WriteProperty(Name As String, Value, [DefaultValue])

The `WriteProperty` method is used to store a property value in the property bag.

`Name` is a `String` containing the name of the property value stored in the property bag.

`Value` is a `Variant` containing the value of the property.

`DefaultValue` is a `Variant` containing the default value of the property.

**Default consistently:** You must specify the same value for `DefaultValue` in the `ReadProperty` and the `WriteProperty` methods, since the property value is stored only when it is different from the specified default value. By doing this, Microsoft reduces the amount of data you need to store in the property bag. However, if you're not careful, you can get different property values each time you save and restore an object.

# Building a Data Source

The classic data source that everyone builds when creating the first data source is a clone of the ADO Data Control. It's an ideal control to build, since it offers a visual component that the user can interact with, and not much code is necessary beyond what is required to manage the `DataBindings` collection. My implementation of this control is called the `DataSpinner` control (see Figure 17-1).

The `DataSpinner` control consists of two command buttons and a text box. The command buttons are used to scroll forward and backward through the data in the control, while the text box simply represents a way to display information in the control.

**UserControls vs. Classes:** All of the steps described here apply equally when creating a UserControl data source or a Class module data source.

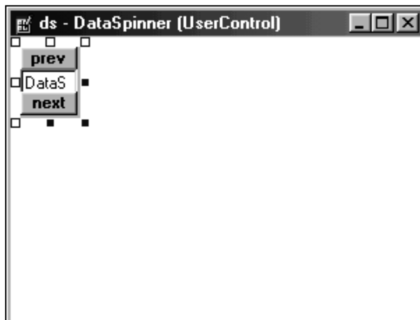


**Figure 17-1:** Designing the DataSpinner control

## Module-level declarations

Before I dig into the code for the DataSpinner control, I want to go over the module-level declarations (see Listing 17-1). Of the six variables declared, four are used to hold property values. The `ConnectionString` is stored in `cn`; `Recordsource` is kept in `ds`; `UserName` is stored in `us`; and `Password` is stored in `pw`. The other two variables hold a connection to the database server and the current recordset that supplies the bound data.

**Listing 17-1: Module-level declarations in DataSpinner**

```
Private cn As String
Private db As ADODB.Connection
Private ds As String
Private pw As String
Private rs As ADODB.Recordset
Private us As String

Event Click()
Event Scroll()
```

Two events are also defined. The `Click` event is triggered whenever someone clicks in the text box, while the `Scroll` event is triggered whenever someone presses either the `prev` or `next` button.

## Binding data

Since the `DataSpinner` control is a data source, you must set the `DataSource Behavior` to `vbDataSource` (1) in the `UserControl` properties. This will expose the names of the columns retrieved from the database and allow them to be bound to the control. This information is gathered from the `GetDataMember` event (see Listing 17-2).

**Listing 17-2: The UserControl_GetDataMember Event in DataSpinner**

```
Private Sub UserControl_GetDataMember _
   (DataMember As String, Data As Object)

If db Is Nothing Then

   If Len(cn) > 0 And Len(us) > 0 Then
      Set db = New ADODB.Connection
```

*Continued*

**Listing 17-2** *(continued)*

```
      db.Open cn, us, pw

      Set rs = New ADODB.Recordset
      rs.Open ds, db, adOpenStatic, adLockPessimistic
      If Not (rs.EOF And rs.BOF) Then
         rs.MoveFirst
      End If

   End If

End If

Set Data = rs

End Sub
```

The `GetDataMember` event is called whenever Visual Basic needs a reference to the underlying `Recordset` object. This allows the caller to find out the names of the columns returned at design time or to retrieve the current row of information to be displayed in the bound controls.

This event will also be called each time the `DataMember` property in the control is changed. Typically, you would return a different `Recordset` based on the value of `DataMember` argument: however, in this case, I'm always going to return the same `Recordset` no matter what value is supplied in the `DataMember` argument.

I start this event by checking to see if the module-level variable `db` is `Nothing`. If it is, it means that I haven't opened a connection to the database yet. Then I can verify that someone entered a connection string (`cn`) and a user name (`us`) before opening a database connection.

Once the connection is open, I can open the `Recordset` object. In this case, I choose to always specify a static cursor and pessimistic locking. However, I could have easily created properties for these values also. Then I do a `MoveFirst` to ensure that the current record pointer is pointing to the first record in the `Recordset`.

At the end of the routine, I'll return an object pointer to the `Recordset` object using the `Data` parameter. If I couldn't open the `Recordset`, then the `rs` will have a value of `Nothing` and the program trying to bind to the control will get an error. Otherwise, the information contained in `rs` will be used in the binding process.

## Moving through the recordset

Once the control has been initialized, you write your program just like you would normally write a Visual Basic program. For instance, in Listing 17-3, the code you see would be very typical of routine to move current record pointer to the next record in a recordset.

**Listing 17-3: The Command2_Click event in DataSpinner**

```
Private Sub Command2_Click()

If Not rs Is Nothing Then
   rs.MoveNext
   If rs.EOF Then
      rs.MoveFirst

   End If
   RaiseEvent Scroll

End If

End Sub
```

The only part of the code that is different is the `RaiseEvent` statement. In this case, the `RaiseEvent` statement is used to trigger the `Scroll` event that was declared in the module-level declarations. This event allows someone using this control to detect when the current record pointer has changed.

## Exporting recordset information

A programmer using this control might find it useful to look at the underlying `Recordset` object from time to time. The easiest way to handle this is to create a `Property Get` routine, like the one shown in Listing 17-4. This routine merely returns an object reference to `rs`. Because I don't want anyone changing the object, I didn't include a `Property Set` routine. This makes the `Recordset` property read-only, and the programmer using the control can't substitute their recordset for the one in the control.

**Listing 17-4: The Recordset Property Get routine in DataSpinner**

```
Public Property Get Recordset() As ADODB.Recordset

Set Recordset = rs

End Property
```

## Using the DataSpinner control

Adding the `DataSpinner` control to your application is merely a matter of dragging the control onto your form and setting a few properties. In this case, I set the `Connection` property to `provider=sqloledb;data source=athena;initial catalog=VB6DB` and the `RecordSource` property to `Select * From Customers`. This will return all of the records in the Customer table.

The `Scroll` event is a good place to display the `AbsolutePosition` property of the underlying `Recordset` (see Listing 17-5). Note that I use the `Recordset` property described previously in this chapter under "Exporting Recordset Information" to access this information.

**Listing 17-5: The DataSpinner1_Scroll event in Customer Viewer**

```
Private Sub DataSpinner1_Scroll()

DataSpinner1.Text = _
   FormatNumber(DataSpinner1.Recordset.AbsolutePosition, 0)

End Sub
```

# Building a Data Consumer

To go along with the data source I just built, I created a simple data consumer called `AddressDisplay` (see Figure 17-2). The control is composed of six text boxes and six label controls. Each text box has its own unique property. Assigning a value to any of these properties simply displays the value in the appropriate text box.

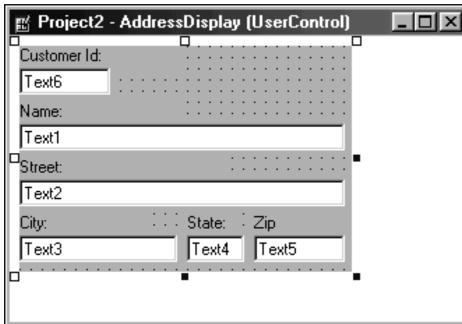


**Figure 17-2:** Designing the AddressDisplay control

## Exposing properties

Each text box on the control has a `Property Get` routine and `Property Let` routine that manage the information associated with each control. Listing 17-6 shows a typical `Property Get` routine that retrieves the value from the text box that is used to display the `Name` field from the database.

**Listing 17-6: The CName Property Get routine in AddressDisplay**

```
Public Property Get CName() As String

CName = Text1.Text

End Property
```

Note

**What's in a name:** You may be wondering why I named this property `CName`, rather than calling it `Name` after the database field. The reason is simple. Each ActiveX control already comes with a property called `Name`, and you can't override this property.

Changing a property is somewhat more complicated than you might expect. In the `CName Property Let` routine (see Listing 17-7), you notice that I call the `CanPropertyChange` method to determine if I can change the value. This prevents errors from occurring if someone chooses to bind the control to a read-only `Recordset`.

**Listing 17-7: The CName Property Let routine in AddressDisplay**

```
Public Property Let CName(s As String)

If CanPropertyChange("CName") Then
   Text1.Text = s
   PropertyChanged "CName"

End If

End Property
```

I also use the PropertyChanged method to notify the control that this property has changed. This is important, since it ensures that the control knows when a property has changed. If the control isn't aware that the property has changed, it may not properly save the property values in the Recordset.

You should closely examine your code and the objects on your UserControl to make sure that the value of the property can't be changed without calling the PropertyChanged method. In the case of this control, it is possible for a user to change the contents of the text box on the control. So I need to include a Change event for each text box to indicate that the value of the property has been changed (see Listing 17-8).

**Listing 17-8: The Text1_Change event in AddressDisplay**

```
Private Sub Text1_Change()

PropertyChanged "CName"

End Sub
```

## Setting property attributes

In order to allow a property to be bound to a data source, you have to identify the property as data bound. To set the attribute on the property, you need to use the Procedure Attributes tool (see Figure 17-3). To start the tool, choose Tools ➪ Procedure Attributes from the Visual Basic main menu.

To mark a property as data bound, select the name of the property in the Name drop-down box and press the Advanced button. This will display a window similar to the one shown in Figure 17-4. At the bottom of the window is the Data Binding section.

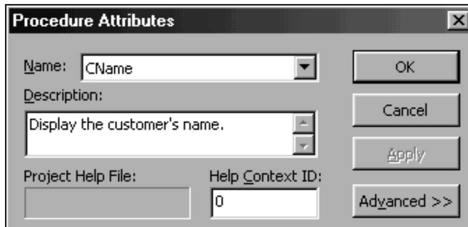


**Figure 17-3:** Setting property attributes

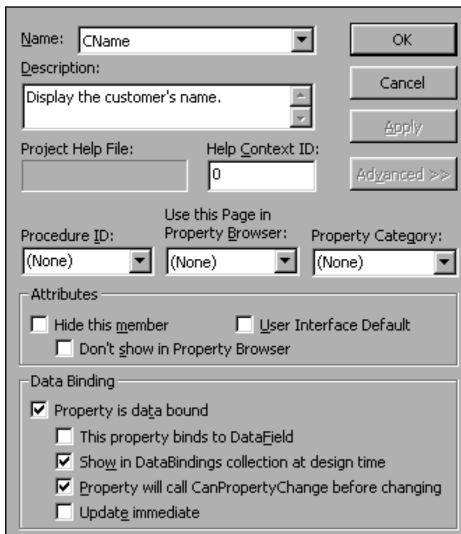


**Figure 17-4:** Viewing advanced procedure attributes

In the Data Binding section, place a check mark in the Property is data bound check box. This will enable the check boxes below it. Then you should check Show in DataBindings collection at design time and Property will call CanPropertyChange before changing check boxes. This will allow you to bind the property to a data source at design-time and let Visual Basic know that you are using the `CanPropertyChange` method in the property routines.

You don't have to close the window after selecting the information for a single property. Just select a different property in the Name drop-down box and set the desired values. Once you enter all of this information, you can verify it by adding your control to a simple program and displaying the Data Bindings window (see Figure 17-5).

If you check the This property binds to DataField check box in the window shown in Figure 17-4, you can bind the property to the control's `DataField` property. This means that the programmer using your control doesn't have to use the Data Bindings window to bind a field in a data source to this property.

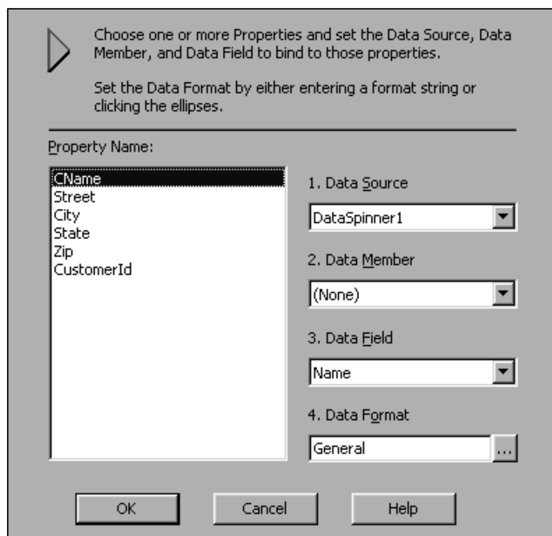


**Figure 17-5:** Binding properties in the Data Bindings window

Tip

**Data binding and other stuff too:** The Procedure Attributes tool performs many useful functions, in addition to allowing you to mark a property as data bound. You can add a description to each property that will show up when you view the component in the Object Browser window. You can mark a property as hidden, assign the property to a specific property page (if you implement custom property windows) and you can assign the property to a specific category so that it can be separated out in the Properties window.

## Persisting properties

One of the important housekeeping duties you need to worry about in an ActiveX control is making sure that the values someone assigns to the control at design-time are properly saved between development sessions, and also available at run-time. This is managed by using the `PropertyBag` object (introduced earlier in this chapter under "Persistable Objects") and the `InitProperties`, `ReadProperties`, and `WriteProperties` events.

### Initializing properties for the first time

The `InitProperties` event is triggered the first time a control (or any other persistent ActiveX component) is instantiated. You should include code in this event to make sure that all of the property values are properly initialized. In this example, I choose to assign descriptive values for each of the fields in the control (see Listing 17-9).

While I could have assigned these values directly using the Properties window for each of the controls used in this control, I wanted to show you the types of things you might do in this event.

**Listing 17-9: The UserControl_InitProperties event in AddressDisplay**

```
Private Sub UserControl_InitProperties()

Text1.Text = "CName"
Text2.Text = "Street"
Text3.Text = "City"
Text4.Text = "State"
Text5.Text = "Zip"
Text6.Text = "CustomerId"

End Sub
```

## Saving property values

When the control is destroyed, the `WriteProperties` event is triggered so you can save your current property values (see Listing 17-9). A `PropertyBag` object is passed to this event to hold all of the property values. To save the current value of each property, you must call the `WriteProperties` event and specify the property name, the property value, and the default value.

**Listing 17-9: The UserControl_WriteProperties event in AddressDisplay**

```
Private Sub UserControl_WriteProperties(PropBag As PropertyBag)

PropBag.WriteProperty "CName", Text1.Text, "CName"
PropBag.WriteProperty "Street", Text2.Text, "Street"
PropBag.WriteProperty "City", Text3.Text, "City"
PropBag.WriteProperty "State", Text4.Text, "State"
PropBag.WriteProperty "Zip", Text5.Text, "Zip"
PropBag.WriteProperty "CustomerId", Text6.Text, "CustomerId"

End Sub
```

If the current value of the property is different than the default value, it will be saved in the property bag. Otherwise, the value will be discarded. While this saves space in the `PropertyBag` object, it may cause problems if you don't use the same default value consistently.

## Reading properties after the first time

The `InitProperties` event is only called once, when the control is instantiated for the first time. Each time after that, the `ReadProperties` event will be called. In this

**event, you just need to load the properties you saved in the** `WriteProperties` **event (see Listing 17-10). As you might expect, you need to use the** `ReadProperty` **method to retrieve each property value from the** `PropertyBag` **object.**

### Listing 17-10: The UserControl_ReadProperties event in AddressDisplay

```
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)

Text1.Text = PropBag.ReadProperty("CName", "CName")
Text2.Text = PropBag.ReadProperty("Street", "Street")
Text3.Text = PropBag.ReadProperty("City", "City")
Text4.Text = PropBag.ReadProperty("State", "State")
Text5.Text = PropBag.ReadProperty("Zip", "Zip")
Text6.Text = PropBag.ReadProperty("CustomerId", "CustomerId")

End Sub
```

Note

**Initialize ain't gone:** The `Initialize` event is still present in a persistent component and you should use it to initialize various aspects to the control that need to be initialized each time the control is instantiated. You should save the `ReadProperties` and `InitProperties` events for situations where you want to keep a memory of various property values.

# Pulling It All Together

**With both the Data Spinner and the Address Display controls available, it is a simple matter to create a test program (see Figure 17-6). In this case, I simply created a new program and added both controls to the form. I then entered the appropriate values for the** `Connection`, `RecordSource`, `Username`, **and** `Password` **properties in the** `DataSpinner` **control, and bound the various properties of the** `AddressDisplay` **control to the** `DataSpinner` **control. I also added the code for the** `Scroll` **event to display the current record number.**

**Figure 17-6:** Running the Customer Viewer program

## Thoughts on Using ActiveX DLLs

Building your own COM components isn't difficult once you have a working template to follow. In this chapter I focused on how to create a data source and a data consumer using ActiveX controls. However, the steps I went through to expose the properties of a data consumer and returning `Recordset` information from a data source can be used to build other types of COM components.

In many ways, you'll find that ActiveX DLLs may be even more useful in database programming than ActiveX controls. After all, ActiveX controls are much more useful in a regular Visual Basic program than in an IIS Application. ActiveX DLLs can be used to represent information abstracted from a database rather than just presenting the collection of `Fields` from a `Recordset` object. They also are a convenient place to include application logic that can be used to validate information in the object or perform useful calculations with the data in the object. ActiveX DLLs are also easy to migrate to COM+ transactions, a topic that I'll explore in Chapter 18.

# Summary

**In this chapter you learned the following:**

- **You can easily create your own data sources similar to the ADO Data Control selecting the appropriate property values.**
- **You can build data consumers by configuring the properties of the object using the Procedure Attributes window.**
- **You can build COM components using the Visual Basic Class module.**
- **You can make an ActiveX control persistable by using the `PropertyBag` object to save and restore the values for each property.**